Vorwort

Die vorliegende Dissertation betrifft den algorithmischen Kern der künstlichen Intelligenz. Hier wird das Problemlösen als Suche in einem Zustandsraum modelliert, daß heißt, man geht aus von einem Startzustand, der das gegebene Problem beschreibt und hat Übergangsregeln, um von einem Zustand zu einem oder mehreren Nachfolgezuständen zu kommen. Diese Übergangsregeln müssen dann solange angewandt werden, bis ausgehend vom Startzustand schließlich ein gewünschter Endzustand erreicht ist. In der Regel ist man an einer kürzestmöglichen Folge derartiger Übergänge interessiert. Zwar kann man relativ einfach zeigen, daß es kein allgemeines Verfahren gibt, das für eine Menge von Übergangsregeln feststellt, ob der Zielzustand vom Startzustand mit Hilfe der Übergangsregeln überhaupt erreichbar ist. In der Praxis hat sich aber diese Problemsicht insbesondere zur Lösung von Ein-Personen-Spielen (Puzzles) doch bewährt, und man hat viele Techniken entwickelt, um in konkreten Fällen zu einer Lösung zu kommen. Die vorliegende Dissertation hat dazu einige wichtige Fortschrittte beigetragen. So werden beispielsweise allgemeine Schiebepuzzles behandelt. Die bekanntesten Vertreter sind die $(n^2 - 1)$-Puzzles; allgemeinere sind das Esel-Puzzle, das Jahrhundert-Puzzle, Man-and-Bottle und andere.

Es werden die grundlegenden Verfahren zur impliziten Graphsuche beschrieben und neue, effizientere Implementationen der klassischen Suchverfahren und von Suchverfahren mit Speicherplatzbeschränkung angegeben.

Eine interessante, neue Beobachtung ist, daß sich große Mengen von Zuständen mit Hilfe geordneten binären Entscheidungsdiagrammen (*OBDDs*) kompakt darstellen lassen. Genauer werden die Zustände des Zustandsraumes binär kodiert und die Übergangsrelation zwischen verschiedenen Zuständen durch eine Boolesche Funktion charakterisiert, die letztendlich durch ein solches *OBDD* repräsentiert wird. Es zeigt sich, daß in konkreten Fällen, also etwa für das (n^2-1)-Puzzle und das Sokoban-Spiel, man zu *OBDDs* durchaus handhabbarer Größe kommt und schon sehr einfache Heuristiken ausreichen, um zu Problemlösungen zu kommen.

Den Kern der Arbeit bilden neue Verfahren zur Erkennung und Elimination von Duplikaten. Dazu werden Suffixbäume benutzt, um darin Worte zu speichern, die Zugfolgen beschreiben, die als Abkürzungen bei der Exploration von Zustandsräumen dienen. Die Arbeit enthält eine ganze Reihe weiterer Techniken zur Beschneidung des Suchraumes, zum Lernen von Abkürzungen und zum Entdecken von Sackgassen, die schließlich zur Steigerung der Effizienz für klassische Suchverfahren in diesem Kontext genutzt werden. Schließlich wird für eine ganze Reihe von Zustandsraumsuchproblemen die Größe des Suchbaumes dadurch abgeschätzt, daß der asymptotische Verzweigungsgrad berechnet wird.

Insgesamt enthält die vorliegende Arbeit nicht nur eine Fülle von Einzelresultaten sowohl experimenteller wie theoretischer Natur, sondern liefert auch eine durchaus umfassende Sicht- und Darstellung des gegenwärtigen *State of the Art* im Bereich der heuristischen Suche.

Prof. Dr. Thomas Ottmann

Danksagung

In der Promotionszeit bin ich von der Deutschen Forschungsgemeinschaft im Rahmen des Graduiertenkollegs *Menschliche und Maschinelle Intelligenz* finanziert worden. Die vom Kolleg organisierten Veranstaltungen wie Ringvorlesungen, Frühjahrstreffen und Herbstschulen haben mich für Fragestellungen und Forschungsmethoden der Psychologie, Linguistik, Neurologie, Philosophie und Kognitionswissenschaft sensibilisiert. Im Zusammenhang mit dieser Dissertation gilt mein persönlicher Dank insbesondere den folgenden Personen:

Prof. Dr. Thomas Ottmann für die motivierende und professionelle Betreuung der Promotion. Im Verlauf der Arbeit hat Prof. Dr. Ottmann mir viele Steine aus dem Weg geräumt, indem er z. B. Reisen initiiert und bewilligt hat. Seine Lehrerfahrung war außerdem immer sehr hilfreich bei der Vorbereitung von wissenschaftlichen Vorträgen.

Prof. Dr. Richard Korf von der kalifornischen Universität in Los Angeles, mit dem ich einen Monat lang zusammenarbeiten durfte. Sein intensives Interesse an meinen Ausarbeitungen hat mich neu für die Arbeit begeistern können. Prof. Dr. Korfs wissenschaftlichen Ratschläge und Brillanz im Aufschrieb von Ergebnissen bereicherten mich sehr.

Prof. Dr. Bernhard Nebel für die Zweitbegutachtung der Dissertation. Ich hoffe, mit der Arbeit einen weiteren Beleg dafür zu geben, daß Erkenntnisse der theoretischen Informatik und der künstlichen Intelligenz fruchtbar zusammenfinden können.

Dr. Jürgen Eckerle für die inhaltliche Zusammenarbeit im Bereich der *Heuristischen Suche*, insbesondere bei den vielen intensiven Diskussionen am Arbeitsplatz. Das Gespür, wo sich neue Resultate verbergen könnten, hat mich häufig darin geleitet, diese auch zu finden und auszuarbeiten.

Dr. Stefan Schrödl für unzählige Abende der wissenschaftlichen Diskussion und Implementation. Diese *externe* Forschung ist hoffentlich noch lange ein Quell neuer Erkenntnisse.

Frank Reffel für die guten Programmideen und Analyseansätze in einem vielversprechenden Schnitt der zwei Disziplinen *Heuristische Suche* und *Modellprüfung*.

Monika und Hildegard Kleyboldt für die vielen Grammatik-, Rechtschreibungs- und Kommasetzungskorrekturen.

Widmung

Die Dissertation widme ich Prof. Dr. Ingo Wegener, dem Betreuer meiner Diplomarbeit. Er ist ein genialer Wissenschaftler und besitzt eine beeindruckende Persönlichkeit. Leider ist er während meiner Promotion schwer erkrankt. Auf diesem Weg wünsche ich ihm gute Besserung und noch viele Funken der Freude in seinem Leben.

Freiburg, den 10. Juli 1998 Stefan Edelkamp

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel führt anhand einer anschaulichen Alltagssituation in die wichtigsten Begriffe der Zustandsraumsuche ein. Die Schwierigkeit der Problemstellung wird in einer komplexitätstheoretischen Betrachtung erörtert. Es wird argumentiert, daß die geeignete Interpretation des Problems durch eine Suche nach kürzesten Wegen innerhalb eines endlichen Automaten gegeben ist. Weiterhin wird die Notation von Rechenverfahren im Pseudocode erläutert und der Grundstock aller analytischen Laufzeit- und Speicherplatzuntersuchungen gelegt.

1.1 Suchprobleme

Ob materiell oder ideell, wir alle suchen. Der eine sucht sein Kleidungsstück im Schrank, die andere ein geeignetes Fernsehprogramm. Vergeßliche Menschen suchen noch ein wenig mehr. Ein Fußballspieler sucht den Weg in das Tor und viele Menschen suchen Geborgenheit und Ruhe. Forschung ist die Suche nach unbewältigten Problemenstellungen bzw. der Lösung selbiger, und die große Unrast menschlichen Seins ist die Suche nach dem Sinn des Lebens. Kurzum, dieses Grundverständnis des Wortes *Suche* läßt sich zur Begriffsbildung nutzen, denn so allgemein, wie der Begriff Suche im menschlichen Umgang erscheint, ist er auch in der Informatik: Jeder Handlungsablauf oder *Algorithmus* sucht nach Lösungen für ein gestelltes Problem.

Suchprobleme sind eingebettet in einer Umgebung, in der die Suche stattfindet. Um diese *Domäne* für den Suchprozeß greifbar zu gestalten, muß eine eindeutige Beschreibung aller *legalen* Problemzustände erfolgen. Innerhalb dieser Menge werden ein *Startzustand* und mitunter mehrere *Zielzustände* ausgezeichnet.

1.1.1 Das Rendezvous

Wir illustrieren die Begrifflichkeit der Zustandsraumsuche an einem Beispiel aus der Alltagswelt. Ein Mann sucht den Ausgang aus einem großen Amtsgebäude, sagen wir, um ein geplantes Essen mit seiner Geliebten nicht zu verpassen. Der Zustandsraum ist durch die Menge der Räume und Flure, der Startzustand durch den gegenwärtigen Standort und der Zielzustand durch

den Ausgang gegeben. Kann aus verschiedenen Ausgängen alternativ gewählt werden, so gibt es gleich mehrere Zielzustände. Ist die Position der Ausgänge unbekannt, so liegen die Zielzustände nicht *explizit* vor, sondern werden durch die Anwendung einer Regel (eines *Prädikates*) beschrieben.

Desweiteren werden *Übergänge* spezifiziert, die es erlauben, einen Zustands- bzw. Raumwechsel durchzuführen, z.B. *hoch*, *runter*, *vor*, *zurück*, *rechts* oder *links*. Je nach Anschauung und Umgebung wird ein Übergang auch als *Operator*, als *Transition* oder einfach als *Zug* bezeichnet. Die einzelnen Zustandswechsel sind häufig *generisch*, d.h. für mehrere Zustände anwendbar. Mehr noch, man geht in der Zustandsraumsuche zumeist von einer endlichen Menge von Zustandsübergängen aus.

1.1.2 Expansionen und Heuristiken

Die Erzeugung aller Nachfolger zu einem gegebenen Zustand wird als *Expansion* bezeichnet. Dieses entspricht einer Neuorientierung über die zu Verfügung stehenden Möglichkeiten des *ausweglosen* Mannes im Beispiel. Die Menge der *erreichten* (*generierten*), aber noch nicht *betretenen* (*expandierten*) Zustände in einem Problemlöseprozeß heißt *Horizont*.

Von dem Startzustand ausgehend wird durch wiederholte Expansion eines Zustandes am Horizont die Menge der tatsächlich *erreichbaren* Zustände beschrieben, die eine Teilmenge aller möglichen Problemzustände darstellt. In dem Beispiel könnten manche Türen verschlossen sein und den Liebenden an den Rand der Verzweiflung bringen. Er mag sich wünschen, den gesamten Horizont zu überblicken, ein Traum, der sich im Rechner durch die mengentheoretische Beschreibung des Horizonts tatsächlich realisieren läßt (vergl. Kapitel 4).

Der Mann hat dadurch, daß er in das Gebäude hineingegangen ist, eine gewisse Schätzung der Weglänge und damit der einzukalkulierenden Zeit. Diese Einschätzung wird als *Heuristik* bezeichnet. Für eine effiziente Problemlösung ist es günstig, wenn die Heuristik die tatsächliche minimale Weglänge zum Ziel immer unterschätzt. Demnach sprechen wir von einer *unteren Schranke* oder auch von einer *optimistischen Heuristik*.

In einer graphischen Repräsentation der Suche nutzt man Knoten für die einzelnen Zustände und Kanten für die jeweiligen Übergänge. Eine Knotenfolge innerhalb des sich aus diesen Elementen zusammensetzenden *Problemgraphen* (vergl. Abbildung 1.1) bildet einen *Pfad*, während die damit assoziierte Kantensequenz als *Makrooperator* oder kurz als *Makro* bezeichnet wird. Der Problemgraph entspricht in unserem Beispiel einer Übersichtskarte des Gebäudes.

Die Übergänge können *gewichtet* sein, um die unterschiedlichen Schwierigkeiten verschiedener Operatoren zu betonen. So ist die Bewältigung einer Treppe für unseren Hauptdarsteller ungleich schwerer als der Gang über eine Türschwelle.

1.1.3 Duplikate und Abkürzungen

Nun kann es sein, daß ein und derselbe Zustand über zwei (oder mehrere) unterschiedliche Wege erreicht werden kann. Der Mann ist in einen *Zyklus* geraten. Er mag sich hilflos fragen: *War ich hier schon mal oder nicht?* Anders gesprochen, der Problemgraph kann so groß werden, daß sich gleiche Zustände

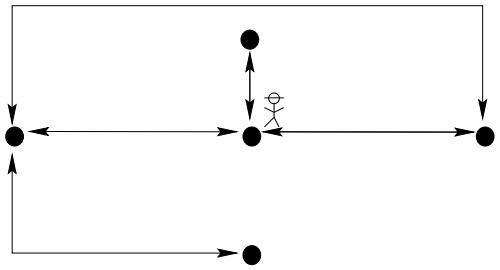


Abbildung 1.1: Der Problemgraph.

im Löseprozeß nicht immer aufspüren lassen. Die möglichst speicherplatzsparende Aufdeckung dieser *Duplikate* ist somit zum Kernproblem der Zustandsraumsuche geworden. Die Dissertation setzt in den Kapiteln 5 und 6 an diesem Punkt an und versucht, im Problemlöseprozeß Regelmäßigkeiten innerhalb der entdeckten Zyklen aufzudecken und im weiteren Verlauf der Suche zu nutzen.

1.1.4 Suchbaumgröße

Durch die wiederholte Expansion von Knoten wird eine Vorgängerbeziehung der Zustände untereinander beschrieben und ein *Explorations-* bzw. *Suchbaum* (vergl. Abbildung 1.2) aufgebaut, der die Menge aller Wege (*Generierungspfade*) beschreibt.

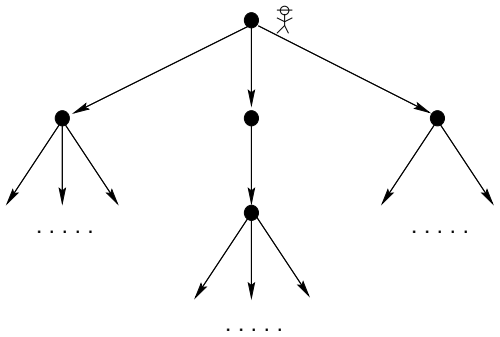


Abbildung 1.2: Der aufgespannte Suchbaum.

Die Größe des Suchbaumes hängt von der *Suchtiefe* und dem *Verzweigungsgrad* der einzelnen Knoten ab, wobei letzterer durch die Anzahl von Nachfolgern festgelegt ist. Die Berechnung des durchschnittlichen Verzweigungsgrads innerhalb des Suchbaumes von Problemräumen, deren Zustände eine unterschiedliche Anzahl von Nachfolgerpositionen (*Kinder*) besitzen, wird im Kapitel 10 aufgegriffen. Dabei werden wir insbesondere auf die Reduktion des Verzweigungsgrads durch die verschiedenen Abschneidestrategien eingehen.

1.1.5 Lernverfahren

Wie bei der Entdeckung von Zyklen soll unser Akteur weitere Information auffangen und zur Verbesserung seines Weges hinzuziehen. So bekommt der

Suchende mehr und mehr einen Eindruck der ihn umgebenden Umwelt und entwickelt Strategien, damit er sich auf seinem Weg immer sicherer wird. Damit wird der umherirrende Mann nicht nur beim nächsten Mal den Ausgang schneller finden, sondern schon beim Erreichen der nächsten Etage die gewonnenen Erkenntnisse über die Gebäudestruktur nutzen. Getragen von dieser Analogie sprechen wir in diesem Fall von einem *Lernverfahren*.

Häufig kann man einen Zustandsübergang in die Gegenrichtung durchführen. Manche Zwischentüren schnappen jedoch nach dem Durchschreiten ins Schloß. Damit ergeben sich in gerichteten Graphen *Sackgassen*, also Positionen, in denen das Ziel unerreichbar geworden ist. Das Kapitel 7 widmet sich dem Aufspüren von Indikatoren für Sackgassen. Auch dieses Verfahren soll dem Mann wenigstens auf Dauer das Leben versüßen.

Die Suchaufgabe in dem so erarbeiteten Formalismus besteht für den Liebenden nun darin, die kostengünstigste Übergangsfolge zu finden, die den Startzustand in einen der Zielzustände überführt. Gelingt dieses, so wird das Suchverfahren, nach dem der Mann handelt, *zulässig* genannt.

1.1.6 Realzeitanforderungen

Eines kann der Mann jedoch nicht. Es ist ihm unmöglich, innerhalb der Domäne in Nullzeit von einem Raum in einen anderen weit entfernten zu gelangen. Diese zusätzliche sogenannte *Realzeitanforderung* im Problemlöseprozeß wird in Kapitel 9 untersucht.

Wir hoffen nicht, daß die sicherlich mißmutig gelaunte Geliebte die Verabredung platzen läßt und weggeht. In diesem Fall bleibt dem Liebenden nur noch die Verfolgung eines sich somit bewegenden Zieles, ein Thema, das in Kapitel 8 behandelt wird.

Wir gehen vielmehr davon aus, daß die beiden Liebenden einmütig beieinander sitzen und sich tief in die Augen schauen. Ziehen wir uns deshalb dezent zurück und widmen uns einer generellen Betrachtung der Suchaufgabe.

1.2 Komplexität des Problems

Es gibt eine Vielzahl von Anwendungen der Zustandsraumsuche, so daß wir uns die Frage stellen, ob jede Berechnungsaufgabe als Zustandsraumproblem aufgefaßt werden kann. Dazu möchten wir die Mächtigkeit des Zustandsraummodells untersuchen und unternehmen einen Exkurs in die Komplexitätstheorie (Wegener 1993c). Wir werden fragen, inwieweit sich gefundene Komplexitätsresultate eingliedern lassen und welche Sichtweise für Zustandsraumprobleme die geeignete ist.

1.2.1 Turingmaschinen

Das in der theoretischen Informatik anerkannte Berechenbarkeitsmodell sequentieller Rechenmaschinen ist das der *Turingmaschine*. Die Turingmaschine (vergl. Abbildung 1.3) hat (zumindest zu einer Seite hin) ein unendlich langes Band, auf der die Eingabe steht und wird traditionellerweise als 7-Tupel formalisiert: Endliche Kontrolle Q, Eingabealphabet Σ, Bandalphabet Γ, Anfangszustand q_0, Blankbuchstabe B, Zielzustandsmenge F und Übergangsfunktion

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, N\}$ (L steht für eine Links-, R für eine Rechts- und N für eine Nichtbewegung des Kopfes).

In jedem Schritt liest die Maschine ein Zeichen an der Kopfposition auf dem Band ein. Je nach Fund und Verfassung (derzeitiger Zustand) wird der Bandbuchstabe manipuliert und ein Zustands- wie auch ein Positionswechsel durchgeführt.

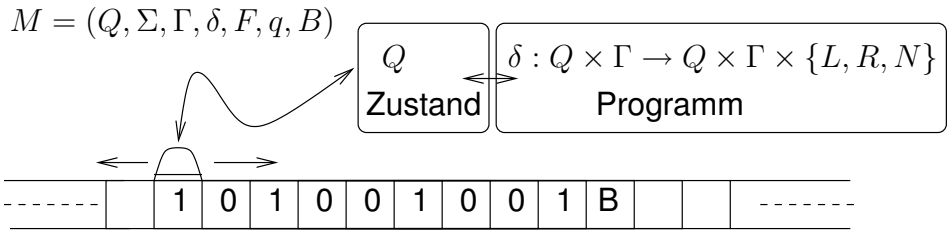


Abbildung 1.3: Eine Turingmaschine.

Über die *Churche These* stimmt die Klasse der intuitiv berechenbaren Funktionen mit der Klasse der mit Hilfe einer Turingmaschine berechenbaren Funktionen überein. Die ebensowenig beweisbare *erweiterte Churche These* geht davon aus, daß die Rechenzeiten von allen sequentiellen Rechenmaschinen polynomiell mit der Laufzeit einer Turingmaschine verknüpft sind, d.h. sich gegenseitig höchstens durch die Anwendung eines Polynoms (im Gegensatz zu einer exponentiellen Funktion) voneinander unterscheiden.

Das Halteproblem der Turingmaschine ist im Grunde schon als Zustandsraumproblem formalisiert: Den Zuständen entsprechen Konfigurationen der Turingmaschine, die Startkonfiguration und die Zielkonfigurationen sind gegeben und die Zustandsübergänge werden durch die Übergangsfunktion δ beschrieben.

Damit ist das Modell des Zustandsraumproblems *universell*, d.h. es kann jedes berechenbare Problem beschreiben. Durch die somit gefundene Äquivalenz zum unentscheidbaren *Halteproblem* einer Turingmaschine ergeben sich allerdings auch Zustandsraumprobleme, für die kein allgemeiner Algorithmus existiert, der sagen kann, ob die Suche nach einem Ziel in einem Zustandsraum ein Ziel finden wird oder nicht.

1.2.2 Beschränkte Zustandsbeschreibung

Das Problem liegt in der Größe der Zustandsbeschreibung. Ist sie prinzipiell unendlich, so kann sie beliebige Konfigurationen der Turingmaschine beinhalten. Fordern wir hingegen ihre Beschränktheit, so bewegen wir uns in Richtung platzbeschränkter Turingmaschinen und entscheidbarer Sprachen. (Platzbeschränkte Turingmaschinen besitzen nur eine endliche Anzahl von Konfigurationen.) Je nachdem, wieviel Platz wir der Zustandsbeschreibung zur Eingabegröße n zugestehen, erreichen wir unterschiedliche Komplexitätsklassen, z.B. *PSPACE* bei polynomieller Beschränkung des Platzes und $DTAPE(n)$ bei linearer Beschränkung.

Bei einer deterministischen Turingmaschine steht der Nachfolger für eine gegebene Konfiguration schon fest. Dies entspricht nicht gerade unserem Bild von einem Suchvorgang.

Bei einer nichtdeterministischen Turingmaschine hingegen wird eine Menge von Nachfolgezuständen alternativ untersucht. Die Laufzeit der nichtdeterministischen Maschine ist definiert als der kürzeste Weg zu einem akzeptierenden Zustand. Ein einzelner Suchprozeß entspricht demnach der deterministischen Simulation einer nichtdeterministischen ausgelegten Spezifikation, die bekanntlicherweise mit exponentiellem Aufwand möglich ist. Bei einem polynomiellen Aufwand wäre das unwahrscheinliche Resultat $P{=}NP$ bewiesen. Dabei ist P bzw. NP die Klasse der mit einer deterministischen bzw. nichtdeterministischen Turingmaschine in polynomieller Zeit berechenbaren Probleme. Ein Problem p heißt NP-vollständig, wenn es zu den schwersten Problemen in NP gehört, d.h. falls *alle* Probleme in NP mit Hilfe eines (in der Laufzeit nicht bewerteten) Unterprogramms für p in polynomieller Zeit zu berechnen sind. Ist p nicht unbedingt in NP enthalten, sagen wir p ist *NP-hart*.

Es ist bekannt, daß die Klassen der mit nichtdeterministischen bzw. deterministischen Turingmaschinen effizient berechenbaren Probleme in $PSPACE$ enthalten sind. Doch haben wir mit allgemeinen Turingmaschinen wirklich das adäquate Modell für die Zustandsraumsuche gefunden?

1.2.3 Einzelne Probleminstanzen

Wenn wir sagen, ein Zustandsproblem gehört einer speziellen Komplexitätsklasse an, so meinen wir im Grunde eine Folge von immer größer skalierten Zustandsraumproblemen. So ist die NP-Vollständigkeit des (n^2-1)–Puzzles (vergl. Kapitel 2) für wachsendes n formuliert. Uns interessiert zumeist nur die Lösung einer einzelnen Probleminstanz. Damit gestehen wir der Beschreibung nur eine konstante Länge zu und gelangen nahezu unweigerlich in das Gebiet der endlichen Automaten.

Hier ist alles endlich, die Anzahl der Zustände Q, das Alphabet der Zustandsübergänge Σ, die Anzahl der Finalzustände F und die Anzahl der verschiedenen Transitionen $\delta : Q \times \Sigma \rightarrow Q$. Dies scheint eine geeignete Beschreibung eines einzelnen Zustandsraumproblem zu sein. Die Menge der von dem Zustandsraumautomat erkannten Übergangsketten heißt gemeinhin *reguläre Sprache*. Doch wollen wir diese Sprache nicht aufzählen, sondern das kürzeste Wort ausfindig machen. Auf der Suche nach einem akzeptierenden Zustand in einem endlichen Automaten übersteigen wir allerdings leicht die zur Verfügung stehenden Platzressourcen, da die Zustandsmenge, wenn auch endlich, so doch riesig groß ist, d.h. *Endlichkeit* bedeutet in unserem Fall gerade nicht *durch den Speicherplatz begrenzt*.

1.3 Anmerkung zur Notation

Rechenverfahren beschreiben Handlungsabläufe, wie wir sie zum Beispiel von Kochrezepten aus dem Alltagsleben her kennen. Die Beschreibung eines Rezeptes sollte so detailliert sein, daß bei genauer Befolgung ein genauso schmackhaftes Gericht entsteht, wie es dasjenige Essen war, das zu dem Rezept Anlaß gab. Auf der anderen Seite ist, je nach Kochkunst des Lesenden, eine Abstraktion bekannter Handlungsschemata sehr nützlich, um die Beschreibung knapp und lesbar zu gestalten. Letztendlich zeichnet sich ein gutes Rezept durch möglichst große Unabhängigkeit von den gegebenen Kochutensilien aus, z.B.

sollte sich die Zubereitung eines Gerichtes auf einem Gasherd nicht wesentlich von der Zubereitung selbigen Mahles auf einem Elektroherd unterscheiden.

1.3.1 Pseudocode

In der Informatik verhält es sich ähnlich - Verfahren sollen so allgemein wie möglich und so detailliert wie nötig dargestellt werden. Die Ungenauigkeit einer Algorithmenbeschreibung führt jedoch schnell zu dramatischen Konsequenzen durch fehlerhafte Programme. Deshalb hat sich eine von dem Rechner unabhängige Sprachregelung eingebürgert, der *Pseudocode*. Die Programmiersprache gibt es nicht, sondern sie steht für einen gemeinsamen Konsens der Programmpräsentation. Aufgrund der Funktionsweise einer Turingmaschine setzen sich Programme grundlegend aus Zuweisungen und bedingten Sprüngen zusammen.

Diese Grundelemente werden in dem Architekturprinzip der *Registermaschine* (engl. random access machine) weiter hervorgehoben. Registermaschinen bestehen aus einem Programmspeicher und aus den den Hauptspeicher repräsentierenden Registern und entsprechen schon eher den derzeitigen Realisierungen sequentieller Rechner. Höhere Konzepte sind Fallunterscheidungen, Unterprogrammaufrufe und Schleifenkonstrukte. Anbei eine kleine Sammlung der in dieser Arbeit genutzten Schlüsselworte:

if (⟨*Bedingung*⟩) ⟨*Anweisungen*⟩ **else** ⟨*Alternative*⟩
Verzweigung des Programmablaufes gemäß des in der *Bedingung* vorliegenden Falles.

and, or, not
Logische Verknüpfung von Bedingungen.

while (⟨*Bedingung*⟩) ⟨*Anweisungen*⟩
Vor dem Durchlauf geprüfte Schleife.

repeat ⟨*Anweisungen*⟩ **until** (⟨*Bedingung*⟩)
Nach dem Durchlauf geprüfte Schleife.

for all ⟨*Element*⟩ **in** ⟨*Menge*⟩
Die Variable *Element* durchläuft nach und nach eine geordnete *Menge*.

return Rücksprung aus einem Unterprogramm.

In jeder Programmzeile erleichtern natürlichsprachliche, rechtsbündig gesetzte Kommentare das Verständnis des Konzentrats. Zuweisungen werden durch einen nach links gerichteten Pfeil beschrieben, Unterprogrammaufrufe werden kursiv gedruckt und Operationen auf den zugrunde liegenden Datenstrukturen entweder durch einen Punkt hervorgehoben (objektorientierte Schreibweise) oder mit der Struktur als Parameter aufgerufen (funktionale Schreibweise). Der Zugriff auf indizierte Sequenzen (sogenannte Arrays) wird durch eckige Klammerung oder durch indizierte Variablen aufgezeigt. Letztendlich werden Programmabläufe durch den begleitenden Text motiviert und gestützt.

1.3.2 O-Notation

Ein gutes Rechen- bzw. Suchverfahren zeichnet sich neben der Einfachheit und Korrektheit insbesondere durch Ressourcenfreundlichkeit (Zeit und Platz) aus.

Leider sind die exakten Zeit- und Platzbedürfnisse schlecht zu ermitteln bzw. sehr vom verwendeten Rechnermodell abhängig. Deshalb schreibt man üblicherweise den Grundoperationen, wie der einer Zuweisung von Variableninhalten oder der Addition zweier Zahlen, konstante Kosten zu. Im allgemeinen definiert man ein Kostenmaß, das die Laufzeit bzw. den Speicherplatz durch eine Funktion innerhalb der natürlichen Zahlen beschreibt.

Die genaue Anzahl der durch das Kostenmaß definierten Elementaroperationen variiert bei geringen Änderungen des Rechenablaufes so enorm, daß ein Vergleich der verschiedenen Ansätze untereinander schlecht zu gewährleisten ist. Deshalb hat sich eine Gruppierung in Klassen von Funktionen nach der Landau'schen O-Notation bewährt.

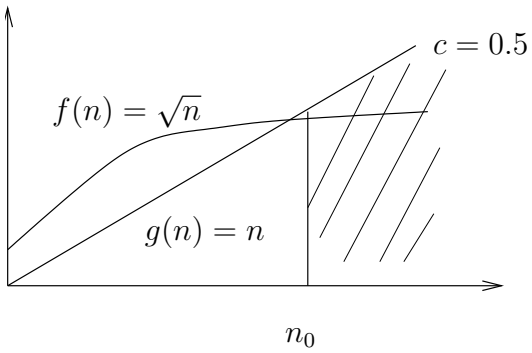


Abbildung 1.4: Funktionenklassen definiert durch die O-Notation.

Eine Funktion f (sagen wir $\sqrt{n}$), die die Anzahl der Elementaroperationen des Programms mißt, soll nach oben durch eine andere Funktion g (sagen wir durch n) beschränkt werden. Der erste Ansatz ist, den Quotienten aus der Funktion f und g für steigendes n darauf zu testen, ob er eine Nullfolge bildet. Ist dies der Fall, dann sagen wir $f(n) \in o(g(n))$. Ist diese Folge nur durch eine Konstante beschränkt, so schreibt man $f(n) \in O(g(n))$. Formal bedeutet der letzte Ausdruck, daß es eine Konstante n_0 und eine reelle Zahl c gibt, so daß für alle n oberhalb dieser Konstante $f(n)$ wirklich kleiner als $c \cdot g(n)$ ist (vergl. Abbildung 1.4). Analog schreibt man $f(n) \in \omega(g(n))$ bzw. $f(n) \in \Omega(g(n))$ für eine untere asymptotische Beschränkung der Funktion f. Letztendlich ist $f(n) \in \Theta(g(n))$ genau dann, wenn f sowohl in $O(g(n))$ als auch in $\Omega(g(n))$ liegt. Somit kann von Vorfaktoren und größenordnungsmäßig kleineren Termen abstrahiert werden (z.B. ist $4n^3 + 2n^2 - 10 \in \Theta(n^3)$). Einige grundlegende so gewonnene Funktionenklassen sind die Klassen der konstanten ($O(1)$), der logarithmischen ($O(\log n)$), der linearen ($O(n)$), der polynomiellen ($O(n^k)$) und der exponentiellen ($O(2^n)$) Funktionen.

Kapitel 2

Herausforderungen im Einpersonenspiel

In diesem Kapitel stellen wir einige populäre Einpersonenspiele vor. Trotz der Einfachheit der Aufgabenstellungen erweisen sich die Lösungen angesichts der enormen Größe der Zustandsräume dieser Puzzles als beweisbar schwer. Die Ausführungen zum allgemeinen Schiebepuzzle sind ausführlicher, da es hier erstmalig analysiert wird. Die Beispiele aktueller Herausforderungen im Einpersonenspiel führen uns an typische Fragestellungen und Lösungsstrategien der Zustandsraumsuche heran. Dabei werden wir insbesondere auf die Entwicklung von unteren Schranken für diese Spiele eingehen.

2.1 Das $(n^2 - 1)$–Puzzle

Das $(n^2 - 1)$–Puzzle (siehe Abbildung 2.1) wurde durch Amerikas berühmten Spieleerfinder und Rätselspezialisten Samuel Loyd (1841-1911) bekannt und ist so populär, daß nahezu jedes Kind es in irgendeiner Form in den Händen gehalten hat. Ein Zug besteht darin, die Quadrate in das einzig vorhandene Leerfeld zu schieben, mit der Aufgabe, eine ausgezeichnete Zielstellung zu erreichen. Eine andere, von uns bevorzugte Möglichkeit, ist es, einen Zug eindeutig durch die Bewegung des Leerfeldes nach *links*, *rechts*, *oben* oder *unten* zu beschreiben.

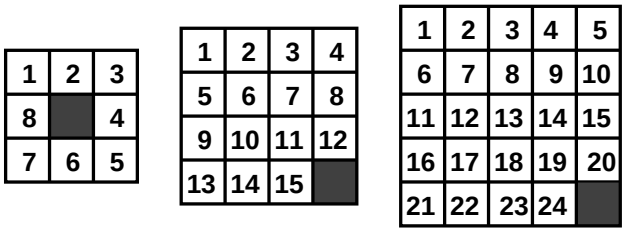


Abbildung 2.1: Das *Acht*–, das *Fünfzehn*– und das *Vierundzwanzig*–Puzzle.

2.1.1 Konfigurationenanzahl und Komplexität

Erste wissenschaftliche Arbeiten zum (n^2-1)-Puzzle reichen zurück bis ins letzte Jahrhundert (Storey 1879; Johnson 1879). Genau die Hälfte der $(n^2)!$ bzw. $1{\cdot}2{\cdot}\ldots{\cdot}n^2$ verschiedenen Permutationen der Spielsteine und des Leerfeldes sind von einer gegebenen Stellung aus erreichbar (dies sind ca. 10^5 Zustände für das Acht–, ca. 10^{13} Zustände für das Fünfzehn– und ca. 10^{25} Zustände für das Vierundzwanzig–Puzzle). Vertauscht man zwei Spielsteine, so wird das Puzzle unlösbar (Gardner 1959). Wir begeben uns auf die andere Seite eines Spiegels. Die volle Permutationsgruppe wird genau dann erreicht, wenn ein Kreis ungerader Länge in dem durch das Schiebepuzzle beschriebenen Graphen existiert (Wilson 1974). Andernfalls erreicht man durch den Tausch zweier Spielsteine die sogenannte *alternierende Gruppe*. Die einzige Ausnahme ist der sogenannte *trickreiche Sechser* (Berlekamp, Conway & Guy 1982).

Das (n^2-1)–Puzzle ist *NP*-hart und somit ist mit keinem effizienten Algorithmus zur Lösung des Problems zu rechnen (Ratner & Warmuth 1990).

2.1.2 Heuristik

Für das (n^2-1)–Puzzle lassen sich schnell einfache untere Schranken bzw. optimistische Heuristiken finden, z.B. die Anzahl der falschplazierten Spielsteine oder die sogenannte *Manhattendistanz*, die für zwei Spielsituationen $S = ((x_1, y_1), (x_2, y_2), \ldots, (x_{n^2-1}, y_{n^2-1}))$ und $S' = ((x'_1, y'_1), (x'_2, y'_2), \ldots, (x'_{n^2-1}, y'_{n^2-1}))$ durch

$$MD(S, S') = \sum_{i=1}^{n^2-1} (|x_i - x'_i| + |y_i - y'_i|)$$

gegeben ist. In Abbildung 2.2 ist die Anzahl der falschplazierten Spielsteine gleich vier (die Steine 5,6,7 und 8 liegen nicht an ihrem Platz) und die Manhattandistanz gleich sieben.

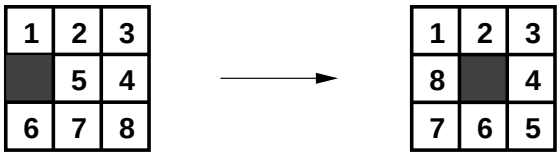


Abbildung 2.2: Zwei untere Schranken: Die Anzahl der falschplazierten Spielsteine und die Manhattendistanz.

Die Manhattandistanz entspricht der getrennten Lösung des auf einen Stein reduzierten Problems. Eine naheliegende Verbesserung ist es demnach, die Lösungen für alle *Zweisteinprobleme* zu berechnen und diese dann zu einer Gesamtlösung zusammenzufügen. Hierbei darf jeder Spielstein nur einmal innerhalb eines Paares auftauchen. Repräsentieren wir jeden Spielstein als Knoten innerhalb eines Graphens, und bewerten wir die Kanten zwischen den Knoten durch die Lösungen der Zweisteinprobleme, so suchen wir eine Auswahl sich nicht berührender Kanten, deren Gesamtbewertung maximal ist. Dies ist ein gewichtetes *Zuordnungs-* oder *Matching Problem* innerhalb eines gewichteten

und ungerichteten Graphen, das effizient in Zeit $O(n^3)$ berechenbar ist (Simon 1992).

Abbildung 2.3 veranschaulicht ein maximales Matching für unser Beispiel aus Abbildung 2.2. Dabei wurde der Graph zur vereinfachten Darstellung auf Zeilen- bzw. Spaltenkonfliktpaare beschränkt. Der *lineare Konflikt*, in dem die Spielsteine 6 und 7 in Abbildung 2.2 stehen, wird durch diesen Ansatz erkannt und führt wiederum zu einer Verbesserung der unteren Schranke um zwei Züge. Aufbauend auf diesem Zuordnungsansatz beschreiben Korf und Taylor ein Verfahren, das zur halbautomatischen Generierung unterer Schranken für das (n^2-1) Puzzle führt (Korf & Taylor 1996).

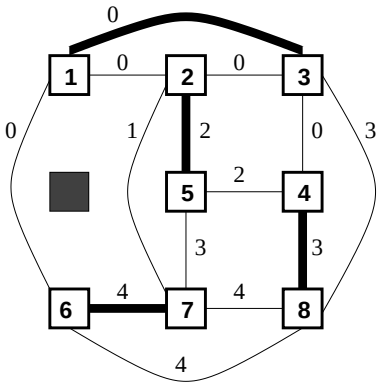


Abbildung 2.3: Ein gewichtetes Matching zur Berechnung der verbesserten Heuristik im (n^2-1)-Puzzle.

Eine Verallgemeinerung dieser Idee auf drei oder mehrere zusammen betrachtete Spielsteine erweist sich als problematisch, da die Berechnung eines drei- oder mehrdimensionalen Matchings *NP*-hart ist (Garey & Johnson 1979).

Weitere obere und untere Schranken, insbesondere in der parallelen Suche nach Lösungen des (n^2-1)–Puzzles, werden von Gasser und Brüngger beschrieben (Gasser 1993; Brüngger 1996).

2.1.3 Lösungen

Bei der Bewertung von heuristischen Suchverfahren diente in den Veröffentlichungen der letzten Jahrzehnte, das (n^2-1)–Puzzle nahezu als alleiniges Standartbeispiel. Ist für ein n die Berechnung von optimalen Lösungswegen gelungen, so versucht man sich an der Spielversion für $n+1$. Meilensteine sind Schofield für $n=3$, Korf für $n=4$ und Korf und Taylor für $n=5$ (Schofield 1967; Korf 1985a; Korf & Taylor 1996).

Korf nutzt in der Lösung des Fünfzehn-Puzzles die Ergebnisse, um den *IDA** Algorithmus vorzustellen, der in Kapitel 3 analysiert wird. Das Verfahren nutzt in einer mit wachsender Tiefenschranke initiierten Tiefensuche die Manhattendistanz als Schätzer für die zu erwartende Lösungslänge. Da eine Elimination erneut erreichter Spielzustände (mit Ausnahme der Vorgängersituation) nicht erfolgt, benötigt das *IDA** Verfahren nur Speicherplatz, der proportional zur Tiefe des Suchbaumes wächst.

Wir stellen beispielhaft eine mit *IDA** und der Manhattandistanz erzielten Lösung eines ausgewürfelten Fünfzehn-Puzzles (vergl. Anhang A) vor. Die im

HSF-Light Projekt (vergl. Anhang B) gemessenen Anzahlen von Knotenexpansionen und -generierungen sind in Tabelle 2.1 angegeben. Die Laufzeit auf einer Sun Ultra Wokstation betrug ca. 20 Sekunden.

Tiefenschranke	Anzahl expandierter Knoten	Anzahl generierter Knoten
43	68	197
45	1002	2970
47	13148	39351
49	135600	410020
51	1188663	3611260
53	9212825	28057343
55	17113528	52064803

Tabelle 2.1: Knotenexpansionszahlen bei der Lösung des *Fünfzehn*-Puzzles mit *IDA**.

Auch haben wir die Instanz (17, 1, 20, 9, 16, 2, 22, 19, 14, 5, 15, 21, 0, 3, 24, 23, 18, 13, 12, 7, 10, 8, 6, 4, 11) des *Vierundzwanzig* Puzzles mit der durch das Matching verbesserten Heuristik gelöst. Sie besitzt eine optimale Lösungslänge von einhundert Zügen. Dabei wurden in ca. vier Stunden insgesamt 19865967282 Knoten auf einer Sun Ultra Workstation generiert. Die Lösung ist gegeben durch die Bewegung der Spielsteine 19, 20, 9, 14, 3, 12, 13, 19, 20, 3, 5, 16, 14, 9, 3, 5, 12, 13, 7, 24, 13, 7, 4, 11, 24, 13, 7, 4, 19, 6, 8, 18, 21, 20, 6, 21, 20, 22, 2, 17, 1, 2, 5, 6, 22, 15, 17, 5, 6, 12, 4, 7, 16, 14, 9, 4, 7, 16, 13, 19, 11, 8, 18, 20, 23, 10, 20, 23, 15, 17, 10, 15, 21, 22, 16, 11, 8, 18, 23, 21, 17, 16, 11, 8, 18, 23, 22, 17, 16, 11, 12, 7, 8, 13, 14, 9, 4, 3, 2 und 1.

Korf und Taylor nutzen zur Lösung des *Vierundzwanzig* Puzzles zusätzlich eine Technik zur *Duplikatselimination*, die wir im Kapitel 6 besprechen. Die von Korf und Taylor in ca. 2,5 Stunden auf einer Sun Ultra Workstation gefundene Lösung für die angegebene Probleminstanz generierte 8110532608 Knoten.

2.1.4 Kritik

Im folgenden wollen wir die erzielten Erfolge in der Lösung des (n^2-1)–Puzzles kritisch hinterfragen.

1. Die Suchraumstruktur des (n^2-1)–Puzzles kommt dem Verfahren der iterierten Tiefensuche sehr entgegen, da der nächst größere Zyklus als der, der sich durch einen Zug zum Vorgänger ergibt, die Länge zwölf hat. Es gibt demnach recht wenig Duplikate innerhalb der ersten Schichten des Suchbaumes.

2. Die Zeitkomplexität von Suchalgorithmen wird (analog zur Schlüsselvergleichsanzahl allgemeiner Sortierverfahren) in der Anzahl der Knotenexpansionen gemessen. Das (n^2-1)–Puzzle bietet jedoch eine sehr schnelle Expansion von Knoten, die, gemessen in real verbrauchter Rechenzeit, wesentlich kostengünstiger durchgeführt werden kann als zum Beispiel die Datenstrukturoperationen einer Duplikatssuche.

3. In einer Tiefensuche kann eine Expansion in konstanter Zeit durchgeführt werden, da nur die neue Position eines Spielsteines und des Leerfeldes bestimmt werden müssen. Auch der optimistische Schätzwert der Manhattandistanz ist inkrementell in $O(1)$ aus dem Vorgängerwert zu ermitteln.

4. Der Verzweigungsgrad ist mit zwei bis vier sehr klein. Der Suchbaum ist demnach recht schmal. Sackgassen, d.h. Spielstellungen mit nur einem Nachfolger, gibt es nicht.

2.2 Das allgemeine Schiebepuzzle

Es scheint somit folgerichtig, die Güte der heuristischen Suchalgorithmen auf eine breitere Datenbasis zu stellen. Hierzu stellen wir das *allgemeine Schiebepuzzle* vor, das aus verschieden geformten Spielsteinen, sogenannten *Polyminos*, besteht.

Da wir eine Beschriftung der Spielsteine zulassen, ist das *allgemeine Schiebepuzzle* eine echte Generalisierung des (n^2-1)–Puzzles. In Abbildung 2.4 sind das *Esel*-, das *Jahrhundert*- und das *Dad*-Puzzle aus dem Buch *Gewinnen, Band 4, Solitärspiele* als typische Vertreter des *allgemeinen Schiebepuzzles* dargestellt (Berlekamp, Conway & Guy 1982). Ziel dieser Puzzles ist, das 2×2 große Quadrat zur Pfeilspitze hin zu bewegen.

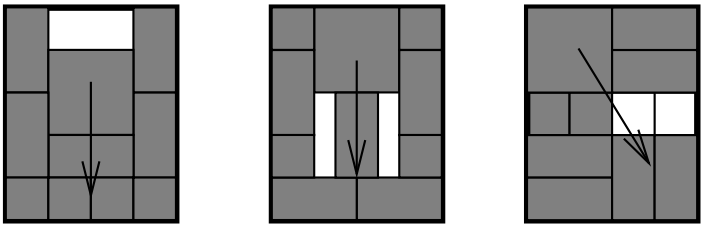


Abbildung 2.4: Das *Esel*, das *Jahrhundert*- und das *Dad*-Puzzle.

Abbildung 2.5 illustriert zwei weitere in Spielzeugläden käufliche Schiebepuzzles. Im *Mann-und-Flasche*–Puzzle soll ein Mann, der durch die drei Spielsteine am linken Spielfeldrand ausgemacht werden kann, zu einer Flasche am rechten Spielfeldrand gebracht werden. Da die wesentliche Schwierigkeit die Zusammenführung der vertikalen 2 mal 1 großen Spielsteine ist, kann auch diese vereinfachte Version betrachtet werden. Im *Harlekin*-Puzzle müssen die schwarz eingefärbten Winkelsteine letztendlich nebeneinander liegen.

Weitere allgemeine Schiebepuzzleprobleme finden sich in einem PC-Sharewareprogramm, namens *Klotzki*. Trotz intensiven Literaturstudiums sind wissenschaftliche Arbeiten zur Komplexität und zur effizienten automatischen Lösung des *allgemeinen Schiebepuzzle* nicht aufzuspüren.

Mit einer *Kachel* soll im folgenden ein einzelnes Spielsteinquadrat bezeichnet werden, aus denen ein Spielstein besteht. Ein Spielstein besteht aus einer Vielzahl von miteinander verbundenen Kacheln und läßt sich durch einen ausgezeichneten Bezugspunkt (obere linke Ecke), auf den hin die relativen Positionen aller in dem Spielstein vorkommenden Randkacheln festgelegt werden,

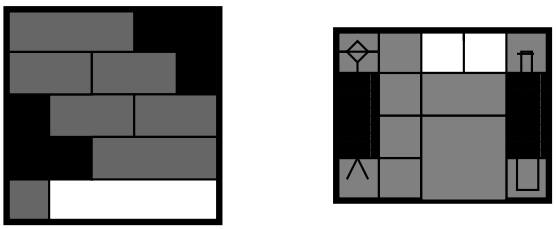


Abbildung 2.5: Das *Harlekin*- und das *Mann-und-Flasche*-Puzzle.

und die horizontalen bzw. vertikalen Ausdehnungen der Spielsteinschichten in beide Koordinatenrichtungen beschreiben. Der im Speicher beanspruchte Platz ist proportional zur Anzahl der Kacheln auf dem Rand.

2.2.1 Anzahl der Konfigurationen

Es gibt verschiedene Darstellungsarten eines allgemeinen Schiebepuzzles: Die Belegung des umgebenden Brettes mit Kacheln ist zwar eine adäquate Ausgabemöglichkeit, doch werden die Beziehungen der Kacheln bzw. der durch sie beschriebenen Spielsteine untereinander nicht einsichtig.

Eine kompakte Darstellung wird durch die Plazierungsreihenfolge der Spielsteine auf dem Spielbrett gegeben, z.B. *1x2*, *Leerfeld*, *Leerfeld*, *1x2*, *2x2*, *2x1*, *1x1*, *1x1*,*1x1*, *1x1* für das *Esel*-Puzzle (Bezugspunkte von links oben nach rechts unten). Damit läßt sich leicht eine obere Schranke für der Konfigurationen ermitteln: Ist s die Anzahl der sich insgesamt auf dem Brett befindenden Spielsteine und f_i für i aus $\{1,\ldots,k\}$ die Größe der Spielsteinfamilie F_i, so ist die Anzahl der Konfigurationen durch den Multinomialkoeffizienten

$$\binom{s}{f_1,\ldots,f_k} = \frac{s!}{f_1!\cdots f_k!}$$

beschränkt. In einem rekursiven, d.h. in einem sich selbst aufrufenden Programm *Count* werden diese Konfigurationen nun durch ein Urnenexperiment nacheinander generiert, auf Legalität getestet und in der Variablen *Configs* gezählt. Jeweils wird ein aus der Urne ausgewählter Spielstein der aktuellen Position hinzugefügt und nach Beendigung des rekursiven Aufrufs wieder entfernt. In der Informatik spricht man deshalb von einer *Versuch-und-Irrtum* Strategie bzw. von einem *Backtracking* Algorithmus.

Die Urne selbst wird durch ein Array *urn* dynamisch verwaltet. Dabei gibt urn_i für i aus $\{1,\ldots,k\}$ die Anzahl der in der Urne verbliebenen Spielsteine vom Typ i an. Initial ist die Urne mit s Kugeln gefüllt, von denen jeweils f_i den gleichen Typ haben. Ziel ist es, Plazierungssequenzen ausfindig zu machen, bei denen alle nacheinander auf das Spielbrett eingesetzten Steine auch passen.

Die Prozeduren *Fits* überprüft von der Stelle *pos* beginnend, ob alle Kacheln sich auf ein nicht belegtes Feld setzen lassen. Wird eine Kollision zwischen den einzufügenden und den schon auf dem Brett befindlichen Spielsteinen entdeckt, so gibt *Fits* den Wahrheitswert *false* zurück. Der Backtrackansatz garantiert, daß *Fits* genau $s!/(f_1!\cdots f_k!)$ mal aufgerufen wird.

Mit Hilfe dieses Berechnungsansatzes ergeben sich folgende Stellungsanzahlen: *Harlekin*-Puzzle: 176250, *Dad*-Puzzle: 18504 und *Mann-und-Flasche*-

Programm 2.1 *Count*: Mittels Urnen wird die Anzahl der Konfigurationen in allgemeinen Schiebepuzzles gezählt. Eingabe: Tiefe *depth* der derzeitigen Iteration, Koordinaten *pos* für den zu plazierenden Spielstein. Ausgabe: Anzahl verschiedener Stellungen *Configs*.

```
if (depth= s)                    {alle Spielsteine passen auf das Brett}
    Configs ← Configs + 1        {erhöhe die Anzahl der Konfigurationen}
else                        {es fehlen noch zu plazierende Spielsteine}
    for all j in {1,...,k}       {ziehe aus der Urne den nächsten Stein}
      if (urn_j > 0)                          {sofern es noch einen gibt}
        if (Fits(j,pos))                {falls Spielstein auf das Brett paßt}
          urn_j ← urn_j − 1                 {nehme Spielstein aus der Urne}
          Set(j,pos)                         {plaziere Stein auf dem Brett}
          pos' ← GetNewPos(pos)                {suche neuen Bezugspunkt}
          Count(depth+1,pos')          {rekursiver Aufruf des Programms}
          ReSet(j,pos)                        {entnehme Stein vom Brett}
          urn_j ← urn_j + 1                {lege Spielstein zurück in Urne}
```

Puzzle: 143100. Für das *Esel*-Puzzle (65880) und das *Jahrhundert*-Puzzle (109260) decken sich die Ergebnisse mit denen, die in dem Buch *Gewinnen* angeführt sind.

2.2.2 Normierung einer Konfiguration

Für die Ausführung eines Zuges ist die Darstellung als Spielsteinsequenz ungeeignet, da sich die Positionen der einzelnen Spielsteine auf dem Brett nur aufwendig rekonstruieren lassen. Eine *Konfiguration* besteht aus den Positionsbeschreibungen der Spielsteine bzw. deren Bezugspunkten, die als ein abschnittsweise sortiertes Array verwaltet werden. Der Bereich der Leerfelder in dem Array wird nicht sortiert. Der somit benötigte Speicherplatz bei s Spielsteinen auf dem Brett ist $O(s)$.

Eine wichtige Operation für Konfigurationen ist die *Normierung*, die mit dem Index des sich verändernden Bezugspunktes und der Bewegungsrichtung des gewählten Zuges aufgerufen wird. Wird ein Spielstein bewegt, so kann die aufsteigende Reihenfolge der Positionen innerhalb der *Familie* gleicher Spielsteine zerstört werden. Sei a die Länge des Arraybereiches, der für diese Familie von Spielsteinen steht. Dann wird durch einen auf- oder abwärtsgerichteten Ringtausch die Sortierung innerhalb einer Familie in $O(a)$ wieder hergestellt.

Die Normierung kann in zwei Fällen umgangen werden. Wird keine Duplikatselimination vorgenommen, so erübrigt sich die Erzeugung einer eindeutigen Darstellung. Auf der anderen Seite kann bei geeignet gewählter Darstellung eines Zustandes im Speicher die normierte Stellung sich unmittelbar aus der Hin- und Rücktransformation der Positionsbeschreibung ergeben.

2.2.3 Expansion einer Spielstellung

Eine Stellung wird expandiert (siehe Programm 2.2), indem all ihre Nachfolgerstellungen erzeugt werden.

Programm 2.2 *Expand*: Die Expandierung einer Stellung im allgemeinen Schiebepuzzle. Eingabe: Konfiguration *u*. Ausgabe: Nachfolgermenge $\Gamma(u)$.

```
Pebble                    {plaziere Leerfelder und lege Kieselsteine aus}
for all i ∈ {1,...,s}     {für alle Steine auf dem Brett und}
  for all d ∈ {up,down,left,right}   {für alle Richtungen}
    if (CanMove(S_i, d))  {falls S_i-Zug in Richtung d möglich ist}
      Copy(u, v)          {erzeuge neue Kopie v von u}
      DoMove(v, S_i, d)   {führe Zug in Richtung d aus}
      Norm(v, i, d)       {sortiere Teil im Positionsarray}
      Γ(u) ← Γ(u) ∪ {v}   {erweitere die Nachfolgermenge}
Unpebble                  {entferne Kieselsteine rund um alle Leerfelder}
```

Dabei werden in einem ersten Schritt Kieselsteine rund um ein Leerfeld gelegt (engl. *pebble*, vergl. Abbildung 2.6), die signalisieren, ob und aus welcher Richtung ein Leerfeld anliegt. Die Zugmöglichkeiten der Spielsteine wird dann durch das Aufsammeln der Kiesel an den Rändern der Spielsteine geprüft. Die dazu benötigte Zeit ist linear in der Anzahl der ausgelegten Kiesel und damit auch linear in der Gesamtzahl aller Leerfelder. Die Ausführung eines Zuges wird durch die horizontalen und vertikalen Breiten der Spielsteine festgelegt.

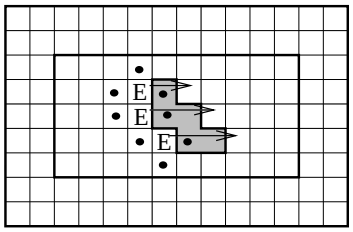


Abbildung 2.6: Das Auslegen von Kieselsteinen.

Sei a_0 die Anzahl der Leerfelder und s die Anzahl der Spielsteine auf dem Brett. Dann ist das Auslegen und Entfernen der Kieselsteine in Zeit $O(a_0)$ möglich. Die Operation *CanMove* wird $O(s)$ mal aufgerufen und führt insgesamt maximal soviele Vergleiche durch, wie es an den Spielsteinen anliegende Leerfelder gibt. Im dem Fall, daß ein Kiesel in einer Richtung fehlt, schlägt die Anfrage der Zugmöglichkeit in eben diese Richtung sofort fehl. Ein Leerfeld kann aber nur in maximal vier Richtungen Nachbar anderer Spielsteine sein. Deshalb benötigen die *CanMove*–Aufrufe einer Expansion *insgesamt* $O(a_0+s)$ bzw. $O(s)$ viele Operationen. Die Durchführung *aller DoMove* Operationen ist in Zeit $O(a_0)$ möglich, da der Spielstein um ein Feld und alle in die Zugrichtung anliegenden Leerfelder entsprechend den Ausdehnungen des Spielsteines

Schicht für Schicht verschoben werden. Sei a_i die Größe der Familie, in der der bewegte Spielstein S_i liegt. Dann liegt der Aufwand für eine Normierung in $O(a_i)$ bzw. in $O(s)$. Existieren m Nachfolgerkonfigurationen zu u, so ist der Gesamtaufwand von *Expand* somit $O(sm)$.

Dieser Ansatz ist optimal unter der Annahme, daß jeder Nachfolgezustand vollständig abgelegt werden muß und ist jedem Algorithmus vorzuziehen, der in Linearzeit zur Gesamtgröße des umgebenden Spielbrettes agiert. Für das (n^2-1)–Puzzle ist m gleich vier und die Laufzeit somit proportional zur Anzahl der Spielsteine.

Ohne Duplikatselimination entfällt der Aufruf *Norm*. Mit dem erwähnten *Backtrackprinzip* kann auf die Anforderung eines neuen Zustandes durch den Aufruf *Copy* verzichtet werden, da ein Zug jeweils durch eine zu *DoMove* inverse Funktion *MoveBack* wieder zurückgenommen werden kann. In diesem Fall ist die gesamte Expansion eines Knotens in Zeit $O(s)$ möglich.

2.2.4 Heuristik

Für das allgemeine Schiebepuzzle sind es die Bezugspositionen, die zu einer veränderten Manhattendistanz $MD_{GST}(S, S')$ zwischen zwei Stellungen S und S' führen sollen.

Um diesen Distanzbegriff auf das allgemeine Schiebepuzzle zu erweitern, ist das Problem der Normierung zu lösen, durch die sich die Reihenfolge der Spielsteine innerhalb einer Familie F_j verändern kann. In Abbildung 2.7 ergibt sich auf der linken Seite für die Konfiguration $(32, 38)$ eine Manhattandistanz zu $(29, 32)$ von insgesamt sieben, obwohl nur ein Zug notwendig ist, um die linke in die rechte Seite zu überführen.

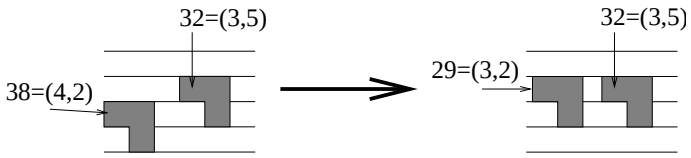


Abbildung 2.7: Das Problem der Manhattendistanz bei einer Normierung.

Die Idee zur Vermeidung dieser Schwierigkeit ist es, den Abstand *dist* aller k_j Familienmitglieder aus F_j gemeinsam zu bewerten, d.h. wir betrachten:

$$dist(F_j, F_j') = \left| \sum_{i=1}^{k_j} x_i - \sum_{i=1}^{k_j} x_i' \right| + \left| \sum_{i=1}^{k_j} y_i - \sum_{i=1}^{k_j} y_i' \right|.$$

Dies ist eine untere Schranke für die Familie F_j, in ihren Zielzustand F_j' zu gelangen. Damit ergibt sich für das *allgemeine Schiebepuzzle* insgesamt die untere Schranke

$$MD_{GST}(S, S') = \sum_{j=1}^{k} dist(F_j, F_j').$$

Der Berechnungsaufwand für die $MD_{GST}(S, S')$ ist linear in der Anzahl der beteiligten Spielsteine.

Eine Verbesserung der Heuristik besteht in dem Ansatz, für alle Spielsteinfamilien das vereinfachte Problem zu lösen, daß durch die Entnahme der anderen Spielsteine entsteht. Durch die gegenseitigen Konflikte ergibt sich ein größerer Wert für $dist(F_j, F'_j)$ und damit für $MD_{GST}(S, S')$.

Die Qualität der Heuristik steht in einem solchen Teillösungsansatz im Widerstreit zur effizienten Berechenbarkeit; immerhin muß für jeden Suchbaumknoten eine Teilsuche durchgeführt werden. Deshalb gehen wir dazu über, alle Teillösungslängen in einer Tabelle abzulegen, die mit den Positionen der Familienmitglieder adressiert wird. Diese Tabelle wächst exponentiell mit der Größe der Familie. Im Vergleich zu einer solchen auf Teillösungslängen basierenden *heuristischen Musterdatenbank* (engl. *heuristic pattern database*) für das $(n^2 - 1)$-Puzzle (Culberson & Schaeffer 1996), sind wir im allgemeinen Schiebepuzzle aufgrund der Unterschiedlichkeit der Spielsteine in der vorteilhaften Situation, die ermittelten Werte aufsummieren zu können.

Die Berechnung der Heuristik kann leicht auf die Berücksichtigung der folgenden zwei Besonderheiten erweitert werden.

1. Die Zielbedingung im *Esel*-, *Jahrhundert*- und *Dad*-Puzzle beinhaltet sogenannte Plazebo- (engl. *don't care*) Variablen, die anzeigen, daß die Position einzelner Spielsteine irrelevant für die Lösung ist.

2. Im *Harlekin*-Puzzle sind die Zielzustände nicht immer durch absolute Koordinaten gegeben. Relative Koordinatenvergleiche werden durch eine gemeinsame Verschiebung realisiert.

2.2.5 Lösungen

Alle angegebenen Beispielprobleme wurden innerhalb des *HSF-Light* Systems (vergl. Anhang B) gelöst. Wir experimentierten mit der ursprünglichen Manhattandistanz MD_{GST}. Tabelle 2.2 gibt die Anzahl der expandierten Knoten der verschiedenen Puzzles mit dem A^* Suchverfahren (vergl. Kapitel 3) an.

Puzzledomäne	Lösungslänge	Anzahl expandierter Knoten mit Heuristik	ohne Heuristik
Dad	45	1956	2176
Esel	120	24022	24077
Jahrhundert	131	41805	42371
Harlekin	29	2597	3071
Mann-und-Flasche	31	15604	17047

Tabelle 2.2: Lösungen im allgemeinen Schiebepuzzle.

Für alle Probleme lag der Zeitaufwand der Suche auf einer Sun Ultra Workstation im Sekundenbereich. Die erzeugten Lösungen zu den einzelnen Puzzles werden im Anhang angegeben.

Die Knotenersparnis, die sich durch die veränderte Manhattandistanz ergibt, ist gering und weist auf die Aussageschwäche der Heuristik hin. Dies läßt sich auch dadurch belegen, daß mit Ausnahme des Harlekin Puzzles ein sehr großer Teil (1/10 bis ca. 1/4) des gesamten Problemgraphen besucht wird.

Das *IDA** Verfahren kam aufgrund der vielen Duplikate und der recht einfachen Heuristik nicht zu einer Lösung. Im *Dad* Puzzle wurden schon bei Tiefe 16 mehr als eine 1.5 Millionen Knoten expandiert.

2.2.6 Komplexität

Da das (n^2-1)–Puzzle schon als *NP*-vollständig nachgewiesen wurde (Ratner & Warmuth 1990), verwundert es nicht, daß sich auch das *allgemeine Schiebepuzzle* in dem Sinne als schwer erweist. In diesem Falle ist der Beweis so illustrativ, daß er kurz vorgestellt werden soll. Wir beziehen uns auf die Entscheidungsvariante des *allgemeinen Schiebepuzzles*, die allein danach fragt, ob das Spiel überhaupt eine Lösung besitzt.

Die Eingabe des *allgemeinen Schiebepuzzles* besteht aus der Brettgröße B, der Anzahl s und der Brettpositionen $p_1, \ldots, p_s$ der Spielsteine und Leerfelder. Sei $p_{\max}$ das Maximum der p_i, dann ist die Länge der Eingabe demnach durch $O(\log B + s \log p_{\max})$ beschränkt (wir verwenden fairerweise das logarithmische Kostenmaß zur Beschreibung der Eingabe). Die Struktur der Spielsteine kann bei der folgenden Argumentation automatisch generiert werden und wird nicht zur Eingabe gezählt.

Die Idee der *polynomiellen Reduktion* $\leq_p$ auf *Partition* (Wegener 1993c) wird in Abbildung 2.8 dargestellt. *Partition* fragt nach einer Aufteilung der Zahlen a_1 bis a_s in zwei Mengen S und S', so daß $\sum_{i \in S} a_i$ gleich $\sum_{i \in S'} a_i$ ist.

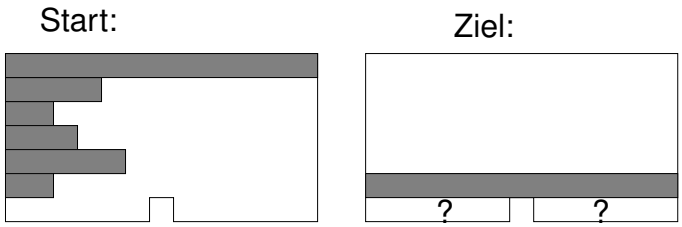


Abbildung 2.8: *NP*-härte des *allgemeinen Schiebepuzzles*.

Aus den Längen a_1 bis a_s für *Partition* werden Spielsteine der Größe *1* mal a_1 bis *1* mal a_s konstruiert und horizontal auf ein Spielbrett der Breite $b = 1 + \sum_{i=1}^{s} a_i$ gelegt, das zwei Ausbuchtungen der Länge 1 und der Breite $(\sum_{i=1}^{s} a_i)/2$ hat. Zusätzlich gibt es einen Spielstein der Größe *1* mal b. *Partition* hat genau dann eine Lösung, falls sich der Spielstein der Größe *1* mal b sich von oben nach unten bewegen läßt. Dieses kann wiederum als Zielbedingung in dem *allgemeinen Schiebepuzzle* formuliert werden.

2.3 Sokoban

Das Sokoban Puzzle ist eines der wenigen Einpersonenspiele, in denen die menschliche Lösungsqualität noch immer alle maschinellen Strategien übersteigt. Die Startposition in Sokoban (siehe Abbildung 2.9) besteht aus b Bällen, die von irgendwo innerhalb eines Labyrinths auf die b Zielfelder gebracht werden sollen. Ein Männchen, gesteuert durch den Spieler, kann sich innerhalb des Spielbrettes bewegen und die Bälle auf anliegende Leerfelder schieben.

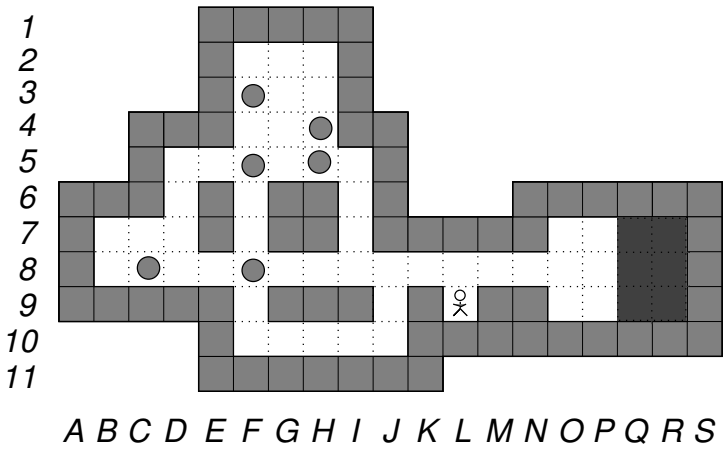


Abbildung 2.9: Das *Sokoban* Puzzle.

2.3.1 Komplexität

Wir definieren drei Probleme, *Decide*, *Pushes* und *Moves*: *Decide* ist das Entscheidungsproblem, das Puzzle zu lösen. *Pushes* fragt zusätzlich nach der minimalen Anzahl von Ballbewegungen, während *Moves* eine optimale Anzahl der Männchenbewegungen erfordert.

Sokoban ist *PSPACE*-vollständig (Culbersone 1997). Sei B die Größe des Spielbrettes, dann bleibt für eine feste Anzahl von Bällen b die Anzahl $K(B)$ der Spielstellungen polynomial:

$$K(B) = B\binom{B-1}{b} = B \cdot (B-1) \cdot \ldots \cdot (B-b)/b! \in O(B^{b+1}).$$

Eine Lösung für *Moves* oder für *Pushes* impliziert eine Lösung für *Decide*, aber optimale Lösungen der beiden erstgenannten Probleme bedingen sich gegenseitig nicht. Dennoch ist *Moves* die unzweifelhaft schwerere Variante von Sokoban.

Mensch und Maschine können am besten in der weltweiten Bestenliste von 90 Problemen (siehe http://xsokoban.lcs.mit.edu/xsokoban.html) verglichen werden.

2.3.2 Zustände und Übergänge

Bei der Lösung von *Moves* empfiehlt es sich, den Suchraum als einen gewichteten Graphen darzustellen, in dem jede Kante einer Ballbewegung entspricht. Das Gewicht der Kante ergibt sich aus dem durch eine Breitensuche (*BFS*) aufgespürten kürzesten Pfad von der derzeitigen Position des Männchens zu dem zu bewegenden Ball (Cormen, Leiserson & Rivest 1990).

Manchmal läßt sich die Suche an *Artikulationsknoten* des Labyrinths (Felder, die bei der Belegung durch eine Mauer das Labyrinth in zwei oder mehrere Leerfeldbereiche aufteilen) in voneinander unabhängige Teilsuchen gliedern, deren Lösungen letztendlich nur noch zusammengefügt werden müssen. Hierzu führen wir *Löcher* innerhalb des Labyrinths ein, an denen Bälle hinterlegt

bzw. entnommen werden können. In Abbildung 2.9 ist *K8* ein Artikulationsknoten und eine geeignete Koordinate, um den Suchraum in eine *Einbring*- und eine *Einsortierungsphase* zu unterteilen.

Desweiteren führen wir *Tunnelmakros* ein. Makrozüge fassen mehrere Züge zu einem einzelnen zusammen und führen zu einer weiteren drastischen Reduktion des Suchraumes (Korf 1985b). Dabei ergeben sich Tunnel aus der Überlegung, daß in einer optimalen Lösung das Männchen keinen Ball in einen Korridor hineinschieben und selbigen Flur wieder zurücklaufen muß, um erst später den Ball im Flur weiterzuschieben. Dabei gibt es zwei mögliche Tunnelausgänge: Der Tunnel endet an einer Kreuzung oder an einer Abzweigung. Im ersten Fall wird das Ende des Makrozuges auf die Kreuzung selbst gelegt (wie z.B. *F6-F8*), während im zweiten Fall das Ende des Zuges bis hinter die Abzweigung vorverlegt werden kann (wie z.B. *G8-J8*). An Artikulationen können Makros verlängert werden (z.B. auf *G8-P8*), da das Männchen den Ball sowieso von der anderen Seite nicht zu greifen bekommt.

Ein nicht an den Zielbereich grenzendes Feld ist *verboten*, wenn es keine Ecke darstellt oder an einer u-förmig eingefaßten Wand steht, an der kein weiteres Zielfeld angrenzt. Ein hier abgelegter Ball ist von der Wand nicht mehr loszulösen. So sind z.B. *F2*, *G2* oder *H2* genauso wie *D5*, *O7* und *O9* verboten, während *P7*, *Q7* und *R7* wiederum erlaubt sind.

In speichersensiblen Suchverfahren sollte die Zustandsbeschreibung sehr klein sein. Wir bevorzugen eine *Bitmap* des Brettes, also einem Vektor Boole'scher Werte, die die Existenz eines Balles an einer gegebenen Brettposition bezeugen. Dabei können die verbotenen Felder weggelassen werden. Die Position des Männchens wird binär codiert.

2.3.3 Heuristik

Die untere Schranke für *Pushes* wird durch ein kostenminimal bewertetes Matching in denjenigen bipartiten Graphen bestimmt, dessen Knoten auf der einen Seite durch die Kugeln und und auf der anderen Seite durch die Zielfelder gegeben sind (vergl. Abbildung 2.10). Die Kanten entsprechen den Wegen aller Bälle von ihren jetzigen Positionen auf die unterschiedlichen Zielfelder. Die Bewertungen werden durch die kürzesten Lösungen dieser Einballprobleme bestimmt. Sei b die Anzahl der Bälle, dann hat der bipartite Graph $2b$ Knoten und $b^2/4$ Kanten.

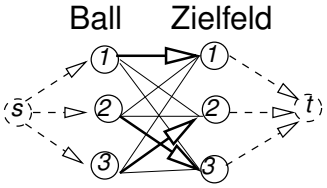


Abbildung 2.10: Ein bipartites Matching zur Berechnung der unteren Schranke im Sokobanpuzzle.

Junghanns und Schaeffer schlagen einen kostenminimierenden Netzwerkflußalgorithmus (*minimum cost network flow algorithm*), vor (Junghanns & Schaeffer 1998b). Dazu wird ein initial leeres Matching wieder und wieder

durch die Anwendung des originalen $O(b^2)$ Algorithmus von Dijkstra vergrößert (Mehlhorn 1984; Dijkstra 1959).

Allerdings muß das Verfahren von Dijkstra um die Bewältigung negativer Kanten ergänzt werden. Mehlhorn schlägt eine Potentialfunktion zur Lösung des Problems vor, aber der von uns in Kapitel 3 untersuchte Ansatz zum Wiederöffnen schon betrachteter Knoten (engl. *reopening*) ist völlig ausreichend.

In einer Erweiterung der Heuristik können wir die relative Position des Männchens zum Ball berücksichtigen. Versperrt ein Ball nämlich einen Eingang, so ist es von entscheidender Bedeutung, ob das Männchen auf der einen oder anderen Seite des Raumes steht.

Die Heuristik kann weiter verbessert werden, indem diejenigen Konflikte betrachtet werden, die sich durch eng aneinanderliegende Bälle ergeben. Angenommen, der Ball auf *C8* in Abbildung 2.9 läge auf *D8*. Dann läge er im linearen Konflikt zu dem Ball auf *F8*, denn dieser müßte eine Bewegung entgegen seines kürzesten Lösungsweges machen, um Ball *D8* das gleiche zu ermöglichen.

Als besondere Schwierigkeit erweist sich die Unabhängigkeit dieser Erweiterung zum ermittelten Matching, das die Abhängigkeit der Bälle untereinander nicht berücksichtigen kann. Wir erkennen einen Konflikt in den erweiterten horizontalen und vertikalen Nachbarschaften der Bälle, die sich in ihrer Zugausführung gegenseitig behindern. So verhindern die Bälle auf *D8* und *F8* gegenseitig eine horizontale Zugausführung, die für die kürzesten Wege beider Bälle zum Ziel erforderlich ist.

In einem *gierigen* Ansatz (engl. *greedy*) entnehmen wir zur Berechnung der Gesamtzahl linearer Konflikte der Stellung jeweils denjenigen Ball, der die meisten Konflikte auflöst und demnach die größte Anzahl der von ihm abhängigen Bälle besitzt und bewerten ihn mit zwei (Hin- und Rückzug). Dann berechnen wir wieder die Menge der sich bedingenden Bälle usw.

Eine einzelne Konfliktauflösung ist mit Hilfe einer Datenstruktur Schlange (*first in first out*-Speicher) in Zeit $O(b)$ möglich, so daß die Berechnung der Zugabe zur unteren Schranke insgesamt in Zeit $O(b^2)$ möglich ist.

Bei der Heuristik in *Moves* können wir zu der Heuristik einen Wert addieren, der sich durch die Bewegung des Männchens von den Zielfeldern zu den jeweils als nächstes einzusammelnden Bällen ergibt. Dieser Wert kann mit Hilfe einer ähnlichen *minimum matching* Heuristik wie bei der Einbringung der Bälle gefunden werden. Die Kantenbewertungen ergeben sich aus den möglichen Männchenwegen zwischen den Zielfeldern und den Bällen. Dabei muß die Distanz des Männchens einmal abgezogen werden, denn bei n Bällen muß das Männchen genau $(n-1)$ mal zurück.

2.3.4 Sackgassen

Mit den vorgebrachten Ideen läßt sich das Problem aus Abbildung 2.9 problemlos in ein paar Sekunden mit dem *HSF-Light* System lösen. Die für *Moves* und *Pushes* optimale Lösung ist im Anhang A angegeben. Um jedoch mehr als nur eine Handvoll der 90 Benchmarkprobleme zu lösen, ist eine weitere Verfeinerung des Verfahrens nötig.

Sokoban liegt ein gerichteter Problemgraph zugrunde, da ein Ziehen der Bälle ausdrücklich nicht gestattet ist. Somit ergeben sich Stellungen, die zu keiner Lösung im Spiel mehr führen können. Diese bezeichnen wir als *Sackgassen*.

In dem Kapitel 7 werden wir auf ein neuartiges Lernverfahren eingehen, das uns durch die Aufteilung einer Stellung ermöglicht, Sackgassen aufzuspüren und das verantwortliche Merkmal (hier ist es eine Gruppe von Bällen) auszukristallisieren, um anschließend in einer Aufwärtsbewegung daraus weitere Merkmale für eine Sackgasse zu gewinnen.

2.4 Der Zauberwürfel

Eine andere Herausforderung im Einpersonenspiel ist der Zauberwürfel (engl. Rubik's Cube), erdacht von dem ungarischen Spieleerfinder Ernö Rubik. Jede Seite des Zauberwürfels (vergl. Abbildung 2.11) ist in neun Einzelwürfelchen (sogenannte *Cubis*) aufgeteilt. Die Drehung einer Seite um 90, 180 oder 270 Grad ist ein Zug. Somit ergeben sich durch die sechs Ansichten insgesamt 18 verschiedene Zugmöglichkeiten. Das Ziel in diesem Problem ist es, durch Drehen der einzelnen Seiten die Gleichfarbigkeit der neun Würfelchen auf allen Würfelseiten zu erreichen.

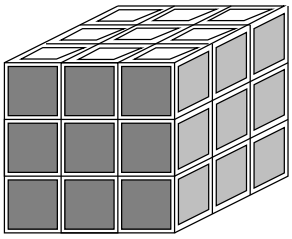


Abbildung 2.11: Der Zauberwürfel.

2.4.1 Zustandsanzahl

Jeder der acht Eckwürfel hat drei mögliche Verdrehungen an seinem Platz, jeder der zwölf Kantenwürfel hat zwei davon, die anderen sogenannten Mittelwürfel sind fest (sie bilden das Drehkreuz, an dem die Steine aufgehängt sind). Somit ergeben sich durch ein Auseinanderbauen und erneutes Zusammensetzen insgesamt $12! \cdot 2^{12} \cdot 8! \cdot 3^8 \approx 5.1903 \cdot 10^{20}$ verschiedene Stellungen. Anne Scott hat bewiesen, daß in Wahrheit genau ein zwölftel davon, also ca. $4.3252 \cdot 10^{19}$, aus einer gegebenen Stellung heraus erreichbar sind (Berlekamp, Conway & Guy 1982). Da es viel mathematische Literatur über den Zauberwürfel gibt (Rubik et al. 1987; Eidswick 1986; Turner & Gold 1985; Larsen 1985), beschränken wir uns auf die Lösungsfindung mit heuristischer Suche.

Der Verzweigungsgrad des durch die wiederholte Expansion der Knoten aufgespannten Suchbaumes für Zauberwürfel ist mit 18 sehr groß. Korf zeigt durch die Lösung von 10 zufällig generierten Instanzen des Zauberwürfels auf, daß die durchschnittliche Lösungslänge wahrscheinlich bei 18 liegt (Korf 1997b). Der *Durchmesser* des Problems (die Lösungslänge der *schwersten* Instanz) ist nicht bekannt. Korf verwendet zur Beschneidung des Suchraumes zwei einfache Beschneidungsregeln: *Drehe keine Seite zweimal hintereinander* und: *Drehe gegenüberliegende Seiten in genau einer Reihenfolge.* Damit läßt sich der Verzweigungsgrad (wie wir in Kapitel 10 sehen werden) auf 13.348 senken. Ein so beschnittener Suchbaum hat in der Tiefe 17 weniger als 19

Trillionen Knoten, so daß der Durchmesser mindestens 18 ist. Liegt die mittlere Lösungslänge tatsächlich bei 18, so verbleiben nur noch wenig Chancen für eine Verbesserung durch eine weitere Beschneidung des Suchbaumes.

2.4.2 Heuristik

Die naheliegende Wahl einer unteren Schranke ist eine dreidimensionale Version der Manhattandistanz. Dabei wird für jeden Würfel die minimale Anzahl der Züge berechnet, um ihn sowohl in eine richtige Position als auch in eine richtige Orientierung zu bringen. Dann werden diese Zahlen aufsummiert. Leider muß die Summe durch acht dividiert werden, da sie der Anzahl der bewegten Würfel in einem Zug entspricht. Dies können wir dadurch verbessern, daß wir die Eck- und die Kantenwürfel getrennt voneinander untersuchen und die jeweiligen Werte durch vier teilen. Der Erwartungswert der so ermittelten heuristischen Funktion für die Kantenwürfel liegt bei ca. 5.5, während der Erwartungswert der Eckwürfel bei ca. 3 liegt. Damit erweist sich die Kantenwürfelheuristik mit 5.5 Zügen als die zu erwartende untere Schranke einer Stellung.

Korf schlägt vor, nicht das Einwürfelproblem zu betrachten, sondern den Suchraum in zwei relaxierte Probleme aufzuteilen, in denen alle Kanten bzw. alle Ecken als unwichtig erklärt (weiß gefärbt) werden. Im ersten Fall verbleiben uns $8! \cdot 3^7$ oder ca. 88 Millionen verschiedene Stellungen (die letzte Orientierung ist durch die anderen festgelegt). All diese Stellungen werden durch eine Breitensuche gelöst und in einer schnell zu adressierenden Tabelle zusammen mit ihrer Lösungslänge abgelegt. Diese Zahl von 88 Millionen zu speichernden Zuständen ist so groß, daß sich eine Breitensuche schwer im Hauptspeicher realisieren läßt. Die Knoten zweier aufeinanderfolgender Schichten werden demnach in Dateien auf der Festplatte des Rechners verwaltet. Im zweiten Fall müssen wir uns mit sechs gemeinsam untersuchten Kantenwürfelchen zufrieden geben und generieren zwei Tabellen mit $12! \cdot 2^6/6!$ bzw. 42 Millionen Stellungen.

Die einzige Art und Weise, diese Teilergebnisse zu einer unteren Schranke zu kombinieren, ist, für eine Stellung das Maximum der Lösungslängen innerhalb dieser Tabellen zu bestimmen. Somit erhalten wir einen mittleren heuristischen Wert von 8.878 Zügen.

Kapitel 3

Grundlegende Algorithmen

In diesem Kapitel stellen wir die wesentlichen Suchverfahren in der Zustandsraumsuche vor. Wir zeigen, wie das A Verfahren durch eine Neugewichtung der Kanten in dem Problemgraphen aus dem Algorithmus von Dijkstra erschlossen werden kann. Diese unübliche Sichtweise des A* Verfahrens zeigt auf, daß A* ein effizientes und theoretisch fundiertes Graphsuchverfahren ist. Desweiteren betrachten wir einfache Tiefensuchverfahren, wie den IDA* Algorithmus, das branch and bound Verfahren und Erweiterungen, wie MEIDA*, die den gesamten Hauptspeicher in die Berechnung integrieren. Dabei zeigen wir auf, wie die Datenstruktur Weakheap als beidseitige Vorrangwarteschlange in der speicherplatzbeschränkten Suche eingesetzt werden kann. Wir schließen das Kapitel mit der Vorstellung einiger alternativer Suchansätze, wie der partiellen Suche und dem Planen.*

3.1 Gerichtete Graphsuche

Das Lösen von Zustandsraumproblemen kann als gerichtete Suche in einem implizit beschriebenen bewerteten Graphen verstanden werden. Den Zuständen entsprechen die Knoten und den Übergängen entsprechen die Kanten des Graphen. Graphtheorie ist ein so zentraler Bereich in der Algorithmengestaltung, daß es nahezu keine Grundvorlesung in der Informatik gibt, die auf das intensive Studium von Graphen verzichtet.

Bei genauerer Betrachtung der entwickelten Verfahren der Zustandsraumsuche finden sich wichtige Ergänzungen der grundlegenden Graphsuchalgorithmen, die dennoch nur selten den Weg in die Lehrbücher der Graphtheorie gefunden haben. Eine Ausnahme bildet das Buch *Data Structures and Algorithms 2: Graph Algorithms and NP Completeness* von Mehlhorn (Mehlhorn 1984), das die Rolle einer *Schätzfunktion* (engl. *estimator*) bei der Suche nach den kürzesten Wegen innerhalb eines Graphen analysiert.

Die Basis aller Suchverfahren der Zustandsraumsuche bilden die beiden Algorithmen A^* (Hart, Nilsson & Raphael 1968) und *IDA** (Korf 1985a). Die Verfahren suchen gerichtet mit unteren Schranken für die noch zu bewältigende Zieldistanz. Der Schätzwert am Knoten u wird häufig mit $h(u)$, der tatsächliche Wert mit $h^*(u)$ bezeichnet. Addiert man die bis dato zurückgelegte Entfernung $g(u)$ hinzu, so erhält man eine Gesamtschätzung $f(u) = g(u) + h(u)$

der Lösungslänge $f^*(u) = g(u) + h^*(u)$ eines optimalen Zielpfades, der mit dem derzeitigen Generierungspfad beginnt.

Reicht der Speicherplatz aus, um die gesamte Exploration zu speichern, so kann ein Rechenverfahren den Schätzwert f einer Position stetig verbessern. Diese in den ersten Abschnitten untersuchte Berechnung bezeichnen wir mit *dynamischer Programmierung.*

Übersteigt die Größe des Graphens den Speicherplatz, so kann man ein und denselben Standort als unterschiedlich ansehen, wenn er von verschiedenen Wegen aus erreicht wird. Damit gelangen wir in den letzten Abschnitten dieses Kapitels in das Gebiet der speicherplatzbeschränkten Algorithmen.

3.2 Speicherplatzsensible Suche

Schrittweise wird erläutert, wie der graphtheoretisch fundierte Algorithmus von Dijkstra (Dijkstra 1959) zum A^* Verfahren ausgeweitet werden kann. Dabei werden insbesondere zwei Probleme analysiert: Die Integration einer Heuristik und, daraus resultierend, Kanten mit negativer Bewertung.

Die Korrektheit von A^* wird in der Literatur häufig mit dem Beweis von Judea Pearl bzw. mit dem Beweis von Nilsson (Nilsson 1982; Pearl 1985) belegt. Leider sind die vorgeschlagenen Beweise nicht vollständig. Dies wird von Eckerle bemerkt und behoben (Eckerle 1997). Die im folgenden vorgetragene Beweisführung von A^* vereinfacht Eckerles Argumentation und lehnt sie an die einschlägige Notation der Graphtheorie an (Cormen, Leiserson & Rivest 1990).

Definition 1 *Sei $G = (V, E, w)$ ein bewerteter Graph mit Kantenbewertungsfunktion $w : E \to R$, s ein ausgezeichneter Startknoten und T eine Menge von Zielknoten t. Der Pfad $p = \langle s = v_0, \ldots, v_k = t \rangle$, mit minimaler Bewertung $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) = C^*$ heißt optimaler Zielpfad. Für Pfade benutzen wir manchmal die Schreibweise $\rightsquigarrow$ z.B., $p = s \rightsquigarrow t$, und für Kanten (u, v) schreiben wir mitunter $u \to v$. Ein Suchverfahren wird zulässig genannt, wenn es beim erstmaligen Erreichen eines Zielknotens C^* berechnet. Die Länge des graphentheoretisch am kürzesten bewerteten Weges von u nach v in dem Zustandsgraphen sei mit $\delta(u, v)$ bezeichnet. Desweiteren sei N die Anzahl der Knoten u, für die $\delta(s, u)$ kleiner-gleich C^* ist. Eine Gewichtsfunktion w, bei der $\min\{\delta(u, t) | t \in T\}$ für alle u aus V nicht negativ ist, wird optimistisch genannt.*

Eine *Vorrangwarteschlange* (engl. *priority queue*) ist eine Datenstruktur für total geordnete Mengen, die die Operationen *Initialize*, *Deletemin, DecreaseKey* und *Insert* anbietet. Dabei kann die einen Schlüsselwert verändernde Funktion *DecreaseKey* als eine Kombination einer *Insert* und *Delete* Anweisung angesehen werden. Zur Implementation einer Vorrangwarteschlange stehen dem Programmierer mehrere Möglichkeiten zu Verfügung: Linksbäume, balancierte Suchbäume, wie z.B. 2-3 Bäume oder Rot-Schwarzbäume (Ottmann & Widmayer 1993).

Die Struktur Halde (engl. *heap*) ist ein in ein Array eingebetteter Baum, in dem der Schlüsselwert an einem Knoten nicht größer ist als an seinen Kindern. Besonders zur Realisation einer Vorrangwarteschlange eignen sich relaxierte Heapdatenstrukturen wie *Weakheaps* (vergl. Abschnitt 3.4.2) und Fibonacci

heaps (Fredman & Tarjan 1987). Weakheaps ermöglichen sehr schnelles Sortieren, während Fibonacci-Heaps erlauben, die Operation *DecreaseKey* sehr effizient (in amortisierter konstanter Zeit) durchzuführen.

3.2.1 Dijkstras Algorithmus

Im Algorithmus von Dijkstra (vergl. Programm 3.1 und 3.2) wird eine Vorrangwarteschlange *Open* genutzt, die die Knoten des Suchhorizontes abspeichert und denjenigen Knoten mit der geringsten Generierungspfadlänge expandiert. Längere Generierungspfade werden einfach durch kürzere überschrieben. Die Kernidee des Algorithmus von Dijkstra ist demnach, eine Abstandsfunktion $f : V \rightarrow R$ solange zu verbessern, bis letztendlich der Minimalabstand $\delta(s,t)$ für $f(t)$ gefunden ist. Sei $\Gamma(u)$ die Menge der durch eine Expansionsfunktion *Expand* erzeugten Nachfolger von u. Für alle Nachfolger v aus $\Gamma(u)$ wird geprüft, ob v in der Vorrangwarteschlange existiert. Ist dies der Fall, so ergibt die Ordnungsnummer von v den minimal bekannten f Wert, also das Minimum aus der alten Bewertung $f(v)$ und dem neu sich eröffnenden Weg über u mit Länge $f(u)$ plus $w(u,v)$. Andernfalls wird v mit der Generierungspfadlänge $f(u)$ plus $w(u,v)$ neu in *Open* eingefügt. Die Gleichung $f(v) \leftarrow \min\{f(u)+w(u,v), f(v)\}$ wird als Bellman'sche Optimalitätsgleichung bezeichnet, dem Kern eines jeden Verfahrens zur Dynamischen Programmierung (Bellman 1957). Ist die Graphgröße kleiner als der zur Verfügung stehende Speicherplatz, so bietet sich an, *Open* durch eine Doppelrepräsentation von Vorrangwarteschlange und ein auf die dort gespeicherten Knoten verweisendes Array zu repräsentieren. Dies ist der Algorithmus von Dijkstra für explizite Graphen, wie er zumeist in den Lehrbüchern zu finden ist.

Übersteigt die Graphgröße die Speicherplatzressourcen, so können schon gespeicherte Knoten mit Hilfe einer *Hashtabelle* gefunden werden. Eine *Hashtabelle* ist eine Arraystruktur fester Länge, die durch eine möglichst streuende und erschöpfende Abbildung von der Menge der Zustände (Universum) aus adressiert wird. Ist die Abbildung nicht injektiv, so muß eine Kollisionsbehandlung vorgenommen werden, in der die Synonyme (Zustände mit gleichem Hashwert) zum Beispiel verkettet werden (Cormen, Leiserson & Rivest 1990; Ottmann & Widmayer 1993).

Zur besseren Beschreibung wird angenommen, daß zu jedem erzeugten v aus V sein aktueller Generierungspfad p_v verwaltet wird. In der konkreten Implementierung kann dieser Pfad rückwärtig über eine Vorgängerfunktion generiert werden.

Hilfssatz 1 *Sei $G = (V, E, w)$ ein positiv gewichteter Graph und f der im Algorithmus von Dijkstra verwaltete Wert. Immer dann, wenn u aus der Vorrangwarteschlange entnommen wird, ist $f(u)$ gleich $\delta(s,u)$,*

Beweis: (Cormen, Leiserson & Rivest 1990) Sei u bei gegenteiliger Annahme der erste aus der Vorrangwarteschlange *Open* entnommene Knoten mit $f(u) \neq \delta(s,u)$, genauer mit $f(u) > \delta(s,u)$, und $p_u = s \rightsquigarrow x \rightarrow y \rightsquigarrow u$ der zu u gehörende kürzeste Pfad, wobei y der erste Knoten ist, der noch nicht expandiert wurde (vergl. Abbildung 3.1).

Es gilt $f(x) = \delta(s,x)$ aufgrund der Minimalität von u. Damit ist:

$$f(u) \leq f(y) \leq f(x) + w(x,y) = \delta(s,x) + w(x,y) = \delta(s,y) \leq \delta(s,u).$$

Programm 3.1 *Dijkstra*: Ein Graphsuchalgorithmus für die kürzesten Wege in einem gerichteten Graphen. Eingabe: Ein gerichteter und gewichteter Graph G, Startknoten s. Ausgabe: Kürzester Pfad zu einem Zielknoten.

```
Initialize(Open)          {leere die f-geordnete Vorrangwarteschlange}
Insert(Open, [s, ⟨s⟩], 0)                  {s wird mit Kosten 0 bewertet}
while (Open ≠ ∅)          {solange noch Horizontknoten existieren}
  [u, p_u] ← Deletemin(Open)        {entferne Knoten mit min. f-Wert}
  if (u ∈ T) return p_u             {Ziel gefunden, gebe Lösung zurück}
  else Γ(u) ← Expand(u)             {Expansion liefert Nachfolgermenge}
    for all v in Γ(u)                    {für alle Nachfolger v von u}
      p_v ← ⟨p_u, v⟩                {erweitere den Generierungspfad zu u}
      Improve(u, v)                                {rufe Unterprogramm auf}
```

Programm 3.2 *Improve*: Ein Unterprogramm zur Abwägung zweier alternativer Pfade zu v durch die Berechnung von $f(v)$ als Minimum von $f(v)$ und $f(u)+w(u,v)$. Eingabe: Knoten u und v. Ausgabe: Verbesserung der oberen Schranke $f(v)$ für $\delta(v)$.

```
if (Search(Open, v))                          {Knoten wurde schon erzeugt}
  if (f(u) + w(u, v) < f(v))                        {neuer Weg ist kürzer}
    DecreaseKey(Open, [v, p_v], f(u) + w(u, v))          {verbessere f(v)}
else                                        {Knoten wurde noch nicht erreicht}
  Insert(Open, [v, p_v], f(u) + w(u, v))          {füge v neu in Open ein}
```

Widerspruch. □

Liegen Kanten mit negativer Bewertung vor, so gilt die letzte Ungleichung im Beweis nicht.

Der Algorithmus von Dijkstra expandiert die Knoten immer nach den kleinsten f-Werten. Deshalb nennt man den Algorithmus auch f– oder *best-first* geordnet.

Satz 1 *(Korrektheit Dijkstra) Der Algorithmus von Dijkstra ist als Suchverfahren zulässig.*

Beweis: Da w positiv ist, gilt $f(v) \geq f(u)$ für alle in die Vorrangwarteschlange *Open* aufgenommenen Nachfolger v von u. Bei der ersten Entnahme eines Zielknotens t ist demnach das Optimum C^* erreicht, denn die kürzesten Wege zu weiteren Zielen t' können nicht abnehmen. □

Die Laufzeit von *Dijkstra* wird im allgemeinen in der Anzahl der Knoten n und in der Anzahl der Kanten e des Graphen G gemessen. Dijkstra selbst gibt eine quadratische Laufzeit $O(n^2)$ für sein Verfahren an, da er den Knoten

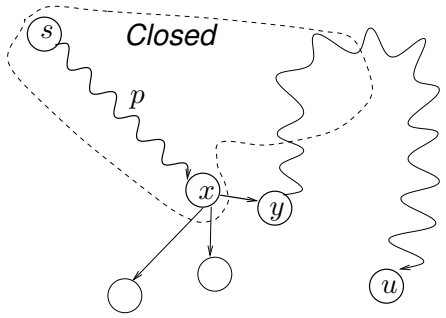


Abbildung 3.1: Entnahme von u aus *Open*.

minimaler Bewertung jeweils durch einen Lauf durch das gesamte Knotenarray gewinnt. Durch die Verwendung von Fibonacci heaps kann die Laufzeit (amortisiert) auf $O(n \log n + e)$ gesenkt werden (Cormen, Leiserson & Rivest 1990). Da wir nur einen Teil G' von G explorieren, übertragen sich die Resultate auf die Anzahl der Knoten n' und auf die Anzahl der Kanten e' des Graphen G'. Wird hingegen, wie in dem Bereich *Heuristische Suche* üblich, die Anzahl der Knotenexpansionen als Komplexitätsmaß gewählt, so erhalten wir folgendes Resultat.

Satz 2 *(Expansionsanzahl Dijkstra) Der Algorithmus von Dijkstra expandiert im ungünstigsten Fall (engl. worst-case) $O(N)$ viele Knoten.*

Beweis: Das Verfahren expandiert alle Knoten u mit Werten $\delta(s,u) \leq C^*$ höchstens einmal. □

3.2.2 Integration einer Heuristik

Eine *Heuristik* h stellt eine richtungsweisende Hilfestellung in der Suche dar.

Definition 2 *Eine Heuristik h ist eine Abbildung von der Knotenmenge V des Graphens in die Menge der positiven reellen Zahlen. Weiterhin ist $h(t)$ für alle t aus T gleich Null.*

Häufig ist $h(t)$ nur für die Zielknoten t aus T gleich Null. In diesem Fall ist der Test für das Erreichen eines Zielzustandes sehr einfach.

Beispiel 1 *Für einen Autofahrer soll der kürzeste Weg von einer Stadt s in eine Stadt t ermittelt werden. Gegeben ist einerseits das als gewichteter Graph $G = (V, E, w)$ repräsentierte Straßennetz und andererseits eine Übersichtskarte, aus der sich die Luftlinienentfernungen $dist$ zwischen den Städten ablesen läßt. Der Wert der Heuristik für eine betrachtete Stadt u ergibt sich als $h(u) = dist(u,t)$. Er wird zu der bisher auf der Straße zurückgelegten Distanz $w(p_u) = \sum_{i=1}^{l} w(v_{i-1}, v_i)$ des bisher untersuchten Weges $p = \langle s = v_0, \ldots, v_l = u \rangle$ hinzuaddiert und ergibt eine Gesamtschätzung $\hat{w}(p_u)$ für die Weglänge eines sich aus p_u ergebenen Weges zum Ziel. Durch diese Zusatzinformation erscheinen Städte, die sich in Richtung des Zieles befinden, vielversprechender als jene, die entgegengesetzt der Zielrichtung liegen.*

Auf dem Weg von der Stadt u nach v wird die Schätzung der Gesamtdistanz wie folgt erweitert:

$$\hat{w}(p_v) = w(p_v) + h(v) = w(p_u) + w(u,v) + h(v) =$$

$$\hat{w}(p_u) - h(u) + w(u,v) + h(v).$$

In der Literatur wird häufig die Bezeichnung $g(u)$ für $w(p_u)$ und $f(u)$ für $\hat{w}(p_u)$ verwendet und die Gleichung $f(u) = g(u) + h(u)$ in den Mittelpunkt der Betrachtungen gestellt. Beachtet werden muß hierbei, daß $g(u)$ als auch $f(u)$ vom Generierungspfad p_u abhängen und somit im Verlauf des Algorithmus verschiedende Werte annehmen.

Die Heuristik h läßt sich in den Algorithmus von Dijkstra integrieren, indem h in die Kantenbewertung eingearbeitet wird.

Satz 3 *(Reweighting) Sei $G = (V, E)$ ein gerichteter mit $w : E \to R$ gewichteter Graph, $p = \langle s = v_0, \ldots, v_k = t \rangle$ ein Pfad von s zu einem Zielknoten t und $h : V \to R$ eine Heuristik. Definiere $\hat{w}(u,v) := w(u,v) - h(u) + h(v)$. Bezeichne mit $\delta(s,t)$ die Länge des kürzesten Weg von s nach t im ursprünglichen und mit $\hat{\delta}(s,t)$ entsprechend im neugewichteten Graphen. Es gilt $w(p) = \delta(v_0, v_k)$, dann und nur dann, wenn $\hat{w}(p) = \hat{\delta}(v_0, v_k)$ gilt. Außerdem genau dann, wenn G mit der Gewichtsfunktion w keine Zyklen mit negativem Gewicht hat, besitzt auch G mit $\hat{w}$ keine.*

Beweis: (Cormen, Leiserson & Rivest 1990) Es gilt

$$\begin{aligned} \hat{w}(p) &= \sum_{i=1}^{k} (w(v_{i-1}, v_i) - h(v_{i-1}) + h(v_i)) \\ &= w(p) - h(v_0). \end{aligned}$$

Beweisrichtung *dann*: In Kontraposition nehmen wir an, daß ein Pfad p' existiert mit $\hat{w}(p') < \hat{w}(p)$ und $w(p') \geq w(p)$. Dies ist gleichbedeutend mit $w(p') - h(v_0) < w(p) - h(v_0)$ und $w(p') < w(p)$. Widerspruch. Der Beweis der Richtung *nur dann* verläuft analog.

Sei $c = \langle v_0, \ldots, v_l = v_0 \rangle$ ein Zyklus negativer Länge. Dann gilt aufgrund der teleskopartigen Summe $\hat{w}(c) = w(c) + h(v_l) - h(v_0) = w(c)$. □

Definition 3 *Sei $G = (V, E, w)$ ein gewichteter Graph. Eine Heuristik h, die die Bedingung $h(u) \leq h(v) + w(u,v)$ für alle Kanten $e = \{u, v\}$ erfüllt, wird konsistent genannt.*

Satz 4 *(Konsistente Heuristik) Sei h eine konsistente Heuristik. Wird im Algorithmus von Dijkstra Insert($Open, [s, \langle s \rangle], h(s)$) statt Insert($Open, [s, \langle s \rangle], 0$) und die Operation DecreaseKey($Open, [v, p_v], f(u) + \hat{w}(u,v)$) statt DecreaseKey($Open, [v, p_v], f(u) + w(u,v)$) gesetzt, bleibt das Verfahren zulässig.*

Beweis: Alle im Algorithmus berechneten f–Werte sind durch die Neuerung *Insert*($Open[s, \langle s \rangle], h(s)$) um $h(s)$ erhöht. Da h konsistent ist, gilt $\hat{w}(u,v) = w(u,v) - h(u) + h(v) \geq 0$. Damit sind die Voraussetzungen des Hilfssatzes 1 für die Gewichtsfunktion $\hat{w}$ erfüllt. Es gilt also immer

$$f(u) = \hat{\delta}(s,u) + h(s),$$

wenn u aus der Vorrangwarteschlange entnommen wird. Nach Satz 3 bleiben die kürzesten Wege bei einer Neubewertung eines Graphen durch eine Heuristik erhalten. Somit gilt für ein aus *Open* entnommenes t aus T:

$$f(t) = \hat{\delta}(s,t) + h(s) = \hat{w}(p_t) + h(s) = w(p_t) = \delta(s,t).$$

Da $\hat{w}$ positiv ist, gilt $f(v) \geq f(u)$ für alle in die Vorrangwarteschlange *Open* aufgenommenen Nachfolger v von u. Die f–Werte der expandierten Zielknoten steigen somit monoton, d.h. bei der ersten Entnahme eines Zielknotens t ist das Optimum C^* erreicht. □

3.2.3 Kanten mit negativem Gewicht

Häufig entstehen gerade durch die Integration einer Heuristik in die Gewichtsfunktion Kanten mit negativer Bewertung $\hat{w}(u,v)$.

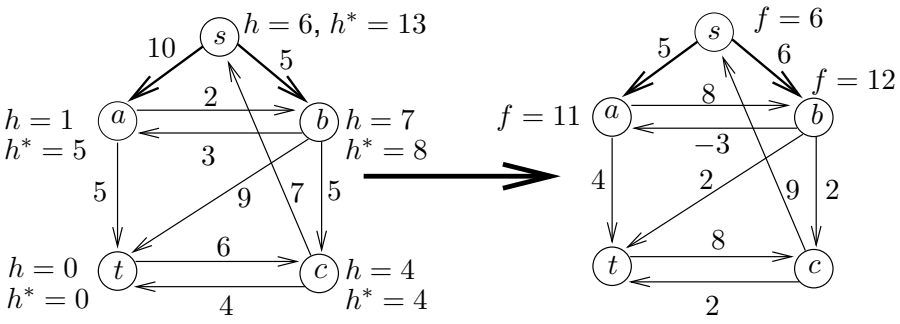


Abbildung 3.2: Ein Graph vor und nach einer Neubewertung der Kanten.

Beispiel 2 *Betrachte den Beispielgraph in Abbildung 3.2 vor und nach der Neubewertung der Kanten durch die Heuristik h und verfolge den geänderten Algorithmus von Dijkstra.*

Der Knoten s sei expandiert, so daß die Knoten a und b in der Vorrangwarteschlange gemäß ihren Gewichten $f(a) = 11$ und $f(b) = 12$ vorliegen. Somit wird a expandiert und t neu in Open aufgenommen. Der f-Wert zu b wird nicht verändert, da die Pfadlänge von s zu b nicht sinkt. Bei der anschließenden Expansion von b liegt a nicht mehr in Open vor und die bis dato ermittelte Information über die Wegstrecke von s zu a ist verloren.

Darum werden in einer weiteren Datenstruktur *Closed*, die die Operationen *Initialize, Search, Insert* und *Delete* anbietet, die schon expandierten Knoten gespeichert und im Falle eines neuentdeckten, kürzeren Weges zurück nach *Open* befördert. Wird zu Beginn des Hauptprogrammes *Initialize*(*Closed*) und nach *DeleteMin*(*Open*) die Operation *Insert*(*Closed*,$[u, p_u], f(u)$) aufgerufen, so gestaltet sich die Prozedur *NewImprove* als Erweiterung von *Improve*, wie in Programm 3.3 ersichtlich.

Wir werden im folgenden eine Bedingung entwickeln, für die dieser *veränderte Algorithmus von Dijkstra* beim Erreichen des Zielknotens auch bei negativen Kantenbewertungen korrekt ist.

Programm 3.3 *NewImprove*: Die Behandlung von Kanten mit negativer Bewertung im Algorithmus von Dijkstra. Eingabe: Knoten u und v. Ausgabe: Verbesserung der oberen Schranke $f(v)$ für $\delta(v)$.

```
if (Search(Open, v))                                  {Knoten wurde schon erzeugt}
  if (f(u) + w(u,v) < f(v))                              {neuer Weg ist kürzer}
    DecreaseKey(Open, [v, p_v], f(u) + w(u,v))            {verbessere f(v)}
  else if (Search(Closed, v))                             {v schon expandiert}
    if (f(u) + w(u,v) ≤ f(v))                          {neuer Weg nicht länger}
      Delete(Closed, v)                                     {Neuöffnung von v}
      Insert(Open[v, p_v], f(u) + w(u,v))                        {(reopening)}
else                                                {Knoten wurde noch nicht erreicht}
  Insert(Open, [v, p_v], f(u) + w(u,v))              {füge v neu in Open ein}
```

Hilfssatz 2 *Sei $G = (V, E)$ ein gerichteter mit $w : E \to R$ gewichteter Graph. Sei f der im veränderten Algorithmus von Dijkstra verwaltete Wert und $\delta(s, v)$ das minimale Gewicht der Wege von s nach v. Vor jeder Entnahme eines Elementes u aus der Vorrangwarteschlange Open gilt die folgende Invarianz:*

(I) *Es gibt einen Knoten v in der Vorrangwarteschlange Open mit $f(v) = \delta(s, v)$, der auf einem optimalen Zielpfad liegt.*

Beweis: Offenbar gilt (I) initial, und wenn der mit $DeleteMin(Open)$ entnommene Knoten u ein Zielknoten ist, ist nichts zu zeigen, da der Algorithmus an dieser Stelle beendet wird. Es wird gezeigt, daß (I) in jedem Schleifendurchlauf erhalten bleibt.

Das Einfügen von u mit $Insert(Closed, (u, p_u), f(u))$ hat keinen Einfluß auf (I). Wir betrachten die folgenden Fälle:

1. u liegt nicht auf einem optimalen Zielpfad. Da kein v aus $Open \cap \Gamma(u)$ mit $f(v) = \delta(s, v) \leq f(u) + w(u, v)$ verändert wird, gilt die Invarianz.

2. u liegt auf einem optimalen Zielpfad p. Sei u über den Generierungspfad p_u erreicht. Wenn u die Eigenschaft (I) sichert, so ist $f(u) = w(p_u) = \delta(s, u)$, sonst wird (analog zu 1.) kein v aus $Open \cap \Gamma(u)$ mit $f(v) = \delta(s, v)$ verändert. Es gibt ein $v \in \Gamma(u)$ auf dem Pfad $p = s \overset{p_1}{\rightsquigarrow} u \overset{e}{\to} v \overset{p_2}{\rightsquigarrow} t$ mit $f(u) + w(u, v) = \delta(s, v)$ (vergl. Abb. 3.3).

Der Knoten v wird so in *Open* aktualisiert, daß (I) gilt, wie folgende Überlegungen beweisen:

- Ist v schon in *Open* enthalten und $f(u) + w(u, v) < f(v)$, dann wird $f(v)$ auf diesen Wert gesetzt. Der optimale Zielpfad p_t, der v enthält, ist gegeben durch $p_t = s \overset{p_u}{\rightsquigarrow} u \overset{e}{\to} v \overset{p_2}{\rightsquigarrow} t$.

  Ist dies nicht der Fall, d.h $f(v) = \delta(s, v)$, so ist auch der gespeicherte Generierungspfad p_v zu v der Anfang eines kürzesten Zielpfades und (I) bleibt gültig für den zusammengesetzten Pfad aus p_v und den Rest des bisher betrachteten Pfades p, d.h. der optimale v enthaltene Zielpfad ist $p_t = s \overset{p_v}{\rightsquigarrow} v \overset{p_2}{\rightsquigarrow} t$.

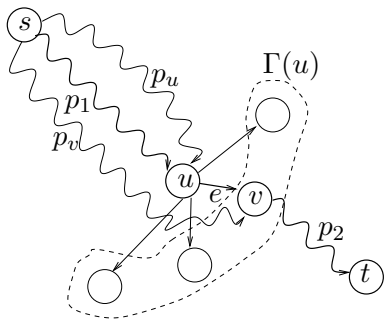


Abbildung 3.3: Beleg der Invarianzeigenschaft.

- Ist v schon in *Closed* enthalten, dann gilt $\delta(s,v) = f(u) + w(u,v) \leq f(v)$ und v wird zurück nach *Open* befördert. Der optimale Zielpfad, der v enthält, ist: $p_t = s \overset{p_u}{\rightsquigarrow} u \overset{e}{\rightarrow} v \overset{p_2}{\rightsquigarrow} t$.
- Ist v weder in *Open* noch in *Closed* enthalten, so wird v neu in *Open* mit $f(v) = \delta(s,v)$ eingefügt. Optimaler v enthaltender Zielpfad: $p_t = s \overset{p_u}{\rightsquigarrow} u \overset{e}{\rightarrow} v \overset{p_2}{\rightsquigarrow} t$.

Somit gilt (I) für alle Iterationen des Algorithmus.

□

Bemerkt sei, daß sich (I) auch für den Algorithmus zeigen läßt, in dem die letzte Fallunterscheidung $f(u) + w(u,v) < f(v)$ lautet (Eckerle 1997).

Satz 5 *(Optimistische Gewichtsfunktion) Sei $G = (V,E)$ ein gerichteter mit $w : E \rightarrow R$ gewichteter Graph und $\delta(u,v)$ das minimale Gewicht der Wege von u nach v. Wenn w optimistisch ist, d.h. für alle u in V ist $\min\{\delta(u,t) |\ t \in T\}$ positiv, dann ist $f(t)$ bei der ersten Entnahme eines Zielknotens t aus der Vorrangwarteschlange Open gleich C^*.*

Beweis: Annahme: Der Algorithmus hält auf einem nicht optimalen Zielknoten t', also $f(t') > C^*$. Nach (I) existiert bei der Entnahme von t' ein Knoten u mit $f(u) = \delta(s,u)$ in *Open*, der auf einem optimalen Zielpfad p_t zu einem Zielknoten t liegt. Es gilt:

$$f(t') > C^* = \delta(s,u) + \min\{\delta(u,t) | t \in T\} \geq \delta(s,u) = f(u),$$

im Widerspruch zur Tatsache, daß t' statt u aus *Open* gewählt wurde. Damit ist bei der ersten Entnahme eines Zielknotens t das Optimum C^* erreicht.
□

Wird die durch die Heuristik gewonnene Gewichtsfunktion $\hat{w}$ entsprechend den obigen Beschreibungen zur Berechnung der f–Werte genutzt, so entsteht dadurch der A^* Algorithmus (Hart, Nilsson & Raphael 1968).

Definition 4 *Eine Heuristik h wird optimistisch genannt, wenn sie die minimale Restdistanz zu Zielknoten t unterschätzt, d.h. wenn für alle u gilt: $h(u) \leq \min\{\delta(u,t) | t \in T\} = h^*(u)$.*

Eine optimistische Heuristik ist also eine untere Schranke für das zu lösende Teilproblem der Suche nach dem minimalen Pfad von u nach t.

Satz 6 *(Optimistische Heuristik) Sei $G = (V, E, w)$ ein gerichteter, gewichteter Graph, h eine Heuristik und $\hat{w}(u, v) = w(u, v) - h(u) + h(v)$ eine Neugewichtung von G. Genau dann, wenn $\hat{w}$ optimistisch ist, ist h optimistisch.*

Beweis: Da $h(t) = 0$ ist und kürzeste Wege nach Satz 3 erhalten bleiben, gilt:

$$0 \leq \min\{\hat{\delta}(u,t)|t \in T\} = \min\{\delta(u,t) - h(u) + h(t)|t \in T\} =$$

$$\min\{\delta(u,t) - h(u)|t \in T\} = \min\{\delta(u,t)|t \in T\} - h(u) = h^*(u) - h(u).$$

□

Folgerung 1 *Der A^* Algorithmus ist für gerichtete und gewichtete Graphen $G = (V, E, w)$ und optimistische Heuristiken h zulässig.*

In der Literatur (Russell & Norvig 1995) findet sich auch eine andere Methode, mit negativen Kantenbewertungen umzugehen.

Für alle Nachfolger v von u sei $f_{\max}(v)$ durch $\max\{f(m)|m \in p_v\}$ bzw. rekursiv durch $\max\{f_{\max}(u), f(v)\}$ festgelegt. Diese Gleichung wird in der Literatur als *pathmax Gleichung* bezeichnet. Offensichtlich ist $f_{\max}$ auf Pfaden p monoton nicht fallend. Für durch konsistente Heuristiken definierte Schätzungen f gilt immer $f_{\max}(u) = f(u)$. Die Methode, den A^* Algorithmus auf $f_{\max}$ anzuwenden, führt beim abermaligen Betreten eines Zustandes zu Problemen.

Beispiel 3 *Sei Open bzgl. $f_{\max}$–geordnet. Betrachte den Graphen in Abbildung 3.2 nach der Expansion des Knotens a. Es ist Open $= \{(b, \langle s, b\rangle, 12), (t, \langle s, a, t\rangle, 15)\}$ und Closed $= \{(s, \langle s\rangle, 6), (a, \langle s, a\rangle, 11)\}$. Nun wird a über b erneut erreicht, d.h. $(b, \langle s, b\rangle, 12)$ von Open nach Closed befördert und $(a, \langle s, b, a\rangle, 12)$ mit der Closed Liste verglichen. Fälschlicherweise gelangt nun $(a, \langle s, a\rangle, 11)$ zurück in Open und die in $(a, \langle s, b, a\rangle, 12)$ steckende Information über den Teil $\langle s, b, a\rangle$ des Lösungspfades $\langle s, b, a, t\rangle$ geht verloren.*

Pearl hat gezeigt, daß A^* *optimal* ist, in dem Sinne, daß jeder andere Suchalgorithmus bis auf *Tiebreaking* die gleichen Knoten wie A^* expandieren muß (Pearl 1985). *Tiebreaking* ist die Entscheidung, an welchem Knoten bei gleicher Bewertung fortgefahren werden soll. Wir werden uns dieses Ergebnis plausibel machen.

Satz 7 *(Untere Schranke Knotenexpansionen) Sei $G = (V, E, w)$ ein optimistisch bewerteter Graph. Ein zulässiger Suchalgorithmus A muß alle Knoten $u \in V$ mit $\delta(s, u) < C^*$ expandieren.*

Beweis: Annahme: A findet einen optimalen Zielpfad p_t mit $w(p_t) = C^*$ und expandiert ein u mit $\delta(s, u) < C^*$ nicht. Wir zeigen, daß es dann in G einen unberücksichtigten Zielpfad q geben kann, für den $w(q) < C^*$ gilt, was im Widerspruch zur Zulässigkeit von A steht. Dazu sei q_u der Pfad in G, für den $w(q_u) = \delta(s, u)$ gilt. Weiterhin sei $(u, t) \in E$ mit $w(u, t) = 0$ dem Graphen G und t der Zielmenge T hinzugefügt.

Da u nicht expandiert wird, ist A nicht bekannt, ob es in G die Kante (u,t) gibt. Da w optimistisch ist, muß $w(u,t) \geq \min\{\delta(u,t)|t \in T\} \geq 0$ gelten, was trivialerweise für $w(u,t) = 0$ erfüllt ist. Sei $q = \langle q_u, (u,t)\rangle$. Dann gilt

$$w(q) = w(q_u) + w((u,t)) = w(q_u) = \delta(s,u) < C^*.$$

□

Wir rekapitulieren, daß N als die Anzahl der Knoten u festgelegt ist, für die $\delta(s,u)$ kleiner-gleich C^* ist.

Folgerung 2 *Seien die Werte $\delta(s,u)$, $u \in G$, untereinander paarweise verschieden. Ein zulässiger Suchalgorithmus A muß für optimistisch bewertete Graphen $G = (V,E,w)$, $\Omega(N)$ viele Expansionen durchführen.*

Beweis: Es ist $|\{u \in G|\delta(s,u) < C^*\}| = N-1$. Nach Satz 7 muß A mindestens $N-1$ Knoten expandieren. □

Ein paar abschließende Bemerkungen über den A^* Algorithmus richten sich an eine geeignete Implementation.

Wenn die Kantenbewertungen und auch die heuristische Schätzung ganzzahlig sind, so kann die Vorrangwarteschlange durch ein Array der Größe $f_{\max}$ realisiert werden, an dessen Einträgen jeweils eine Liste von Knoten gleicher Bewertung abgelegt ist. Dabei ist $f_{\max}$ eine bekannte obere Schranke für die Lösungslänge C^*. Sind n' und e' die Anzahl der Knoten bzw. Kanten des explorierten Teilgraphen G' von G und die zugrunde liegende Heuristik konsistent, dann wird eine Laufzeit von $O(n' + e' + f_{\max})$ erreicht, wobei $f_{\max}$ eine bekannte obere Schranke für die Lösungslänge C^* ist.

Weiterhin läßt sich *Closed* implizit als die Differenz aus der Menge aller jemals generierter Knoten, die in der Hashtabelle H abgelegt sind, und den Horizontknoten aus *Open* verwalten. Diese natürliche Repräsentation von A^* ist in Programm 3.4 angegeben.

Programm 3.4 *A^*-Improve*: Die veränderte *Improve* Funktion im A^* Algorithmus mit implizit verwalteter *Closed*-Liste. Eingabe: Knoten u und v. Ausgabe: Verbesserung der oberen Schranke $f(v)$ für $\delta(v)$.

```
if (Search(H, v))                                  {wurde v schon generiert}
  if (f(u) + w(u,v) < f(v))                {falls neuer Weg eine Abkürzung ist}
    if (v ∈ Open)                              {falls v ein Horizontknoten ist}
      DecreaseKey(Open, [v, p_v], f(u) + w(u,v))          {verbessere f(v)}
    else                                       {v ist voll expandierter Knoten}
      Insert(Open[v, p_v], f(u) + w(u,v))                        {öffne v neu}
else                                                    {v noch nicht generiert}
  Insert(Open, [v, p_v], f(u) + w(u,v))     {füge v erstmalig in Open ein}
  Insert(H, v)                                {füge v (einmalig) in H ein}
```

Ein Zustandsraum wird üblicherweise vom initialen Zustand zu der Menge der Zielzustände hin abgesucht. Es ist genauso möglich, die Suche von den

Zielzuständen hin zum Anfangszustand zu richten, wobei die Operatoren in der Rückrichtung angewendet werden (Bundy 1997; Barr & Feigenbaum 1981). Hierbei empfiehlt es sich, die Vorrangwarteschlange initial mit allen Zielknoten zu füllen, anstatt von jedem Zielknoten erneut auszugehen.

In der *Bidirektionalen Suche* (Korf 1988; Eckerle 1996) werden Vorwärts- und Rückwärtssuche nebenläufig aufgerufen, wobei die beiden Suchhorizonte einander angenähert werden. In der bidirektionalen Breitensuche ergibt sich die Lösung beim Aufeinandertreffen der Suchfronten unmittelbar, während in der bidirektionalen heuristischen Suche alle *konkatenierten* (zusammengefügten) Lösungspfade zur Verringerung einer globalen Lösungsschranke genutzt werden, bis diese mit der assoziierten unteren Schranke zusammenfällt. Die sogenannte *wave shaping* Technik dient zum Abgleich der Suchfronten. Es wird immer derjenige Knoten expandiert, der den Abstand zwischen den beiden Suchhorizonten minimiert (Eckerle 1997).

Einen Kompromiß der uni- und bidirektionalen Suche ist die *Perimetersuche* (Dillenburg & Nelson 1994). Hier wird eine tiefenbeschränkte Breitensuche vom Zielzustand aus aufgerufen und die erreichten Knoten in einer Tabelle abgelegt. Daraufhin wird eine Vorwärtssuche aufgerufen, deren Suchfront mit den Knoten im Perimeter abgeglichen wird.

3.3 Speicherplatzbeschränkte Suche

Da die heuristische Suche nach dem A^* Verfahren immer mit einem optimalen Zielpfad terminiert, wird es unter anderem zur Lösung von Einpersonenspielen eingesetzt. Dieses führt zu Problemen, da die Zustandsgraphen schnell die Speicherplatzobergrenzen eines Rechners sprengen: Die Anzahl der in den Datenstrukturen *Open* und *Closed* verwalteten Knoten wächst je nach Güte der Heuristik h bei einer effizient implementierten Expansionsfunktion schnell an. Nehmen wir an, daß für die Speicherung eines Knotens ca. 20 Bytes benötigt werden und das Suchverfahren pro Sekunde ca. zehntausend Zustände bearbeiten kann. Dann ist ein Hauptspeicher der Größe 100 MByte schon nach weniger als einer Minute erschöpft.

Mittlerweile sind viele Ansätze entwickelt worden, die mit beschränktem Speicherplatz operieren und die sich grob in zwei Kategorien einordnen lassen: Verfahren, die Speicherplatz proportional zur Suchtiefe benötigen und Verfahren, die sich adaptiv zur Speicherplatzobergrenze verhalten.

In der ersten Gruppe finden sich Verfahren, die auf einer iterierten Tiefensuche basieren oder die eine *branch and bound* Tiefensuche durchführen.

Die zweite Verfahrensklasse ist ein erst in der letzten Zeit genauer untersuchtes Gebiet, auf das wir in Abschnitt 3.4 eingehen.

3.3.1 Der Graph der Generierungspfade

In der impliziten Graphsuche ist es günstig, die Sichtweise des Problemlöseprozesses anzunehmen und statt der einzelnen Knoten die unterschiedlichen Generierungspfade zu betrachten, die zu diesen Knoten führen.

Die *Expansion* $Exp(G, s)$ eines Graphen $G = (V, E)$ zum Knoten s aus V ist ein Graph mit einer aus Generierungspfaden gebildeten Knotenmenge V' und einer Kantenmenge E' als Teilmenge von $V' \times V'$. Die *Expansion* $Exp(G, s)$ ist wie folgt rekursiv festgelegt: 1. $\langle s \rangle$ ist in V'. 2. Aus p_u in V' und (u, v) in

E folgt, daß $\langle p_u, v\rangle$ in V' und $(p_u, \langle p_u, v\rangle)$ in E' ist. Die Knoten von $Exp(G, s)$ entsprechen den durch den Graphen G induzierten Generierungspfaden, so daß im weiteren von Pfaden aus V' gesprochen wird.

Der Begriff *Expansion* ist graphentheoretisch bedingt und überlagert somit leider den eingeführten Begriff der *Expansion* von Knoten. Mehr noch, wir wollen in der Sichtweise der Generierungspfade auch von der *Expansion* eines Pfades sprechen, die der Expansion des Endknotens entspricht und die Menge der Nachfolgerpfade bildet.

Die Expansion eines endlichen Graphen ist prinzipiell unendlich. Doch selbst bei Vorgabe einer Tiefenschranke können Graphexpansionen im Verhältnis zum begründenden Graphen exponentiell groß werden, wie zwei Beispiele belegen:

Beispiel 4 *Sei $G = (V, E)$ mit $|V| = 2n$ ein ungerichteter Graph. Die ersten n Punkte $v_1, \ldots, v_n$ bilden eine n-Clique, die restlichen einen Pfad $p = \langle v_n, \ldots, v_{2n}\rangle$, d.h. (v_{i-1}, v_i) ist Kante in E für $i = \{n+1, \ldots, 2n\}$ (vergl. Abbildung 3.4). Die Expansion $Exp(G, s)$ hat über n^n Knoten bis zu der Tiefe, die den Zielknoten v_{2n} enthält.*

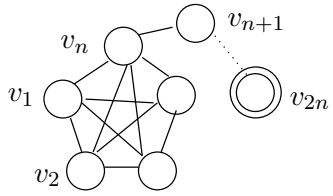


Abbildung 3.4: Ein Graph mit exponentiell großer Expansion.

Aber auch ohne Zyklen im Graphen läßt sich leicht ein exponentieller Zwischenraum (engl. Gap) zwischen der Graph- und Expansiongröße konstruieren.

Beispiel 5 *Sei $G = (V, E)$ gerichteter Graph mit $V = \{v_1, \ldots, v_{3n+1}\}$. Seien für $k \in \{0, \ldots, n-1\}$ die Kanten (v_{3k+1}, v_{3k+2}), (v_{3k+1}, v_{3k+3}), (v_{3k+2}, v_{3k+4}) und (v_{3k+3}, v_{3k+4}) in E enthalten (vergl. Abbildung 3.5). Die Expansion des Graphens G zum Startpunkt v_1 hat bis zu der Tiefe mit dem Zielknoten v_{3n+1}*

$$-1 - 2^n + 2\sum_{i=0}^{n} 2^i = -2^n + 2^{n+2} - 3 = 3(2^n - 1) \in \Omega(2^n)$$

viele Knoten.

3.3.2 Suchalgorithmen ohne Duplikatselimination

Suchalgorithmen, die keine Duplikate von Knoten eliminieren, betrachten die Expansion $Exp(G, s)$ bestehend aus der Knotenmenge V' und der Kantenmenge E' eines Graphen. Die im Algorithmus verwaltete Kostenfunktion ist eine Abbildung $w : V' \to R$. Dies begründet sich damit, daß zu jedem v aus V im Algorithmus mehrere Generierungspfade p_v unabhängig voneinander existieren können. Die folgenden Definitionen erweitern demnach nur die bisher erarbeitete Nomenklatur für Graphen.

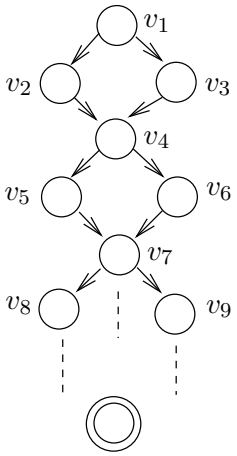


Abbildung 3.5: Ein azyklischer Graph mit exponentiell großer Expansion.

Definition 5 *Sei $G = (V, E, w)$ ein gerichteter und gewichteter Graph, s ein ausgezeichneter Startknoten, T eine Menge von Zielknoten t und $Exp(G, s) = (V', E', w)$ die durch G induzierte Expansion. Dabei ist die Kostenfunktion w' eine Abbildung $w' : V' \to R$ für einen Pfad $p = \langle v_0, \ldots, v_k \rangle$ durch $w'(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$ definiert. Die Bewertung w' wird vereinfachenderweise auch mit w bezeichnet. Die induzierten Pfade p_t mit $t \in T$ heißen Zielpfade und die Menge aller Zielpfade wird mit T' bezeichnet. Ein Suchverfahren wird zulässig genannt, wenn es beim erstmaligen Erreichen eines Zielpfades $C^* = min\{w(p_t)|p_t \in T'\}$ berechnet. Eine Gewichtsfunktion w, bei der der Wert $\min\{w(q)|p_t = \langle p_u, q \rangle \in T'\}$ für alle p_u aus T' nicht negativ ist, wird optimistisch genannt. Sei $w_{\max}(p_v) = \max\{w(p_m)|p_v = \langle p_m, q \rangle\}$. Ein Suchalgorithmus, der die erreichten und nicht expandierten Knoten entsprechend ihren $w_{\max}$–Werten expandiert, wird $w_{\max}$–geordnet genannt. Desweiteren sei N' die Anzahl der Pfade $p_u \in T'$, für die $w_{\max}(p_u)$ kleiner-gleich C^* ist.*

Besitzt G eine Baumstruktur, so gilt unmittelbar N gleich N', da jedem Generierungspfad in G genau ein Knoten aus G als *Endposten* entspricht. Die vorangegangenen Beispiele zeigen allerdings auf, daß N' in $\Omega(2^N)$ liegen kann.

Hilfssatz 3 *Genau dann, wenn die Gewichtsfunktion w optimistisch ist, gilt für alle Zielpfade p_t, daß $w_{\max}(p_t)$ gleich $w(p_t)$ ist.*

Beweis: Beweisrichtung *Dann*: Wenn $\min\{w(q)|p_t = \langle p_m, q \rangle \in T'\}$ für alle p_m aus V' nicht negativ ist, dann gilt für alle Zielpfade $p_t = \langle p_m, q \rangle$ mit p_m aus V', daß $w(q)$ größer gleich Null ist, insbesondere für den Teilpfad p_m mit $w_{\max}(p_t)$ gleich $w(p_m)$. Es gilt:

$$w(p_t) - w_{\max}(p_t) = w(p_t) - w(p_m) = w(q) \geq 0.$$

Auf der anderen Seite ist per Definition $w(p_t)$ kleiner-gleich $w_{\max}(p_t)$ und damit $w(p_t) = w_{\max}(p_t)$.

Beweisrichtung *Genau dann.* Annahme: Es gilt $w(p_t) = w_{\max}(p_t)$ für alle Zielpfade p_t und es existiert ein p_m und ein $p_t = \langle p_m, q \rangle$ mit $w(q)$ kleiner

Null. Dann ist $w(p_m) = w(p_t) - w(q)$ größer als $w(p_t)$. Widerspruch zu $w_{\max}$ monoton. □

Der folgende Satz begründet die Zulässigkeit von Algorithmen, die auf der Expansion $Exp(G, s)$ von G agieren. Die Beweisführung ist recht einfach, da $Exp(G, s)$ eine Baumstruktur aufweist.

Satz 8 *(Zulässigkeit bei $w_{\max}$ Ordnung) Sei $G = (V, E, w)$ ein gerichteter, optimistisch gewichteter Graph. Für jeden $w_{\max}$–geordneten Algorithmus A gilt beim Erreichen des ersten Zielpfades p_t: $w(p_t) = C^*$. Mit anderen Worten: A ist zulässig.*

Beweis: Annahme: $w(p_t) > C^*$, d.h. es existiert ein Zielpfad p'_t mit $w(p'_t) = C^*$, der noch nicht expandiert wurde. Zum Zeitpunkt der Terminierung von A existiert ein erreichter, aber noch nicht expandierter Pfad p_u mit $p'_t = \langle p_u, q \rangle$. Es gilt $w_{\max}(p_u) \leq w_{\max}(p'_t) = C^* < w(p_t) = w_{\max}(p_t)$ im Widerspruch dazu, daß der Algorithmus $w_{\max}$–geordnet ist und p_t gewählt wurde. □.

3.3.3 Iterierte Tiefensuche und *IDA**

Die Idee der iterierten Tiefensuche ist es, eine dynamisch wachsende, globale Kostenschranke U für die Kostenfunktion w zu nutzen, bis zu der ein rekursiver *Tiefensuchalgorithmus* (engl. *depth first search*) die Knoten von s aus expandieren soll. Der *DFS* Algorithmus (vergl. Programm 3.5) sucht einen optimalen Zielpfad p_t in dem durch eine Tiefenschranke begrenzten Suchbaum. Die Rückgabe U' von *DFS* ist die minimale Bewertung aller aufgespürten Horizontknoten und liefert den *Schwellwert* U (engl. *threshold*) für die nächste Iteration.

Programm 3.5 *DFS*: Die von *IDA** aufgerufene rekursive Tiefensuche. Eingabe: Derzeit betrachteter Pfad p_u, Tiefenschranke U. Ausgabe: Neuer Schwellwert U' (untere Schranke) für die Lösungslänge oder Lösungspfad sol zum Ziel.

```
Γ(u) ← Expand(u)                          {Expansion liefert Nachfolgermenge}
for all v in Γ(u)                                    {für alle Nachfolger}
  p_v ← ⟨p_u, v⟩                                  {ergänze Generierungspfad}
  if (goal(v) and w(p_v) ≤ U')                                 {Ziel gefunden}
    goalFound ← true; sol ← p_v                              {setze Lösungspfad}
  else if (w(p_v) ≤ U) DFS(p_v)          {Schwellwert nicht überschritten}
    else if (w(p_v) < U') U' ← w(p_v)                        {neuer Schwellwert}
```

Wird die Kantenbewertung $\hat{w}$ durch die Neugewichtung von w aufgrund einer vorliegende Heuristik h in der iterierten Tiefensuche genutzt, entsteht der bekannte *IDA** Algorithmus (vergl. Programm 3.6). Auch hier wollen wir auf die neuen Kantengewichte im folgenden vereinfachenderweise mit w verweisen. Der Lösungspfad sol kann durch den Rekursionsstapel der Tiefensuche gebildet werden.

Programm 3.6 *IDA**: Das Grundgerüst des iterativen Tiefensuchalgorithmus.
Eingabe: Gewichteter und mit Heuristik h bewerteter Graph G, Startknoten s.
Ausgabe: Kürzester Lösungsweg von s zu einem Zielknoten.

```
U ← h(s)                          {Tiefenschranke initialisieren}
U' ← ∞                            {noch kein neuer Schwellwert gesetzt}
goalFound ← false                 {Ziel wurde noch nicht gefunden}
while (U ≠ ∞)                     {solange bis ein Ziel gefunden}
  U ← U'                          {setze unteren Schwellwert}
  U' ← ∞                          {setze oberen Schwellwert}
  DFS(⟨s⟩)                        {starte Tiefensuche bei s}
  if (goalFound) return sol       {falls Ziel erreicht, gib Lösungs aus}
```

Satz 9 *(Zulässigkeit IDA***) Der Algorithmus IDA** *ist für Graphen mit einer optimistische Bewertungsfunktion zulässig.*

Beweis: (Korf 1985a) Es reicht zu zeigen, daß *IDA** $w_{\max}$–geordnet ist. Induktion über die Anzahl der *while*–Iterationen k. Sei E_k die Menge der neu expandierten Pfade der Iteration k und R_k die Menge der erreichten, aber nicht expandierten Pfade. Weiterhin sei U_k der Schwellwert der Iteration k.

Nach der ersten Iteration ist für alle p in E_1 der Wert $w_{\max}(p)$ gleich U_1 bzw. gleich $h(s)$. Weiterhin ist für alle $q \in R_1$ der Wert $w_{\max}(q)$ echt größer als U_1. Sei für alle $p \in E_k$ der Wert $w_{\max}(p)$ gleich U_k. Für alle q in R_k haben wir $w_{\max}(q) > U_k$. Also ist

$$U_{k+1} = \min_{q \in R_k} \{w_{\max}(q)\}.$$

Für alle p in E_{k+1} ist $w_{\max}(p)$ gleich U_{k+1}. Die gegenteilige Annahme würde der Monotonie der Kostenfunktion $w_{\max}$ widersprechen, da nur Pfade p mit $w(p) \leq U_{k+1}$ neu expandiert werden. So folgt, daß für alle $p \in E_{k+1}$ die Beziehung $U_k < w_{\max}(p) = U_{k+1}$ gilt. Damit ist *IDA** im Übergang von einer auf die nächste Iteration $w_{\max}$–geordnet. □

Achtung: Der direkte Abbruch beim Erreichen eines Zielzustandes in *IDA** führt im allgemeinen zu einem nicht zulässigen Verfahren. Bei einer konsistenten Heuristik oder monotonen Kostenfunktion ist ein Abbruch beim Generieren eines Zielzustandes hingegen erlaubt.

Satz 10 *(Expansionsanzahl in IDA***) IDA** *expandiert im ungünstigsten Fall* $\Omega((N')^2)$ *viele Knoten.*

Beweis: Der ungünstigste Fall tritt ein, wenn *IDA** alle schon in Iteration $i-1$ gefundenen Knoten und nur einen neuen Knoten expandiert. □

Betrachten wir als Beispiel den uniform gewichteten Pfad $p = \langle v_1, \ldots, v_{N'} \rangle$. Die heuristische Bewertungfunktion h sei die Nullfunktion. Von v_1 aus expandiert *IDA** bis zum Erreichen des Zieles $v_{N'}$ insgesamt $\sum_{i=1}^{N'} i = \binom{N'}{2}$ bzw. $\Omega(N'^2)$ viele Knoten.

3.3.4 Branch and bound

Dem Problem des zur Tiefe exponentiell großen Suchbaumes für ein gegebenes Optimierungsproblem behandelt man in der Algorithmengestaltung üblicherweise mit der *branch and bound* Methode (Schöning 1997).

Dazu betrachten wir das Problem des Handlungsreisenden (engl. traveling salesman problem, siehe Abbildung 3.6), der eine kreuzungsfreie Rundreise mit minimalen Kosten innerhalb eines gewichteten Graphen mit n Knoten und e Kanten sucht. Das Problem ist *NP*-vollständig und gehört zu den schwersten Problemen in dieser Klasse von Problemen (Wegener 1993c).

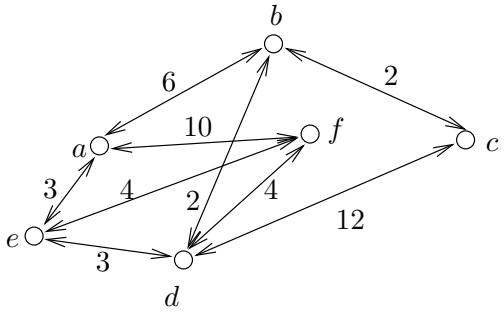


Abbildung 3.6: Das Problem des Handlungsreisenden.

Eine in $O(e + n \log n)$ mit *Fibonacci heaps* effizient berechenbare untere Schranke L für die gesuchte Rundreise ist die des *minimalen Spannbaumes*, der wiederum ein zusammenhängender, alle Knoten umfassender Baum mit minimaler Kantenbewertung ist (Cormen, Leiserson & Rivest 1990).

Die obere Schranke kann auf zwei Arten berechnet werden: Durch einen Approximationsalgorithmus oder durch das Erreichen eines Blattes im *branch and bound* Baum, wobei wir uns in der Zustandsraumsuche auf den zweiten Fall konzentrieren (Korf 1993): In einer Tiefensuche nehmen wir jeweils eine Kante zu einer initial leeren Rundreise hinzu, so daß der aufgespannte Suchbaum maximal die Tiefe $n - 1$ erreicht. Damit finden wir schnell eine, wenn auch nicht die optimale Lösung. Diese Lösung kann eine globale Variable U immer weiter verbessern, bis letztendlich $L = U$ gilt. Die zulässige Lösung, die zu der oberen Schranke des Problems gehört, ist eine optimale Lösung.

Sei initial $U \leftarrow \infty$. Dann löst das initial mit der Eingabe s aufgerufene rekursive Programm 3.7 das gegebene Zustandsraumproblem. Die Sortierung der Nachfolger dient dazu, schnell eine möglichst gute Lösung zu finden.

Die Restriktion $L(u_{i_j}) < U$ führt zum Abschneiden nicht zur optimalen Lösung führender Teilbäume. Ist $L(u_{i_j}) \geq U$, dann existiert eine Problemlösung v mit Höchstkosten $L(u_{i_j})$. Damit sind alle Lösungen t aus dem von u_{i_j} aufgespannten Teilbaum T größer als v, d.h. T braucht nicht expandiert zu werden.

Wir betrachten als Beispiel die aus den Entfernungen 0 bis 9999 ausgewürfelte Distanzmatrix für 15 Städte in Tabelle 3.1.

Die vom *branch and bound* Algorithmus im *HSF-Light* System (vergl. Anhang B) auf einer Sun Ultra in weniger als 20 Sekunden berechnete stete Verbesserung der oberen Schranke ist in Tabelle 3.2 aufgezeigt. Insgesamt wurden 245403 Knoten expandiert. Dabei wurde die aus dem minimalen Spannbaum

Programm 3.7 *BnB*: Der *branch and bound* Algorithmus. Eingabe: Derzeit betrachteter Knoten u und Generierungspfad p_u. Ausgabe: Verbesserung der globalen Variablen L, U und des Zielpfades sol.

```
Γ(u) = {u_1, ..., u_k} ← Expand(u)          {berechne die Nachfolgermenge}
{u_i1, ..., u_ik} ← Sort(Γ(u))  {sortiere gemäß ansteigender L(u_i) Werte}
for all j in {1, ..., k}                      {betrachte alle Nachfolger}
  if (L(u_ij) < U)       {untere Schranke kleiner dem globalen Maximum}
    p_uij ← ⟨p_u, u_ij⟩                      {erweitere Generierungspfad}
    if (goal(u_ij))                                      {Ziel erreicht}
      if (w(p_uij) < U) sol ← p_uij                {aktuell-kürzester Weg}
      U ← min{U, w(p_uij)}            {berechne neues globales Maximum}
    else                                               {Ziel nicht erreicht}
      BnB(u_ij, p_uij)                               {suche rekursiv weiter}
```

6838	5758	0113	7515	1051	5627	3010	7419	6212	4086	2749	2767	9084	2060	2225
7543	5089	1183	5137	5566	6966	4978	0495	0311	1367	0054	7031	3145	9882	5736
0524	8505	8394	2102	4851	9067	2754	1653	6561	7096	3628	5188	2085	4143	6967
1406	4165	3403	5562	4834	1353	0920	0444	4803	7962	9318	1422	1327	0457	1945
4479	9983	8751	3894	8670	8259	6248	7757	5629	3306	8606	3990	1738	2516	1414
5262	7116	2825	3181	3134	5343	8022	1233	7536	9760	9979	9071	1201	1336	3061
2160	4005	0729	7644	7475	1693	5514	4139	2088	6521	5202	9171	4434	8317	4582
6815	4586	9653	6306	7174	8451	3448	6473	2434	8193	4110	4748	8210	9320	2049
2956	4162	4166	4997	7793	2310	1391	9799	7926	4905	5885	2582	5610	5000	8052
0965	0120	2380	5639	6204	4385	2475	5725	7265	3214	9471	1376	4697	5543	3297
4619	3087	3123	1549	8065	4256	8973	0901	5613	6157	9899	9267	2413	9598	8526
3711	0046	4566	8536	5988	9878	3626	4273	8387	1171	2017	3752	2388	1191	1483
8122	1744	8528	5585	5363	0159	5641	8176	9575	8578	7363	7685	9344	9489	7713
5511	1461	2626	8645	3496	6703	6270	3870	1529	7499	4500	8607	5808	5725	2457
6542	6474	1531	7222	3952	7024	9894	4015	8247	1276	6278	9365	8746	1976	8092

Tabelle 3.1: Eine Distanzmatrix für das Problem des Handlungsreisenden.

resultierende untere Schranke genutzt.

Ein wichtiger Vorteil von *branch and bound* Algorithmen ist es, immer eine gute Kontrolle über die Güte der zu erwartenden Lösung zu haben, denn der Nutzen der optimalen Lösung ist nur um $U - L$ kleiner als der Nutzen der besten berechneten Lösung.

3.4 Nutzung des gesamten Hauptspeichers

Bei den auf einer Tiefensuche aufbauenden Verfahren nutzen wir nur Speicherplatz, der linear mit steigender Suchtiefe wächst. Bei derzeitigen Hauptspeicherplatzgrößen von mehreren hundert Millionen Zeichen erscheint dies eine wahre Verschwendung. In diesem Abschnitt wenden wir uns deshalb Verfahren zu, die den gesamten Hauptspeicher zur Abspeicherung von Knoteninformationen nutzen. Der zu Verfügung stehende Platz wird dabei in der Anzahl speicherbarer Knoten m gemessen. Wir beginnen mit einer durch m Knoten begrenzten Hashtabelle, die in der IDA^* Suche genutzt werden kann.

Viele der in der Literatur vorgestellten platzbeschränkten Algorithmen, wie z.B. *ITS* (Ghosh, Mahanti & Nau 1994) und *SMA* (Russell 1992) führen nicht

obere Schranke	korrespondierende Tour
22438	(0 2 7 14 9 1 10 3 13 8 6 5 12 4 11)
21396	(0 2 7 14 9 1 10 3 13 8 6 5 4 11 12)
18988	(0 2 7 14 9 1 10 3 6 8 11 13 4 12 5)
18912	(0 2 7 14 9 1 10 3 11 13 8 6 5 4 12)
18058	(0 2 7 14 9 11 1 10 3 13 8 6 5 4 12)
17173	(0 2 12 5 13 4 14 9 11 1 10 3 7 8 6)
16777	(0 4 14 9 1 10 7 8 6 2 3 11 12 5 13)
15203	(0 4 14 9 1 10 3 11 13 8 6 2 12 5 7)
14349	(0 4 14 9 11 1 10 3 13 8 6 2 12 5 7)

Tabelle 3.2: Verbesserung der Lösung im Handlungsreisendenproblem.

zu einer asymptotischen Verringerung der Zeitkomplexität, wogegen der hier vorgestellte Algorithmus *MEIDA** auf Bäumen maximal $O((N')^2/m)$ Knoten expandiert. Zur Erinnerung: A^* besitzt eine Komplexität $O(N')$ und *IDA** benötigt $O((N')^2)$ viele Knotenexpansionen. Der zu *MEIDA** von der Komplexität vergleichbare Ansatz *DBIDA** wird von Eckerle beschrieben und analysiert (Eckerle 1997).

Wir werden auf die bisher unbeachteten Implementationsfrage von *MEIDA** nach einer effizienten Verwaltung von Knoten in einer doppelseitigen Vorrangwarteschlange eingehen.

3.4.1 Duplikatselimination in *IDA**

Das Problem von *IDA** sind nicht die erneuten Expansionen der Knoten, die durch die Erhöhung des Schwellwertes entstehen, wie eine kurze mathematische Überlegung zeigt. Wir nehmen an, der Suchraum wachse exponentiell mit steigender Suchtiefe mit einem uniformen Verzweigungsgrad b an. Dadurch wird ein vollständiger b-ärer Baum aufgespannt. Sei T_d der explorierte Teilbaum, der in der d-ten Iteration von *IDA** durchsucht wird, dann ist die Anzahl der Knoten in T_d nach der geschlossenen Formel für die geometrische Reihe $\sum_{i=0}^{d} b^i$ gleich $(b^{d+1}-1)/(b-1)$ oder $O(b^d)$. *IDA** expandiert $\sum_{i=0}^{d} |T_i|$ also $\sum_{i=0}^{d}(b^{i+1}-1)/(b-1)$ viele Knoten. Dieser Wert liegt offensichtlich in $O(b^d)$. Die Schwierigkeit im *IDA** Verfahren liegt demnach vor allem darin, daß die Pfade mit gleichem Endknoten nicht erkannt werden.

Im ersten Schritt können Generierungspfade p in $Exp(G, s)$, die Knoten aus G mehrfach enthalten, in *IDA** leicht vermieden werden, indem die Bedingung: *if* $(v \in p_u)$ *return* in die Prozedur *DFS* nach der Expansion von u aufgenommen wird. Der Suchbaum eines endlichen Graphen besitzt somit auch nur endlich viele Knoten. Da ein optimaler Zielpfad keinen Zyklus enthält, bleibt *IDA** mit dieser Einschränkung zulässig.

Es bietet sich demnach in einem weiteren Schritt an, den günstigsten der bisher generierten Pfade zu einem Endknoten in eine Hashtabelle H, die die Operationen *Insert*, *Delete* und *Search* anbietet, einzutragen. Wir werden dabei darauf achten, daß sich die abermalige Expansion aufgrund eines neuen Schwellwertes im *IDA** Algorithmus und die Duplikatselimination nicht gegenseitig behindern. Das Verfahren *IDA** wird um $Insert(H, [s, \langle s \rangle])$ ergänzt.

Die aufgerufene Prozedur *HashDFS* ist in Programm 3.8 dargestellt.

Programm 3.8 *HashDFS*: Die rekursive Tiefensuche wird mittels einer Hashtabelle beschnitten. Eingabe: Derzeit betrachteter Knoten u. Ausgabe: Neuer Schwellwert (untere Schranke) für die Lösungslänge von s oder Lösungspfad sol zum Ziel.

```
Γ(u) ← Expand(u)                                  {erzeuge die Nachfolger}
for all v in Γ(u)                                    {für alle Nachfolger}
  p_v ← ⟨p_u, v⟩                                {verlängere den Generierungspfad}
  p'_v ← Search(H, v)                           {suche Äquivalent in Hashtabelle}
  if (p'_v = nil) Insert(H, [v, p_v])                  {kein Gegenpart gefunden}
  else                                          {alter Generierungspfad ist p'_v}
    if (w(p'_v) ≥ w(p_v))     {alter Pfad mindestens so teuer wie neuer}
      Delete(H, [v, p'_v])                    {lösche alten Pfad in Hashtabelle}
      Insert(H, [v, p_v])                             {und füge neuen Pfad ein}
    else return                               {alter Generierungspfad billiger}
  if (goal(v) and w(p_v) ≤ U')                                  {Ziel gefunden}
    goalFound ← true; sol ← p_v                 {aktualisiere Lösungspfad}
  else if (w(p_v) ≤ U) HashDFS(p_v)                      {rekursiver Aufruf}
    else if (w(p_v) < U') U' ← w(p_v)  {aktualisiere neuen Schwellwert}
```

Beispiel 6 *Abbildung 3.7 illustriert das Verhalten des Algorithmus in der Expansion $Exp(G, s)$ des Graphen G von Beispiel aus Abbildung 3.2. In der ersten while–Iteration ist $U = 6$ und es wird HashDFS($\langle s \rangle$) aufgerufen. Die Expansion von $\langle s \rangle$ führt zu den Nachfolgerpfaden $p_v = \langle s, a \rangle$ und $p_v = \langle s, b \rangle$, die beide in H eingefügt werden, da die Suche in H in beiden Fällen fehlschlägt. Die minimale Bewertung $w(p_v)$ der beiden Pfade ist 11 und wird in U' abgelegt.*

In der zweiten Iteration ist U somit gleich 11. Die von $\langle s \rangle$ erzeugten Nachfolger $\langle s, a \rangle$ und $\langle s, b \rangle$ werden aufgrund der Gleichheit von $w(p_v)$ und $w(p'_v)$ in H ersetzt. Die Bewertung von $\langle s, a \rangle$ ist gleich U, also wird HashDFS($\langle s, a \rangle$) aufgerufen. Die generierten Nachfolger $\langle s, a, t \rangle$ und $\langle s, a, b \rangle$ werden in H aufgenommen. Die Bewertungen dieser Pfade sind 15 bzw. 19 und damit größer als U, so daß U' nach dem Backtrack-Schritt zu $\langle s, b \rangle$ auf 12 gesetzt wird.

Die dritte Iteration (U ist gleich 12) expandiert für den Suchbaumzweig $\langle s, a \rangle$ alle Knoten der zweiten Iteration, indem es die alten gleichwertigen Werte in H überschreibt. Für den Zweig $\langle s, b \rangle$ ergeben sich die Nachfolger $\langle s, b, a \rangle$ $\langle s, b, t \rangle$ und $\langle s, b, c \rangle$. Der Wert von $\langle s, b, a \rangle$ wird mit $\langle s, a \rangle$ in H verglichen und da $w(\langle s, b, a \rangle) = 9 \leq 11 = w(\langle s, a \rangle)$ ist, wird für den Schlüssel a der günstigere Generierungspfad $\langle s, b, a \rangle$ in H eingefügt. Analog wird in H der Pfad $\langle s, a, t \rangle$ durch $\langle s, b, t \rangle$ überschrieben. Der Pfad $\langle s, b, c \rangle$ ist noch nicht in H und wird demnach neu aufgenommen. Da $w(\langle s, b, a \rangle)$ kleiner als U ist, wird HashDFS($\langle s, b, a \rangle$) initiiert und da $w(\langle s, b, a, b \rangle)$ größer als $w(\langle s, b \rangle)$ ist, wird die Suche direkt abgebrochen und für den Pfad $\langle s, b, a, t \rangle$, der $\langle s, b, t \rangle$ in H überschreibt, ergibt sich letztendlich der Wert von $U' = 13$.

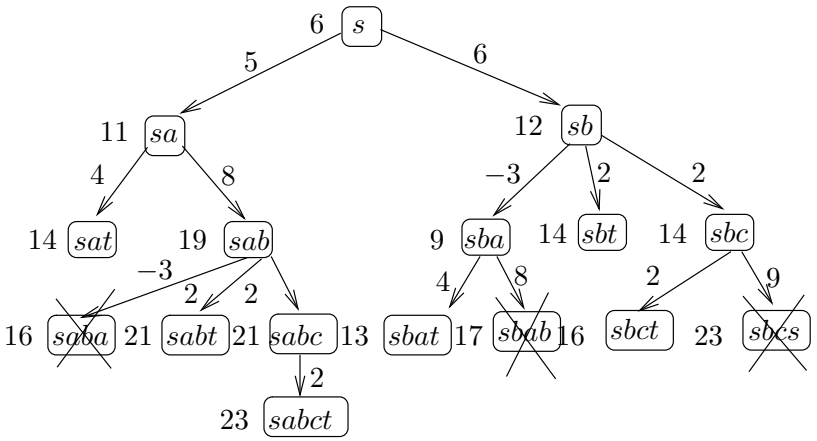


Abbildung 3.7: Die Expansion des Graphen aus Abbildung 3.2.

Die vierte Iteration wird $\langle s, a, b, t\rangle$ als optimale Lösung mit goalFound gleich true und dem Wert 13 ausgeben.

Satz 11 *Der Algorithmus IDA* mit Duplikatselimination ist für optimistisch bewertete Graphen zulässig.*

Beweis: Die Erzeugung von Pfaden von *IDA** mit Duplikatselimination unterscheidet sich von *IDA** nur durch die Abbruchbedingung von *HashDFS*, in der die weiteren von p_v ausgehenden Pfade nicht mehr untersucht werden.

In diesem Fall gibt es einen Pfad $p'_v \neq nil$ zu v in H mit $w(p'_v) < w(p_v)$. Angenommen, es gibt einen optimalen Zielpfad p_t mit $p_t = \langle p_v, q\rangle$, dann gibt es einen Pfad $p'_t = \langle p'_v, q\rangle$ mit

$$w(p'_t) = w(\langle p'_v, q\rangle) = w(p'_v) + w(q) < w(p_v) + w(q) = w(\langle p_v, q\rangle) = w(p_t).$$

Da $w(p_t) = C^*$ gilt, folgt $w(p'_t) < w(p_t) = C^*$, Widerspruch. □

Dieses Verfahren stellt einen speziellen *branch and bound* Ansatz dar. Während U eine global anwachsende untere Schranke für die Lösung des Gesamtproblems beschreibt, beinhaltet der gespeicherte Kostenwert eines jeden Knotens in der Hashtabelle eine obere Schranke, die beim Erreichen dieses Knotens nicht überschritten werden darf.

Damit nicht immer alle Pfade gleichen Gewichtes abermals untersucht werden, ist es naheliegend, die Abfrage $w(p'_v) \geq w(p_v)$ durch $w(p'_v) > w(p_v)$ zu ersetzen. Dadurch wird eine erneute Expansion eines Knotens aber unmöglich. Die Lösung, zumindest einen Generierungspfad zu einem Knoten zu erlauben, sieht wie folgt aus: Falls $w(p'_v)$ gleich $w(p_v)$ gilt, so fahre nur dann im Algorithmus fort, wenn auch p'_v gleich p_v ist, d.h. der Knoten wird auch über den durch den mit p'_v festgelegten, bis dato kürzesten Weg, betreten. Im obigen Beweis würde ein optimaler p_v enthaltender Zielpfad p_t automatisch einen optimalen p'_v enthaltenden Zielpfad p'_t implizieren. Damit geht durch die Duplikatselimination kein optimaler Zielpfad *verloren*.

In *HSF-Light* (vergl. Anhang B) haben wir *IDA** mit *HashDFS* implementiert. Zur Lösung des 15-Puzzles aus dem Anhang A (vergl. auch Tabelle 2.1)

werden bei einer Hashtabelle mit einer Millionen Einträgen in der letzten Iteration 13774606 Knoten (in ca. 1.5 Minuten) expandiert. Ohne Vorgängerelimination steigt die Anzahl der Expansionen nach der Erschöpfung der Hashtabelle in der letzten Iteration allerdings auf 228654367 Knoten an (Laufzeit auf einer Sun Ultra Workstation ca. 20 Minuten).

Reinefeld und Marsland geben eine alternative Implementation von *IDA** mit einer Hash- bzw. *Transpostionstabelle* an (Reinefeld & Marsland 1994). An jedem Knoten u des Suchbaumes wird in einem *postorder* Durchlauf die minimale Distanz zu allen Horizontknoten v in dem von u aufgespannten Teilbaum vermerkt. Dies ist ein Mindestwert, der für eine erneute Expansion überschritten werden muß. Dabei heben die Autoren außerdem den Vorteil einer durch die Nachfolgergewichte festgelegten Durchlaufordnung hervor.

Da wir nicht erwarten können, den gesamten Suchraum abzuspeichern, liegt es nahe, manche Knoten in der Hashtabelle wieder zu entfernen und bei Bedarf notfalls neu zu erzeugen. Man spricht in diesem Fall von *Caching* oder *Memorizing*. Dabei ist ein *Cache* (Stangier 1996) ein Array fester Länge l. Die Elemente des Arrays sind *Schubladen*. Jede Schublade kann k Einträge enthalten. Dabei wird k die *Tiefe* des Caches genannt. Die *Cachegröße* ist $m = kl$. Die Einträge werden mit einer gewöhnlichen Hashfunktion für offenes Hashing berechnet. Der Cache bietet, wie die Hashtabelle, die Operationen *Insert*, *Delete* und *Search* an. Die Einträge werden nach der LIFO-Strategie (engl. last in first out) verwaltet, wobei beim Überschreiten der Obergrenze k die Elemente zyklisch überschrieben werden. Ein *Lesefehler* entsteht, wenn bei einer erneuten Anfrage der gespeicherte Wert nicht mehr gefunden werden kann. Repräsentiert ein Eintrag im Cache einen Zeittakt, so ist die *Cachelebensdauer* eine Zufallsvariable X, für die Zahl der Einträge, die von dem Eintrag bis zum Löschen eines bestimmten Elementes stattfinden.

Satz 12 *(Cache) Der Erwartungswert $E(X)$ für die Cachelebensdauer X bei l Schubladen der Tiefe k beträgt m.*

Beweis (Stangier 1996) Sei $p = 1/l$ die Wahrscheinlichkeit, daß ein ausgewähltes Element in eine Schublade fällt (Erfolg) und $q = (l-1)/l$ die Gegenwahrscheinlichkeit (Mißerfolg). Dann ist die Zufallsvariable X für die Anzahl der Zuweisungen bis zum k-ten Erfolg negativ binomialverteilt, d.h.

$$P(X = r) = \binom{r-1}{k-1} p^k q^{r-k}$$

für $r \in \{k, k+1, \ldots\}$. Damit ist $E(X) = k/p = kl$ □.

3.4.2 Memory Extended *IDA**

*MEIDA** (Eckerle & Schuierer 1994) besitzt eine zusätzliche $O(m)$ große Datenstruktur *Deque* (engl. double ended priority queue), also eine Vorrangwarteschlange mit den Operationen *Initialize*, *DeleteMax*, *DeleteMin*, *ExtractMax* und *Insert*. Dabei fügt *Insert* ein Element gemäß der vorgegebenen Ordnung ein und führt bei einem Speicherüberlauf die Operation *DeleteMax* aus. Die Hauptprozedur (siehe Programm 3.9) besteht aus zwei einander sich abwechselnden Phasen, die in den Programmen 3.10 (*CollectHorizon*) und 3.11 (*ExpandHorizon*) dargestellt werden.

Programm 3.9 *MEIDA**: Grundgerüst eines effizienten speicherplatzbeschränkten Algorithmus. Eingabe: Speicherbegrenzung m, Problemgraph G und Startknoten s. Ausgabe: Optimaler Lösungweg von s zu Zielknoten in G.

```
Initialize(Deque,m)                          {initialisiere Datenstruktur}
U ← h(s); U' ← ∞; goalFound ← true             {initialisiere Variablen}
while (U ≠ ∞)                                {bis ein Ziel gefunden ist}
  U ← U'; U' ← ∞                             {aktualisiere Schwellwerte}
  CollectHorizon(⟨s⟩)          {sammle die m kleinsten Horizontknoten}
  if (goalFound) return sol                {Abbruch, wenn Lösung gefunden}
  ExpandHorizon                                      {vergrößere Horizont}
  if (goalFound) return sol                {Abbruch, wenn Lösung gefunden}
```

Programm 3.10 *CollectHorizon*: Die Horizontknoten mit den kleinsten Werten werden eingesammelt. Eingabe: Generierungspfad p_u. Ausgabe: Mit m Knoten gefüllte Datenstruktur *Deque*.

```
Γ(u) ← Expand(u)                          {berechne die Nachfolgermenge}
for all v in Γ(u)            {betrachte die Nachfolger hintereinander}
  p_v ← ⟨p_u, v⟩                          {verlängere Generierungspfad}
  if (goal(v) and w(p_v) ≤ U')                              {Ziel erreicht}
    goalFound ← true; sol ← (p_v)                {bilde Lösungspfad}
  else if (w(p_v) ≤ U)                {Schwellwert nicht überschritten}
    CollectHorizon(p_v)                                  {rekursiver Aufruf}
    else Insert(Deque, p_v, w(p_v))            {Einfügen (mit Überlauf)}
```

Dabei bestimmt *CollectHorizon* die Pfade p_u mit $w_{\max}(p_u)$ größer als U und trägt sie in die Datenstruktur *Deque* ein.

In der Prozedur *ExpandHorizon* werden die in der Struktur *Deque* gespeicherten Knoten expandiert. Vielfach sind die $w_{\max}$ Werte der erzeugten Nachfolgerpfade p_v von p_u kleiner als das Maximum der in der *Deque* gespeicherten Pfade. In diesem Fall wird p_v in der *Deque* mit der Operation *Insert* eingefügt. Damit reichen wir tief in den Suchraum hinein.

Eckerle und Schuierer zeigen: Der Algorithmus *MEIDA** ist zulässig und expandiert auf Bäumen $O((N')^2/m)$ viele Knoten. Im folgenden werden wir eine effiziente Realisation der Datenstruktur *Deque* untersuchen.

Ein *Weakheap* (Edelkamp 1996) ist eine Datenstruktur, die durch die Relaxierung der Heapbedingung entsteht. Dabei ist ein Heap ein Array a, in dem für jede Position i die Bedingungen $a_i < a_{2i}$ und $a_i < a_{2i+1}$ gelten. Auch der Weakheap kann als ein binärer Baum aufgefaßt werden und durch zwei Arrays a und r beschrieben wird. In a_i ist der Schlüssel des i-ten Elementes abgelegt und r_i ist ein einzelnes (Reverse-)Bit, das angibt, ob der zu a_i assoziierte Teilbaum rotiert ist oder nicht. In einem Weakheap gelten die folgenden

Programm 3.11 *ExpandHorizon*: Mit Hilfe einer doppelseitigen Vorrangwarteschlange werden die Knoten des Horizonts nacheinander expandiert. Eingabe: Datenstruktur *Deque*. Ausgabe: Lösung *sol* oder verbesserter Schwellwert U'

```
while (Deque≠ ∅)                          {solange noch Knoten vorhanden sind}
  p_u ← DeleteMin(Deque)                  {entnehme aussichtsreichsten Knoten}
  if (goal(u)) goalFound ← true; sol ← p_u; return sol     {Ziel erreicht}
  Γ(u) ← Expand(u)                        {erzeuge die Nachfolgermenge}
  for all v in Γ(u)                       {betrachte die Nachfolger}
    p_v ← ⟨p_u, v⟩                        {erweitere den Generierungspfad}
    w(p_v) ← max{w(p_v), w(p_u)}          {bestimme den ungünstigsten Pfad}
    p_w ← ExtractMax(Deque)               {bestimme längsten Pfad}
    if (w(p_v) ≤ w(p_w)) Insert(Deque,p_v, w(p_v))   {Einfügen mit Überlauf}
    U' ← w(p_u)                           {aktualisiere den Schwellwert}
```

Bedingungen:

1. Jeder Schlüssel im rechten Teilbaum eines Knotens ist größer als der Schlüssel an der Wurzel des Teilbaumes selbst.

2. Die Wurzel hat kein linkes Kind.

3. Blätter finden sich nur auf den letzten beiden Schichten des Baumes.

In der Arrayrepräsentation definieren wir das linke Kind eines Indexes i als $2i+1-r_i$ und das rechte Kind als $2i+r_i$. Durch das Negieren der Boole'schen Werte r_i ändern wir die Adressierung des rechten und linken Kindes. Der durch i beschriebene Teilbaum wird demnach rotiert.

Weakheaps erweisen sich sowohl in der Theorie als auch in der Praxis als sehr geeignete Datenstruktur im Bereich des sequentiellen Sortierens. Die Schlüsselvergleichsanzahl im ungünstigsten Fall ist durch $n \log n + 0.1n$ beschränkt (Dutton 1992; 1993). Empirische Studien von Edelkamp belegen, daß der mittlere Fall bei $n \log n + d(n)n$ mit $d(n)$ aus dem Intervall $[-0.47, -0.42]$ liegt. Die bekannte untere Schranke für sequentielles Sortieren für worst und average case liegen bekanntlich bei $\lfloor \log n! \rfloor$ bzw. bei $\lfloor \log n! \rfloor - 1$, also ungefähr bei $n \log n - 1.44n$. Sei $k = \lceil \log n \rceil$. Dann zeigt Edelkamp auf, daß die *exakte* Anzahl an Vergleichen in dem auf Weakheaps basierenden Sortieransatz im besten Fall bei $nk - 2^k + 1$ (Eingabe in aufsteigender Sortierung) und im ungünstigsten Fall bei $nk - 2^k + n - k$ (Eingabe in aufsteigender Sortierung mit Ausnahme der letzten beiden Elemente) Schlüsselvergleichen liegt.

Zum Vergleich liegt die Vergleichsanzahl in bottom-up Heapsort (Wegener 1993b) bei $1.5n \log n + \Theta(n)$ im ungünstigsten Fall. Die bottom-up Heapsort Variante Mdr Heapsort von McDiarmond and Reed (McDiarmid & Reed 1989) benötigt maximal $n \log n + 1.1n$ Vergleiche (Wegener 1992). Ein vor allem in der Praxis entscheidender Vorteil beim Sortieren mit Weakheaps ergibt sich daraus, daß der Aufbau der Struktur aus einer ungeordneten Menge optimal in $n - 1$ Vergleichen möglich ist.

Die Nachfolger eines Knotens auf dem linken Schenkel des unterliegenden Teilbaumes werden *Enkel* genannt. Ein in einen Weakheap einzufügendes Element v wird hinter dem letzten vergebenen Arrayindex x in a gelegt, und in einer jeweils zu den Großeltern (engl. *grand parents* (gp)) gerichteten Aufwärtsbewegung wird die Weakheapstruktur wieder hergestellt (siehe Programm 3.12). Die dabei maximal beschrittene Pfadlänge ist im Mittel durch die Hälfte der Tiefe des Weakheap gegeben und durch $\lfloor \log n \rfloor$ beschränkt.

Programm 3.12 *Insert*: Das Einfügen eines Elementes in einen Weakheap. Eingabe: Einzufügendes Element v.

$x \leftarrow size$; $a_x \leftarrow v$; $r_x \leftarrow 0$ {plaziere einzufügendes Element}
while ($x \neq 0$ **and** $a_{gp(x)} > a_x$) {solange noch Aufwärtstrieb vorhanden}
  $Swap(gp(x), x)$; $r_x = 1 - r_x$ {tausche Elemente und Teilbäume}
  $x \leftarrow gp(x)$ {gehe einen Schritt hinauf}
$size \leftarrow size + 1$ {erhöhe die Anzahl der Elemente}

Die Transformation eines Weakheaps in sein duales Komplement ist in optimalen $\lfloor (n-1)/2 \rfloor$ Schlüsselvergleichen möglich (siehe Programm 3.13).

Programm 3.13 *Transform*: Der Aufbau einer inversen Vorrangwarteschlange. Ein- und Ausgabe: Weakheap Datenstruktur.

for all i **in** $\{size - 1, \ldots, \lfloor (size-1)/2 \rfloor + 1\}$ {gesamte untere Schicht}
  $Swap(gp(i), i)$ {tausche alle Elemente mit ihren Großeltern}
for all i **in** $\{\lfloor (size-1)/2 \rfloor, \ldots, 1\}$ {für alle übrigen Elemente}
  $Merge(gp(i), i)$ {baue Weakheap wie bekannt auf}

Die Operation *Merge* wird mit den Indizes i und j aufgerufen und vergleicht die Schlüssel a_i und a_j miteinander. Falls a_i sich als kleiner erweist, werden die Arrayinhalte an den Stellen i und j vertauscht und das Reversebit gekippt.

Sei M ein abwärts und M' aufwärts sortierter Weakheap der Elemente a_1 bis a_n. Zur Unterscheidung werden wir mit a_i das i-te Arrayelement in M und mit a'_j das j-te Arrayelement in M' bezeichnen. Im folgenden werden wir zwei (als Arrays zu repräsentierende) Bijektionen ϕ und ϕ' auf der Menge der Indizes $\{1, \ldots, n\}$ verwalten, so daß $a_i = a'_{\phi(i)}$ und $a'_i = a_{\phi'(i)}$ für alle i aus $\{1, \ldots, n\}$ gilt. Die Bedingungen werden als Invarianz *(I)* bezeichnet.

Eine entscheidende Beobachtung ist, daß jede Datenstrukturoperation auf Weakheaps in mehrere *Merge* und damit in mehrere Vergleichs- und Vertauschungsschritte (engl. compare and exchange) zerlegt werden kann. Sei $Swap_M$ für die Indizes j und k durch die folgenden drei Operationen festgelegt: Erstens vertausche a_j mit a_k und $\phi(j)$ mit $\phi(k)$, zweitens setze $\phi'(\phi(j))$ auf j und drittens setze $\phi'(\phi(k))$ auf k.

Satz 13 *Die Operation $Swap_M(j, k)$ respektiert die Invarianzbedingung (I) und*

benötigt konstante Zeit.

Beweis: Wir gehen davon aus, daß *(I)* vor dem Aufruf $Swap_M(j,k)$ gilt. Der erste Schritt vertauscht Schlüsselinhalte a_j und a_k miteinander. Damit gilt *(I)* weiterhin für alle Positionen außer j und k. Der zweite Schritt eröffnet sowohl $a_j = a'_{\phi(j)}$ als auch $a_k = a'_{\phi(k)}$. Letztendlich werden im dritten Schritt die Bedingungen $a'_j = a_{\phi'(j)}$ und $a'_k = a_{\phi'(k)}$ wiederhergestellt, so daß die Invarianz *(I)* gilt. Insgesamt werden nur vier Zuweisungen durchgeführt. □

Ein analoges Resultat läßt sich für die Operation $Swap_{M'}$ herleiten. Dadurch lassen sich die beiden Bijektionen ϕ und ϕ' von M nach M' bzw. zurück ohne großen Mehraufwand verwalten, und wir gewinnen somit eine effiziente Vorrangwarteschlange in beide Richtungen.

3.4.3 Partielle Suche

Alle bis hierher aufgeführten Algorithmen sind zulässig, d.h. sie finden eine optimale Lösung, wenn sie existiert. Wir werden im folgenden andeuten, wie sich in schweren Problemstellungen, wie der Protokollvalidation (Holzmann 1991), (gute) Lösungen zumindest mit einer hohen Wahrscheinlichkeit auffinden lassen.

Die grundlegende Idee des *Supertrace*-Verfahrens (Holzmann 1987; 1988) ist einfach: Anstatt den gesamten Zustandsvektor in einer Hashtabelle abzuspeichern, nutzen wir nur ein einzelnes Bit pro Adresse (engl. *bitstate hashing*). Offensichtlich wird der Speicheraufwand auf einen Bruchteil reduziert, was wiederum eine wesentliche Vergrößerung der Tabelle ermöglicht. Das Bit gibt an, ob der Zustand mit der gegebenen Hashadresse schon besucht wurde oder nicht. Ist beim Einfügen ein Bit schon gesetzt worden, so wird keine Veränderung vorgenommen. Durch den dadurch verbundenen Informationsverlust ist das Finden gespeicherter Zustände nicht mehr fehlerfrei möglich, da mehrere Zustände ununterscheidbar auf dieselbe Hashadresse abgebildet werden. Folglich kommt es zu einer übermäßigen Beschneidung des Suchraumes, in der der (optimale) Zielpfad unter Umständen übersehen wird.

Die Hoffnung ist, durch die wiederholte Anwendung des Verfahrens mit anderen Hashfunktionen eine gute Abdeckung des gesamten Suchraumes zu erzielen (*sequential hashing*). Dabei bietet es sich insbesondere an, die Funktionen aus einer zufällig erzeugten Menge H von Hashfunktionen zu ziehen. Sei die Anzahl der Hashadressen durch m gegeben. Führt nur der m-te Teil aller Funktionen in H für je zwei Schlüssel zu einer Adreßkollision, so nennen wir H universell. Diese Eigenschaft wird z.B. für die Klasse

$$H = \{((ax + b) \bmod p) \bmod m \mid 1 \leq a < p, 0 \leq b < p\}$$

erfüllt (Ottmann & Widmayer 1993), wobei die Größe der Schlüsselmenge p eine Primzahl sein soll. Somit erhalten wir bei zufälliger Wahl von a und b immer eine *gute* Hashfunktion.

Eckerle und Lais zeigen, daß die Durchlaufordnung im Suchbaum einen erheblichen Einfluß auf den Gesamtüberdeckung hat (Eckerle & Lais 1998). Sie schlagen eine Randomisierung der Nachfolgerauswahl vor.

Supertrace eignet sich insbesondere für den Bereich der Protokollvalidation, da die Zustände sehr groß und stark unstrukturiert sind. Die Komprimie-

rung der Zustände, wie wir sie in Kapitel 4 beschreiben, gibt dennoch auch hier den Forschungstrend vor (Meinel & Stangier 1997; Holzmann & Puri 1998).

3.4.4 Suche durch Planen

In dem Gebiet der Künstlichen Intelligenz gibt es zwei Forschungszweige, die eine starke Verwandschaft aufweisen: Die *Heuristische Suche* und das *Planen* (Russell & Norvig 1995). Es ist in diesem Rahmen nicht möglich, detailliert auf die verschiedenen Planungsverfahren einzugehen, doch werden wir der Vollständigkeit halber einige Querverbindungen aufzeigen und die Unterschiede beider Ansätze charakterisieren.

Einfache Planungssysteme beruhen auf der *STRIPS*-Beschreibungssprache (Stanford Research Institute Planning System, (Fikes & Nilsson 1971)), in der sich jeder Operator in drei Bestandteile zerlegen läßt: Der Menge von *Vorbedingungen* und den zwei Mengen von *Konsequenzen*: Die zusätzlich geltenden und die nicht mehr geltenden Prädikate. Betrachten wir als Beispiel das Sokoban-Puzzle. Hier ergeben sich für einen einfachen Zug des Männchens nach unten die folgenden Regeln:

1. $Man(x, y) \wedge Free(x, y + 1) \rightarrow$
   $Man(x, y + 1) \wedge \neg\, Man(x, y)$

2. $Man(x, y) \wedge Ball(x, y + 1) \wedge Free(x, y + 2) \rightarrow$
   $Man(x, y + 1) \wedge Ball(x, y + 2) \wedge \neg\, Man(x, y) \wedge \neg\, Ball(x, y + 1)$

Dabei sind *Man*, *Ball* und *Free* zweistellige Prädikate, die dann wahr werden, wenn das Männchen, ein Ball oder nichts sich auf dem durch die Koordinaten beschriebenen Kästchen befindet. Ein Planungsverfahren versucht nun, durch stetes Einfügen von Operatoren in einen gerichteten azyklischen Graphen vom Ausgangszustand in den Zielzustand zu gelangen. Dabei werden (wie in der Suche) entweder vom Startzustand ausgehend die Operatoren ausgewählt, deren Vorbedingungen erfüllt sind (progressive Planung), oder rückwärtig geschaut, welche Bedingungen erfüllt sind und welche Vorbedingung als nächstes erfüllt werden müssen (regressive Planung). Ein *vollständiger*, *konsistenter* Plan ist dann erreicht, wenn jede Vorbedingung der Operatoren im Plan durch andere Operatoren gesichert ist und auf alternativen Wegen im azyklischen Graphen keine Widersprüche auftreten, d.h. diese unabhängig voneinander durchzuführen sind.

Auf die Frage hin, wodurch sich Planungssysteme und Suchverfahren unterscheiden, finden wir folgende Antworten:

Variablenbindung In Planungssystemen wird häufig darauf verzichtet, stark typisierte Variablen zu verwenden. Einen Vektor aus Zustandsattributen gibt es somit nicht. Die Vorgehensweise zur Variablenbindung basiert vielmehr auf die der Logikprogrammierung bekannten Unifikation.

Optimalität In Planungssystemen konzentriert man sich, vergleichbar mit der Realzeitsuche, hauptsächlich darauf, überhaupt eine Lösung zum gestellten Problem zu finden.

Dekomposition Das Planen von Handlungsschritten beinhaltet die Aufteilung der Gesamtaufgabe in möglichst voneinander unabhängige Teilaufgaben, bis letztendlich ein einfacher Weg vom Start- zum Zielzustand

gefunden wird. Diese rekursive Problemlösung ist im *allgemeinen Problemlöse-Ansatz* (engl. *general problem solver*) von Newell und Simon verwurzelt (Newell & Simon 1963).

Parallelität Viele Aktionen (wie das Drehen gegenüberliegender Würfelseiten) können unabhängig von einer vorgegebenen Reihenfolge durchgeführt werden. Demnach zielen sogenannte *partial order* Planungssysteme darauf, diese Parallelität in einer kompakten Graphbeschreibung zu nutzen.

Allgemeinheit Planungssysteme sollen universell einsetzbar sein, so daß die Beschreibungssprache nicht auf die effizient zu gestaltenden Eigenheiten der Domäne, wie z.B. untere Schranken, eingeht.

Konkatenation Seien Vorbedingungs-, Hinzufüge- und Löschliste für zwei aufeinanderfolgende Züge m_1 und m_2 mit $P(m_i)$, $A(m_i)$ und $D(m_i)$ für i aus $\{0,1\}$ bezeichnet, so ergibt sich für die konkatenierte Sequenz $m = m_1m_2$: $P(m) = P(m_1) \cup (P(m_2) - A(m_1))$, $A(m) = A(m_2) \cup (A(m_1) - D(m_2))$ und $D(m) = D(m_2) \cup (D(m_1) - A(m_2))$.

Jedes Planungsproblem kann unmittelbar als Suchproblem aufgesehen werden, indem jeder einzelne (partielle) Plan als Suchzustand aufgefaßt wird (McAllester & Rosenblitt 1996). Alternativ läßt sich eine Probleminstanz direkt durch Variablenbelegungen erzeugen. Die zu einem Zustand möglichen Übergänge sind durch die (*STRIPS*-) Operatoren festgelegt.

Auf der anderen Seite kann jedes Suchproblem als Planungsproblem beschrieben werden, da sich alle Grundelemente, wie Startzustand, Erzeugung von Nachfolgeoperatoren und Zielbedingungsabfrage problemlos spezifizieren lassen.

Reine auf die Dekomposition aufbauende Devide-and-Conquer Planungssysteme wie *PRODIGY* (Carbonell et al. 1992) und *UCPOP* (Penberty & Weld 1992) haben Schwierigkeiten mit der Lösung selbst einfacher Zustandsräume (Bonet, Loerincs & Geffner 1997). Deshalb werden allgemeine Heuristiken für eine Planfindung vorgeschlagen (McDermott 1996).

Erfolgreiche *STRIPS* Planungssysteme wie *GRAPHPLAN* (Blum & Furst 1995) verwenden zusätzlich zur Dekomposition explorative Suchstrategien, die direkt auf den Probleminstanzen agieren. Der Ansatz von Bonet, Loerincs und Geffner nutzt zur erfolgreichen Lösung von Planungsproblemen einen Realzeit-Suchansatz, den wir in Kapitel 9 beschreiben.

Kapitel 4

Zustandsraumsuche mit *OBDDs*

In diesem Kapitel erörtern wir, wie binäre Entscheidungsdiagramme (OBDDs) in der heuristischen Suche eingesetzt werden können. Die einzelnen Problemsituationen werden binär codiert und die charakteristischen Funktionen von mehreren Stellungen zusammengefaßt in einem OBDD darstellt. In dem Algorithmus BDDA werden der traditionelle A* Algorithmus und die auf OBDDs basierende Breitensuche zu einem neuen Mittelweg zwischen Zeit- und Speicheranforderungen zusammengefaßt. Die Komplexität von BDDA* wird untersucht und im (n^2-1)-Puzzle und im Sokobanspiel evaluiert.*

4.1 Entscheidungsdiagramme

Geordnete binäre Entscheidungsdiagramme (engl. ordered binary decision diagrams), kurz *OBDDs*, sind graphische Repräsentationen Boole'scher Funktinen. Sie wurden von Bryant (Bryant 1985) vorgestellt. Ein *OBDD* $G(f,\pi)$ der Funktion f zur Variablenordnung π ist ein azyklischer Graph mit einer Quelle und zwei Senken, die mit 0 und 1 beschriftet sind. Alle inneren Knoten haben zwei ausgehende Kanten (0 und 1) und sind mit den Boole'schen Variablen x_i der Funktion f bezeichnet. Für alle Kanten von einem mit x_i zu einem mit x_j beschrifteten Knoten gilt $\pi(i) < \pi(j)$, so daß auf jeden Pfad in G jede Variable maximal einmal getestet wird. Die Auswertung $f(a)$ im *OBDD* $G = (f,\pi)$ bei der Variablenbelegung a geschieht von der Wurzel beginnend durch stetes Verzweigen an den Variablenknoten bis eine Senke erreicht wird. Die Beschriftung der Senke entspricht dem gesuchten Funktionswert $f(a)$.

Es gibt zwei Reduktionsregeln für ein *OBDD*. Die Regel (R1) löscht diejenigen Knoten innerhalb des Baumes, deren Null- und Einsnachfolger identisch sind. Die Regel (R2) löscht einen von zwei Knoten mit gleichem Variablenindex und gleichem Nachfolgerpaar. Ein *OBDD* $G(f,\pi)$ ist isomorph zu dem minimalen *OBDD* $G^*(f,\pi)$, wenn keine der beiden Reduktionsregeln mehr angewendet werden kann. Das minimale *OBDD* ist eindeutig, da jeder Knoten eine unterschiedliche Subfunktion von f darstellt. Da reduzierte *OBDDs* eindeutig sind, ist der Erfüllbarkeitstest in konstanter Zeit durchführbar. Damit

heben sich diese Darstellungsformen deutlich von Boole'schen Formeln ab, für die das Erfüllbarkeitsproblem bekanntlicherweise *NP* schwer ist (Cook 1971). Einige weitere Operationen für *OBDDs* sind (Sieling 1994):

Erfüllbarkeitsgröße Eingabe: *OBDD* $G = (f, \pi)$. Ausgabe $|f^{-1}(1)|$ in $O(|G|)$ Zeit und Platz (Idee: topologische Sortierung des *OBDDs*, Einsinkenlassen der Zahl aller möglichen Belegungen an der Wurzel mit Aufsplittung und Aufaddition an den einzelnen Variablenknoten).

Synthese Eingabe: *OBDDs* G_f und G_g, Boolescher Operator $\otimes$. Ausgabe: *OBDD* für $f \otimes g$ in $O(|G_f||G_g|)$ Zeit und Platz (Idee: paralleler Lauf durch beide Eingangsgraphen und Kopplung der postorder Verarbeitung mit beiden Reduktionsregeln).

Ersetzung durch Konstante Eingabe: *OBDD* G_f, Variable x_i und Konstante c. Ausgabe: *OBDD* $f|_{x_i=c}$ in $O(|G_f|)$ Zeit und Platz (Idee: Festsetzung der $(1-c)$ Nachfolger auf die Nullsenke).

Negation Eingabe: *OBDD* G_f. Ausgabe: *OBDD* für $\neg f$ in $O(1)$ Zeit und Platz (Idee: Vertauschung von Null- und Einssenke).

Die Synthese ermöglicht den sukzessiven Aufbau eines *OBDDs* aus einer gegebenen Boole'schen Formel oder einem Schaltkreis.

Die Variablenordnung π hat einen starken Einfluß auf die Größe des *OBDD*, z.B. besitzt die Funktion $f = x_1x_2 \wedge x_3x_4 \wedge x_{2n-1}x_{2n}$ bei aufsteigenden Variablenindizes genau $2n+2$ Knoten und exponentielle Größe bezüglich der Anordnung $x_1, x_3, \ldots x_{2n-1}, x_2, x_4, \ldots, x_{2n}$. Das Finden einer optimalen Variablenordnung ist *NP*-schwer (Tani, Hamaguchi & Yajima 1993; Bollig & Wegener 1996) und nur für eine geringe Anzahl von Variablen durch einen exponentiellen Ansatz direkt lösbar (Friedman & Supowit 1990). Es gibt Funktionen wie $HWB(x) = x_{||x||}$, die in jeder Variablenanordnung exponentielle Größe haben (Bryant 1991) und der Anteil der Variablenordnungen, für die eine n-stellige Funktion reduzierte *OBDDs* exponentieller Größe hat, konvergiert für n gegen unendlich gegen eins (Wegener 1993a), aber für viele praktische Anwendungen besitzen *OBDDs* eine durchaus handhabbare Größe. Die Implementation eines sehr effizienten Pakets zur Verwaltung von *OBDDs* wird in Anhang **??** beschrieben.

4.2 Breitensuche mit *OBDDs*

OBDDs werden erfolgreich im Bereich der Hardwareverifikation (Bryant 1985), der Modellprüfung (McMillan 1993) und der Protokolvalidation (Meinel & Stangier 1997) eingesetzt. Eine der wichtigsten Aufgaben in diesen Forschungszweigen ist die Bestimmung der Menge aller erreichbaren Zustände.

4.2.1 Positionsbeschreibung

Wir werden zur Illustration ein sehr einfaches Beispiel eines Schiebepuzzles untersuchen (vergl. Abbildung 4.1). Dabei ist ein Spielbrett durch vier aneinandergereihte, aufsteigend numerierte Quadrate und einem darauf zu bewegenen Spielstein gegeben. In dem Startzustand befindet sich der Spielstein an Stelle null, in dem Zielzustand hingegen an der Stelle drei.

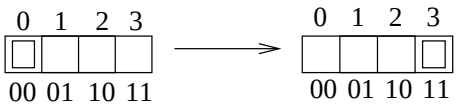


Abbildung 4.1: Ein einfaches Schiebepuzzle.

Offensichtlich sind nur drei Schritte zur Lösung des Problems notwendig. Die vier verschiedene Situationen lassen sich problemlos mit zwei Bits beschreiben. Die charakteristische Funktion einer Position ist definiert als der Minterm dieser binären Codierung. So beschreibt z.B. der Minterm $x_0 \wedge x_1$ die Zielposition (11). Die charakteristische Funktion von zwei und mehr (insbes. Ziel-) Positionen ist die Disjunktion ($\vee$-Verknüpfung) der charakteristischen Funktionen der Einzelstellungen.

4.2.2 Transitionsfunktion

Im nächsten Schritt beschreiben wir die durch die Spielregeln festgelegten Zustandsübergänge. Dazu bilden wir eine Boole'sche Transitionsfunktion T mit einer zu der Stellungsbeschreibung doppelten Anzahl an Variablen. Dann und nur dann, wenn x die charakteristische Funktion der gegebenen Situation ist und x' die charakteristische Funktion eines Nachfolgers von x, soll die Auswertung von $T(x, x')$ den Wert eins liefern. Damit ist T die Disjunktion aller Einzelregeln innerhalb des Problems. Im Beispiel finden wir die sechs möglichen Bewegungen (00) $\rightarrow$ (01), (01) $\rightarrow$ (00), (01) $\rightarrow$ (10), (10) $\rightarrow$ (01), (10) $\rightarrow$ (11) und (11) $\rightarrow$ (10). Das diese Regelmenge darstellende *OBDD* ist in Abbildung 4.2 dargestellt.

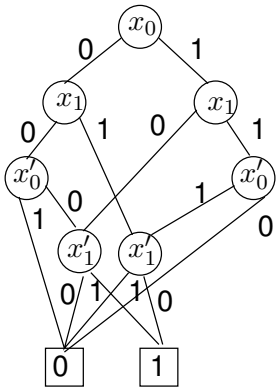


Abbildung 4.2: Das *OBDD* für die Transitionsfunktion.

Um die charakteristische Funktion aller von dem Startzustand s aus erreichbaren Potitionen zu beschreiben, wenden wir T auf die charakteristische Funktion von s an. Damit fragen wir nach allen Nachfolgern x', die die Funktion $T(s, x')$ erfüllen. Sei $\Phi(S)$ die charakteristische Funktion der Menge S und S^i die Menge der in i Schritten erreichbaren Zustände. Weiterhin sei S^0 mit $\{s\}$ initialisiert. Dann läßt sich $\Phi_{S^{i+1}}(x')$ durch folgende Boole'sche Formel rekursiv berechnen:

$$\Phi_{S^{i+1}}(x') = \exists x\ (T(x, x') \wedge \Phi_{S^i}(x)).$$

Dabei müssen wir in jedem Schritt eine Variablentransformation von x' nach x vornehmen, was sich durch eine Indexverschiebung innerhalb des die Funktion $\Phi_{S^{i+1}}(x')$ repräsentierenden *OBDDs* realisieren läßt.

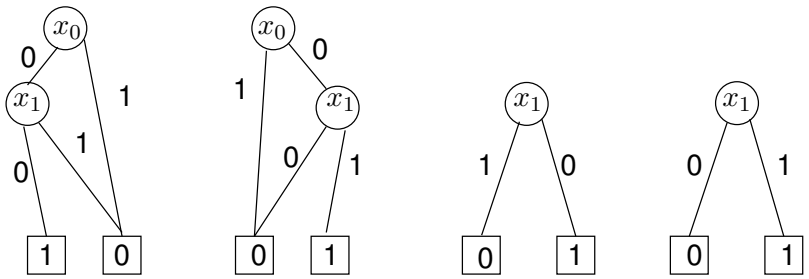


Abbildung 4.3: *OBDDs* für Φ_{S^0}, Φ_{S^1}, Φ_{S^2} und Φ_{S^3}.

4.2.3 Existenz-Quantifizierung

Die noch zu beantwortende Frage richtet sich an die effiziente Implementation des Existenzquantors. In der zweiwertigen Boole'schen Algebra gilt offensichtlich: $\exists x_i f(x) = f|_{x_i=0} \vee f|_{x_i=1}$. Damit ergeben sich für die gesamte Quantifizierung mit x in einem einfachen Ansatz $O(|x|)$ Syntheseoperationen und $O(|x|)$ Ersetzungen durch eine Konstante. Betrachten wir die Abbildung 4.2 genauer, so stellen wir fest, daß für die Berechnung des Quantors die Disjunktion aller Wurzelknoten im zweiten (unteren) Teil des *OBDDs* gebildet werden muß. Dies führt zu einer Einsparung an unnötigen Syntheseaufrufen. Beachtet man, daß vor der Quantifizierung noch eine Konjunktion mit Φ_{S^i} durchzuführen ist, so verringert sich die Anzahl potentieller Wurzelknoten noch weiter.

Wir haben in diesem Fall ausgenutzt, daß die Variablenordnung in T durch die Aneinanderreihung von x und x' Variablen gegeben ist. Da die Spielregeln im allgemeinen nur lokale Veränderungen hervorrufen, ist es vorteilhafter, die Variablen alternierend abzufragen, so daß x'_i unmittelbar auf x_i folgt. Die effiziente Realisierung des Existenzquantors (siehe Programm **??**) ergibt sich in diesem Fall durch eine Mischung aus des Synthesealgorithmus für die Konjunktion, dem Löschen von x_i Variablen und der *vee*-Verbindung der sich ergebenen Teilbäume. Die Prozedur nutzt zur effizienten Berechnung zwei Hashtabellen: Die Transpositionstabelle TT zur Wiedererkennung schon besuchter Knoten und die Eindeutigkeitstabelle UT (engl. unique table) zur Anwendung der Reduktionsregel (R2). Der Pseudocode unterscheidet sich nur in der Zeile *if* ($xindex(m)$) *return* $Apply(s_0, s_1, \vee)$ von einer normalen $\wedge$-Synthese.

4.3 Heuristische Suche mit *OBDDs*

In der heuristischen Suche druchforsten wir den unterliegenden Problemgraph. In vielen Fällen ist dieser Graph uniform gewichtet (das Gewicht der Kanten beträgt jeweils eins). Mit jedem Zustand im Zustandsraum assoziieren wir eine effizient berechenbare untere Schranke. Der Unterschied der Bestensuche zur

Programm 4.1 ExistAnd: Zwei *OBDDs* $T(x, x')$ und $\Phi(x)$ werden rekursiv mit $\wedge$ verbunden und das Resultat mit $\exists x$ quantifiziert. Eingabe: *OBDDs* G_T und G_Φ. Ausgabe: *OBDD* $G_{\exists x(T(x,x')\wedge\Phi(x))}$.

if (TT.Search$(top(G_T), top(G_\Phi))$ {Knoten erneut besucht}
  return TT.Search$(top(G_T), top(G_\Phi)$ {gebe gefundenen Knoten zurück}
if (sink$_0(G_T)$ **or** sink$_0(G_\Phi)$) **return** sink$_0$ {gebe null Senke zurück}
if (sink$_1(G_T)$) **return** sink$_1$ {gebe eins Senke zurück}
if $(\pi(top(G_T)) < \pi(top(G_\Phi)))$ {Variablenindex in G_T kleiner}
  $s_0 \leftarrow$ ExistAnd(left$(G_T), G_\Phi)$ {nutze 0-Kante in G_T}
  $s_1 \leftarrow$ ExistAnd(right$(G_T), G_\Phi)$ {nutze 1-Kante in G_T}
  else if $(\pi(top(G_T)) > \pi(top(G_\Phi)))$ {Variablenindex in G_Φ kleiner}
    $s_0 \leftarrow$ ExistAnd$(G_T,$ left$(G_\Phi))$ {nutze 0-Kante in G_Φ}
    $s_1 \leftarrow$ ExistAnd$(G_T,$ right$(G_\Phi))$ {nutze 1-Kante in G_Φ}
  else {Variablenindex in G_T und G_Φ gleich}
    $s_0 \leftarrow$ ExistAnd(left$(G_T),$ left$(G_\Phi))$ {nutze 0-Kante in G_T und G_Φ}
    $s_1 \leftarrow$ ExistAnd(right$(G_T),$ right$(G_\Phi))$ {nutze 1-Kante in G_T und G_Φ}
$m \leftarrow \min\{\pi(top(G_T)), \pi(top(G_\Phi))\}$ {minimaler Index von G_T und G_Φ}
if $(s_0 = s_1)$ **return** s_0 {Reduktionsregel (R1)}
if (UT.Search(s_0, s_1, m)) **return** UT.Search(s_0, s_1, m) {Reduktionsregel (R2)}
if (xindex(m)) **return** Apply$(s_0, s_1, \vee)$ {berechne *vee*-Synthese}
return new$(s0, s1, m)$ {neuer Knoten in G, Eintrag in TT und UT}

Breitensuche ist eine kombinierte Gewichtung der Knoten aus Generierungspfadlänge und heuristischen Schätzwert.

4.3.1 Darstellung von Relationen

In A^* (vergl. Kapitel 3) wird eine Vorrangwarteschlange Open genutzt, in der die Zustände gemäß eines ansteigenden f Wertes verwaltet werden, der sich aus der Summe des Generierungspfades g und der heuristischen Schätzwertes h ergibt. Initial beinhaltet die Vorrangwarteschlange nur den Startzustand. In jedem Schritt wird der Zustand mit dem kleinsten f Wert entnommen und expandiert und die Nachfolger entsprechend ihren neu ermittelten f Werten eingefügt, bis letztendlich der Zielzustand erreicht ist.

Für den entnommenen Zustand x haben wir $f(x) = g(x) + h(x)$ und für den Folgezustand x' ergibt sich die folgende zentrale Beziehung:

$$f(x') = g(x') + h(x') = g(x) + 1 + h(x') = f(x) + 1 - h(x) + h(x'). \quad (4.1)$$

Der Schätzer h kann als eine Relation von Tupeln der Form (*estimate*, *state*) angesehen werden, die genau dann den Wert wahr ergibt, wenn h(*state*) geich *estimate* ist. Wir nehmen an, daß h sich als *OBDD* für den gesamten Problemraum darstellen läßt - eine realistische Annahme, falls h nicht allzu kompliziert ist.

Die Vorrangwarteschlange *Open* kann ebenso als *OBDD* dargestellt werden, die auf eine Relation von Tupeln der Form (*merit*, *state*) aufzeigt (*merit*

ist der zu einem Zustand *state* assoziierte f Wert). Die Variablen sollten so angeordnet werden, daß die signifikanteste Variable in *merit* zuerst abgefragt wird und auch die anderen *merit* Variablen den *state* Variablen vorausgehen. Somit erhalten wir ein initial kleines *OBDD*, daß uns viele Iterationen des A^* Algorithmus simulieren läßt. Weiterhin finden wir in dieser Variablenanordnung ein intuitives Verständnis des *OBDDs* und der Verbindung der damit realisierten Vorrangwarteschlange.

4.3.2 Das *BDDA** Verfahren

Im folgenden beschreiben wir den Hauptalgorithmus *BDDA** (für A^* mit *OBDDs*), dessen Pseudocode in Programm 4.2 angeführt ist. Initial wird das *OBDD Open* mit der heuristischen Bewertung des Startzustandes gleichgesetzt.

Programm 4.2 Der A^* Algorithm mit *OBDDs*. Eingabe *OBDD* für die heuristische Bewertungsfunktion h eines in der Übergangsrelation T beschriebenen Zustandsraumproblems, charakteristische Funktionen Φ des Start und der in G zusammengefaßten Zielzustände. Ausgabe: Lösungslänge des kürzesten Weges von Start zum Ziel.

$\texttt{Open}(f,x) \leftarrow h(f,x) \wedge \Phi_{S^0}(x)$ {initialisiere die datenstruktur}
while ($\texttt{Open} \neq \emptyset$) {solange Datenstruktur nicht leer ist}
  $(f_{\min}, \texttt{Min}(x), \texttt{Open}'(f,x)) \leftarrow \texttt{goLeft}(\texttt{Open})$ {entnehme minimales Element}
  if $(\exists x\ (\texttt{Min}(x) \wedge \Phi_G(x))$ **return** $f_{\min}$ {Zielzustand erreicht}
  $\texttt{VarTrans}_{x,x'}(\texttt{Min}(x))$ {Tausche x mit x' Variablen}
  $\texttt{Open}''(f,x) \leftarrow \exists x'\ Min(x') \wedge T(x',x) \wedge$ {bilde Übergang}
    $\exists e'\ h(e',x') \wedge \exists e\ h(e,x) \wedge (f = f_{\min} + e - e' + 1)$ {und bewerte ihn}
  $\texttt{Open}(f,x) \leftarrow \texttt{Open}'(f,x) \vee \texttt{Open}''(f,x)$ {füge Nachfolger ein}

Die Funktion *goLeft* ermittelt den minimalen f Wert $f_{\min}$, das *OBDD Min* aller Zustände der Vorrangwarteschlange mit Wert $f_{\min}$ und das *OBDD Open*$'$ der verbleibenen Zustände. Um den minimalen f Wert zu ermitteln, gehen wir von der Wurzel in *Open* ausgehend immer links (d.h. zum Nullnachfolger), es sei denn dieser Weg führt auf eine Nullsenke. Dann gehen wir entsprechend rechts. Der erste aufgefundene einen Zustand beschreibene *OBDD* Knoten ist die Wurzel des *OBDDs Min*. Um *Min* zu extrahieren, muß letztendlich nur der auf Min verweisende Link auf die Nullsenke gebogen werden.

Sei Φ_G die charakteristische Funktion der Zielzustände G. Die Menge *Min* enthält einen Zielzustand, falls das *OBDD* $\texttt{Min} \wedge \Phi_G$ nicht der trivialen Nullfunktion gleicht. Falls kein Zielzustand gefunden wird, ermitteln wir mit der Übergangsfunktion T die Menge der Nachfolgerzustände *Open*$''$. Da nur zu diesem Zeitpunkt sowohl der Vorgängerzustand x als auch der Nachfolgerzustand x' verfügbar sind, wird gleichzeitig die Berechung der sich ergebenen f-Werte gemäß Gleichung 4.1 durchgeführt. Demnach sind die Zustände in *Open*$''$ schon gemäß ihren neuen Bewertungen adressiert.

Letztendlich werden die beiden *OBDDs Open*$'$ und *Open*$''$ für die nächste Iteration mit einer Disjunktion zu einem neuen *OBDD Open* verbunden.

4.3.3 Beispiel

Um zu sehen, wie der Algorithmus *BDDA** arbeitet, betrachten wir abermals das einfache Schiebepuzzlebeispiel aus Abbildung 4.1. Das *OBDD* für den Schätzer h ist in Abbildung 4.4 dargestellt, wobei wir den Wert auf null für die Zustände null und eins gesetzt haben und auf eins für die Zustände drei und vier.

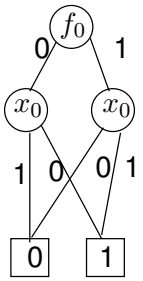


Abbildung 4.4: Das *OBDD* für die heuristische Function.

Da der minimale bwz. maximale f Wert eins bzw. vier ist, benötigen wir nur zwei Variablen f_0 und f_1 für dessen binäre Beschreibung (Subtrahiere eins von dem Binärwertes von f).

Nach dem Initialisierungsschritt ist die Vorrangwarteschlange mit dem Minterm $\overline{x}_0\overline{x}_1$ und der Bewertung eins gefüllt, da gerade dieses dem Schätzwert des Startzustandes entspricht. Es gibt nur einen Nachfolger der Anfangssituation, die durch den Minterm $\overline{x}_0 x_1$ beschrieben wird. auch dieser Zustand besitzt den Schätzwert eins und demnach einen f Wert von zwei.

Wenden wir nun T auf das resultierende *OBDD* an, so finden wir die charakteristische Funktion der Zustande zwei und drei, deren h Wert sich allerdings um eins unterscheidet. Demnach verbinden wir Minterm $x_0\overline{x}_1$ mit dem f Wert zwei und $\overline{x}_0\overline{x}_1$ mit drei. Der Status der Vorrangwarteschlange zu diesem Zeitpunkt ist zur Linken in der Abbildung 4.5 dargestellt.

In der nächsten Iteration extrahieren wir $x_0\overline{x}_1$ mit Wert zwei und ermitteln die Nachfolgermenge, die in diesem Falle aus den Positionen eins und zwei besteht. Kombiniert man die gemeinsame charakteristische Funktion x_1 dieser Zustände mit dem *OBDD* h so teilen wir die Nachfolgermenge wieder in zwei Teile x_0x_1 und $\overline{x_0}x_1$. Das resultierende *OBDD* *Open* ist zur Rechten in der Abbildung 4.5 dargestellt.

Da nun *Min* einen nicht leeren Schnitt mit der Zielzustandsmenge hat, haben wir letztendlich die optimale Lösung der Länge drei gefunden.

4.3.4 Arithmetische Operationen

Eine offene Frage ist, wie wir arithmetische Operationen mit *OBDDs* verwirklichen können. Da die f Werte auf eine endlichen ganzzahligen Bereich beschränkt sind, können wir eine Boole'sche Funktion *add* mit den Parametern a, b und c beschreiben, die genau dann wahr ist, wenn c der Summe von a und b entspricht. Demnach müssen wir lediglich die Disjunktion der charakteristischen Funktionen aller dieser Tripel (a, b, c) bestimmen. Der *minus*

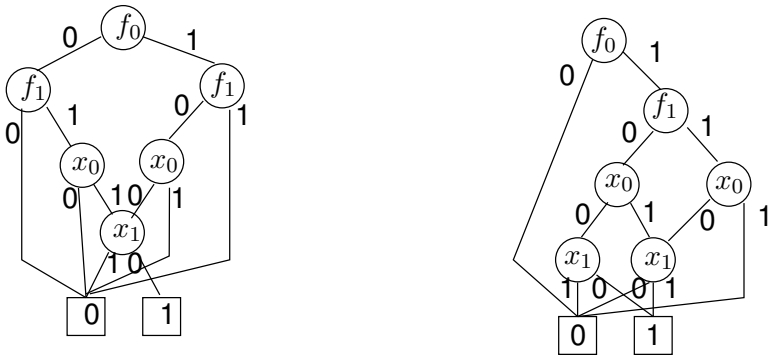


Abbildung 4.5: Die Vorrangwarteschlange `Open` nach zwei und nach drei Schritten.

Operator entsteht dann durch einen einfachen Variablentausch von a und c in *add*.

4.4 Experimente

Da es derzeit eine Menge verschiedener effizienter *OBDD* Pakete gibt, haben wir eine solche Implementation für unseren Algorithmus gewandelt. Wir wählten den μ-Kalkül Modellprüfer μcke (Biere 1997) und ersetzten die klassische Breitensuche durch durch *BDDA**. Da μcke spezielle Funktionen, wie *goLeft* nicht unterstützt mußten wir diese durch mehrere *OBDD* Operationen ersetzen, die im Vergleich zu einer grundlegend neuen Implementation zu einem Mehraufwand in den Berechnungen führen.

4.4.1 $(n^2 - 1)$ Puzzle

Unser erstes Beispiel ist das Acht-Puzzle, die 3 mal 3 Variante der bekannten Schiebepuzzle von Samuel Loyd (vergl. Kapitel 2).

Dabei experimentierten wir mit einer initialen Konfiguration, die in 21 Zügen lösbar ist. Zuerst starteten wir die Breitensuche, die nach exakt 21 Iterationen terminierte. Es wurden insgesamt 40.000 Zwischenkonfigurationen erreicht, deren Repräsentation insgesamt ca. 39.000 *OBDD* Knoten erforderte. Daraufhin wendeten wir den Algorithmus *BDDA** an, der nach 26 Iterationen die minimale Lösung fand. Dabei benutzten wir die bekannte Manhatten Distanz als untere Schranke. Das *OBDD Open* für die Vorrangwarteschlange besaß eine Maximalanzahl von 200 Stellungen die durch 1.200 *OBDD* Knoten dargestellt wurden.

Wir haben außerdem einige Instanzen des Fünfzehnpuzzles gelöst. Für ein Beispiel mit einer minimalen Lösungslänge von 45 Zügen benötigten wir 176 Iterationen mit einer maximalen *OBDD* Größe von 215.00 Knoten zur Darstellung von 136.000 Zuständen. Eine Breitensuche mit *OBDDs* scheiterte an den zur Verfügung stehenden Platzressourcen. Schon nach 19 Iterationen waren mehr als eine Millionen *OBDD* Knoten zur Darstellung von über 1.4 Millionen Zuständen erforderlich.

4.4.2 Sokoban

Unser zweites Beispiel ist Sokoban (vergl. Kapitel 2). Um Sokoban mit klassischen A^* oder IDA^* Algorithmen zu lösen, muß eine sehr verfeinerte Heuristik erstellt werden, die zu einem nicht unerheblichen Design-, Analyse- und Programmieraufwand führt, die sich wiederum in der Komplexität einer Knotenexpansion niederschlägt (siehe Kapitel 7 und Anhang **??**). Die Bestimmung der Minimalanzahl von Ballbewegungen ist aktuelles Forschungsinteresse (Junghanns & Schaeffer 1998a; 1998b). Die minimale Anzahl von Männchenbewegungen ist wesentlich schwieriger. In einer bidirektionalen Breitensuche mit *OBDDs* konnten nur sechs der 90 Benchmarkprobleme von xsokoban gelöst werden (Doppelhamer & Lehnert 1998).

Um eine minimale Lösung in der Suche mit *OBDDs* zu finden, ist eine effiziente Codierung essentiell. In dem von uns betrachteten ersten Level von Sokoban (vergl. Abbildung 2.9 auf Seite 20) gibt es 56 Felder, die vom Männchen erreicht werden können. Die Männchenposition kann demnach durch 6 Bits binär codiert werden. Für die Bälle sind 23 Positionen nicht erreichbar oder die Situation wird unlösbar. Demnach ist ein Bitvektor der Länge 33 ausreichend, um alle Ballpositionen zu beschreiben.

Wir haben eine Lösung selbst mit der Breitensuche gefunden, die jedoch sehr viel Speicher beansprucht. Der $BDDA^*$ Algorithmus ist mit einer sehr bescheidenen Heuristik aufgerufen worden, die allein die Anzahl derjenigen Kugeln zählt, die nicht auf einem Zielfeld liegen. Die Ergebnisse der beiden Ansätze werden in Tabelle 4.1 miteinander verglichen.

	Breitensuche	$BDDA^*$ Algorithmus
Anzahl der Iterationen	230	419
max. Zustandsanzahl in Open	61.000.000	4.300.000
max. *OBDD* Knotenanzahl	250.000	68.000
Zeitverbrauch	3h17min	6h12min

Tabelle 4.1: Vergleich der Resultate in Sokoban

Besonders hervorzuheben ist, daß selbst mit einer so schwachen Heuristik die Anzahl der zu expandierender Knoten im $BDDA^*$ Algorithmus mehr als zehn mal geinger ist als in der Breitensuche. Auch die *OBDD* Größe ist wesentlich kleiner. Dieses positiver Resultat trägt auf der anderen Seite eine etwas größere Zeitlast mit sich. In der den Größendimensionen der Zustandsraumsuche ist der Platz der wichtigerer Parameter als die Zeit, denn dieser ist durch die gegebene Hardware unverrückbar festgelegt. Die Anzahl der Zustände ist bis zu 250 mal größer als die Anzahl der sie repräsentierenden *OBDD* Knoten. Weiterhin werden im allgemeinen mehr Bits zur Beschreibung eines Zustandes als zur Beschreibung eines *OBDD* Knoten benötigt.

4.5 Analyse

Für verfeinerte Heuristiken ist die Anzahl der Iterationen im Vergleich zu einer Breitensuche erstaunlich hoch. Diese Eigenschaft, werden wir im folgenden genauer analysieren.

Sei f^* die optimale Lösungslänge für eine gegebenen Probleminstanz s und h eine untere Schranke. Wir nehmen an, daß für jeden Knoten u und für jeden Nachfolger v von u die Beziehung $h(u) \leq h(v)+1$ gilt. Mit anderen Worten: h ist konsistent (vergl. Kapitel 3). Zusammen mit Gleichung 4.1 sehen wir, daß f dadurch auf allen Generierungspfaden nicht abnimmt.

Für eine optimale Heuristik, d.h. eine Heuristik, die den kürzesten Weg schätzt, benötigen wir maximal $h(s) = f^*$ Iterationen in *BDDA**. Auf der anderen Seite, wenn die Heuristk mit der Null-Funktion übereinstimmt (Breitensuche), sind f^* Iterationen notwendig. Dies verleitet zur Hypothese, daß die Anzahl der Iterationen in *BDDA** immer gleich f^* ist. Unglücklicherweise ist dies nicht wahr. Betrachte Abbildung 4.6, in der die g Werte im Verhältnis zu den h Werten aufgetragen sind, so daß die Zustände mit gleicher Summe aus g und h Wert auf den Geraden $f = g + h$ liegen.

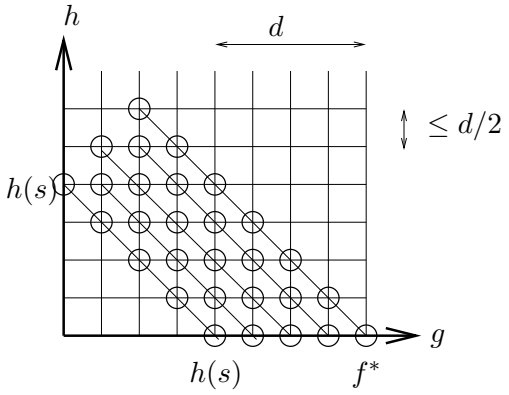


Abbildung 4.6: Die Anzahl der Iterationen in *BDDA**.

Die Anzahl der Nullen auf jeder Geraden ist eine obere Schranke für die Anzahl der Iterationen in *BDDA**, die mit diesem f Wert durchgeführt werden. Damit müssen wir also die Gesamtanzahl der Nullen zählen. Sei d die Distanz $f^* - h(s) + 1$ zwischen f^* und $h(s)$.

Unterhalb des Wertes $h(s)$ befinden sich maximal $dh(s)$ Knoten. Weiterhin hat *Dach* oberhalb von $h(s)$ nicht mehr als $1 + 3 + \ldots + 2(d/2) - 1$ Knoten. Da letztere Summe sich zu $d^2/4$ auswerten läßt, benötigen wir nicht mehr als $dh(s) + d^2/4 + d$ Schritte insgesamt. Da f^* nicht im vornherein bekannt ist mag eine gute obere Schranke zu der Problemlösung als worst-case Prediktor für den zu erwartenden Suchaufwand und der damit einzukalkulierenden Zeit dienen. Ist die Differenz der Schätzwerte zweier aufeinanderfolgender Zustandsknoten wie im Falle der Manhattandistanz immer vom Absolutwert eins, so ergibt sich eine Halbierung der Maximalanzahl an Iterationen.

4.6 Verwandte Arbeiten

Holzmann und Puri beschreiben die zu den *OBDDs* leicht erweiterte Struktur eines geschichteten Automatens (Holzmann & Puri 1998), in dem die Reduktionsregeln (R1) und (R2) nicht angewendet werden. In Abbildung 4.7 ist ein Beispiel angegeben. Der geschichtete Automat dient alleinig der komprimierten Darstellung des Zustandsraumes. Das Hauptergebnis ist, daß das Einfügen

eines einzelnen Zustandes in Zeit linar zur Codierungslänge des Zustandes möglich ist, indem man rückwärtig von der Einssenke den Zustand ermittelt, der den größtmöglichen Suffix abdeckt und dann von der Wurzel aus einen neuen Pfad generiert.

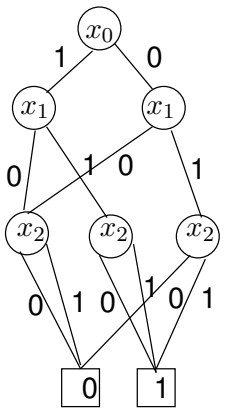


Abbildung 4.7: Ein minimierter geschichteter Automat für die Zustände 010, 011 und 111.

Aufbauend auf diese Datenstruktur läßt sich der A^*-Algorithmus direkt verwirklichen.

4.7 Ausblick

Der A^* Algorithmus wurde von den *IDA** Methoden in vielen Anwendungsbereichen verdrängt, da für die Zulässigkeit alle jemals expandierten Knoten im Hauptspeicher verwaltet werden müssen. Wir haben aufgezeigt, wie sich dieses Problem mit Hilfe von *OBDDs* bewältigen läßt, da sie eine sehr große Menge von Zuständen recht kompakt beschreiben.

Zustände, die sich durch einige wenige Züge ergeben, führen häufig nur zu lokale Veränderungen in der binären Codierung. Demnach nutzen wir die Beobachtung, daß in A^* die Zustände sich untereinander nicht um viele Züge unterscheiden und somit platzeffizient darstellbar sind. Die Zustände *scharen* sich entlang des optimalen Lösungsweges, was in einer simplen Zustandsaufzählung nicht gegeben ist.

Ein großer Vorteil von *BDDA** im Vergleich bestehender Ansätze ist, daß schon eine sehr einfache Heuristik ausreicht, um Lösungen für Probleme zu erhalten, die ohne zusätzliches problem-spezifisches Domänenwissen nicht zu erreichen sind.

Das Design nicht-trivialer Heuristiken erhöht unmittelbar die Iterationstiefe. Auf der anderen Seite wird der Speicherverbrauch erniedrigt. Ein Schätzwert nahe dem Optimum ist schwer zu finden und erfordert zeitintensive Berechnungen. Demnach sollte das Ziel eine untere Schranke sein, die zu keiner allzu großen Iterationstiefe führt, dennoch aber die Platzansprüche unter die zu Verfügung stehenden Ressourcen drückt.

In A^* wird jeder Zustand maximal einmal dargestellt, während unser Ansatz Duplikate nur mit gleichem f Wert erkennt. Die Verbindung dieser Me-

thode mit existierenden Suchbaumbeschneidungsansätzen (vergl. Kapitel 6) soll somit in Zukunft genauer studiert werden.

Ausgehend von den Erfolgen in der bidirektionalen Breitensuche ist eine weitere Forschungszielsetzung, die bidirektionale heuristische Suche mit *OBDDs* zu untersuchen.

Kapitel 5

Multi Suffix Bäume

Übergangssequenzen in den Zustandsraumproblemen können als Zeichenketten über einem endlichen Alphabet aufgefaßt werden. In diesem Kapitel wird eine Datenstruktur für die komfortable Verwaltung von Zeichenketten eingeführt und analysiert. Die angebotenen Operationen sind die eines Wörterbuches: Einfügen, Löschen und Suchen. Die Suche dient hierbei dem Aufspüren einer bzw. aller gespeicherten Zeichenketten, die einen Teil einer Anfrage darstellen. Die Früchte dieser allgemein gehaltenen Untersuchung werden in dem hierauf folgenden Kapitel geerntet.

Eine zeit- und platzeffizient gestaltete Repräsentation des Wörterbuchs für Zeichenketten wird mit dem Namen *dynamic dictionary matching problem*, kurz *DDMP* bezeichnet (Amir et al. 1994). Arbeiten, wie z.B. von (Ferragina & Luccio 1996) zum parallelen, von (Choi & Lam 1996) zum 2-dimensionalen, von (Amir & Farach 1991) zum adaptiven und von (Amir, Farach & Matias 1992) zum randomisierten Wörterbuch-Matching belegen, daß der Themenbereich aktuelles Forschungsinteresse ist.

Sei d die Summe der Längen, der im Wörterbuch gespeicherten Zeichenketten und $|m|$ die Länge der Zeichenkette m. Amir et al. (Amir et al. 1994) und Idury und Schäffer (Idury & Schaeffer 1994) stellen (sequentielle) Algorithmen für das *DDMP* vor. Im schlechtesten Fall benötigen die Verfahren eine Zeit von $O(|m| \log d)$ für das Einfügen und Löschen von m und $O((|x| + tocc) \log d)$ für die Suche nach allen Teilstrings von x ($tocc$ ist die Anzahl der Teilstrings in der Ausgabe). Amir et al. verbessern diese Resultate auf $O(|m| \log d / \log\log d)$ bzw. $O((|x| + tocc) \log d / \log\log d)$ (Amir et al. 1995). Die in diesem Kapitel erzielten Ergebnisse lassen sich wie folgt zusammenfassen:

1. Die Einfüge– und Löschoperationen können in zur Länge der eingefügten Strings proportionalen Zeit durchgeführt werden. Diese Zeit ist optimal, wenn man bedenkt, daß die Eingabe gelesen werden muß. Gegenüber Amir et al. bedeutet dies eine Ersparnis von einem Faktor $\log d / \log\log d$. Dabei muß fairerweise angemerkt werden, daß der Faktor auf einer von ihnen zusätzlich verwalteten recht komplizierten Struktur (u.a. wird die Verwaltung geordneter Listen nach Dietz und Sleator (Dietz & Sleator 1987) verwendet) beruht, die letztendlich die effiziente Suche *aller* Teilstrings einer Anfrage ermöglicht.

2. Die Suchzeit *eines* Teilstrings im Wörterbuch ist linear zur Größe der Anfrage (engl. query) und damit bestmöglich. Auch dieses Ergebnis erweist sich auf den ersten Blick als Gewinn eines Faktors der Größe $\log d / \log\log d$ gegenüber Amir et al.. Die effiziente Berechnung des längsten Teilstrings, der von einer Textstelle ausgeht, ist den Autoren indes bekannt und wird von ihnen als das vermeintlich einfache *dictionary representative problem* bezeichnet.

   Die Neuerung ist darin zu sehen, die Suche inkrementell zu organisieren. Sie kann somit als Folge von Zustandsübergängen bei Eingabe des jeweils folgenden Zeichens interpretiert werden. Da immer nur ein neu eingelesenes Zeichen verarbeitet wird, spricht man in diesem Zusammenhang auch von einer *on-line Suche*.

   Das Wörterbuch stellt somit die Operation eines endlichen Automaten zur Verfügung. Der Übergang von einem Automatzustand in einen nachfolgenden wird in amortisierter konstanter Zeit ermöglicht. Dabei bedeutet *amortisiert*, daß eine einzelne Operation zwar aufwendig sein kann, dieser Aufwand jedoch durch die sich ergebende Vereinfachung der folgenden Operationen ausgeglichen wird. Erst durch diesen inkrementellen Suchansatz kann das Wörterbuch effizient zur Vermeidung von Zustandswiederholungen im Suchprozeß genutzt werden.

3. Unter der Einschränkung, nur die Existenz eines Teilstrings zu bestimmen, kann das Wörterbuch *teilstringfrei* verwaltet werden. Dabei bedeutet *teilstringfrei*, daß kein gespeicherter String Teil eines anderen ist. Bevor ein String m in das Wörterbuch eingefügt wird, können alle *Superstrings* (das Gegenstück eines Teilstrings) von m gelöscht werden, denn das Entfernen dieser Strings betrifft nur redundante Information (Ein Superstring von m ist auch immer Superstring eines Teilstrings von m).

4. Es wird untersucht, wie *alle* Teilstrings einer Anfrage x in dem Wörterbuch gefunden werden können.

   Wird die Lösung auf die Ausgabe des *dictionary representative problems* aufgebaut, so sucht man nach allen Präfixen (Amir et al. 1994; 1995) bzw. nach allen Suffixen (in dem hier vorgestellten Suchansatz) des gefundenen längsten Teilstrings ab einer bzw. bis zu einer gegebenen Textstelle. Diese Probleme werden als *dictionary prefix* bzw. als *dictionary suffix problem* bezeichnet. Wir stellen einen Algorithmus mit der Laufzeit $O(|x| + \sum_{j=1}^{|x|} |h_j|)$ vor. Dabei ist h_j der längste Suffix im Multi-Suffixbaum, der an der Stelle j endet. Leichte Modifikationen des Verfahrens erlauben es uns, diesen Wert durch $O(|x|+d)$ zu beschränken. Unter der Annahme, daß der Multi-Suffixbaum balanciert ist, beweisen wir, daß die Suche nach allen Teilstrings durch $O(|x| \log d)$ beschränkt ist. Dies ist um einen Faktor $\log\log d$ schlechter als das worst case Resultat von (Amir et al. 1995). Der entscheidende Vorteil ist jedoch, daß sich die Linearzeitresultate der Einfüge- und Löschoperationen nicht ändern.

5. Die Platzkomplexität wird auf $O(d)$ (mit d als die Summe der Längen der in dem Wörterbuch gespeicherten Zeichenketten) gesenkt. Unter der Annahme, daß jede Zeichenkette wenigstens einen Platz proportional zu ihrer Größe verbraucht, ist dieser Wert sogar optimal. Im Gegen-

satz zu der Löschoperation im Wörterbuch von (Amir et al. 1994; 1995) bekommen wir demnach den gesamten von dem gelöschten String m belegten Speicherplatz, zurück, da wir m nicht mehr als Referenz benötigen. Der Algorithmus von Amir et al. berührt dieses Problem nicht, so daß wir von einem Speicherplatzbedarf von $O(d^*)$ ausgehen müssen, mit d^* als Summe der Längen der *jemals* eingefügten Strings. Der Speicherbedarf akkumuliert sich in mehreren aufeinanderfolgenden Einfüge– und Löschoperationen, so daß die Speicherressourcen bald aufgebraucht sind.

Entgegen üblicher Darstellungsformen, werden die Implementationsdetails der Algorithmen angegeben. Sie sind für das Textverständnis nicht unmittelbar notwendig und können überlesen werden, doch erleichtern sie es der praktisch orientierten Leserschaft, die Verfahren zügig auf einer konventionellen Rechenmaschine in lauffähige Programme umzusetzen. Aus Platzgründen wird ein solcher Anspruch in der Literatur zumeist nur unzureichend verfolgt. Die vorgeschlagene Datenstruktur der Multi-Suffixbäume ist jedoch elaboriert, so daß in dieser Arbeit die auf der vorgestellten Datenstruktur agierenden Algorithmen ihre verborgenen technischen Fragestellungen *beraubt* werden.

5.1 Teilstringsuche für mehrere Strings

Das gleichzeitige Suchen von gespeicherten Teilstrings für eine gegebene Anfrage kann in einer Vielzahl von Anwendungen genutzt werden, wie z.B. beim Suchen in mehreren Dateien, bei der Textkorrektur, in der Molekularbiologie oder in Datenbankabfragen. Gewöhnlicherweise wird der Algorithmus von Aho und Corasick (Aho & Corasick 1975) oder eine seiner Verfeinerungen (Commentz-Walter 1979; Fan & Su 1993; Aoe 1994) verwendet.

Der Algorithmus von Aho und Corasick verwaltet einen Vielwegsuchbaum (engl. *Trie*) als Repräsentation der gespeicherten Zeichenketten. Der Begriff *Trie* wurde aus *retrieval* entnommen und die phonetische Gleichheit zu einem Baum (engl. *tree*) ist gewollt. Knuth spricht in dem Zusammenhang von einem *digital search tree* (Knuth 1973). Die Knoten eines Vielwegsuchbaumes haben entsprechend der Alphabetsgröße einen beschränkten Verzweigungsgrad, d.h. von jedem Knoten geht nur eine begrenzte Anzahl von Kanten aus einem Fundus aller möglichen Übergangsmöglichkeiten aus. Die Beschriftung (engl. *label*) der Kanten ist unter Geschwisterknoten paarweise unterschiedlich. Jedem Zustand im Trie wird ein Fehlerfunktionswert zugeordnet, der es erlaubt, von einem Zweig des Mehrwegsuchbaumes zum anderen zu springen. Sei t ein Knoten im Trie und $v(t)$ der durch t beschriebene String. Der Fehlerfunktionswert $f(t)$ ist definiert als das längste Endstück (engl. *suffix*) von $v(t)$, das gleichzeitig Anfangsstück eines (engl. *prefix*) im Wörterbuch gespeicherten Strings ist.

Aho und Corasick zeigen, daß damit eine Teilstringsuche in linearer Zeit möglich ist. Alle Werte von f können in einem Breitendurchlauf in $O(d)$ bestimmt werden. Auch ist die Speicherausbeute mit $O(d)$ für den Trie optimal. Leider ist eine dynamische Aktualisierung von f bei eingefügten oder gelöschten Zeichenketten nicht unmittelbar einsichtig, wie folgende Überlegung zeigt.

Sei t' der an eine bestehende Triestruktur an t hinzuzufügende Knoten und z der zugehörige Übergang von t nach t' (vergl. Abbildung 5.1). Weiterhin sei die transitive Hülle TH der Fehlerfunktion an t durch die Menge $\{t, f(t), f(f(t)), \ldots\}$ beschrieben. Der Fehlerfunktionswert von t' ist über $TH(t)$ mit dem Übergang z leicht ermittelt.

Das Problem der Neuberechnung von f ist, daß die Fehlerfunktionswerte aller Zustände t^* verändert werden müssen, deren Vorgänger den Knoten t in ihrer transitiven Hülle der Fehlerfunktionswerte besitzen und die mit z auf t^* weisen. Die Schwierigkeiten, die Knoten t^* bei einer inkrementellen Konstruktion (Zeichenketten werden nur eingefügt) des Tries und der Fehlerfunktion zu ermitteln, sind zuerst von Meyer aufgezeigt worden (Meyer 1985).

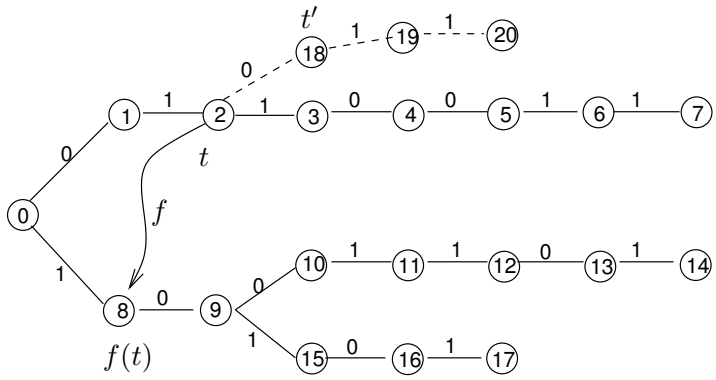


Abbildung 5.1: Ein Trie mit Fehlerfunktion für die Zeichenketten *0110011*, *1001101*, *10101* beim Einfügen von *01011*.

5.2 Meyers Ansatz

Meyer verwaltet zu der Fehlerfunktion an jedem Knoten auch die Urbilder f^{-1}. Das Inverse der Fehlerfunktion besitzt eine Baumstruktur, da für jeden Knoten außer der Wurzel der Fehlerfunktionswert auf einen Knoten geringerer Triefe im Trie verweist (vergl. Abbildung 5.2).

Die zentrale Prozedur in Meyers Ansatz ist die rekursive Funktion *Compute*f^{-1} (siehe Programm 5.1), die mit den alten und dem neuen Knoten t bzw. t' des Tries aufgerufen wird. Sie bestimmt alle potentiellen Vorgänger t_{neu} in $TH^{-1}(t)$ und prüft, ob die Übergänge $g(t_{neu}, z)$ definiert sind.

Sei $m_{\max}$ die maximale Länge aller Zeichenketten im Wörterbuch. Meyer postuliert eine Gesamtzeit von $O(m_{\max}d)$ für sein Knoten-für-Knoten Einfügeschema und argumentiert, daß $m_{\max}$ normalerweise nicht auftaucht. Eine Inkonsistenz in seiner Zeitanalyse wurde gefunden und ihm mitgeteilt: Er argumentiert, daß die Summe der Teilbäumgrößen im f^{-1}-Baum durch $O(m_{\max}d)$ begrenzt ist und damit auch die Gesamtanzahl der erreichten Knoten t_{neu}. Allerdings mag ein Knoten t aus der einen Iteration genauso in einer vorhergehenden Iteration gewählt werden. Unabhängig davon kann $m_{\max}$ nicht als konstant angesehen werden, so daß der Algorithmus nicht optimal ist.

Eine mögliche Alternative, den Knoten t' im f^{-1}-Baum einzufügen, ist es, alle Nachfolger von $f(t^*)$ zu betrachten. Obwohl t^* gesucht werden soll, ist

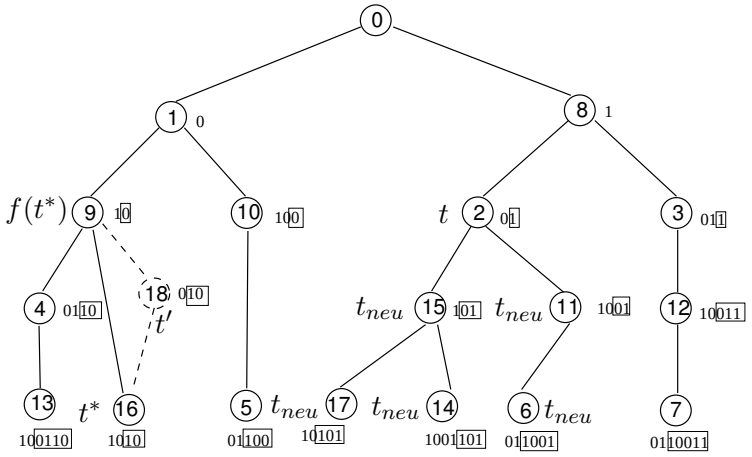


Abbildung 5.2: Der zu dem Trie korrespondierende f^{-1}–Baum.

Programm 5.1 *Compute*f^{-1}: Die Berechnung der inversen Fehlerfunktion für den Algorithmus von Aho und Corasick. Eingabe: Vorgängerknoten t, einzufügender Knoten t'. Ausgabe: Korrekter f^{-1}-Baum.

```
for all t_neu in f^{-1}(t)        {für alle Fehlerfunktionsurbilder von t}
  t* ← g(t_neu, z)                      {führe einen Zustandsübergang aus}
  if (t*)                       {falls ein Nachfolger t* von t_neu existiert}
    Insert(t', t*, f(t*))                {füge t' zwischen t* und f(t*) ein}
  else                             {Nachfolger t* von t_neu existiert nicht}
    Compute f^{-1}(t_neu, t')            {rekursiver Aufruf des Programms}
```

dieser Knoten bekannt - er fällt mit dem neuberechneten Wert $f(t')$ zusammen. Sei $v(t,i)$ das i-te Zeichen von $v(t)$. Von $f(t^*)$ ausgehend betrachten wir alle Knoten t^* aus der Nachfolgermenge $f^{-1}(f(t^*))$ und vergleichen solange die Zeichen $v(t^*,i)$ mit $v(t',i)$ für i aus $\{|v(t^*)|-|v(f(t^*))|,\ldots,1\}$, bis eine Unstimmigkeit festgestellt wird. In Abbildung 5.2 findet sich das untersuchte Intervall $[1,|v(t^*)|-|v(f(t^*))|]$ durch Streichen der Zahlen in den Kästchen. Je nach Größe des Alphabets, wird nur eine sehr geringe Anzahl von Vergleichen benötigt. In einigen Fällen kann $f^{-1}(f(t^*))$ jedoch sehr groß werden, z.B. für die einelementige Menge $M=\{(011\ldots1)\}$ und $f(t^*)$ als die Wurzel des assoziierten Tries.

Die im nächsten Abschnitt besprochene Struktur kann als eine Lösung des Problems des großen Verzweigungsgrads im f^{-1}-Baum angesehen werden. Für die Darstellung der Zeichenketten werden mehrere Knoten *spendiert*. Letztendlich bleibt die Gesamtzahl der Knoten aber linear durch die Gesamtlänge der gespeicherten Zeichenketten beschränkt.

5.3 Suffixbäume

Ein String m besitzt genau $|m|$ unterschiedliche Endstücke, bestehend aus insgesamt $\binom{|m|}{2} \in \Omega(|m|^2)$ unterschiedlichen Zeichen. Ergänzt man m um ein spezielles Endsymbol \$, und fügt man die Suffixe von m\$ in eine Mehrwegsuchbaumstruktur ein, so erhält man einen Suffixtrie (siehe Abbildung 5.3). Der Suffixtrie hat folgende grundlegende, nahezu definierende Eigenschaft.

Satz 14 *(Suffixtrie) Jeder Knoten im Suffixtrie für m\$ entspricht genau einem der unterschiedlichen Teilstrings von m\$.*

Beweis: Sicherlich entspricht jeder Knoten im Suffixtrie mindestens einem Teilstring von m\$, da er durch ein Anfangsstück eines Suffixes von m\$ erreicht werden kann. Unterschiedliche Zeichenketten führen in einem Mehrwegsuchbaum aber auch zu unterschiedlichen Knoten. □

Leider gibt es für den Suffixtrie noch immer Beispiele mit $\Omega(|m|^2)$ Knoten. Zeichenketten der Form 1^n0^n\$ haben die Länge $2n+1$ und n^2+4n+2 voneinander unterschiedliche Teilstrings.

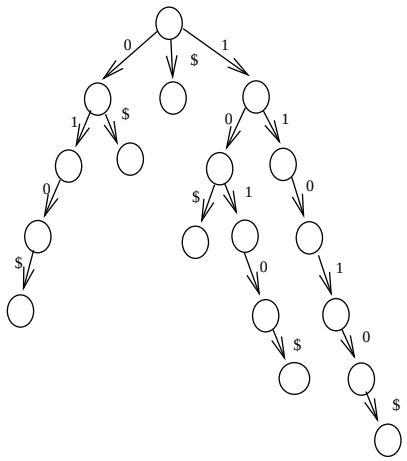


Abbildung 5.3: Ein Suffixtrie für den String *11010$*.

Ein *Suffixbaum* (vergl. Abbildung 5.4) ist eine kompakte Darstellungsform eines Suffixtries, in dem jeder Knoten mit nur einem Nachfolger mit seinem Elternteil verschmolzen ist. Die durch die Elimination von Knoten mit Grad zwei gewonnene Struktur wird im allgemeinen Patricia-Baum genannt. Der Begriff Patricia ist von Morrison geprägt worden und ist ein Akronym aus dem die Struktur beschreibenden Satz *practical algorithm to retrieve information coded in alphanumeric* (Morrison 1968). Weiner war der erste, der den Suffixbaum genauer studierte und somit einen expliziten Index für alle unterschiedlichen Teilstrings eines gegebenen Strings angab (Weiner 1973). Jeder Knoten im Suffixbaum für m\$ hat mehr als einen Nachfolger und m\$ viele Blätter. Die Kantenbeschriftungen werden als Intervall des zugrundeliegenden Strings an den Knoten verwaltet, auf dem die Kante einläuft, so daß der Suffixbaum nur $O(|m|)$ Platz benötigt.

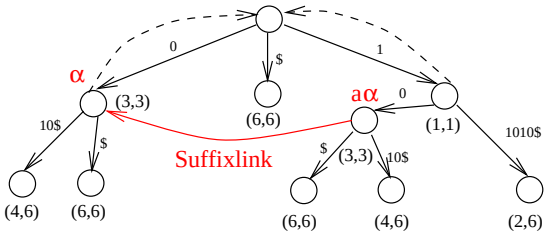


Abbildung 5.4: Ein Suffixbaum für den String *11010$*. Der Suffixlink von $a\alpha$, $a \in \Sigma$, $\alpha \in \Sigma^*$ zeigt auf α und kann als Abkürzung verwendet werden.

Unterschiedliche Konstruktionsalgorithmen für den Suffixbaum wurden angegeben. Die Algorithmen von (Slisenko 1983) und (Chen & Seiferas 1985) beruhen auf den Indexgedanken Weiners, während (McCreight 1976) eine speicherplatzfreundliche Konstruktion des Suffixbaumes vorschlägt. Sie fußt auf dem sogenannten *Suffixlink*, der von einem den String $a\alpha$ repräsentierenden Knoten auf den α repräsentierenden Knoten zeigt. Während McCreight den Suffixbaum von dem längsten zu dem kürzesten String konstruiert, schlägt Ukkonen einen Linearzeitalgorithmus für den gegenteiligen Aufbau vor (Ukkonen 1995).

Die definierende Eigenschaft aus Satz 14 liest sich für Suffixbäume wie folgt:

Satz 15 *(Suffixbaum) Jeder innere Knoten t im Suffixbaum für $m\$$ entspricht dem größten gemeinsamen Präfix zweier Suffixe von $m\$$. Die diese Suffixe beschreibenden Blätter finden sich in zwei unterschiedlichen Zweigen des Teilbaumes zu t.*

Beweis: Der größte gemeinsame Präfix p zweier Suffixe m^i und m^j von $m\$$ ist Teilstring von m^i und m^j und bildet demnach einen p beschreibenden Knoten t im Suffixtrie. Die die Suffixe m^i und m^j darstellenden Blätter befinden sich aufgrund der Präfixeigenschaft von p in dem Teilbaum des Suffixtries, der durch t beschrieben wird.

Annahme, der Knoten t wird durch die Kontraktion gelöscht. Dann hat t nur einen Nachfolger t', der - sagen wir - eine Erweiterung p' von p repräsentiert. Die Zeichenkette p' ist jedoch Präfix von m^i und m^j, was im Widerspruch dazu steht, daß p der größte gemeinsame Präfix ist.

Analog führt man die Annahme, daß sich m^i und m^j in einem Zweig des Teilbaumes zu t befinden, zu einem Widerspruch. □

5.4 Multi–Suffixbäume

Die Idee, McCreights Algorithmus zur Konstruktion eines Suffixbaumes auf mehrere Strings auszuweiten, stößt zuerst auf Widerstand.

Zwar können der Suffixbaum S' und damit die Indizes aller unterschiedlichen Teilstrings von $m_1\$_1, \ldots, m_k\$_k$ für die im Wörterbuch zu speichernden Strings $m_1, \ldots, m_k$ nach McCreights Ansatz aufgebaut werden, doch ist die Struktur ein für allemal festgelegt und müßte, wie die Fehlerfunktion im Algorithmus von Aho und Corasick, für jede Einfüge- und Löschoperation immer

wieder erneut berechnet werden. Auf der anderen Seite kann der Patricia–Baum S für alle Endstücke der Zeichenketten $m_1\$_1$ bis $m_k\$_k$ konstruiert werden.

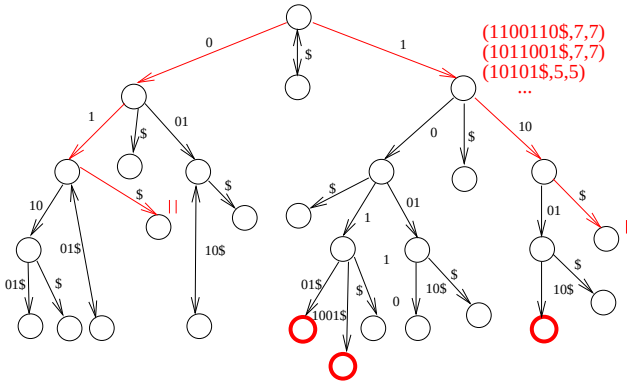


Abbildung 5.5: Der Multi–Suffixbaum für die Zeichenketten *1100110$*, *1011001$*, *10101$*.

Es ist der Verdienst von Amir et al. zu erkennen, daß die Baumstrukturen von S und S' *isomorph* (strukturell gleich) sind und daß die Kantenbeschriftungen bis auf die an den Blättern einlaufenden Kanten übereinstimmen (Amir et al. 1994).

Diese Beobachtung ermöglicht es, einen String $m\$$ in einen schon bestehenden Suffixbaum einzufügen. Um eine unbeschränkte Anzahl von Zustandsübergängen zu vermeiden, beschreiben wir mit \$ jedes der $\$_i$, lassen jedoch den Vergleich von \$ mit sich selbst scheitern.

Die Menge der in dem Wörterbuch zu speichernden Strings sei mit M bezeichnet. Im Multi-Suffixbaum (vergl. Abbildung 5.5) werden hingegen die Suffixe von $m\$$ für m aus M gespeichert. Diese Menge sei mit $M\$$ bezeichnet. Da die Länge von $m\$$ linear zur Länge von m ist, ist auch die Summe der Längen der in S gespeicherten Strings linear der Summe der Längen der ursprünglichen Zeichenketten.

Die an den Knoten verwalteten Kantenbeschriftungen werden durch die Angabe des Intervalls *eines* ausgewählten Strings dargestellt. Wir schreiben kurz $m(j,k)$ für den Teilstring $m[j],\ldots,m[k]$ von m. Der Anfangsindex j des Intervalls wird mit $m.f$ (für *first*), der Endindex k mit $m.l$ (für *last*) adressiert. Weiterhin wird an jedem Knoten die Länge (engl. *length*, kurz *len*) des Pfades verwaltet, der von der Wurzel ausgehend auf ihn zeigt. Letztendlich merken wir und an jedem Blatt die Anzahl c (für engl. *count*) der unterschiedlichen Suffixe, die durch dieses Blatt dargestellt werden. An den inneren Knoten steht c für die Größe der Nachfolgermenge. Eine Boole'sche Variable *acc* (für engl. *accepting*) vermerkt, ob ein Blattknoten einen vollständigen String aus der gespeicherten Stringmenge repräsentiert.

Der Speicherverbrauch für einen Suffixknoten ist damit konstant. Das Problem ist, die geeignete Auswahl der Zeichenkette m zu treffen, auf die ein Knoten verweist, bzw. die der Knoten *repräsentiert*.

Satz 16 *(Bijektion Trie-Suffixbaum) Sei $M\$$ die Menge der gespeicherten Zeichenketten $m\$$ im Suffixbaum S und sei T der Vielwegsuchbaum, der diese*

Strings in umgekehrter Reihenfolge $(m\$)^R$ *darstellt. Dann gibt es eine Bijektion zwischen der Menge der Knoten in* T *(mit Ausnahme der Wurzel) in die Menge der Blattknoten von* S*.*

Beweis: Auf der einen Seite entspricht jedem Suffix der Zeichenketten $m\$$ ein Blatt in S. Auf der anderen Seite existiert für jeden Präfix $(m\$)^R$ ein Knoten in T. □

Die in Abbildung 5.6 gewählten Zahlen verdeutlichen diesen Zusammenhang.

Folgerung 3 *Die Anzahl der Knoten in* S *ist durch die Summe der Längen der gespeicherten Zeichenketten beschränkt.*

Beweis: Jeder innere Knoten (außer der Wurzel) in einem Suffixbaum hat mindestens zwei Nachfolger. Weiterhin besitzt S für $m\$$ nicht mehr Blätter als der Vielwegsuchbaum für $(m\$)^R$ Knoten hat. Diese Anzahl ist durch die Summe der Längen der gespeicherten Zeichenketten beschränkt. □

Satz 16 ermöglicht es, Verweise von T zu den Blättern in S zu verwalten. Diese Verweise werden innerhalb der Einfügeoperation gesetzt und bei der Löschoperation genutzt.

Die nächste Betrachtung ermöglicht uns, die Anzahl derjenigen Endstücke der Strings in S zu zählen, durch die ein Blatt im Multi-Suffixbaum beschrieben werden kann.

Folgerung 4 *Ist der mit einem Knoten* t *in* T *assoziierte String* $v(t)$ *Präfix von* k *Strings in* T*, so kann die Kante des durch die Bijektion aus Satz 16 beschriebenen Blattes in* S *auf* k *unterschiedliche Weisen repräsentiert werden.*

Beweis: Sei $v(t)$ ein Präfix von k unterschiedlichen Strings $\$m_1, \ldots, \$m_k$ in T. Sei q das durch die Bijektion aus Satz 16 von t aus beschriebene Blatt in S. Dann ist $(v(t)^R)$ der auf q endende Suffix der in S gespeicherten Strings $(\$m_1)^R$ bis $(\$m_k)^R$. Demnach kann die in q einlaufende Kante auf k unterschiedliche Weisen beschrieben werden. □

Jede Kante in T endet in genau einem Knoten und stellt zumindest ein Zeichen eines der Strings aus T dar. Wir können demnach an jedem Knoten t aus T (außer der Wurzel) einen ihn repräsentierenden String auswählen.

Folgerung 5 *Die ausgewählte Repräsentation an* t *aus* T *läßt sich direkt auf die Repräsentation des Blattes in* S *übertragen.*

Beweis: Stellt t ein Zeichen von $(m\$)^R$ dar, so kann das durch die Bijektion aus Satz 16 festgelegte Blatt durch einen Suffix von $m\$$ erreicht werden. Damit läßt sich die einlaufende Kante in diesem Blatt durch ein Intervall in $m\$$ beschreiben. □

Damit ist es überhaupt erst möglich, von einer eindeutigen (weil durch T festgelegten) Repräsentation der Teilstrings an den Blattknoten zu sprechen.

Die folgenden nützlichen Begriffe für Umgang mit Suffixbäumen wurden von McCreight geprägt.

Definition 6 *Der Ort einer Zeichenkette ist der Knoten im Suffixbaum, auf dem der Pfad für diesen String endet. Eine Erweiterung eines Strings m ist jeder String, der m als Anfangsstück bzw. Präfix besitzt. Der erweiterte Ort eines Strings m ist der Ort der kleinsten Erweiterung von m, die im Suffixbaum existiert. Analog ist der kontraktierte Ort des Strings m der Ort des längsten Anfangsstückes von m. Sei m^j der Suffix von $m\$$, der von der Stelle j ausgeht. Der längste Präfix von m^j, der gleichzeitig Präfix eines der im Multi–Suffixbaum gespeicherten Suffixe ist, wird mit $head_j$ bezeichnet.*

5.5 Einfügen in einen Multi–Suffixbaum

Sei $m\$$ der einzufügende String. Wir beschreiben im folgenden die Veränderungen des Multi–Suffixbaumes S, die sich durch das Einfügen der Suffixe m^j von $m\$$ ergeben. Dabei sei S^{j-1} die Multi-Suffixbaumstruktur vor und S^j die Struktur nach der Einfügung von m^j.

Wir können annehmen, daß $(m\$)^R$ schon in T eingefügt wurde. Da die Einfügeroutine die längeren Suffixe zuerst einfügt, werden die Links von T aus rückwärts gehend gesetzt.

Als Beispiel wählen wir den Multi–Suffixbaum S aus Abbildung 5.5, d.h. wir fügen die Endstücke von 11010\$ nacheinander in S ein.

Programm 5.2 *Scan*: Sukzessiver Vergleich eines Strings mit den Kantenbeschriftungen im Multi-Suffixbaum. Eingabe: Teilstring m^j, gegeben durch m und j, und derzeitiger Standort el. Ausgabe: Wahrheitswert, ob Scanning an einem Knoten endet und Stelle, gegeben durch cl,el und at, an der ein Mismatch vorliegt.

```
while(true)                                          {Endlosschleife}
  cl ← el                                  {erster bzw. nächster Knoten}
  el ← getChild(cl.m[j + cl.len])               {finde ausgehende Kante}
  if (el = nil) return true                  {Scan endet an einem Knoten}
  k ← el.f                                        {absoluter Index in el}
  at ← −1                          {zurückgelegte Distanz auf der Kante}
  while(k <= el.l)               {solange Ende der Kante nicht erreicht}
    k ← k + 1; at ← at + 1                    {erhöhe Index und Distanz}
    if (m[j + cl.len + at] ≠ el.m[k − 1])              {Mismatch gefunden}
      return false                           {Scan endet auf einer Kante}
```

Zum Einfügen des ersten Suffixes $m^0 = 11010\$$ (siehe Abbildungen 5.6) wird ein Zeichen-für-Zeichenvergleich (engl *scanning*, vergl. Programm 5.2) in dem Multi–Suffixbaum durchgeführt, bis eine Nichtübereinstimmung (engl. *mismatch*) festgestellt wird. Als Rückgabe liefert das Scanning somit den kontraktierten wie auch den erweiterten Ort (cl bzw. el) des Präfixes 110\$ von m^0 und eine Ganzzahl at, die die Anzahl der gelesenen Zeichen auf der Kante von cl nach el beschreibt. In unserem Beispiel ist $at = 0$ und ein neuer Knoten mit der Beschriftung 110\$ wird an cl angehängt. Diesen Vorgang wird mit *InsertAfter* (vergl. Programm 5.3) bezeichnet.

Programm 5.3 *InsertAfter*: Das Endstück eines Suffixes wird direkt hinter einem bestehenden Knoten eingefügt. Dabei wird ein neues Blatt erzeugt. Eingabe: m^j, gegeben durch m und i) und derzeitiger Standort cl. Ausgabe: Kopf $head_j$, von dem aus der Suffixlink noch gesetzt werden muß.

$q \leftarrow new(m)$ {ein neues Blatt wird erzeugt}
if $(j = 0)$ $q.acc \leftarrow 1$ {Blatt für $m\$ = m^0$ wird gekennzeichnet}
$q \leftarrow m(j + cl.len, m.len - 1)$ {Teilstring der Kante in p}
$q.len \leftarrow m.len - j$ {die Länge des eingefügten Suffixes}
$setChild(cl, m[q.f], q)$ {Blatt q wird in die Struktur eingefügt}
return cl {der Kopf $head_j$ ist mit cl gefunden}

Nun wird der Suffixlink von cl genutzt, um $m^1 = 1010\$$ einzufügen (vergl. Abbildung 5.7). Damit kürzt sich der Weg gegenüber einem Neuanfang an der Wurzel ab. Startend von dem durch den Suffixlink beschriebenen Knoten erreichen wir durch abermaliges Scanning den kontraktierten Ort 101 und den erweiterten Ort 10101\$ von 1010. Die Anzahl at der noch übereinstimmenden Zeichen auf der Kante von cl nach el ist gleich eins. Daraufhin wird ein neuer innerer Knoten (der Ort von 1010) und ein neuer Blattknoten (der Ort von 1010\$) in den Multi–Suffixbaum eingefügt. Diese Operation wird mit *InsertBetween* bezeichnet (vergl. Programm 5.4).

Programm 5.4 *InsertBetween*: Das Endstück eines Suffixes wird zwischen zwei bestehenden Knoten eingefügt. Dabei werden zwei neue Knoten erzeugt. Eingabe: m^j und derzeitiger Standort (cl,el,at). Ausgabe: Position $head_j$, von dem aus der Suffixlink noch gesetzt werden muß.

$p \leftarrow q \leftarrow new(m)$ {erzeuge 2 neue Knoten}
if $(m[j + cl.len + at] = el.m[el.f + at] = \$)$ {Mismatch von \$ mit \$}
  $el.c \leftarrow el.c + 1$; **return** cl {erhöhe Darstellungszähler}
if $(j = 0)$ $q.acc \leftarrow 1$ {Blatt für $m\$ = m^0$ wird gekennzeichnet}
$p \leftarrow m(j + cl.len, j + cl.len + at - 1)$ {Teilstring der Kante an p}
$p.len \leftarrow cl.len + at$ {Länge des Suffixes bis p}
$q \leftarrow m(p.len + j, m.len - 1)$ {Teilstring der einlaufenden Kante}
$q.len \leftarrow m.len - j$ {Länge des Suffixes bis q}
$el.m.f \leftarrow el.f + at$ {Kantenbeschriftung in el wird verkürzt}
if $(getChild(cl, m[p.f]))$ $removeChild(el, m[p.f])$ {Kante löschen}
$setChild(p, m[q.f], q)$ {q wird Nachfolger von p}
$setChild(p, el.m[el.f], el)$ {el wird Nachfolger von p}
$setChild(cl, m[p.f], p)$ {p wird Nachfolger von cl}
return p {Kopf $head_j$ mit p gefunden.}

Der Rückgabewert von *InsertAfter* ist der Knoten cl und von *InsertBet-*

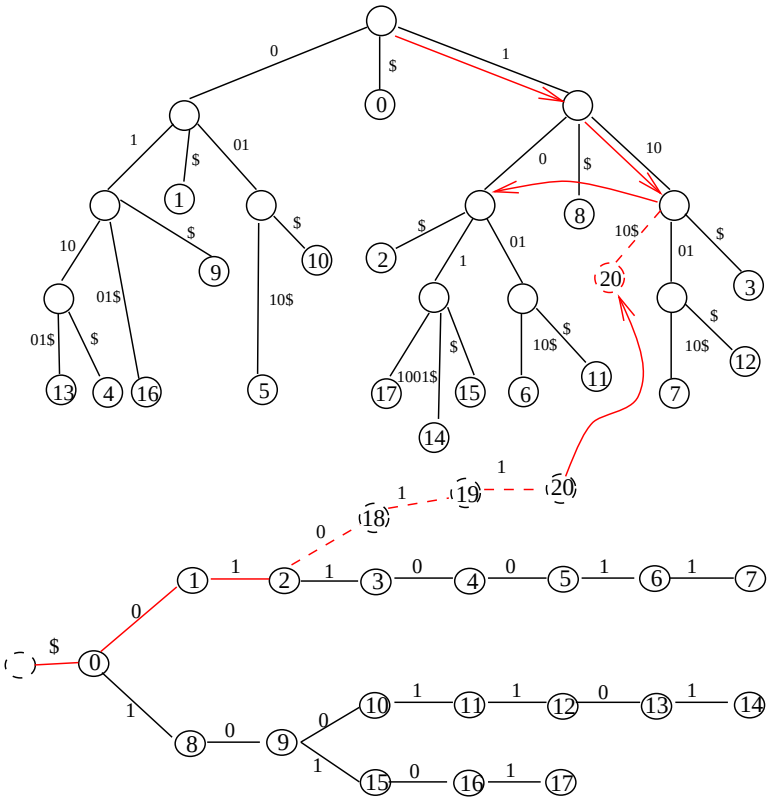


Abbildung 5.6: Der Multi-Suffixbaum für *1100110*\$, *1011001$*, *10101$* bei der Einfügung von *11010$*.

ween der neu erzeugte innere Knoten. Sie repräsentieren den längsten Präfix von m^j eines gespeicherten Suffixes im Multi–Suffixbaum und damit $head_j$.

Die entscheidende Idee der Einfügeoperation ist es, zwei Invarianzen zu garantieren:

(I1) In dem Multi-Suffixbaum S^j (der Baum, in dem m^j eingefügt wurde) besitzt höchstens der Ort von $head_j$ einen fehlerhaften Suffixlink.

(I2) Der kontraktierte Ort von $head_j$ wird besucht.

In dem Beispiel haben wir $head_0 = 110$ und $head_1 = 1010$. Die kontraktierten Orte 110 von $head_0$ bzw. 101 von $head_1$ wurden in der ersten bzw. zweiten Iteration besucht. Der Suffixlink von $head_0$ braucht nicht aktualisiert zu werden, da er schon korrekt war. Der Suffixlink von $head_1$ wird in der nächsten Iteration gesetzt.

Sei w mit $|w| = at$ der String, der die schon korrekt beschriebenen Zeichen auf der Kante von cl nach el beschreibt. Da nur der Suffixlink von cl genutzt werden kann, muß w nach dem Übergang erneut eingelesen werden (vergl. Abbildung 5.8). Wichtig ist, daß w auch tatsächlich in einem abwärtsgerichteten Vergleich von dem durch den Suffixlink von cl beschriebenen Knoten eingelesen werden kann. Dies gilt aufgrund der Definition des Suffixlinks, da alle Suffixe des Strings für den Ort el schon in den Multi-Suffixbaum eingefügt worden sind.

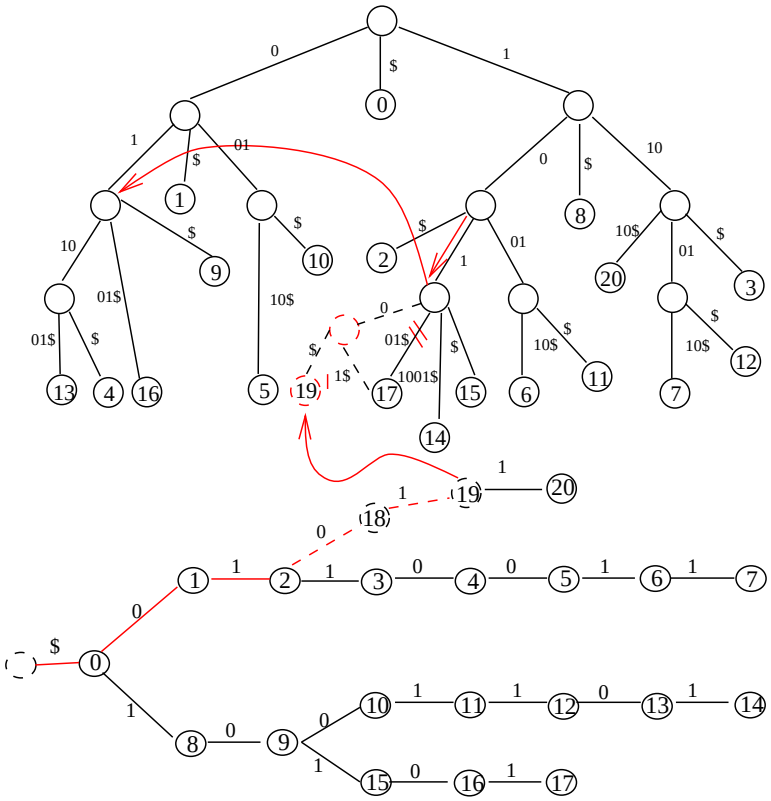


Abbildung 5.7: Der Multi–Suffixbaum bei der Einfügung von *1010$*.

Die *Rescanning* Funktion (vergl. Programm 5.5), die den erneuten Vergleich mit w durchführt, ist entgegen dem *Scanning*-Aufruf unabhängig von m. Das Rescanning von w endet genauso wie in der letzten Iteration auf einer Kante, so daß zwei neue Knoten mit *InsertBetween* eingefügt werden. Da das auf w folgende Zeichen schon in der vorherigen Iteration nicht mit dem zu lesenden Zeichen übereinstimmt, paßt es auch in dieser Iteration nicht. Der Suffixlink von $head_1$ kann auf $head_2$ gesetzt werden.

Nach dem Gang über den Suffixlink von cl gelangen wir zu dem Ort von 1 (vergl. Abbildung 5.9). Ein erneutes Rescanning von w ist nun erfolgreich und endet an einem Knoten. Wir können den Suffixlink von $head_3 = 010$ auf den so gefundenen Ort von 10 setzen. Nun wird das *Scanning* des Eingabestrings fortgesetzt und wir finden, daß der Ort des einzufügenden Strings 10$ schon existiert. Dies ist auch der Fall für die noch ausstehenden Einfügungen von 0$ und $.

Zum Verständnis des Gesamtansatzes beschreiben wir vorerst die Aufteilung (engl. *decomposition*, siehe Programm 5.6) von $head_{j-1}$ in u,v und w. Vor jedem Iterationsschritt gilt (Stephen 1994):

1. Wenn der kontraktierte Ort von $head_{j-1}$ nicht die Wurzel ist, dann ist es der Ort von uv, sonst ist v der leere String ϵ.

2. Die Länge von u ist bis auf die Ausnahme $head_{j-1} = \epsilon$ gleich eins.

Somit können wir einen Iterationsschritt (Prozedur *Iterate*, vergl. Pro-

Programm 5.5 *Rescan*: Ein Teilstring muß im Multi-Suffixbaum nochmal eingelesen werden. Da er passen muß, können Vergleiche eingespart werden. Eingabe: Wiedereinzulesendes w und derzeitiger Standort cl. Ausgabe: Wahrheitswert, ob Rescanning erschöpfend ist und die Stelle (cl,el,at), an der w endet.

```
j' ← j ← w.f                    {j' merkt sich die alte Position von j in w}
if (w = ε) return true          {leerer String endet immer an einem Knoten}
while (j <= w.l)                      {w noch nicht vollständig bearbeitet}
  cl ← el;                      {die Kante von cl nach el wird übersprungen}
  el ← cl.getChild(w.m[j])                      {neuer Nachfolger gesucht}
  j' ← j                                       {merke alte Position von j}
  j ← j + el.w.len                {Position von j plus die Kantenlänge}
at ← w.l − j' + 1                          {Distanz auf Kante als Rückgabe}
if ( at < el.l − el.f + 1 )     {at kleiner als Kantenbeschriftungslänge}
  return false                                {Rescan endet auf einer Kante}
else return true                              {Rescan endet auf einem Knoten}
```

gramm 5.7) der Einfügeprozedur eines Strings $m\$$ wie folgt umreißen.

1. Als erstes wird $head_{j-1}$ entsprechend der obigen Betrachtung in u, v und w zerlegt.
2. Daraufhin wird ein Rescanning von w ausgeführt. Es ergeben sich zwei Fälle.
   (a) Das Rescanning endet an einem Knoten v. Der ausstehende Suffixlink von $head_{j-1}$ wird auf v gesetzt. Je nachdem, ob das Scanning an einem Knoten endet, wird das Endstück von m^j hinter einem Knoten (*InsertAfter*) oder zwischen zwei Knoten (*InsertBetween*) eingefügt und somit $head_j$ festgelegt.
   (b) Das Rescanning endet auf einer Kante. Dann wird unmittelbar entsprechend dem Rest von w das Endstück von m^j zwischen den zwei sich ergebenden Knoten eingefügt. Der innere Knoten repräsentiert $head_j$ und gleichzeitig die Position, auf den der noch ausstehende Suffixlink von $head_{j-1}$ gesetzt wird.

Eine Analyse der Korrektheit und Laufzeitanalyse von *Iterate* wird von Amir et al. gegeben, die sich wiederum auf die Ergebnisse von McCreight stützen (Amir et al. 1994).

Die im folgenden vorgeführten wesentlichen Beweisideen führen zu einem tieferen Verständnis des Ansatzes.

Satz 17 *(Korrektheit Iterate) Das Verfahren Iterate fügt den Suffix m^j korrekt in den Multi-Suffixbaum S^{j-1} ein.*

Beweis: (vergl. (Stephen 1994)) Für die Korrektheit des Verfahrens muß gezeigt werden, daß die Invarianzen *(I1)* und *(I2)* nach dem Einfügen von m^j nach wie vor gelten.

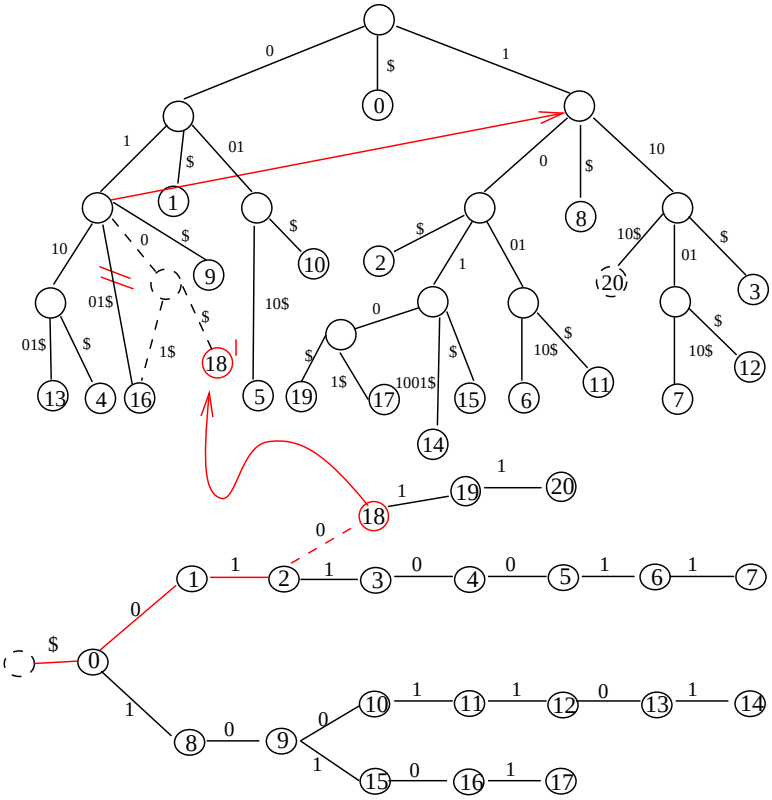


Abbildung 5.8: Der Multi–Suffixbaum bei der Einfügung von *010$*.

Da der kontraktierte Ort von $head_{j-1}$ ein Anfangsstück von $head_{j-1}$ darstellt, muß der genutzte valide Suffixlink von dem kontraktierten Ort auch Anfangsstück des noch zu setzenden Suffixlinks von $head_{j-1}$ sein. Durch diese Abkürzung wird das erste Zeichen von $head_{j-1}$ abgestreift, und nach Beendigung des Rescannings ist der Ort des Suffixlinks von $head_{j-1}$ gefunden.

Wenn das Rescanning nicht an einem Knoten endet, so ist sichergestellt, daß das *fehlgeschlagene* Zeichen auf der alten Kante auch mit dem Zeichen auf der neuen Kante nicht übereinstimmt. Es braucht in diesem Fall kein Scanning der Eingabe mehr zu erfolgen, und der Suffixlink von $head_{j-1}$ ist der Ort von $head_j$.

Damit sind alle Suffixlinks der Knoten aus S^{j-1} korrekt. Da Suffixlinks aber nur für innere Knoten bestimmt sind und nur ein innerer Knoten hinzugefügt wird, der gleichzeitig den Ort von $head_j$ darstellt, gilt *(I1)*.

Der kontraktierte Ort $head_j$ wird besucht, da er unterhalb von dem Suffixlink des kontraktierten Ortes von $head_{j-1}$ liegt. Von dort an sind alle Vergleiche von m^j mit einem Pfad in S^j im Scanning und Rescanning bis zu dem Rückgabewert der Einfügeoperationen *InsertAfter* und *InsertBetween* korrekt. Dieser Wert ist der Ort des längsten Präfix von m^j, der gleichzeitig Präfix eines gespeicherten Suffixes im Multi–Suffixbaum ist. Dies ist die Definition von $head_j$, und cl stellt somit den kontraktierten Ort von $head_j$ dar. Somit gilt die Invarianz *(I2)*. Die Invarianzen *(I1)* und *(I2)* erlauben es wiederum, $head_j$ im nächsten Schritt aufzuteilen. □

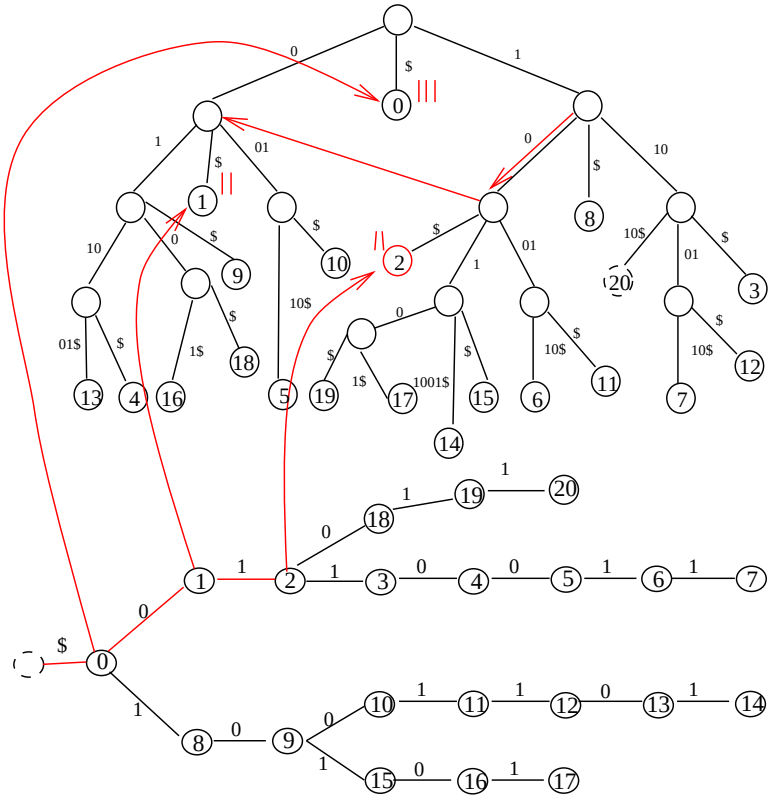


Abbildung 5.9: Der Multi–Suffixbaum bei der Einfügung von *10$*, *0$* und *$*.

Satz 18 *(Laufzeit Iterate) Die Prozedur Iterate benötigt amortisiert konstante Zeit zur Berechnung der Transformation des Multi-Suffixbaumes S^{j-1} in S^j.*

Beweis: (vergl. (McCreight 1976)) In der amortisierten Laufzeitanalyse wird gezeigt, daß die Gesamtanzahl der Schritte beim Einfügen eines String $m\$$ durch $O(|m|)$ beschränkt ist. Die einzigen Prozeduren, die mehr als konstante Zeit benötigen, sind *Scan* und *Rescan*. Somit bleibt zu beweisen, daß der Gesamtaufwand für *Scan* und für *Rescan* durch $O(|m|)$ beschränkt ist. Beide Prozeduren selbst werden maximal $O(|m|)$ mal aufgerufen:

1. *Scan*: Die Gesamtanzahl der Operationen in *Scan* ist proportional zu der Anzahl der durchgeführten Zeichenvergleiche. Der Indexwert $j + cl.len + at$ von m wird bei jedem Zeichenvergleich erhöht. Sollte at erniedrigt werden, dann wird $cl.len$ wiederum um den gleichen Wert erhöht. Wenn sich durch den Suffixlink der Wert $cl.len$ um eins senkt, wird j um eins erhöht. Demnach wird jedes Zeichen in $m\$$ maximal einmal zum Zeichenvergleich in *Scan* herangezogen.

2. *Rescan*: Die Gesamtanzahl der Operationen in *Rescan* ist durch die Gesamtlänge der neu einzulesenden Teilstrings beschränkt. Mehr noch, sie ist durch die Anzahl der Knoten beschränkt, die bei dem Rescanning-Schritt traversiert werden, da in jedem Vergleichsschritt die Länge einer Kantenbeschriftung verarbeitet wird. Damit nimmt die Länge des Teilstrings für jeden durchgeführten Übergang zum nächsten Knoten um die

Programm 5.6 *Decomposition*: Ein String wird in drei Bestandteile zerlegt. Eingabe: String uvw und derzeitiger kontraktierte Ort cl. Ausgabe: Teilstrings v,w von uvw, Variable uv entspricht String uv.

if $(uvw = \epsilon)$ {da uvw leerer String ist, ist u}
  $v \leftarrow w \leftarrow \epsilon$; $uv \leftarrow root$ {als auch v bzw. w leer}
else {andernfalls ist $|u| = 1$}
  if $(cl.IsRoot)$ {kontraktierter Ort von $head_{j-1}$ ist Wurzel}
    $v \leftarrow \epsilon$ {v ist der leere String}
    $w \leftarrow uvw.m(f+1, l)$ {u wird abgestreift}
  else {kontraktierter Ort von $head_{j-1}$ nicht die Wurzel}
    $uv \leftarrow cl$ {von uv geht Abkürzung aus}
    $v \leftarrow uvw.m(f+1, f+uv.len-1)$ {u wird abgestreift}
    $w \leftarrow uvw.m(f+uv.len, l)$ {w muß neu eingelesen werden}

Länge der durch die Kante beschriebenen Zeichenkette ab. Weiterhin nehmen die Längen der Teilworte nur wieder mit jedem in Scan übereinstimmenden Zeichen zu. Also ist die gesamte Rescanningzeit durch die gesamte Scanningzeit beschränkt.

□

5.6 Löschen in einem Multi–Suffixbaum

Amir et al. finden alle Blattknoten eines zu löschenden Strings $m\$$ mit der Hilfe der Suche von $m\$$ in dem Wörterbuch (Amir et al. 1994).

Mit Hilfe der in Satz 16 gezeigten Bijektion kann diese Menge jedoch direkt durch den Trie T bestimmt werden. Dies führt zu der folgenden einfachen Löschroutine (siehe auch Prozedur 5.8):

1. Lösche alle Knoten in T und fülle Stack mit Links zu den Blättern in S.
2. Lösche die Blätter in S und evtl. Vorgänger mit nur noch einem Kind.

Dabei wird der Stack zur Entkopplung der Löschoperationen in S und T genutzt. Dies ist nicht unbedingt notwendig, denn die Verfahren können abwechselnd aufgerufen werden.

Das Löschen der Suffixe kann von den kürzesten Suffixen ausgehend hin zu den längsten geschehen und stände damit umgekehrt zur Einfügereihenfolge. Es ist allerdings einfacher, die Strings vom längsten Suffix zum kürzesten Suffix zu löschen. Dann sind die Suffixlinks nach jedem einzelnen Löschschritt im Suffixbaum korrekt, da S der Menge von Strings entspricht, in die ersten Buchstabe der löschenden Zeichenkette fehlen.

Das von Amir et al. vernachlässigte Problem ist, daß der entfernte String (insbesondere an einem inneren Knoten) noch gebraucht werden kann. Die Grundidee der vorgeschlagenen Verbesserung ist es, die durch T eindeutig

Programm 5.7 *Iterate*: Das Einfügen eines Suffixes im Multi Suffixbaum. Eingabe: String m_j, Positionszeiger $head_{j-1}$ aus der Einfügung von m_{j-1} und assozierter String uvw zu $head_{j-1}$. Ausgabe: Positionszeiger $head_j$ und assozierter uvw.

```
Decomposition(uvw, cl, uv, v, w)                    {teile uvw in u und v}
if (v = ε) cl ← el ← root else cl ← el ← uv.sl                {Abkürzung}
if (Rescan(w, at, cl, el))              {falls Rescan an einem Knoten endet}
  if (head.sl = 0)head.sl ← el            {setze Link von head_{j-1} auf el}
  if (Scan(m, j, at, cl, el))            {falls Scan auf einem Knoten endet}
    head ← InsertAfter(m, j, cl)                {setze Kopf head_j auf cl}
  else                                                      {oder auf den}
    head ← InsertBetween(m, j, cl, el, at)          {neuen inneren Knoten}
else                                          {w paßt nicht, z = ε, kein Scan}
  inner ← InsertBetween(m, j, cl, el, at)                {inner ist Variable}
  if (head.sl = 0) head.sl ← inner          {setze Suffixlink von head_{j-1}}
  head ← inner                    {und head_j auf den neuen inneren Knoten}
uvw ← head.m(l − len + 1, l)                      {head_j legt neues uvw fest}
```

Programm 5.8 *Delete*: Das Löschen eines Strings im Multi-Suffixbaum unter Zuhilfenahme eines Stapels. Eingabe: String m.

```
T.remove(m$^R, St)                          {lösche Knoten und fülle Stapel St}
while (St ≠ ∅)                  {nacheinander vom längsten zum kürzesten}
  q ← St.pop                                  {entferne Knoten vom Stapel und}
    RemoveLeaf(q)                  {korrigiere Falschrepräsentationen}
```

festgelegte Repräsentation an den Blättern auf die inneren Knoten zu übertragen. Leider wandelt sich die Vorgängerrelation durch das Einfügen neuer Strings, sodaß eine direkte Beziehung gleicher Repräsentation zwischen den Blätten und ihren Vorgängern nicht gewährleistet werden kann.

Deshalb führen wir *Zwillinge* ein, die die Historie der Blattgenerierungen beschreiben. Jeder Knoten t aus T besitzt einen Zwilling, dessen Nachfolger und Vorgänger wieder auf einen Knoten in T weist. Wird *InsertAfter* aufgerufen, so wird ein Nachfolger wie auch ein Zwillingsnachfolger von dem kontraktierten Ort cl auf das neue Blatt hinzugefügt. Bei einem *InsertBetween*-Aufruf wird nur ein Zwillingsnachfolger des inneren Knotens gebildet, der auf das neue Blatt zeigt. Der Zwillingsnachfolger des kontraktierten Knotens bleibt bestehen.

Für den Multi–Suffixbaum aus Abbildung 5.5 wird die kombinierte Struktur aus Nachfolgern und Zwillingsnachfolgern in Abbildung 5.10 dargestellt.

Nun können wir die Löschprozedur *RemoveLeaf* (vergl. Programm 5.9) genauer beschreiben. Dazu sei q das zu löschende Blatt in S und s der Vorgänger

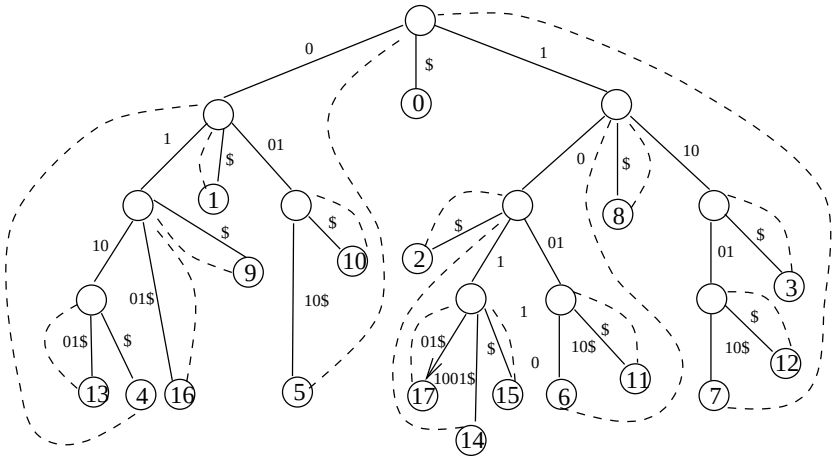


Abbildung 5.10: Die Zwillingsstruktur in dem Multi–Suffixbaum für *1100110$*, *1011001$*, *10101$*.

von q. Weiterhin sei p der eindeutige Zwillingsvorgänger von q und r bzw. r' zwei weitere Zwillingsnachfolger von s (falls vorhanden). Dann erhalten wir durch das Löschen von q in der ursprünglichen Struktur und in der Zwillingsstruktur die folgende Fallunterscheidung:

1. Der Vorgänger s ist gleich dem Zwillingsvorgänger p. Falls s keinen weiteren Nachfolger hat, so lösche s aufgrund der Kontraktionsbedingung für Patricia-Bäume. Sonst nehme einen beliebigen Zwillingsnachfolger r und übertrage dessen Stringrepräsentation auf die von s.

2. Der Vorgänger s ist ungleich dem Zwillingsvorgänger q. Nehme einen weiteren Zwillingsnachfolger r von s und mache ihn zum Zwillingsnachfolger von p. Falls s keinen weiteren Nachfolger hat, so lösche s. Sonst nehme einen beliebigen Zwillingsnachfolger r' und übertrage dessen Stringrepräsentation auf die von s.

Hilfssatz 4 *Seien I und L die Mengen aller inneren bzw. Blattknoten in dem Multi–Suffixbaum S. Sei $\Gamma(p)$ die Menge der Nachfolger von p und $\Gamma'(p)$ die Menge der Zwillingsnachfolger von p.*

In dem Multi–Suffixbaum gilt die folgende Invarianz:

(Ia) *Für alle inneren Knoten p aus I existiert ein Blattknoten q aus L der zugleich Zwillingsnachfolger von p ist ($q \in \Gamma'(p)$) und dessen Stringrepräsentation mit p übereinstimmt ($p.m = q.m$).*

(Ib) *Für alle inneren Knoten p aus I gilt: $|\Gamma'(p)| = |\Gamma(p)| - 1$, d.h. die Anzahl der direkten Nachfolger ist immer um eins größer als die Anzahl der Zwillingsnachfolger.*

Beweis: Die Struktur verändernden Operationen sind *InsertAfter*, *InsertBetween* und *RemoveLeaf*

Programm 5.9 *RemoveLeaf*: Löschen eines einzelnen Blattes im Multi- Suffixbaum. Eingabe: Zu löschendes Blatt q.

```
s ← q.pred; p ← q.twin.pred                    {(Zwillings-) Vorgänger von q}
j ← findLabel(p.twin, q)                       {Beschriftung der ausgehenden Kante}
if (q.c > 0)                                   {mindestens 2 Suffixe zeigen auf q}
  q.c ← q.c − 1; return                        {erniedrige Zähler}
removeChild(s, q); removeChild(p.twin, q)      {lösche Blattknoten q}
r ←anyChild(s.twin)                            {finde Zwillingsnachfolger von s}
if (s = p)                                     {direkte Vorgänger nicht gleich}
  if (s.c > 1)                                 {s hat mehr als einen Nachfolger}
    ChangeLeaf(r)                              {adjustiere Repräsentation}
  else                                         {s hat nur noch einen Nachfolger}
    RemoveInner(s)                             {lösche inneren Knoten s}
else                                           {direkte Vorgänger gleich}
  removeChild(s.twin, r); setChild(p.twin, r, j)   {gib Zwilling ab}
  ChangeLeaf(r)                                {adjustiere Repräsentation für p}
  if (s.c > 1)                                 {s hat mehr als einen Nachfolger}
    r' ← anyChild(s.twin)                      {finde weiteren Zwillingsnachfolger}
    ChangeLeaf(r')                             {adjustiere Repräsentation}
  else                                         {s hat nur einen Nachfolger}
    RemoveInner(s)                             {lösche inneren Knoten s}
```

InsertAfter Seien die Parameter der Prozedur durch m, j und cl gegeben. Das neu zu erzeugende Blatt vergrößert die Mengen $\Gamma(cl)$ und $\Gamma'(cl)$ um je ein Element. Somit gilt *(Ib)*. Die Stringdarstellung des Blattes und von cl wird auf m gesetzt und somit wird *(Ia)* erfüllt.

InsertBetween Seien die Parameter der Prozedur durch m, j, cl, el und at gegeben. Da die Stringrepräsentation beider neu erzeugter Knoten auf m gesetzt wird, gilt *(Ia)* unmittelbar. Der innere Knoten bekommt zwei Nachfolger (el und den neuen Blattknoten) und einen Zwillingsnachfolger (den neuen Blattknoten). Dadurch ist $2 = |\Gamma(p)| = |\Gamma'(p)| + 1$.

RemoveLeaf Seien q der zu löschende Knoten, s sein Vorgänger und p sein Zwillingsvorgänger. Der Algorithmus betrachtet zwei Fälle (vergl. Abbildung 5.11):

- $s = p$ Da q in $\Gamma(s) \cap \Gamma'(s)$ liegt ist *(Ib)* erfüllt. Wenn $|\Gamma(s)| > 1$ gilt, dann existiert ein Blatt r in $\Gamma'(s)$. Dieses Blatt r bestimmt durch die *ChangeLeaf* Operation die Stringrepräsentation von s, so daß s nicht mehr auf $q.m$ zugreift. Damit ergibt sich *(Ia)* für den Knoten s. Wenn hingegen $|\Gamma(s)| = 1$ ist, so wird s gelöscht und es muß nichts mehr gezeigt werden.
- $s \neq p$ Dieser Fall ist der schwierigste. Wenn $|\Gamma(s)| = 1$ ist, dann wird s entfernt. Weiterhin wird $\Gamma'(p)$ auf $\Gamma'(p) - \{q\} \cup \{r'\}$ gesetzt, so daß $|\Gamma'(p)|$ unverändert bleibt. Anderenfalls ist $|\Gamma(s)| = k > 1$. Somit gab es nach Invarianz *(Ib)* zu dem Zeitpunkt als q gelöscht

wurde genau $k+1$ Nachfolger und k Zwillingsnachfolger von s. Demnach gibt es neben r' einen weiteren Zwillingsnachfolger r von s. Dieser kann genutzt werden, um die Stringrepräsentation von p zu bestimmen, d.h $\Gamma'(p)$ wird auf $\Gamma'(p) - \{q\} \cup \{r\}$ gesetzt. Die Invarianten *(Ia)* und *(Ib)* gelten somit wie zuvor.

□

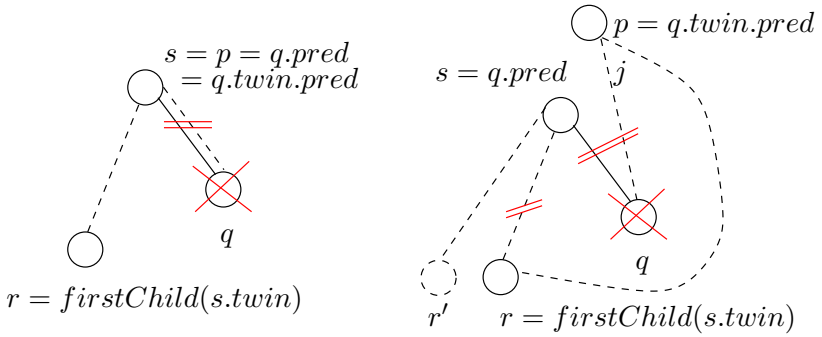


Abbildung 5.11: Der Beweis der Gradinvarianz für Zwillinge.

Die Invariante *(Ib)* dient demnach im Beweis nur, um *(Ia)* in einem Fall zu garantieren. Wir kommen zum Hauptresultat dieses Abschnittes.

Satz 19 *(Platzoptimalität) Sei M die Menge von Zeichenketten, die in dem durch S repräsentierten Wörterbuch nach einer beliebigen Anzahl von Einfüge- und Löschoperationen gespeichert sind und d die Summe der Längen aller Strings m aus M. Dann ist der benötigte Speicherplatz im Multi-Suffixbaum S durch $O(d)$ beschränkt.*

Beweis: Die Invariante *(Ia)* aus Hilfssatz 4 belegt, daß sich nach dem Einfügen und Löschen an den inneren Knoten in S nur Stringrepräsentationen finden, die auch an den Blättern gegeben sind. In Verbindung mit der Folgerung 5 aus Satz 16 heißt dies, daß jeder Knoten in S nur einen String repräsentiert, der auch in T repräsentiert ist. Damit sind die Verweise valide, d.h. nach *Delete(m)* kann m endgültig gelöscht werden. Aufgrund von Folgerung 3 ist die Platzkomplexität in $O(d)$. □

5.7 Suche in einem Multi–Suffixbaum

Wir analysieren den Fall der Suche nach einem Teilstring bei einer gegebenen Anfrage zuerst.

5.7.1 Suche eines Teilstrings

Amir et al. berechnen den längsten Präfix h in ST bestimmen, der mit x *beginnend* an Position j übereinstimmt (Amir et al. 1994). Wir wollen in diesem Abschnitt hingegen den längsten Teilstring h, der mit x *endend* an Position j, $j \in \{1, \ldots, |x|\}$, übereinstimmt.

Wenn die Strings *teilstringfrei* sind, ist in beiden Ansätzen nur zu testen, ob der gefundene längste Präfix h auch gleichzeitig ein String im Wörterbuch ist. Die Darstellung von h als Teilstring ist üblicherweise durch die Repräsentation des erweiterten Ortes von h in S gegeben. Damit brauchen wir nur zu testen, ob h maximal ist, d.h., ob die Länge von $h.m$ gleich der Länge von h ist.

Wir haben gesehen, daß bei der Suche, ob ein Teilstring existiert, die Forderung nach *Teilstringfreiheit* keine Einschränkung darstellt, da ein Teilstring von m gleichzeitig Teilstring eines Superstrings von m ist.

Ein *Suffixzustand* (vergl. Abbildung 5.12) ist eine Momentaufnahme des Suchvorganges innerhalb des Multi-Suffixbaumes. Durch die Darstellung als Patricia-Baum ist die Beschreibung durch einen einzelnen Knoten nicht ausreichend. Demnach legen wir einen Suffixzustand als ein Tripel aus Vorgänger cl, Nachfolger el und der Position at fest. Um eine bessere Beschreibung des Suchverfahrens zu gewährleisten, fügen wir die Intervall-Darstellung des erkannten Teilstrings h hinzu. Offensichtlich benötigt die Darstellung eines Suffixzustandes nur $O(1)$ also konstanten Platz.

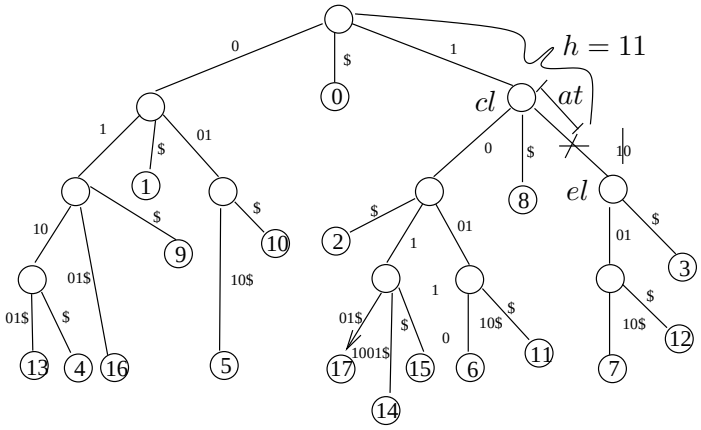


Abbildung 5.12: Ein Zustand im Multi-Suffixbaum.

Wir beschreiben nun den Übergang δ (vergl. Programm 5.10) von einem Suffixzustand q in einen Suffixzustand q'. Dabei werden wir immer dann durch einen Buchstaben am Anfang des maximal zu bestimmenden Präfix h_j (durch den Gebrauch des Suffixlinks in S) streichen, wenn wir keine Übereinstimmung mit dem nächst zu untersuchenden Zeichen x_j finden.

1. Zuerst versuchen wir, x_j von der Stelle an zu lesen, an der wir uns mit q in S gerade befinden. Dazu erhöhen wir at um eins.

2. Bei dem Erreichen eines neuen Knotens müssen wir einen neuen erweiterten Ort für das zu bildende h_j suchen. Dieser Versuch kann fehlschlagen, und wir müssen so lange von h_{j-1} einen Anfangsbuchstaben entfernen und den Suffixlink nutzen, bis wir Erfolg haben.

3. Wir prüfen, ob der in el für die Kante beschriebene Teilstring an der Stelle $at+f$ mit x_j übereinstimmt. Schlägt dieser Versuch fehl, so müssen wir einen Anfangsbuchstaben in h_{j-1} streichen und über den Suffixlink des kontraktierten Ortes zu h_{j-1} unser Glück an anderer Stelle versuchen.

Das wiederum kann zum Rescanning der Kantenbeschriftung führen, an der wir den *mismatch* festgestellt haben. Ist w verbraucht, gehen wir zu (2), sonst iterieren wir (3). Wenn wir terminieren, wird h_j festgelegt als $el.m(f + at - cl.m, f + at)$, was wiederum der Beschriftung des Weges entspricht, an dem wir das Zeichen x_j letztendlich einlesen konnten.

Programm 5.10 δ: Der Übergang von einem Suffixzustand in einen anderen zur inkrementellen Suche eines Teilstrings. Eingabe: Suffix Zustand q, einzulesendes Zeichen x_j. Ausgabe: Suffix Zustand q'.

```
at ← (at + 1) mod |el.l − el.f + 1|          {initialisiere h_j: i_j ← i_{j−1}}
if (at = 0) return SearchChild(q, x_j)       {neuer Knoten wird erreicht}
while (el.m[at + f] ≠ x_j)                   {solange Zeichen nicht paßt}
  if (cl.IsRoot) at ← at − 1                 {Spezialfall, cl ist Wurzel}
  w ← el.m(f, f + at − 1)                    {w ist Teilstring auf Kante {cl, el}}
  cl ← cl.sl                                 {streiche Buchstaben, i_j ← i_j + 1}
  if (Rescan(w, at, cl, el))                 {lese w erneut ein}
    return SearchChild(q, x_j)               {w erschöpft Kantenbeschriftung}
h ← el.m(f + at − cl.len, f + at)            {maximaler Suffix gefunden}
```

Die Verfeinerung von Punkt 2 (siehe Prozedur 5.11) ergibt folgendes Ablaufschema:

2a. Versuche einen Nachfolger *el* zu finden, dessen Beschriftung mit x_j beginnt.

2b. Falls ein Nachfolger gefunden ist, so lege h_j durch $el.m(f - cl.len, f)$ (also durch den Weg von der Wurzel in S zu dem ersten Zeichen auf der in *el* einlaufenden Kante) fest. Sonst benutze den Suffixlink und gehe zu (2a).

Ein Rescanning erfolgt in diesem Fall nicht, da Punkt 2 nur ausgeführt wird, wenn *at* gleich null ist.

Satz 20 *(Korrektheit δ) Sei x aus Σ^n bis x_{j-1} gelesen. Der Rückgabewert h_j von $\delta(x_j)$ ist der längste Suffix von $x(1, j)$, der auch Teilstring eines m im Multi-Suffixbaum ist.*

Beweis: Der Korrektheitsbeweis erfolgt induktiv. Das heißt, h_{j-1} ist der längste Suffix von $x(1, j - 1)$, der auch Teilstring eines Strings m im Multi-Suffixbaum S ist. Sei i_j definiert als der Anfang von h_j in x, dann ist h_{j-1} längster Teilstring in S mit $h_{j-1} = x(i_{j-1}, j - 1)$.

Der Algorithmus δ nutzt eine Reihe von Suffixlink- und Rescanningaufrufen, um letztendlich h_j mit $el.m(f + at - cl.len, f + at)$ festzulegen. Damit ist h_j Teilstring eines der in S gespeicherten Zeichenketten.

Bei einem Rescanning muß der String w von der Stelle des genutzten Suffixlinks an in S eingelesen werden können, da zu jedem Suffix in S auch alle weiteren Endstücke enthalten sind.

Programm 5.11 *SearchChild*: Suche einen anliegenden Kindknoten im Multi-Suffixbaum, der mit einem einzelnen Zeichen übereinstimmt. Eingabe: Suffix Zustand q, einzulesendes Zeichen x_j. Ausgabe: Suffix Zustand q'.

```
cl ← el; el ← cl.getChild(x_j)        {versuche Nachfolger zu finden}
while (not el and el ≠ cl)            {solange kein Nachfolger existiert}
  cl ← cl.sl                          {streiche Buchstaben, i_j ← i_j + 1}
  el ← cl.getChild(x_j)               {versuche Nachfolger zu finden}
if (el = cl)                          {Sonderfall, el ist Wurzel}
  at ← −1; h ← ϵ                      {initialisiere Suffixzustand}
else h ← el.m(f − cl.len, f)          {maximaler Suffix gefunden}
```

Annahme, der Suffix $x(i_j, j)$ von $x(1, j)$ wäre nicht maximal.

Das Abbruchkriterium für δ ist $x_j = el.m[f + at]$. Durch die Verwendung von Suffixlinks und dem damit verbundenen Abstreifen der Anfangsbuchstaben werden alle Teilstrings von $x(i_{j-1}, j)$ bis $x(i_j, j)$ überprüft, ob sie ein Präfix eines im Suffixbaum vorkommenden Strings bilden.

Damit ist k kleiner als i_{j-1}. Dies führt jedoch zu einem Widerspruch zur Maximalität von $x(i_{j-1}, j-1)$, denn $x(k, j-1)$ ist dann auch ein Suffix von $x(1, j-1)$, der länger ist als $x(i_{j-1}, j-1)$ und einen String in S repräsentiert. □

Satz 21 *(Laufzeit δ) Sei x aus Σ^n bis x_{j-1} gelesen. Die amortisierte worst case Zeit für $\delta(x_j)$ ist konstant.*

Beweis: In der amortisierten Analyse wird gezeigt, daß der Gesamtaufwand für die Suche nach Teistrings in x durch $O(|x|)$ beschränkt ist.

Die Laufzeit eines Aufrufes der Prozedur δ ergibt sich aus der Summe der Anzahl der Aufrufe von (2a) und (3) (entsprechend den Schleifendurchläufen in δ und *SearchChild*) und des gesamten Rescanningaufwandes.

1. Zur Bestimmung der Gesamtanzahl der Aufrufe von (2a) und (3).

   In jeder Iteration wird zumindest ein Anfangsbuchstabe von h_{j-1} bei der Bestimmung von h_j durch die Verwendung des Suffixlinks abgestreift. Sei i_j der Anfang von h_j in x. Dann ist die Anzahl der Aufrufe von (2a) und (3) durch $i_j - i_{j-1}$ beschränkt, wobei i_0 initial auf eins gesetzt wird. Summiert man diesen Wert für $j \in \{1, \ldots, |x|\}$ auf, so kürzen sich die meisten Terme heraus und wir erhalten $i_{|x|} - i_0$. Jeder Anfangsindex eines Teilstrings von x ist durch die Länge von x beschränkt.

2. Zur Bestimmung des gesamten Rescanningaufwandes.

   Die Gesamtanzahl der Operationen in *Rescan* ist beschränkt durch die Anzahl der Knoten, die bei den Rescanning-Schritten traversiert werden. Die Länge des Teilstrings nimmt demnach in jedem Schritt um die Länge der durch die Kante beschriebenen Zeichenkette ab. Diese Länge

wird in der Variablen at verwaltet. Sei at_j der Inhalt von at nach der Bestimmung von h_j. Da at nur am Anfang der Prozedur um den Wert eins erhöht wird, ist der Rescanningaufwand durch $1+at_{j-1}-at_j$ beschränkt, wobei wir hier at_0 mit -1 festlegen. Die Summe der Werte $1+at_{j-1}-at_j$ für $j \in \{1,\ldots,|x|\}$ ergibt teleskopartig den Wert $at_0 - at_{|x|} + |x|$. Da at durch $|x|$ beschränkt ist, benötigen wir für das gesamte Rescanning weniger als $O(|x|)$ Zeit.

□

Chang und Lawler beschreiben ein der vorgeschlagenen on-line Suche ähnliches Verfahren *matching statistics*, daß die längsten Anfangsstücke h_i von den Stellen i in x aus sucht (Chang & Lawler 1994). Sie nutzen, daß die Endemarkierung $k = i + |h_i|$ monoton wächst. Bevor das nächste Zeichen x_{k+1} angeschaut wird, ist der längste Teilstring bis zu x_k gefunden.

Nichtsdestotrotz bewegt sich in ihrem Ansatz ein weiterer Zeiger j zwischen i und k, der dazu führt, daß, entgegen unserer Anforderung an eine on-line Suche, zusätzlich zu x_k auf x_j mit $j < k$ zugegriffen wird.

5.7.2 Suche aller Teilstrings

In diesem Abschnitt suchen wir *alle* gespeicherten Teilstrings in S, die in einer Anfrage enthalten in. Demnach macht es keinen Sinn mehr, den Multi-Suffixbaum S teilstringfrei zu verwalten.

Amir et al. bezeichnen diese Problemstellung als das *dictionary prefix problem*, da sie zusätzlich zu dem gefundenen längsten Präfix h auch alle Anfangsstücke von h darauf testen, ob sie durch einen String aus S dargestellt werden können (Amir et al. 1995).

Unsere Aufgabe könnte somit als *dictionary suffix problem* bezeichnet werden und steht dual zum *dictionary prefix problem*. Dennoch kann uns die Suffixbaumstruktur auf der Suche nach den Suffixen unterstützen.

Definition 7 *Ein String h heißt akzeptierend, wenn der Ort des um das Zeichen \$ erweiterten Strings h akzeptierend ist, d.h. einen vollen String m\$ repräsentiert. Der String h heißt genau akzeptierend, wenn der Ort von h existiert.*

Sei i_j das Anfangszeichen von h_j und $h(i,j)$ der Suffix von h_j, der durch das Abstreifen des Anfangsbuchstabens von h_j entsteht, $i \in \{i_j,\ldots,j\}$. Dann kann Strategie *FindSuffix* (bei Initialisierung von i als i_j) zur Lösung des *dictionary suffix problem*s wie folgt beschrieben werden:

1. Bestimme den kontraktierten Ort cl von $h(i,j)$ und den Teilstring w, der sich durch die Kontraktion ergibt.
2. Nutze den Suffixlink von cl und lese w neu ein. Prüfe ob $h(i+1,j)$ akzeptierend ist. Gebe in diesem Fall die Stringrepräsentation von el aus.
3. Erhöhe i um eins und fahre mit Punkt 1 fort.

In dem Fall, daß ein $h(i,j)$ genau akzeptiert, ist das Rescanning vollbracht und wir können über die Suffixlinks direkt die \$ Kanten auf Akzeptanz testen.

Satz 22 *(Suche aller Teilstrings) Die Suche im dynamic dictionary matching problem wird durch die Kombination der inkrementellen Suche mittels δ und FindSuffix in $O(|x| + \sum_{j=1}^{|x|} |h_j|)$ gelöst. Dabei ist h_j der maximale Suffix von $x(1, j)$, der gleichzeitig ein Teilstring eines der im Multi-Suffixbaum S gespeicherten Strings ist.*

Beweis: Aufgrund Satz 21 bleibt nur zu zeigen, daß die Prozedur *FindSuffix* zur Berechnung aller Suffixe von h_j insgesamt $O(|h_j|)$ viele Schritte benötigt.

Der Gesamtaufwand des Rescannings von w in *FindSuffix* ist durch $O(|w|)$ und die Anzahl der Suffixlinkoperationen durch $O(|h_j|)$ beschränkt. □

Hilfssatz 5 *Ein String m in M ist dann, und nur dann, Teilstring eines weiteren Strings aus M, wenn ein innerer Knoten als Ort von m in S existiert.*

Beweis: Wenn m Teilstring von m' ist, dann ist m größter gemeinsamer Präfix von $m\$$ und eines Suffixes von $m'\$$. Aufgrund Satz 15 existiert der Ort von m.

Wenn der Ort von m in S existiert und innerer Knoten ist, dann gibt es zumindest einen Blattknoten als Ort eines Strings ungleich $m\$$ in dem Teilbaum, der durch diesen Knoten beschrieben wird. Der dort repräsentierte String ist Superstring von m. □

Wir wollen eine mögliche Alternative *FindPrefix* zu der Lösung des *dictionary prefix problems* angeben. Dabei sei h_j, $j \in \{1, \ldots, |x|\}$, der längste Prafix von $x(j, k)$ und die Position von h_j im Multi-Suffixbaum durch die Lösung des *dictionary representative problems* bekannt. Für alle j aus $\{1, \ldots, |x|\}$

1. Bestimme den Pfad p von der Wurzel des Multi-Suffixbaumes zu dem kontraktierten Ort von h_j.
2. Traversiere p bottom-up und gebe die exakt akzeptierenden Teilstrings auf p aus.

Satz 23 *(Korrektheit FindPrefix) Der Algorithmus FindPrefix ist korrekt, d.h. er bestimmt alle Präfixe der gegebenen Eingaben h_j, $j \in \{1, \ldots, |x|\}$, die einen vollständigen String im Multi-Suffixbaum S darstellen.*

Beweis: Der Pfad p repräsentiert alle Präfixe zum Suffix h_j. Weiterhin kann nach Hilfssatz 5 ein String m aus S nur dann Teilstring eines anderen Strings m' sein, wenn der Ort für m existiert. □

Satz 24 *(Laufzeit FindPrefix) Sei als Modellannahme der Multi-Suffixbaum S für die Darstellung der Menge $M\$$ balanciert, d.h. die Tiefe von S durch $O(\log d)$ beschränkt. Dabei ist d die Gesamtlänge der Strings m aus M. Dann benötigt der Algorithmus FindPrefix zur Lösung des dictionary prefix problems $O(|x| \log d)$ Schritte.*

Beweis: Die Traversierung des Pfades p benötigt $O(\log d)$ Schritte, da jeder Knoten des Pfades in (3) genau einmal untersucht wird und die Länge von p nach der Annahme durch $O(d)$ beschränkt ist. Die Schritte (1)-(3) des Algorithmus *FindPrefix* werden $|x|$ mal aufgerufen. □

Wir können jedoch durch eine geringe Modifikation des Algorithmus *FindPrefix* eine Schranke von $O(d)$ gewinnen. Auf der Suche nach genau akzeptierenden Knoten müssen wir jeden Knoten auf der Suche nach allen Teilstrings von h_j maximal einmal besuchen.

Folgerung 6 *Die Suche im DDMP kann durch Vermeidung von doppelt besuchten Knoten im Algorithmus FindSuffix in $O(|x|+d)$ gelöst werden, wobei d die Summe der Längen der im Multi-Suffixbaum S verwalteten Strings angibt.*

Abschließend bauen wir eine Brücke zwischen dem den Suffixbäumen und dem Fehlerfunktionansatz von Aho und Corasick bzw. Meyer.

5.8 Multi-Suffixbäume als Fehlerfunktion

Der Zusammenhang ist durch die Baumstrukturen in Abbildung 5.5 und 5.2 motiviert. Diese Bäume sehen einander sehr ähnlich, insbesondere, wenn die Blätter, in die eine $-Kante einläuft, mit ihren Elternteilen verbunden werden.

Der Trie in Abbildung 5.1 ist mit dem Trie aus Abbildung 5.6 bis auf die Wurzel deckungsgleich. Wir werden ihn deshalb in beiden Fällen mit T bezeichnen.

Satz 25 *(Fehlerfunktion im Multisuffixbaum) Sei M eine Menge von Strings und T der aus dieser Menge konstruierte Mehrwegsuchbaum. Sei $v(t)$ der assoziierte Teilstring zum Trieknoten t aus T. Weiterhin sei S der zur Menge $(\$M)^R$ konstruierte Multi-Suffixbaum. Wenn der Ort von $v(t')^R$ eines Knotens t' aus T, kurz $l(v(t')^R)$, im Multi-Suffixbaum S existiert und in dieser Eigenschaft tiefster Vorfahre von $l(v(t)^R\$)$ ist, dann gilt $f(t)$ gleich t'.*

Beweis: Da $l(v(t')^R)$ ein Vorfahre von $l(v(t)^R\$)$ ist, ist $v(t')^R$ Präfix von $v(t)^R$ und somit ist $v(t')$ Suffix von $v(t)$.

Zur Erinnerung: Der Wert $f(t)$ ist definiert durch den Knoten t' aus T, dessen Darstellung $v(t')$ gleichzeitig maximaler Präfix eines Strings aus T als auch Suffix von $v(t)$ ist.

Bleibt zu zeigen, daß $v(t')$ maximal ist. Bei gegenteiliger Annahme finden wir eine Verlängerung $v(t'')$ von $v(t')$, die einem Präfix von einem String in T und gleichzeitig einem Suffix von $v(t)$ entspricht. Somit ist $v(t'')^R$ Präfix von $v(t)^R$, und nach Satz 16 (Bijektion Trie-Suffixbaum) existiert der Ort von $v(t'')^R\$$ in S. Weiterhin ist $v(t'')$ eine Verlängerung von $v(t')$, also $|v(t'')^R| > |v(t')^R|$. Da sowohl $v(t')^R$ als auch $v(t'')^R$ Präfixe von $v(t)^R$ sind, ist der Ort $l(v(t'')^R)$ tiefer gelegen als der Ort $l(v(t')^R)$. Dies ist ein Widerspruch dazu, daß $l(v(t')^R)$ tiefster Vorfahre von $l(v(t)^R\$)$ im Multi-Suffixbaum S ist. Somit gilt $f(t) = t'$. □

Der Satz 25 ermöglicht uns, den Suffixbaum zur effizienten Berechnung der Fehlerfunktion einzusetzen. Wie in Abbildung 5.5 ersichtlich, benötigt die Berechnung der Fehlerfunktion in vielen Fällen nur konstante Zeit. Ausnahmen sind allerdings möglich, wie man sich am Beispiel $M = \{(011\ldots1)\}$ verdeutlichen kann.

Damit läßt sich der Speicherplatz für die inkrementelle Teilstringsuche weiter verringern, indem nicht der Suffixzustand, sondern die Position im Trie vermerkt wird. Dies ist in der Zustandsraumsuche sehr wünschenswert, da der Zustand in den meisten Suchverfahren an jeden gespeicherten Suchbaumknoten angehängt wird.

5.9 Experimente

Wir haben den Multi-Suffixbaumakzeptor als *c++*-Klasse für die Verwaltung von Zeichenketten implementiert. Als Beispiel haben wir ein englisches Wörterbuch mit 25143 Einträgen verwendet, das wir für das Betriebssystem Unix (unter */usr/dict/words*) gefunden haben. Aufgrund der lexikalischen Ordnung innerhalb der Datei haben wir diejenigen Wörter (vom Fileende ausgehend) eingefügt, die keine Verkürzung und auch keine Verlängerung innerhalb des schon bestehenden Wörterbuchs hatten.

Die Teilstringsuche wurde mit dem (inkrementellen) Ansatz aus Abschnitt 5.7.1, die Superstringsuche durch einen Scanaufruf durchgeführt. Genau 14344 der 25143 englischen Worte waren teilstringfrei und wurden in die Datenstruktur eingefügt. Der Multi-Suffixbaum besaß 69207 Knoten, und der auf die Blätter verweisende Trie bestand aus 48854 Knoten. Folglich hatte der Suffixbaum nur 20353 oder in etwa 29 Prozent innere Knoten.

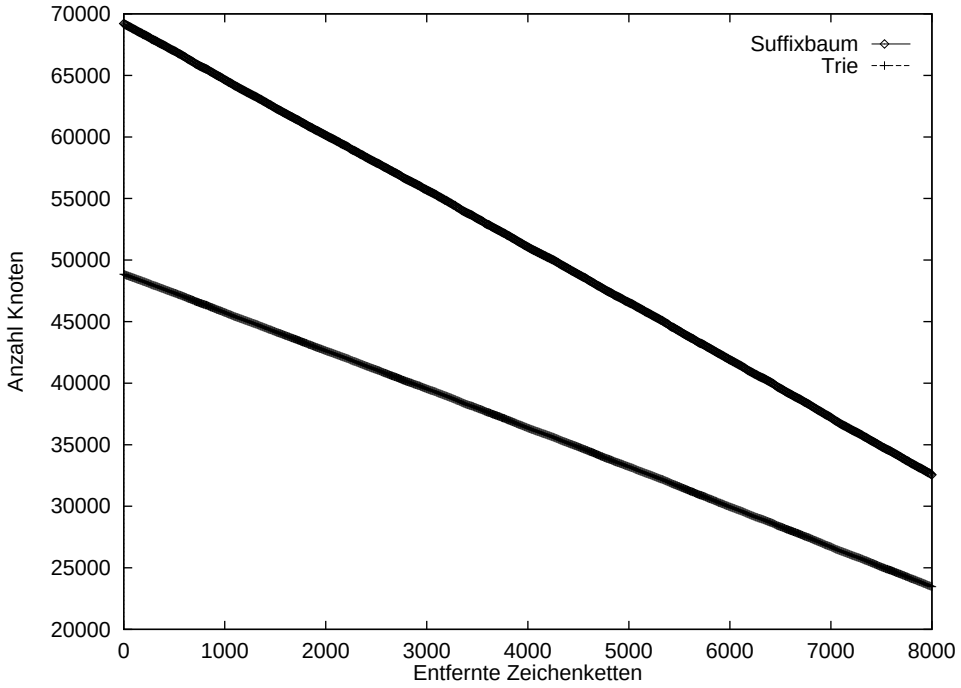


Abbildung 5.13: Experimentelle Daten zum Entfernen von Knoten im Multi-Suffixbaum.

In Abbildung 5.13 wird die Abnahme der Knotenanzahlen dargestellt, die sich im Suffixbaum und im Trie durch das Löschen von 6000 zufällig gezogenen Zeichenketten ergab. Dabei haben wir den im Abschnitt 5.6 beschriebenen Entfernungsalgorithmus verwendet, der eine optimale Speicherplatzausbeute garantiert. Trotz der Diversität des Korpus und der zufälligen Wahl der zu entfernenden Zeichenketten, ist der Funktionsverlauf augenscheinlich linear.

Kapitel 6

Lernen von Abkürzungen

In diesem Kapitel studieren wir die Gemeinsamkeiten von Zugsequenzen innerhalb des Suchraumes, die sich durch verschiedene Wege zu den Zuständen ergeben. Dabei gehen wir von einer endlichen Anzahl von Operatoren aus. Den längeren Weg zu einem Zustand bezeichnen wir als Duplikat, den kürzeren als Abkürzung. Ziel ist es, einen (möglichst speicherplatzsparenden) Akzeptor zu bauen, der uns parallel zur Suche bei der Erkennung eines ausgeführten Duplikats eine Abkürzung garantiert.

6.1 Abkürzungen im Suchraum

Wir Menschen erkunden die Welt eher in der impliziten Problemsicht, indem wir Aktionen durchführen und uns an die Reihenfolge der Aktionen erinnern. So wird eine Frage eines Autofahrers nach dem geeigneten Weg etwa wie folgt beantwortet: *Dritte Straße rechts, an der nächsten Ampel links, über den Bahnübergang, den Wald entlang, bis zur zweiten großen Kreuzung usw.*.

Sollen wir hundert Schritte voraus und dann hundert Schritte nach rechts gehen, so haben wir direkt eine Abkürzung im Sinn, die uns diagonal schneller zum Ziel bringt. Auch wenn eine Häuserwand uns bei diesem Versuch behindern sollte, das Konzept der Abkürzung falsifizieren wir dabei nicht, hat es uns doch schon so häufig geholfen. Abkürzungen sind so verlockend, daß eine Henne nicht in der Lage ist, um einen Zaun herumzugehen, wenn das Futter schon im Gesichtsfeld lockt (Koehler 1927).

Die Stärke des Verlangens nach einer Abkürzung läßt sich in dem Löseverhalten des Menschen belegen, der versucht, den Abstand zwischen der aktuellen Position und dem Ziel möglichst rasch zu minimieren. In dem *Kannibalen und Missionare Problem* geht es darum, drei Kannibalen (C) und drei Missionare (M) mit einem Boot über einen Fluß zu bringen, so daß die Missionare an keinem Flußufer in der Minderzahl sind. Die Lösung des Problems ist gegeben durch MC(hin), M(zurück), CC(hin), C(zurück), MM(hin), MC(zurück), MM(hin), C(zurück), CC(hin), M(zurück), MC(hin), wobei ein Zug jeweils durch die Belegung des Bootes gegeben ist. Menschen (Greeno 1974; Jeffries et al. 1977) haben Schwierigkeiten, den Zug MC(zurück) auszuführen, der zwei Personen entgegen der gewünschten Richtung befördert. Das bewährte Schema *zwei hin, einen zurück*, geht nicht auf. Eine einfache Symmetriebetrachtung

der Situation belegt aber, daß zur Lösung ein Zug durch diesen Flaschenhals immer notwendig ist.

Menschen vermeiden es, zu früheren Problemzuständen zurückzukehren, selbst wenn diese zur Lösung des Problems notwendig wären (Anderson 1995). Die Begründung liegt nach unserem Dafürhalten in der Erkenntnis, daß eine optimale Lösung keine Kreise enthält und demnach eine Abkürzung existieren muß.

In der Zustandsraumsuche werden wir durch die Größe der Problemräume gezwungen, eine implizite Sichtweise anzunehmen. Da die Regelmenge der von uns untersuchten Probleme viel kleiner als der aufgespannte Suchraum ist, entspricht ein Zug der Auswahl von Operatoren aus einer endlichen Menge, die wir mit Σ bezeichnen. So ist z.B. im (n^2-1)-Puzzle Σ durch die Bewegungen des Leerfeldes nach *links*, *rechts*, *oben* und *unten* gegeben. Wir fassen eine Operatorsequenz m als eine Abbildung der Zustandsmenge in sich auf und schreiben kurz $m(q)$.

Wie wir in Kapitel 3 gesehen haben, ist das hohe Ziel, alle erneut erreichten Zustände innerhalb der Zustandsraumsuche aufzuspüren, angesichts der großen Zustandsräume durch einfache Speicherung der Zustände innerhalb einer Tabelle nicht zu erreichen. In diesem Abschnitt versuchen wir, das Problem zu umgehen, in dem wir die unterschiedlichen Operatorfolgen betrachten, die zu ein und demselben Zustand führen. Dabei ergibt sich immer ein kürzerer Weg, den wir als als *Abkürzung* bezeichnen, und zumindest ein längerer Weg, den wir *Duplikat* nennen.

In reversiblen Problemräumen hat jeder Operator ein Inverses, so daß eine Abkürzung zusammen mit einem Duplikat als ein Zyklus aufgefaßt werden kann. Unser Wunsch ist es nun, daß der Zyklus uns unabhängig von der gegebenen Situation immer wieder zum Ausgangszustand zurückbringt, also die lokalen Erkenntnisse auf den gesamten Suchraum zu verallgemeinern.

Betrachten wir die Sequenzen von Zügen innerhalb des *Acht*-Puzzles genauer. Wir kürzen die Zugrichtungen mit L für *links*, R für *rechts*, U für *oben* (engl. *up*) und D für *unten* (engl. *down*) ab. Wir definieren eine lexikographische Ordnung auf den Zügen, in der U kleiner R kleiner D kleiner L gilt. Das (n^2-1)-Puzzle ist reversibel, da L invers zu R und U invers zu D ist. Als Beispielzyklus wählen wir RDLURDLURDLU, der einer dreimaligen Umrundung des Leerfeldes auf einem zwei mal zwei großen Teilbrett entspricht. Als Duplikat halten wir DRULDR fest, worauf sich für die Abkürzung RDLURD ergibt (vergl. Abbildung 6.1).

Die grundlegende Idee ist es, den Suchraum immer dann zu beschneiden, wenn wir auf dem Endstück des Generierungspfades zu einem gegebenen Zustand ein Duplikat erkennen. Wir müssen dann eine Abkürzung garantieren, über die der derzeitige Zustand zuvor besucht wurde oder später noch besucht wird. Um diese Verallgemeinerung zu gewährleisten, fordern wir von einem Duplikat und der vorgeschlagenen Abkürzung die folgenden drei Eigenschaften.

(D1) Das Gewicht des Duplikats ist größer gleich dem Gewicht der Abkürzung. Bei einer Gleichgewichtung ist die lexikographische Ordnung des Duplikats auf der Menge der Züge größer als die der Abkürzung.

(D2) Für alle Elemente des Zustandsraumes läßt sich die Abkürzung immer dann durchführen, wenn ein Duplikat vorliegt.

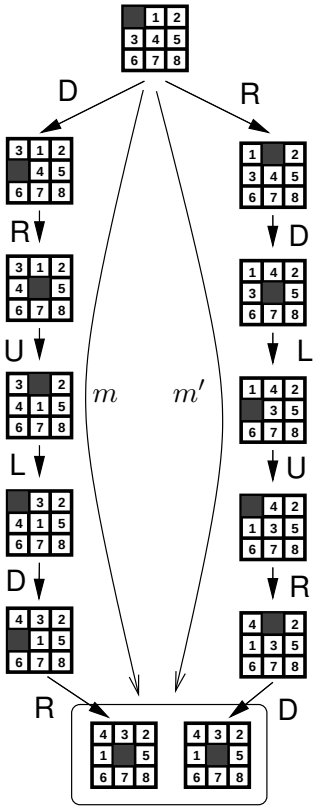


Abbildung 6.1: Ein Duplikat und eine Abkürzung.

(D3) Für alle Elemente des Zustandsraumes führt die Anwendung des Duplikates und der Abkürzung zu dem gleichen Zustand.

Aus (D1) folgt, daß sich Duplikat und Abkürzung unterscheiden. So ist der Operator R im *Acht*-Puzzle kein Inverses zum Operator R, obwohl RL einen Zyklus bildet. Um jedoch Vorgänger zu erkennen, wird RL als ein Duplikat der leeren Sequenz ϵ angesehen.

Die Bedingung (D2) gilt in dem Beispiel der durch eine Mauerwand versperrten Abkürzung nicht. Falls die Eigenschaft nicht für den gesamten Suchraum gewährleistet werden kann, müssen wir sie explizit bei der Entdeckung eines Duplikates prüfen.

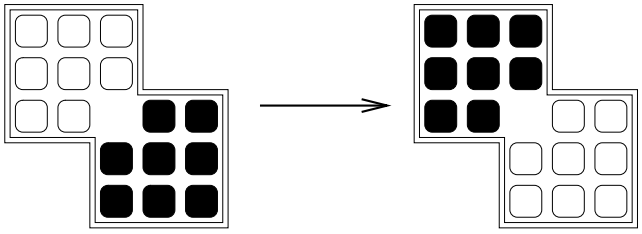


Abbildung 6.2: Das Seemannsspiel.

Auch (D3) erweist sich als essenziell. Dazu betrachten wir das *Seemanns-*

spiel aus Abbildung 6.2 (Gardner 1959).

Ziel dieses Spieles ist, die schwarzen und weißen Steine miteinander zu vertauschen. Ein Stein kann von seinem Feld auf ein angrenzendes Leerfeld geschoben werden oder über ein anliegendes besetztes Feld springen. Wie üblich bezeichnen wir eine Verschiebung des Leerfeldes nach rechts bzw. nach links mit R bzw. L und einen Sprung (engl. jump) nach rechts bzw. links mit JL bzw. JR. Im Ausgangszustand ist R JL kein Duplikat zu L. Sind die beteiligten Steine allerdings gleichfarbig, so ist R JL ein Duplikat zu L.

Das Seemannsspiel bis dato wissenschaftlich nicht untersucht worden. Die Anzahl der Spielstellungen ist mit $17!/(8! \cdot 8!) = 218790$ recht gering, doch der Verzweigungsgrad ist mit bis zu 8 verschiedenen Zügen sehr groß. Die optimale Lösung von 46 Zügen wurde auf einer Sun Ultra Workstation mit dem *HSF-Light* System in weniger als einer Sekunde gefunden. Dabei wurde das A^* Verfahren genutzt und 3272 Knoten expandiert. Die detaillierte Lösung findet sich in Anhang A.

Die Variante des Seemannsspiels mit 31 Feldern, 15 schwarzen und 15 weißen Spielsteinen besitzt immerhin $31!/(15! \cdot 15!) \approx 4.8 \cdot 10^9$ verschiedene Stellungen. Sie ist in 117 Zügen lösbar: L, JR, R, JL, JU, L, JD, JR, JD, R, JU, JL, U, JD, JR, JD, U, JU, JL, JU, U, JD, JL, JU, JR, L, D, JD, JR, JR, JD, D, L, U, JU, JL, JU, JL, R, JD, JR, JR, JD, L, JU, JL, JL, R, JR, JR, L, JL, D, JU, JL, D, L, JR, JR, D, R, JL, JU, U, JD, JD, D, JU, JU, JU, D, JD, JD, U, JU, L, JL, JU, JR, D, JD, JR, D, JR, L, JL, JU, L, JL, D, JU, JR, JD, JR, D, L, JU, L, D, JR, JD, D, L, JU, JU, D, JL, L, JR, JR, JD, L, JU, JL, R, JR und L. Dabei haben wir angenommen, daß sich die Spielsteine in der Lösung sich nicht entgegen ihrer Zielfelder bewegen müssen.

Als Heuristik haben wir die Manhattandistanz der gleichfarbigen Spielsteine durch zwei geteilt: $(|\sum_{i\in W} x_i - \sum_{i\in W} x'_i| + |\sum_{i\in W} y_i - \sum_{i\in W} y'_i| + |\sum_{i\in B} x_i - \sum_{i\in B} x'_i| + |\sum_{i\in B} y_i - \sum_{i\in B} y'_i|)/2$. Dabei ist W die Menge der weißen und B die Menge der schwarzen Spielsteine. Ein Paar (x, y) stellt die Koordinaten eines Spielsteines im gegebenen Zustand und (x', y') in der Zielstellung dar.

Verbessert wurde der Schätzwert durch die Anzahl der Einfachzüge, die in einer Sequenz von Doppelzügen notwendigerweise vorhanden sind, da diese jeweils nur einen Teil der Felder abdecken. Ein zweifaches Matching der Spielsteine einer Farbe auf ihre Zielfelder lieferte hingegen eine schlechtere untere Schranke. Mit A^* wurden in *HSF-Light* für die 31 Felder-Variante in ca. 30 Sekunden insgesamt 2138818 Knoten expandiert.

6.2 Suchraumbeschneidung mit endlichen Automaten

Betrachten wir das Beispiel des $(n^2 - 1)$-Puzzles mit den Operatoren U, R, D und L für die Bewegung des Leerfeldes. Die Vorgängerelimination innerhalb dieses Puzzles kann durch den in Abbildung 6.3 dargestellten endlichen Automat durchgeführt werden.

Wir starten in dem Zustand in der Mitte und verfolgen für eine gegebene Zugsequenz jeweils den mit dem derzeitigen Zug beschrifteten Pfeil. Offensichtlich lassen sich keine Züge direkt an ihr Inverses anschließen. In dem Automat gehen wir davon aus, daß ein akzeptierender Zustand immer dort erreicht wird, wo kein in die Zugrichtung weisender Pfeil existiert.

Der Automat kann parallel zur Suche genutzt werden, um die Suche jedesmal dann zu beschneiden, wenn sich ein akzeptierender Zustand findet.

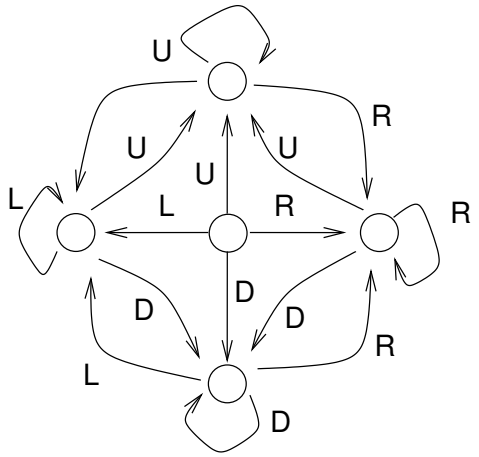


Abbildung 6.3: Automat zur Vorgängerelimination der Züge U, R, D und L .

In Kapitel 10 werden wir sehen, wie man die sich ergebende Größe des durch diesen Automat beschnittenen Suchbaumes ermitteln kann. Für das *Fünfzehn*-Puzzle erhalten wir durch die Vorgängerelimination eine Verbesserung des zu erwartenden Verzweigungsgrads von 3.23 auf 2.13.

Als weiteres Beispiel interessieren wir uns für das *Gitter* (vergl. Abbildung 6.4), sprich den unendlichen Graphen, bestehend aus der Gesamtzahl aller ganzzahligen Paare. Zwei Paare sind genau dann miteinander verbunden, wenn sie sich in genau einer Komponente um eins unterscheiden. Die vier möglichen Bewegungen innerhalb des Gitters werden wieder mit U, R, D und L bezeichnet.

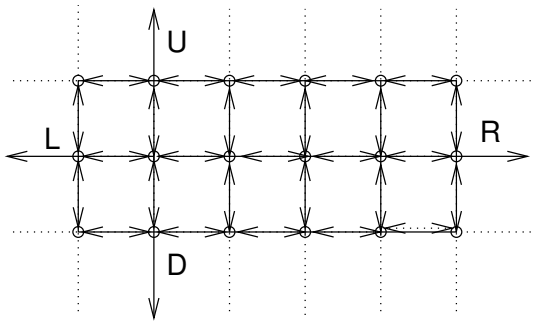


Abbildung 6.4: Das *Gitter*.

Der in Abbildung 6.5 dargestellte Automat erlaubt sowohl keine Bewegung in die Gegenrichtung der gerade gemachten wie auch keine Bewegung L oder R, wenn U oder D gezogen sind.

Damit wird, von einem beliebigen Punkt des Gitters ausgehend, jede Position auf genau eine Weise erreicht. Der ursprüngliche Suchbaum zur Tiefe d des Gitters hat die Größe 4^d, da jeder Gitterpunkt jeweils vier Nachfolger erzeugt. Ein durch den Automat aus Abbildung 6.5 beschnittener Suchbaum besitzt hingegen nur $4d$ Knoten in Tiefe d. Also führt dieser aus fünf Zuständen bestehende Automat zu einer optimalen Beschneidung des unendlichen Such-

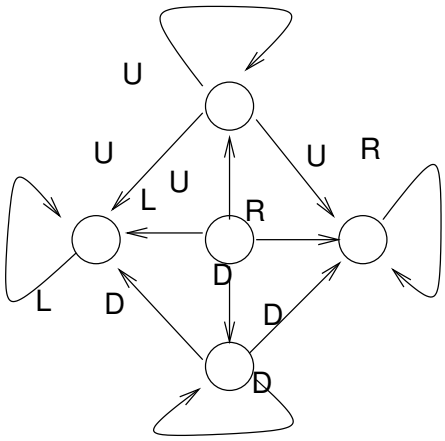


Abbildung 6.5: Ein Automat zur Beschneidung des *Gitters*.

baumes, die über ein rein auf Zustandsvergleiche basierendes Verfahren nicht möglich ist.

6.2.1 Taylor und Korf

Taylor und Korf haben das Aufspüren von Duplikaten innerhalb des Zustandssuchraumes und die Konstruktion eines endlichen Automaten als Akzeptor in der Suchphase erstmalig studiert (Taylor & Korf 1993).

Ihr Ansatz gliedert sich in zwei Teile: In der *Lernphase* werden in einem Breitensuchlauf alle Duplikate m und die zugehörigen Abkürzungen m' gefunden, deren Länge die vorgegebene Suchtiefe nicht überschreiten. Die Gleichheit der durch m und m' vom Startzustand enthaltenen Problemstellungen wird mittels einer Hashtabelle festgestellt.

Der Einfachheit halber nehmen die Autoren einen *unbeschränkten Zustandsraum* an, in dem jeder Zug immer durchführbar ist. Das *Gitter* oder der *Zauberwürfel* liefern uns gute Beispiel einer solchen Domäne.

Die Autoren nutzen den in Kapitel 5 besprochenen Algorithmus von Aho and Corasick, um aus der Menge der Duplikate einen endlichen Automat aufzubauen, der diese Zeichenketten als Teile innerhalb einer Anfrage, bestehend aus dem Generierungspfad eines einzelnen Knotens, erkennt. Die Duplikate dienen somit als Menge verbotener Teilworte innerhalb der Sprache aller durchführbaren Zugsequenzen.

Im Beispiel der Vorgängerelimination in dem Schiebepuzzle ist die Menge der Duplikate durch UD, RL, DU und LR gegeben. Für den Automat zur Beschneidung des Gitters standen zusätzlich noch die Duplikate UL, UR, DL und DR zur Konstruktion des Automaten Pate.

Die grundlegende Idee der Suchraumbeschneidung in der *Suchphase* ist die folgende: Wenn ein Knoten auf dem Generierungspfad liegt, der mit einem Duplikat abschließt, können wir uns die weitere Suche an dieser Stelle ersparen, da dieser Knoten innerhalb der Suche über den günstigsten Pfad auf jeden Fall noch erreicht wird. Ein einzelnes Duplikat kann somit Hunderte von Beschneidungen des Suchraumes bewirken.

6.2.2 Dissertation Taylor

Taylor hat aufbauend auf der Arbeit von Taylor und Korf seine Dissertation verfaßt (Taylor 1997), die wir im folgenden kurz zusammenfassen.

Aufbauend auf dem von ihm als Mitautor erschienenen Artikel referiert Taylor den Aufbau und den Nutzen des Stringerkennungsautomaten. Dabei hebt er insbesondere hervor, daß die Menge der Duplikate durch die Aufteilung von Kreisen erzeugt werden kann, die wiederum in einer *heuristischen Zyklenerkennungssuche* generiert werden: Ausgehend vom Startzustand wird mit einer richtungsweisenden Heuristik selbiger wieder gesucht, wobei sich bei steigender Suchtiefe immer größere Kreise ergeben. Ein einzelner Zyklus kann durch eine Verschiebung des Anfangspunktes und durch die Ausnutzung von Symmetrieeigenschaften des Zustandsraumes zu einer großen Menge von Duplikaten führen. Wir kommen in dem Abschnitt *Symmetrien* darauf zurück, wie die damit verbundene Vervielfachung der Automatzustände vermieden werden kann.

Die Automaten können zur Beschneidung des Suchraumes in der Aufbauphase genutzt werden. Der in einer iterierten Tiefensuche der Tiefe d konstruierte Automat kann somit auch in der Tiefe $d + 1$ genutzt werden. Bei der dynamischen Konstruktion des Automaten verbleibt Taylor jedoch in den Anfängen des durch Meyer vorgezeichneten Weges des inkrementellen Aufbaus des Zeichenkettenwörterbuches stehen.

Taylor zeigt in seiner Dissertation auf, daß die Methode der Suchbaumbeschneidung mit Hilfe des so konstruierten Automaten in dem $(n^2 - 1)$-Puzzle ohne Einbeziehung der vom Leerfeld benutzten Spur im Duplikat *und* in der Abkürzung nicht korrekt ist, da der Suchraum durch den Spielfeldrand restringiert ist. Auf der anderen Seite nutzt er den Automat ohne die Berücksichtigung der Spur. Demnach bieten wir im *HSF-Light* System (vergl. Anhang B) den Anwendbarkeitstest domänenabhängig an.

Der Autor gibt Ergebnisse der Suchbaumbeschneidung in verschiedenen Schiebe- und Zauberwürfelvarianten an, die über die in Kapitel 2 benannten Strategien, wie die der Vorgängerelimination, Reihenfolgetreue und Seitenexklusivität hinausgehen (vergl. Tabelle 6.1). Dabei steht *bfs* für den Aufbau des Automaten durch eine Breitensuche, *hcds* für eine heuristische Zyklenerkennungssuche (engl. heuristic cycle detection search) und *comb* für eine aus beiden Ansätzen kombinierte (engl. combined) Strategie. In *HSF-Light* konnten die Ergebnisse repliziert werden (Verzweigungsgrade 1.96 für das 15 Puzzle und 13.28 für den Zauberwürfel)

Taylor untersucht am Beispiel des *Tower-of-Hanoi* Puzzles (siehe Abbildung 6.6) Operatoren, deren Ausführung an zusätzliche Bedingungen geknüpft sind. In dem Puzzle sollen n Scheiben von der Stange A auf die Stange C durch Umlegen der Scheiben gebracht werden. Hierbei darf allerdings auf keiner der drei Stangen eine größere Scheibe auf eine kleinere gelegt werden. Taylors Idee ist es, nach der Anwendung eines Duplikates durch eine möglichst effiziente Abfrage explizit nachzuprüfen, ob die vorgeschlagene Abkürzung greift. Dabei werden die Einzelbedingungen der Operatoren miteinander verbunden.

Taylor analysiert das Abschneiden des Suchbaumes in parametrisierten Operatorsequenzen, wie sie insbesondere in Planungssystemen eingesetzt werden (Fikes & Nilsson 1971). Er befaßt sich weiterhin mit der Automatkonstruktion für Und/Oder Graphen und damit für Logikprogramme.

Puzzle Domäne	Lern- typ	Verzweigungs- Grad vorher	Anzahl Duplikate	Anzahl Zustände	Verzweigungs- grad nachher
8-Puzzle	bfs	1.732	35858	137533	1.39
15-Puzzle	bfs	2.130	16442	55441	1.98
15-Puzzle	hbs	2.130	58897	246768	1.96
19-Puzzle	comb	2.249	127258	598768	2.095
24-Puzzle	comb	2.368	127258	598768	2.235
2^3-Würfel	bfs	6.0	31999	24954	4.73
3^3-Würfel	bfs	13.34	28210	22974	13.26

Tabelle 6.1: Experimentelle Ergebnisse bei Taylor.

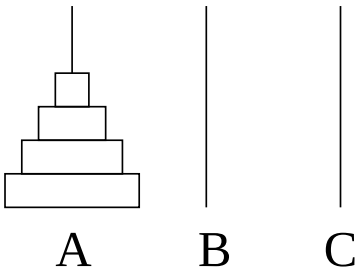


Abbildung 6.6: Das *Tower-of-Hanoi* Problem.

Abschließend vergleicht Taylor das Abschneiden des Suchbaumes mit einem speicherplatzbeschränkten Verfahren, das die Speicherung der vollen Knoteninformationen wenigstens eines Teiles der Knoten in die Suche einbezieht.

6.3 Codierung der Automatzustände

In diesem Abschnitt werden wir den Bereich einer speicherplatzeffizienten Darstellung des Automaten genauer ausloten. Dazu werden wir die üblicherweise als konstant aufgefaßte Anzahl der verschiedenen Zustandsübergänge $|\Sigma|$ explizit in unsere Analysen aufnehmen.

In manchen Zustandsräumen, wie dem allgemeinen Schiebepuzzle, können die Übergänge unterschiedlich häufig vorkommen. So ist im *Esel*-Puzzle (vergl. Abbildung 2.4 auf Seite 13) der 2×2 Block wesentlich unbeweglicher als der 1×1 große Spielstein.

Als Beispiel betrachten wir die Menge Y der Zeichenketten *she, hers* und *his*. Abbildung 6.7 zeigt den sich durch diese Strings ergebenden, in einem Array realisierten Trie. Wir haben in der Abbildung zusätzlich die Fehlerfunktion von Aho und Corasick (vergl. Kapitel 5) gestrichelt dargestellt. Der beanspruchte Speicher ist von der Größenordnung $O(|T_Y||\Sigma|)$.

Eine sehr speichereffiziente Darstellung des Tries ist eine Adjazenzliste (vergl. Abbildung 6.8). Der Speicheraufwand ist durch $O(|T_Y|)$ gegeben. Die Zugriffszeit auf den Nachfolgerknoten ist insbesondere bei großen Verzweigungsgraden wesentlich länger als in der Arraydarstellung, schlimmstenfalls $O(|\Sigma|)$.

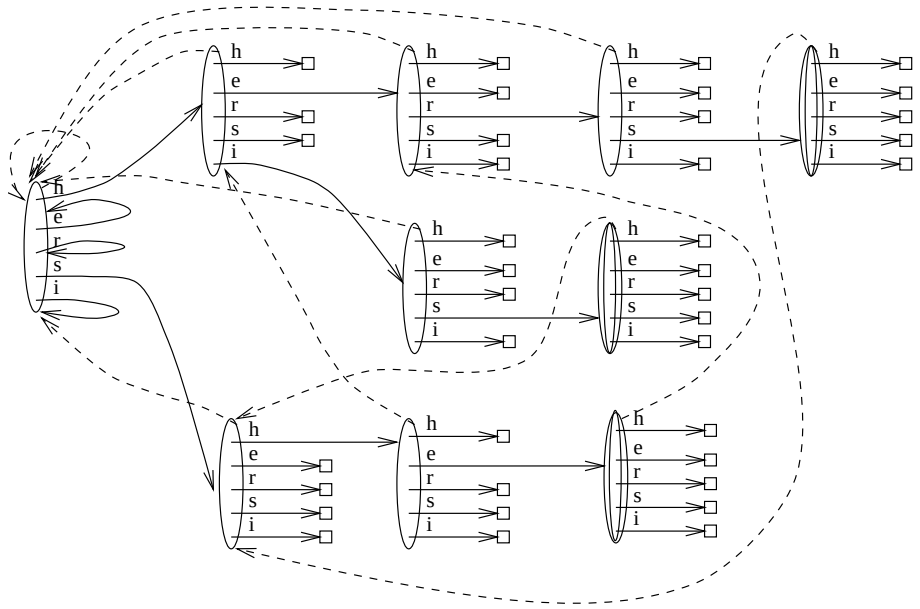


Abbildung 6.7: Der Arrayautomat für die Strings *she*, *hers* und *his*.

Im folgenden versuchen wir eine Darstellung des Automaten zu finden, die uns sowohl einen schnellen Zugriff auf die Nachfolger und eine sehr gute Speicherbilanz bietet.

Zur *Komprimierung des Automaten* wird Σ auf einem Alphabet Γ codiert. Dabei ist eine *Codierung* C eine Abbildung von Σ in Γ^*. Sie kann durch einen Codierungsbaum T_C dargestellt werden werden.

Die Codierung der Züge ist *präfixfrei*, um aus einer gegebenen codierten Sequenz $C(a_1)C(a_2)\ldots C(a_k)$ die Züge a_1, a_2 bis a_k eindeutig ablesen zu können.

Für die speicherplatzeffiziente Codierung ist es günstig, die Wahrscheinlichkeiten $p(a)$, mit der ein Zug a aus Σ zu erwarten ist, gut zu schätzen, denn mit dem Huffman Algorithmus kann ein bezüglich dieser Wahrscheinlichkeiten optimaler Präfixcode erzeugt werden (Huffman 1952).

Dazu werden die Züge als Blattknoten in einer bezüglich der Wahrscheinlichkeiten geordneten Vorrangwarteschlange PQ abgelegt. Die Strukturen T_1 bis $T_{|\Gamma|}$ mit dem geringsten Wahrscheinlichkeitswert in PQ werden aus PQ entnommen und miteinander zu dem Baum $T = \langle T_1, \ldots, T_{|\Gamma|}\rangle$ verbunden. Die Wahrscheinlichkeiten werden addiert und T mit diesem Wert in PQ eingefügt. Der Algorithmus hat die Laufzeit $O(|\Sigma| \log |\Sigma|)$.

Zum Beweis der Optimalität der Huffman Codierung stellt man fest, daß eine optimale Datenkompression o.B.d.A. durch eine Präfixcodierung erreicht werden kann und definiert die Kosten eines durch einen Baum T dargestellten Präfixcodes $\Phi(T_C)$ als $\sum_{a\in\Sigma} p(a)\ |C(a)|$. In dem Beispiel ist $\Phi(T_C) = 3(0.1+0.1)+2(0.3+0.3+0.2)$ also $\Phi(T_C) = 2.2$. Die *gierige* Wahl der Strukturen $T_1, \ldots, T_{|\Gamma|}$ mit den minimalen Kosten ist berechtigt, da ein optimaler Präfixcode C' darstellender Baum $T_{C'(\Sigma)}$ kostenneutral in $T_{C(\Sigma)}$ umgewandelt werden kann, und die durch die Zusammenfassung der Teilbäume $T_1, \ldots, T_{|\Gamma|}$ verbundene Reduktion des Alphabets Σ die Optimalität des Codes erhalten (Cormen, Leiserson & Rivest 1990).

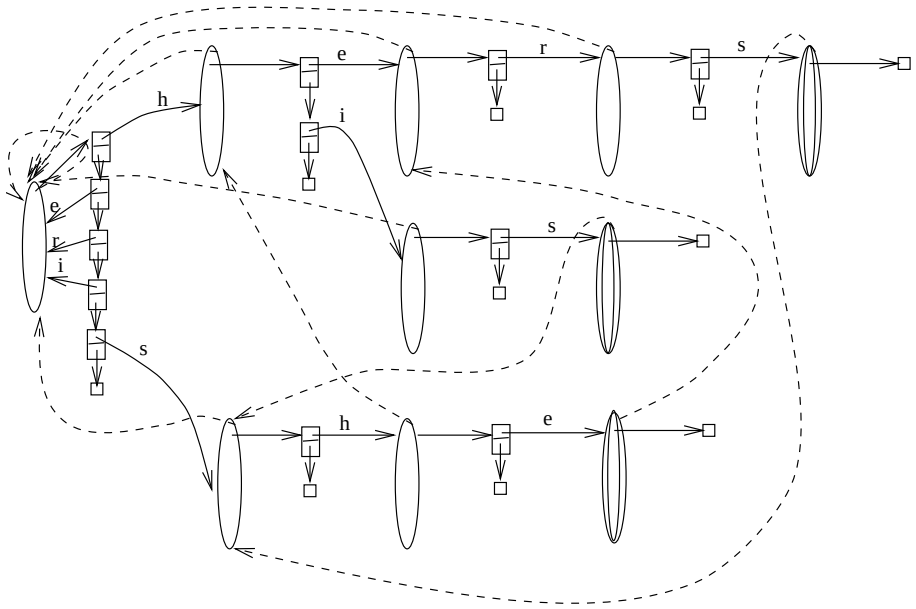


Abbildung 6.8: Der Adjazenzlistenautomat für die Strings *she*, *hers* und *his*.

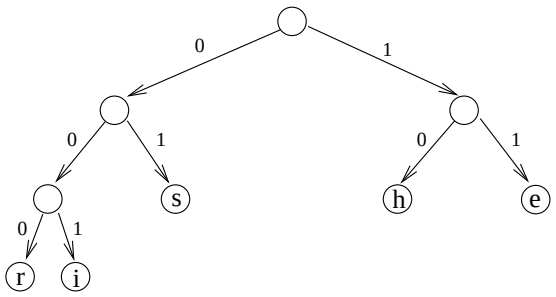


Abbildung 6.9: Ein präfixfreier Codierungsbaum $T_{C(\Sigma)}$ für die Strings *she*, *hers* und *his*.

Ist $z = (z_1, \ldots, z_n)$ aus Σ^n gewählt, so ist $C(z) = (C(z_1), \ldots, C(z_n))$. Demnach ist für die Menge Y aus Strings $y_1, \ldots, y_k$ die Codierungsmenge $C(Y)$ durch $\{C(y_1), \ldots, C(y_k)\}$ gegeben. Ist ein präfixfreier Code C gefunden, so kann der Trie $T_{C(Y)}$ auf dem Alphabet Γ an Stelle von T_Y aufgebaut werden. Abbildung 6.10 stellt den Trie $T_{C(Y)}$ für die Musterstringmenge $Y = \{she, hers, his\}$ und der in Abbildung 6.9 dargestellten Codierung dar. In den Quadraten unterhalb der Knoten ist der zugehörige Fehlerfunktionswert angegeben. Die Codierung ist gemäß der Anteile der Buchstabenvorkommnisse in *she*, *hers* und *his* optimal.

Der Automat $M_{C(Y)}$ entsteht durch den Algorithmus von Aho und Corasick (vergl. Kapitel 5). Es gilt der folgende Satz.

Satz 26 *(Automaterkennung) Sei C eine präfixfreie Codierung. Eine String y wird von M_Y genau dann erkannt, wenn $C(y)$ in $C(Y)$ liegt und $C(y)$ von $M_{C(Y)}$ erkannt wird.*

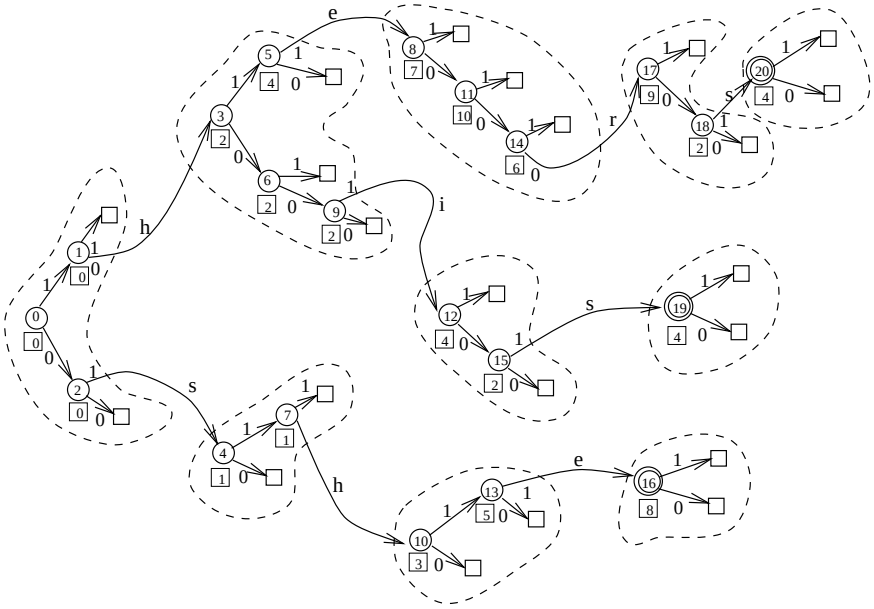


Abbildung 6.10: Der Automat $M_{C(Y)}$ für die Strings *she*, *hers* und *his*.

Beweis: Sei y von M_Y erkannt. Dann existiert ein z aus Y, der Teilstring von y ist. Somit ist auch $C(z)$ Teilstring von $C(y)$ aus $C(Y)$. Damit wird $C(y)$ von $M_{C(Y)}$ akzeptiert.

Ist $C(y)$ aus $C(Y)$, so ist y aus Y durch den präfixfreien Code eindeutig festgelegt. Akzeptiert nun $M_{C(Y)}$ den String y aus $C(Y)$, so gibt es einen z aus $C(Y)$, der Teilstring von $C(y)$ ist. Damit ist $C^{-1}(z)$ aus Y ein Teilstring von y aus Y. Folglich wird y von M_Y akzeptiert. □

Die Bedingung $C(y)$ aus $C(Y)$ ist wesentlich, denn $M_{C(Y)}$ erkennt auch Suffixe z aus $C(Y)$ von Strings, deren Anfangsstücke nicht wohlgeformt sind.

In dem Beispiel läßt sich eine durch die Codierung zu erreichende Kompressionsmöglichkeit nur erahnen. In Abbildung 6.7 gibt es insgesamt 50 Zeiger auf die Nachfolger (allgemein $O(|\Sigma||M_Y|)$) und in Abbildung 6.10 sind es 42 (allgemein $O(|\Gamma||M_{C(Y)}|)$). Wir werden $|M_{C(Y)}|$ nach oben abschätzen.

Definition 8 *Die Modellannahme (M) besagt, daß die Wahrscheinlichkeit $p(a)$, mit der ein Übergang a zu erwarten ist, sich in dem Automat M_Y widerspiegelt, d.h.*

$$p(a) = \frac{|\{q \in M_Y \mid \delta(q,a) \neq \emptyset\}|}{|M_Y|}.$$

Satz 27 *(Komprimierung des Automaten) Sei $\Phi(T_C) = \sum_{a\in\Sigma} p(a)|C(a)|$ die mittlere Tiefe des Codierungsbaumes T_C zur Codierung C. Der Automat $M_{C(Y)}$ besitzt unter der Modellannahme (M) höchstens $|M_Y|\Phi(T_C)$ viele Zustände.*

Beweis: Sei $[p] = 1$, falls p eine wahre Aussage darstellt, sonst 0. Für die Anzahl der Zustände des Automat $M_{C(Y)}$ gilt:

$$|M_{C(Y)}| \quad \leq \quad \sum_{t\in M_Y} \sum_{a\in\Sigma} |C(a)|[\delta(t,a) \neq \emptyset]$$

$$
\begin{aligned}
&= \sum_{a\in\Sigma} |C(a)| \sum_{t\in M_Y} [\delta(t,a) \neq \emptyset] \\
&\overset{(M)}{=} \sum_{a\in\Sigma} |C(a)||M_Y|p(a) \\
&= |M_Y| \left(\sum_{a\in\Sigma} p(a)|C(a)| \right) \\
&= |M_Y|\Phi(T_C).\square
\end{aligned}
$$

□

Der Speicherplatzbedarf für $M_{C(Y)}$ ist demnach durch $(|M_Y|\Phi(T_C)|\Gamma|)/|\Sigma|$ beschränkt. Damit lohnt sich eine Codierung garantiert, falls $\Phi(T_C)|\Gamma| < |\Sigma|$ gilt. Im Beispiel ergibt sich die Ungleichung $2.2 \cdot 2 < 5$. Der Quotient $5/4.4$ ist etwas schlechter als $50/42$. Dies liegt an der Ersparnis von Knoten an den zwei Verzweigungen innerhalb von M_Y, d.h. für Verzweigungsgrade größer zwei ist die Ungleichung im Beweis scharf.

Damit ist es legitim, die Darstellung der Züge zu variieren, um speicher– oder laufzeiteffizientere Automaten $M_{C(Y)}$ zu konstruieren.

6.4 Symmetrie

Vielfach lassen sich Suchprobleme durch Drehungen und Spiegelungen in sich selbst überführen. Culberson und Schaeffer untersuchen, wie sich Symmetrieüberlegungen (Automorphismen) auf die von ihnen konzipierte Musterdatenbank für untere Schranken übertragen lassen (Culberson & Schaeffer 1996).

Wir konzentrieren uns auf die Konstruktion eines die Symmetrien ausnutzenden Automaten. Dabei nennen wir ein Suchproblem *symmetrisch* zur Bijektion $\phi : \Sigma \to \Sigma$, wenn folgende Eigenschaft erfüllt ist: Ist eine Übergangssequenz A eine Abkürzung zu B, so ist auch $\phi(A)$ ein Abkürzung zu $\phi(B)$, wobei $\phi(A)$ für $A = (a_1, \ldots, a_n)$ durch den Vektor $(\phi(a_1), \ldots, \phi(a_n))$ gegeben ist. Die Menge Φ ist durch alle Bijektionen ϕ gegeben, zu denen ein gegebenes Suchproblem symmetrisch ist.

Das $(n^2 - 1)$–Puzzle ist z.B. rotationssymmetrisch zu den Abbildungen $\phi_1, \ldots, \phi_4$, die den Drehungen um 0, 90, 180 und 270 Grad entsprechen und spiegelsymmetrisch zu den Abbildungen ϕ_5 und ϕ_6, die den Spiegelungen an der horizontalen bzw. vertikalen Achse entsprechen.

Es ist $\Sigma = \{\mathsf{U,D,L,R}\}$, $\phi_1(\Sigma) = id_\Sigma$, $\phi_2(\Sigma) = \{\mathsf{L,R,D,U}\}$ (90 Grad), $\phi_3(\Sigma) = \{\mathsf{D,U,R,L}\}$ (180 Grad), $\phi_4(\Sigma) = \{\mathsf{R,L,U,D}\}$ (270 Grad), $\phi_5(\Sigma) = \{\mathsf{D,U,L,R}\}$ (Spiegelung an der x–Achse) und $\phi_6(\Sigma) = \{\mathsf{U,D,R,L}\}$ (Spiegelung an der y–Achse).

Es ist bei ausreichend vorhandenem Speicherplatz möglich, für jedes gefundene Duplikat d und für alle ϕ aus Φ die Werte $\phi(d)$ in den Automat M_Σ einzugliedern. Da die Anzahl der gefundenen sehr schnell anwächst, ist jedoch eine möglichst komprimierte Automatdarstellung von großem Interesse.

Somit wird im folgenden ein Automat beschrieben, der eine einmal gefundene Duplikatssequenz auch nur einmal repräsentiert. Die erforderlichen Symmetrien werden durch die Eindeutigkeit der Darstellung erzwungen. Dazu wird ausgenutzt, daß eine totale Ordnung $<$ auf Σ existiert. Die Züge wer-

den bei der Konstruktion des Automaten immer nach aufsteigender Ordnung ausgeführt.

Als Beispiel sei für das (n^2-1)–Puzzle der Automat $M_\Sigma = (Q, \Sigma, F, q_0, \delta)$ durch das Duplikat LURDLURD zur Abkürzung RDLU erzeugt. Der Trie mit Fehlerfunktion ist in Abbildung 6.11 dargestellt.

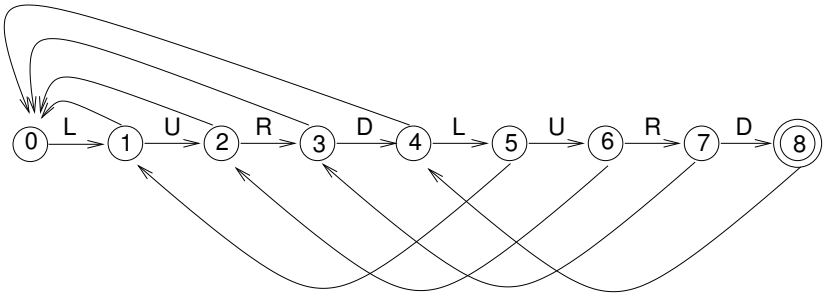


Abbildung 6.11: Ein Trie mit Fehlerfunktion zum Duplikat LURDLURD.

Sei ULRULDRULD die auf ein Duplikat zu untersuchende Sequenz. Die für eine Sequenz noch validen Drehungen $\Phi' \subset \Phi$ werden durch Bitvektoren dargestellt. Es ergibt sich der folgende Ablauf:

$$(0, 111111) \xrightarrow{\mathsf{U}} (1, 000100) \xrightarrow{\mathsf{L}} (2, 000000) \xrightarrow{f(2)} (0, 000100) \xrightarrow{\mathsf{L}} (0, 000000) \xrightarrow{f(0)}$$

$$(0, 111111) \xrightarrow{\mathsf{R}} (1, 001010) \xrightarrow{\mathsf{U}} (2, 000010) \xrightarrow{\mathsf{L}} (3, 000010) \xrightarrow{\mathsf{D}} \ldots \xrightarrow{\mathsf{D}} (8, 000010)$$

Bei einem erreichten Nullvektor wird die Fehlerfunktion aufgerufen, da keine der möglichen Symmetrien einen Zug erlaubt. Die Hinzunahme schon ausgeschlossener Symmetrien ist nur bei Erreichen einer sonst aussichtslosen Situation im Zustand Null erlaubt.

Somit erkennt der Automat die Sequenz RULDRULD, die durch eine Spiegelung an der y–Achse aus LURDLURD hervorgeht.

Hierbei ist eine Besonderheit zu beachten: Durch die Permutation der Züge können gleichlange Duplikate und Abkürzungen nicht repräsentiert werden, da das Duplikat in einer Zugpermutation durch die Ordnung der Züge auf einmal zu einer Abkürzung wird. Hier empfiehlt es sich, zu jedem Duplikat die Abkürzung mit abzulegen. Ist ein Duplikat erst einmal erkannt, kann in einem zweiten Schritt nachgeprüft werden, ob das Duplikat bezüglich der jetzigen Permutation wirklich länger ist als die vorgeschlagene Abkürzung.

Sei a das im angegeben Verfahren gelesene Zeichen für den Übergang von q nach q'. Der aus b_q resultierende Bitvektor $b_{q'}$ kann dann effizient durch eine $\vee$-Verknüpfung mit einem Wert $T[a', a]$ aus einer Tabelle der Größe $|\Sigma|^2$ gebildet werden. Dabei ist $T[a', a] = (b_1, \ldots, b_{|\Phi|})$ mit $b_i = 1$ $(i \in \{0, \ldots, |\Phi|\})$, falls $\phi_i(a) = a'$, sonst gilt $b_i = 0$ (siehe Tabelle 6.2).

Die Konstruktion der Fehlerfunktion eines vorgegebenen symmetriefrei generierten Automaten verändert sich nicht.

6.5 Formale Sprachen Wegeproblem

Die Zustandsraumsuche entspricht der Aufgabe, den kürzesten Weg innerhalb eines gewichteten und mit Elementen aus Σ beschrifteten Graphen zu suchen.

	U	D	L	R
U	100001	001010	010000	000100
D	100001	001010	010000	000100
L	000100	010000	100010	001001
R	010000	000100	100010	001001

Tabelle 6.2: Die Tabelle Boole'scher Verknüpfungsvektoren.

In der Duplikatselimination unterdrücken wir die Generierung von Worten aus einer regulären Sprache (Schöning 1995).

Das Komplement einer regulären Sprache ist wiederum regulär. So können wir das Duplikatserkennungsproblem auch so definieren, daß auf der Suche nach dem kürzesten Weg nur Elemente einer regulären Sprache erlaubt sind. Endliche Automaten sind das Maschinenmodell der regulären Sprachen, wobei eine Sprache eine Teilmenge aller Zeichenketten über dem Übergangsalphabet Σ ist. Formal schreiben wir für eine von einem Automat M erkannte Sprache

$$L = \{x \in \Sigma^* \mid M(x) = 1\}$$

Die von deterministischen und nicht-deterministischen endlichen Automaten erkannten Sprachen unterscheiden sich nicht. Die regulären Sprachen sind eine sehr eingeschränkte Sprachklasse, z.B. kann die Sprache der Palindrome, die Wörter wie *Reliefpfeiler* enthält, nicht von einem endlichen Automat erkannt werden (Wegener 1993c).

Somit gliedern sich die regulären Sprachen weit unten in der Chomsky-Hierarchie ein. Verallgemeinerungen sind die kontextfreien und die kontextsensitiven Sprachen. Reguläre Sprachen sind abgeschlossen gegenüber der Vereinigung, der Schnitt- und der Komplementbildung. Sie lassen sich einfach durch reguläre Ausdrücke beschreiben, die wie folgt rekursiv definiert sind:

1. Das leere Wort ϵ ist ein regulärer Ausdruck.

2. Wenn u und v reguläre Ausdrücke sind, so auch die Konkatenation uv, die Vereinigung $u \cup v$ und die transitive Hülle u^*.

In der Teilstringerkennung der Duplikate m_1 bis m_n läßt sich die von dem Automat von Aho und Corasick erkannte Sprache wie folgt beschreiben:

$$L = \{\Sigma^* m_1 \Sigma^* \ \cup \ldots \cup \Sigma^* m_n \Sigma^*\}$$

Verallgemeinern wir unsere Betrachtung auch auf andere Sprachklassen, so erhalten wir das durch eine formale Sprache eingeschränkte Wegproblem *FLCPP* (engl. *formal language constrained path problem*), das von Barett, Jacob und Marathe wie folgt definiert wird: Gegeben ein beschrifteter und gewichteter Graph G, eine Quelle s, ein Ziel g und eine formale Sprache L (regulär, kontextfrei, kontextsensitiv, etc.). Suche den kürzesten Weg in G, so daß die Beschriftung von p in L liegt (Barrett, Jacob & Marathe 1998).

Die Autoren beweisen, daß das reguläre *FLCPP* in Zeit $O(|R||G| \log |R||G|)$ lösbar ist, wobei R der die Sprache L begründende reguläre Ausdruck ist.

Die Idee ist, einen Produktgraphen aus einem R akzeptierenden nichtdeterministischen endlichen Automat und dem Graphen G zu bilden und dann den kürzesten Wege Algorithmus von Dijkstra (vergl. Kapitel 3) anzuwenden. Der nichtdeterministische endliche Automat für R besitzt $O(|R|)$ viele Zustände (Simon 1992).

Weiterhin zeigen Barett, Jacob und Marathe, daß das kontextfreie *FLCPP* mit Hilfe von dynamischer Programmierung ähnlich dem Algorithmus von Cook, Younger und Kasami (Wegener 1993c) in kubischer Zeit lösbar ist. Das kontextsensitive *FLCPP* ist hingegen nicht entscheidbar.

6.6 Faktorisierung des Automaten

Korfs Erfolg bei der Lösung des Zauberwürfels beruht auf einer Relaxierung des Problems auf ein Eckwürfel- und zwei Kantenwürfelprobleme (vergl. Kapitel 2), die einen stark verkleinerten Suchraum aufweisen.

In diesem Abschnitt werden wir untersuchen, ob und wie sich die Aufteilung des Problems auf die Konstruktion des endlichen Automaten übertragen läßt. Als Beispiel betrachten wir ein 2×2 großes Spielfeld mit vier verschiedenen Spielsteinen (siehe Abbildung 6.12). Es gibt vier Züge U, D, R und L, die einer Vertauschung der Spielsteine in der oberen oder unteren Reihe bzw. in der rechten oder linken Spalte entsprechen.

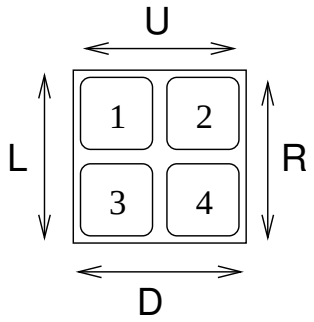


Abbildung 6.12: Ein Vertauschungsspiel.

Ähnlich wie beim Zauberwürfel erlauben wir kein aufeinanderfolgendes Vertauschen gleicher Spielsteine und die Reihenfolgetreue U vor D und R vor L. Wir vereinfachen das Spiel auf zweierlei Art und Weise. In der Spielversion A werden die Spielsteine 1 und 4 voneinander ununterscheidbar eingefärbt und in der Spielversion B entsprechend die Spielsteine 2 und 3.

Beide relaxierte Spielversionen besitzen mit 12 Spielstellungen einen halb so großen Zustandsraum wie das ursprüngliche Puzzle.

In der Version A finden wir Zyklen wie LDL, DLD, RUR und URU, und in der Spielversion B ergeben sich Kreise wie z.B. LUL, ULU, RDR und DRD. Die entscheidende Beobachtung ist, daß sich ein Zyklus im Ausgangspuzzle dann, und nur dann, findet, wenn sowohl ein Zyklus in der vereinfachten Version A und in der vereinfachten Version B existiert, da sich letztendlich alle Spielsteine zurück an ihren Platz bewegt haben. In der Sprechweise formaler Sprachen seien $L(A)$ und $L(B)$ die Mengen aller Zyklen in den Spielversionen A bzw. B. Somit ist die Menge aller Zyklen L im ursprünglichen Puzzle durch den Schnitt von $L(A)$ mit $L(B)$ gegeben. Dabei ergibt sich ein Automat M für die Sprache L aus dem parallelen Lauf von $M(A)$ und $M(B)$, die die Sprachen

$L(A)$ und $L(B)$ erkennen. Der endliche Automat M akzeptiert genau dann, wenn sowohl $M(A)$ und $M(B)$ akzeptieren.

Angenommen, die Speicherkapazität der Hashtabelle innerhalb der Suche wäre begrenzt, so ließen sich durch diesen Faktorisierungsansatz dennoch alle Zyklen innerhalb des Puzzles finden. So sind die zwei Wörter LUL und RDR in $L(B)$, aber nicht in $L(A)$. Die Konkatenation LULRDR liegt hingegen im Schnitt der beiden Sprachen und wird demnach von beiden Teilautomaten $M(A)$ und $M(B)$ erkannt.

In der Duplikatselimination mit endlichen Automaten versuchen wir innerhalb der Suche schon beim Erkennen eines längeren Halbzyklus (Duplikat) anzuhalten. Eine Faktorisierung eines Automaten für Halbzyklen ist jedoch nicht offensichtlich. Die mit einer gleichzeitigen Akzeptanz verbundenen Abkürzungen können höchst unterschiedlich ausfallen und damit nicht unbedingt eine Abkürzung des ursprünglichen Automaten liefern.

Dennoch kann die vorgetragene Faktorisierungsidee gewinnbringend bei der Duplikatselimination angewendet werden. Durch ein sicheres Erkennen von Zyklen in $L(A) \cap L(B)$ können wir innerhalb des Suchraumes *herumstrolchen* (engl. random walk) und mit jedem aufgespürten Kreis die Erkennungsrate des Duplikatsautomaten steigern, indem die mit M gefundenen Zyklen in Duplikate und Abkürzungen zerlegt werden.

6.7 Inkrementelles Lernen von Abkürzungen

In diesem Abschnitt werden wir einen Automat zur Duplikatserkennung konstruieren, der sich sowohl in der Aufbau- und in der Suchphase nutzen läßt. Wir verbinden somit die zwei von Taylor und Korf vorgeschlagenen Phasen zu einer. In dem Bereich *maschinelles Lernen* spricht man von einer inkrementellen Lernstrategie, da die Performanz sich bei der Suche nach dem Ziel selbständig verbessert (Langley 1996).

Warum braucht man in der Duplikatselimination eine inkrementelle Lernstrategie? Betrachten wir restringierte Zustandsräume, wie z.B. der eines Labyrinths. Hier sind selbst einfache Duplikate (wie DU) nicht aus allen Anfangszuständen direkt aufzufinden. Je kürzer die erkannten Duplikate sind, umso größer ist die Verallgemeinerung und damit die resultierende Beschneidung des Suchbaumes. Somit empfiehlt sich das Abstreifen (*Kürzen*) des gemeinsamen Anfangsstücks in Duplikat und Abkürzung (engl. *cancellation of common prefix*). Nun ist jedoch die Länge der Duplikatstrings nicht mehr monoton wachsend, so daß die gesamte Automatstruktur nach jeder Einfügung aktualisiert werden muß.

Durch die Entdeckung kurzer Duplikate kann es dazu kommen, daß schon gespeicherte Verlängerungen (engl. *super strings*) überflüssig werden. Diese sollten aus Platzgründen gelöscht werden, da die kürzeren Duplikate die größere Generalisierung bergen und die längeren Duplikate abdecken.

Duplikate unterschiedlicher Struktur können auch aus Symmetrieüberlegungen und der heuristischen oder anderweitigen Zyklensuche gewonnen werden, so daß die Frage nach einem komfortablen Wörterbuch aufkommt, das sowohl die Operationen *Insert* und *Delete* für gefundene Duplikatssequenzen und *Search* für die Suchbaumbeschneidung anbietet.

Wie wir in Kapitel 5 gesehen haben, führt ein Wörterbuch basierend auf dem Automat von Aho und Corasick jedoch zu großen Effizienzverlusten.

6.7.1 Der *ILA** Algorithmus

Der Algorithmus *ILA** (für engl. *incremental duplicate learning* A^*, vergl. Programm 6.1) ermöglicht die kombinierte Duplikatserkennung und Nutzung. Die zur Zeichenkettenverarbeitung genutzte Datenstruktur ist die eines Multi-Suffixbaumes aus Kapitel 5.

In einem restringierten Zustandsraum nehmen wir an, daß beim Akzeptieren des Automaten die definierenden Eigenschaften (D2) und (D3) für ein Duplikat m und eine Abkürzung m' explizit nachgeprüft werden. Wir schreiben für beide Tests kurz $m^{-1}m'(v) = v$. Der Zustandsraum selbst muß nicht reversibel sein. Wir können m^{-1} entweder durch Rückwärtszüge gewinnen ober aber einfach in dem Suchbaum $|m^{-1}|$ Schritte hinaufklettern. Neben der Anwendbarkeit (engl. applicability) der Abkürzungssequenz wird die Gleichheit (engl. equality) der durch das generalisierte Duplikat und dessen Abkürzung erhaltenen Knoten durch einen Hashfunktionsvergleich gepruüft.

Wir erweitern den bekannten A^* Algorithmus (vergl. Kapitel 3) und verwenden somit zu dem Zeichenkettenwörterbuch D eine Vorrangwarteschlange *Open* und eine Hashtabelle H. Die Eingabe des *ILA**Verfahrens ist ein implizit durch eine Menge von Regeln festgelegtes Zustandsraumproblem und die Ausgabe ein Lösungspfad. Um einem Namenskonflikt zu entgehen, bezeichnen wir Suchraumzustände als Knoten und Automatzustände als Zustände.

Bevor wir einen Knoten in der Hashtabelle suchen, schauen wir erst nach, ob ein akzeptierender Zustand in D vorliegt. In diesem Fall wird die Suche je nach Restriktion des Suchraumes entweder unmittelbar oder erst nach der Prüfung der zum erkannten Duplikat gehörenden Abkürzung abgebrochen.

Akzeptiert D hingegen nicht, so wird in der Hashtabelle nachgeschaut, ob ein Synonym v' des von u generierten Nachfolgeknotens v vorliegt. Ist dies nicht der Fall, so können wir den neuen Knoten bedenkenlos sowohl in die Vorrangwarteschlange als auch in die Hashtabelle einfügen.

Existiert hingegen ein Gegenpart von v in der Hashtabelle, so kürzen wir in den Generierungspfaden das gemeinsame Anfangsstück und halten den längeren Pfad als Duplikat m und den kürzeren Pfad als zugehörige Abkürzung m' fest. Dieses Paar wird in das Wörterbuch D eingefügt und evtl. existierende Verlängerungen von m werden gelöscht.

Die Verbesserung der Bewertung des Gewichtes innerhalb der Vorrangwarteschlange ist aus dem A^* Ansatz hinreichend bekannt. Die Korrektheit des Verfahrens ergibt sich unmittelbar aus der Korrektheit von A^*, da wir den Suchbaum nur dort beschneiden, wo wir eine kostengünstigere Alternative garantieren können.

An einem kleinem Beispiel innerhalb des *Gitters* vergegenwärtigen wir uns die Funktionsweise des Verfahrens. Als heuristische Bewertungsfunktion benutzen wir die aus dem (n^2-1)-Puzzle bewährte Manhattandistanz. Angenommen wir starten bei $s = (3,3)$ und suchen die Zielposition $g = (0,0)$. Demnach wird die Vorrangwarteschlange *Open* mit s und der Bewertung sechs initialisiert. Wir schreiben kurz *Open*$= \{(3,3)_6\}$. Bei der Expansion von s finden wir die Nachfolgermenge $\Gamma(s) = \{(4,3)_8, (3,4)_8, (3,2)_6, (2,3)_6\}$. Kein Nachfolger ist in der Hashtabelle H enthalten, so daß alle Elemente sowohl in *Open* und in H eingefügt werden.

Im nächsten Schritt expandieren wir $u = (2,3)_6$ und finden heraus, daß der Nachfolger $v = (3,3)_8$ einen Gegenpart $v' = (3,3)_6$ in der Hashtabel-

Programm 6.1 *ILA**: Der *A** Algorithmus wird um das inkrementelle Lernen von Duplikaten und Abkürzungen erweitert. Eingabe: Zustandsraumproblem in Form eines gewichteten Graphens G. Ausgabe: Generierungspfad der Lösung.

```
Insert(Open,[s,p_s],h(s)); Insert(H,[s,p_s])     {initialisiere Strukturen}
while (Open ≠ ∅)                                  {wenn Open leer, keine Lösung}
  u ← DeleteMin(Open)                             {Closed ist H ohne Open}
  for all v in Expand(u)                          {für alle Nachfolger}
    if (goal(v)) return p_v                       {wenn Ziel gefunden, gib Lösung aus}
    q(v) ← δ(q(s),p_v)                            {q(v) ist neuer Automatzustand}
    if (q(v) ∈ F)                                 {Automat akzeptiert Anfrage p_v}
      Sei (m,m') mit q(v) verbunden               {m ist Duplikat, m' Abkürzung}
      if (m^{-1}m'(v) = v) continue               {bei Akzeptanz überspringe Rest}
    A*-Improve(u,v)                               {evtl. Verbesserung des Weges zu v}
    v' ← Search(H,v)                              {suche Synonym in Hashtabelle}
    if (v' ≠ nil)                                 {v ist in Hashtabelle}
      if (w(p_v) < w(p_v'))                       {Generierungspfad p_v kürzer}
        m' ← p_v; m ← p_v'                        {v' erzeugt Duplikat, v die Abkürzung}
      else m ← p_v; m' ← p_v'                     {v erzeugt Duplikat, v' die Abkürzung}
      (m,m') ← ccp(m,m')                          {kürze den kleinsten gemeinsamen Präfix}
      while (m'' ←findSuperString(D,m))           {finde eine Verlängerung}
        Delete(D,m'')                             {lösche gefundene Verlängerung}
      Insert(D,(m,m'))                            {füge neues Paar in Wörterbuch ein}
```

le H besitzt, so daß das Duplikat m auf den Generierungspfad LR und die Abkürzung m' auf die leere Sequenz ϵ gesetzt wird. Das Paar (m, m') wird in das Zeichenkettenwörterbuch D eingefügt.

Danach wird $u = (3, 2)_6$ als *Open* Knoten mit geringster Bewertung expandiert. Dabei finden wir ein weiteres Duplikat DU mit Abkürzung ϵ. Zusätzlich ergibt sich das Duplikats-Abkürzungspaar LD-DL.

Die Expansion von $u = (2, 2)_6$ liefert den Nachfolger $(2, 3)_8$, der allein durch die Teilstringerkennung in D als Duplikat ausgemacht wird. Und tatsächlich ist die vorher gelernte Sequenz DU ein Teilstring des Generierungspfades LDU, die im letzten Schritt als Duplikat in D gespeichert wurde.

6.7.2 Experimente

Wir haben einen auf eine Multi-Suffixbaumstruktur aufbauenden Akzeptor und das Verfahren *ILA** implementiert und wenden es in einem auf ein 100 mal 100 Gitter aufbauendes Labyrinth an. Hierbei haben wir mit einer Wahrscheinlichkeit von 35 Prozent die Existenz einer Wand ausgewürfelt.

Unser Hauptinteresse ist, die Abschneidekapazitäten des inkrementellen und des nicht-inkrementellen Ansatzes miteinander zu vergleichen.

In Abbildung 6.13 haben wir 20 Instanzen, die mehr als 1000 Expansionen benötigen, aufgezeichnet. Die ausgewählten Herausforderer des *ILA** Verfahrens sind zwei vordefinierte Automaten zur Vorgängerelimination (vergl.

Abbildung 6.3 auf Seite 97) und zur perfekten Elimination im Gitter (vergl. Abbildung 6.5) und zwei in einer Breitensuche der Tiefen 5 und 25 konstruierte Automaten.

In dem inkrementellen Ansatz erlauben wir im Gegensatz zu den anderen Ansätzen die Kürzung gleicher Anfangsstücke, da sich hier die größte Effizienzdifferenz der beiden Verfahren ergibt. In *ILA** können die einzelnen Einfüge- und Löschoperationen in Linearzeit zur Länge der Zeichenkette durchgeführt werden, während in den nicht-inkrementellen Ansätzen eine Neuberechnung der Fehlerfunktion erforderlich ist.

Die Größen der Automaten (gemessen in der Anzahl der Zustände) sind: 10 für die Vorgängerelimination (die akzeptierenden Zustände sind in dieser Zahl enthalten), 14 für den *Gitter*-Automat, 17 für die Lerntiefe 5 und 43 für die Lerntiefe 25 in der Breitensuche. Der in *ILA** konstruierte Automat besitzt 75 Zustände in der ersten und 135 Zustände in der letzten Iteration.

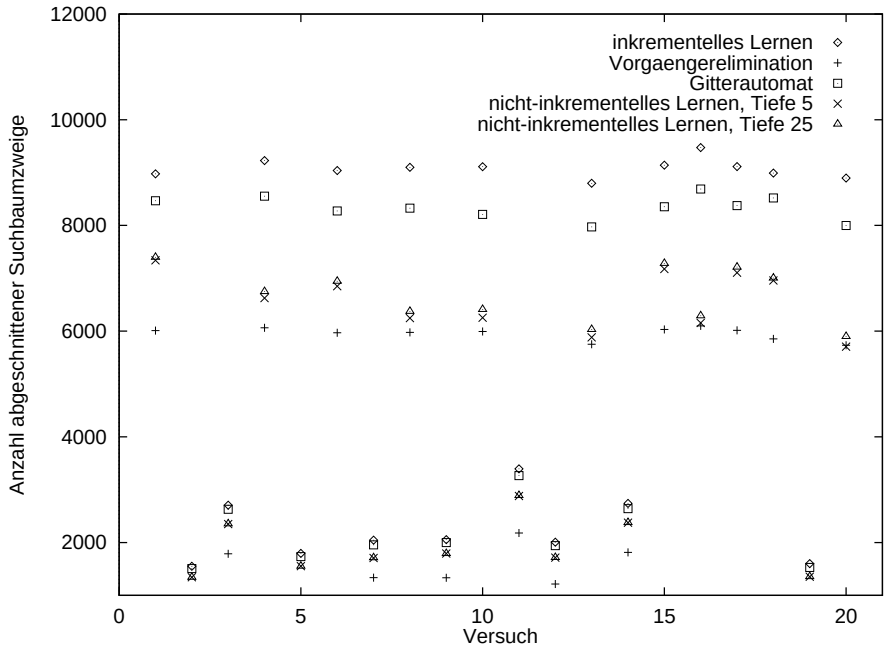


Abbildung 6.13: Experimentelle Resultate zum Vorteil inkrementellen Lernens im (100×100) *Labyrinth*.

6.8 Ausblick

In dem *ILA** Verfahren hindert uns der Wörterbuchautomat nur daran, einen Suchvorgang in der Hashtabelle durchzuführen. Auch wenn wir im schlimmsten Fall nicht von einer konstanten Zugriffszeit auf die Hashtabelle ausgehen können, so ist die Zeit im Verhältnis zu der Berechnung von $\delta(s, p_v)$ durchaus vergleichbar, wenn nicht sogar geringer.

In Kapitel 5 haben wir eine effiziente Realisation eines Verfahrens zur online Suche von Teilstrings mit Hilfe von Multi-Suffixbäumen besprochen. Diesen Algorithmus können wir nun einsetzen, um die Berechnung von $\delta(q(s), p_v)$ über den letzten Zug $a(v)$ und den Zustand $q(u)$ durchzuführen.

Dabei ergibt sich das folgende zentrale Problem: Durch das dynamische Einfügen und Löschen von Zeichenketten innerhalb der Suche, kann ein an einem Zustandsraumknoten gespeicherter Suffixzustand nicht mehr gültig sein, da es die Position, die er beschreibt, nicht mehr gibt. In diesem Fall muß entweder der Generierungspfad neu eingelesen werden oder ein Alternativverfahren zur Herstellung eines passenden Suffixzustandes beschrieben werden.

Der Weg, einen Suffixbaum nur zur Berechnung der Fehlerfunktion zu benutzen, ist aussichtsreich, da der Zustand allein durch die Position im Mehrwegsuchbaum gegeben ist.

Auf der anderen Seite können in großen Suchräumen nicht alle Einfügungen in der Hashtabelle durchgeführt werden. In dem so angesprochenen Gebiet speicherplatzbeschränkter Algorithmen ist die kompakte und generalisierende Beschreibung von Duplikatssequenzen der vielversprechendste Weg, um Zustandswiederholungen innerhalb des Suchbaumes zu vermeiden. Die Kombination des ILA^* Ansatzes mit Cachedatenstrukturen in IDA^* soll in Zukunft genauer analysiert werden.

Kapitel 7

Lernen von Sackgassen

In diesem Kapitel stellen wir eine konservative Erweiterung des A Algorithmus vor, der Merkmale für Sackgassen aufspürt, generalisiert und damit der Suchraumbeschneidung dient. Die Verallgemeinerung beruht auf der Tatsache, daß zumeist nur kleine Teile einer Stellung verantwortlich für deren Unlösbarkeit sind. Wir zeigen auf, wie aussagekräftigere Muster durch Aufteilung und Ausbreitung gefunden werden können. Die erste Methode zerlegt eine Stellung in für den Status einer Sackgasse vielversprechende Komponenten, während die zweite Methode gefundene Merkmale für die Komponenten zu aussagekräftigeren Charakteristiken für die Ausgangsstellung verbindet. Weiterhin stellen wir einen effizienten Teilpositionenspeicher (engl. subposition store) für die Speicherung von und Suche nach Teilstellungen vor. Wir benutzen Sokoban für die empirische Evaluation unseres Algorithmus.*

7.1 Sackgassen

Eine alltägliche Weisheit ist, daß Türen genau dann ins Schloß fallen, wenn wir auf der falschen Seite stehen. Ob bei einer Auto- oder Haustür, nahezu jeder Mensch hat sich selbst schon mal ausgeschlossen.

In der Suche nach Zielen in gerichteten Zustandsgraphen kann es genauso geschehen, daß das Ziel durch einen ungeeigneten Zug auf einmal nicht mehr erreichbar ist. Leider ist es einer Stellung nicht immer direkt anzusehen, ob sie den Lösungsweg nun vereitelt oder nicht. Erst die weitere Exploration des Suchraumes wird uns vermitteln, ob wir auf dem *Holzweg* sind oder nicht. Wenn wir uns nun in einer Sackgasse befinden, gilt es möglichst viel daraus zu lernen, um bei einem neuen Versuch gegen diesen oder ähnliche Fehler gefeit zu sein.

7.2 Finden von Sackgassen

Abbildung 7.1 zeigt ein Beispiel einer Sackgassenposition in Sokoban. In diesem Fall kann keiner der Bälle 1,2,5 und 6 sich bewegen, da sie einander gegenseitig behindern. Die anderen Bälle lassen sich jedoch nach und nach loslösen, so daß die Position trotzdem die Wurzel eines sehr großen Suchbau-

mes darstellt. Daher ist die Erkennung von Sackgassen eine der wichtigsten Objektiven in der (heuristischen) Suche nach Lösungen in Sokoban.

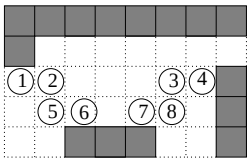


Abbildung 7.1: Eine Sackgasse in Sokoban.

Wir haben die folgende Prozedur *Stuck* implementiert, um einfache Sackgassenpositionen ausfindig zu machen. Wir bezeichnen einen Ball als *frei*, wenn er sich in einer Richtung bewegen läßt, also zwei angrenzende Leerfelder in einer der beiden Koordinatenrichtungen hat. In Abbildung 7.1 ist zum Beispiel nur Ball 4 frei. Wird ein freier Ball entfernt, so können weitere Bälle beweglich werden, so daß wir die Entnahme eines freien Balles wiederholen. Im Beispiel werden die Bälle 4, 3, 8 und 7 nacheinander frei. Die Bälle 1,2,5 und 6 sind nicht zu befreien. Die Position ist eine Sackgasse, da für die Lösung des Sokobanspieles ein jeder Ball bewegt werden muß. Eine Besonderheit ergibt sich im Zielbereich, indem ein unfreier Ball als frei erklärt wird, da er sich möglicherweise schon auf seiner Endposition befindet.

Das Verfahren *Stuck* wird mit Hilfe der Datenstruktur einer *Schlange* (engl. *queue*) verwirklicht, die alle Bälle beinhaltet, deren Status sich noch ändern kann. Initial befinden sich alle Bälle in der Schlange. Im Fall, daß ein aus der Schlange entnommener Ball neu als *frei* erkannt wird, werden seine *unfreien* Nachbarn in die Schlange aufgenommen. Die übrig gebliebenen Bälle bilden die als minimal bezeichnete Sackgassenteilposition.

Da jeder Ball maximal vier Nachbarn hat, kann er sich maximal fünfmal in der Schlange befinden, was letztendlich zu einer $O(n)$ Zeitbeschränkung für *Stuck* führt.

7.2.1 Ausbreitung

Die Prozedur *Stuck* kann nur einen Bruchteil aller Sackgassen erkennen. Eine Position gilt als Sackgasse, falls all ihre Nachfolger eine Sackgasse darstellen.

Unser Verfahren verwendet das Prinzip *Aufwärtsausbreitung*, um aus den verantwortlichen Teilen der Nachfolgerpositionen den verantwortlichen Teil der derzeitigen Stellung und damit weitere Sackgassenmuster zu generieren.

Betrachte den Knoten mit der Nummer 16 in der Abbildung 7.2 mit seinen Nachfolgern 20, 21 und 22, deren jeweilige, durch *Stuck* gefundene, Sackgassenteilposition schwarz gefärbt ist. Um nun selbige für die Position 16 zu finden, markieren wir jeden Ball, der in einem der Nachfolger schwarz markiert ist, gleichfalls mit schwarz, wobei wir den jeweils durchgeführten Zug berücksichtigen. Als Endresultat sind in diesem Fall alle Bälle in Position 16 schwarz eingefärbt. Damit ist die Sackgassenteilposition durch die Vereinigung aller minimalen Sackgassenteilpositionen der Nachfolger gegeben.

7.2.2 Aufteilung

Eine Sokobanposition ist eine Sackgasse, falls sie eine Teilmenge von Bällen enthält, die nicht aufgelöst werden kann, selbst wenn der Rest der Bälle vom Brett entfernt werden. Somit können wir durch eine Aufteilung der Spielstellung eben solche Teilpositionen schneller aufzuspüren. Diese Aufteilung der Spielstellung ist letztendlich eine Daumenregel, inwieweit die Teile die Unlösbarkeit der ursprünglichen Position erben. Intuitiv sind die Positionen *Cliquen* von Bällen, deren Zugmöglichkeiten sehr voneinander abhängen.

Wir untersuchen zwei unterschiedliche Aufteilungsregeln:

Verbundene Komponenten (vk): Bei der Beobachtung unerreichbarer Felder ist die folgende Überlegung zentral in unserem Ansatz, Sackgassen zu finden: Eine Position mit einem vom Männchen unerreichbaren Nicht-Zielfeld ist eine Sackgasse, es sei denn, die Zielfelder teilen die übrigen Felder in mehrere Bereiche. Demnach richtet sich die erste Idee der Aufteilung auf die Partitionierung des Graphen der Leerfelder in voneinander erreichbaren Felder. Diese Aufteilung des Graphen in Zusammenhangskomponenten ist in einem Tiefendurchlauf in Zeit $O(B)$ möglich, da die Größe des Graphens durch die Brettgröße B beschränkt ist (Cormen, Leiserson & Rivest 1990). Zusätzlich kann man alle Bälle aufspüren, die an mehr als einer Komponente anliegen und diejenigen Komponenten, die sich einen gemeinsamen Ball an ihrer Grenze teilen, miteinander verschmelzen. Eine Position wird als lösbar angesehen, wenn alle Leerfelder von dem Männchen aus erreicht werden können.

Streife Bälle ab (sa): Diese Daumenregel zerteilt das n-Ball Problem in n Einball-Probleme und versucht, diese zu lösen. Dabei erstarren die anderen Bälle jeweils zu Wänden. Ist dies für einen Ball möglich, so wird das ursprüngliche Problem auf das $(n-1)$-Ball Problem reduziert.

Die Aufteilung muß gut gewählt sein, um auf der einen Seite eine schnelle Sackgassenbestimmung und damit eine geringe Suche anzustoßen, auf der anderen Seite jedoch die verantwortlichen Merkmale für eine Sackgasse nur an einer Komponente zu vererben. Da der Suchraum gewöhnlich exponentiell mit der Größe der Eingabe (hier der Anzahl der Bällen) wächst, erleichtern wir uns die Suche nach Sackgassen durch die Aufteilung enorm.

Idealerweise würden wir gerne nach dem Teile-und-Herrsche Prinzip (engl. *divide-and-conquer*) die Lösungen der Teilprobleme einfach zu einer Lösung des ursprünglichen Problems zusammenführen. Leider kann bei den gewählten Strategien in Sokoban dabei die Optimalität der zusammengefügten Lösungsteile nicht garantiert werden.

7.2.3 Verbinden der Strategien

Um eine Spielstellung zu explorieren, haben wir nun die Auswahl zwischen einer Expansion (wie im ursprünglichen A^* Algorithmus) und einer Aufteilung in Teilpositionen. Die Aufteilung einer Stellung und die Idee der Aufwärtsausbreitung ergänzen sich gegenseitig. Während die erste Strategie die verantwortlichen Teilpositionen von Sackgassen aufspürt, kombiniert der zweite Ansatz diese Ergebnisse zu erweiterten Strukturen.

Programm 7.1 *BuPropDead*: Aufwärtsverbreitung des *mdsp* Status innerhalb des Suchbaumes. Eingabe: Als Sackgasse erkannter Zustand u. Ausgabe: Erweiterung des Teilpositionenspeichers.

```
if (u = root) return          {falls Wurzel erreicht, erfolgt Abbruch}
Insert(TPS,mdsp(u))                          {füge Teilstellung ein}
p ← predecessor(u)                           {Wechsel zum Vorgänger}
if (decompose(u))              {ein Vorgänger ist Aufteilungsknoten}
  mdsp(p) ← mdsp(u)            {kopiere minimale Sackgassenstellung}
  BuPropDead(p)                               {informiere Vorgänger}
else                              {Vorgänger kein Aufteilungsknoten}
  if (∀v ∈ Γ(p)                              {falls alle Nachfolger}
      ∃ Teilposition v' von v in TPS)                 {Sackgassen sind}
    mdsp(p) ← computeDsp(p, Γ(p))  {bestimme min Sackgassenstellung}
  BuPropDead(p)                               {informiere Vorgänger}
```

Betrachten wir das Beispiel in Abbildung 7.2, in der die Position 1 expandiert und die Nachfolgerstellungen 2 bis 6 generiert werden. Die drei Nachfolger 3, 4 und 5 werden unmittelbar durch *Stuck* als Sackgasse erkannt, wobei die entsprechenden Sackgassenmuster schwarz eingefärbt sind.

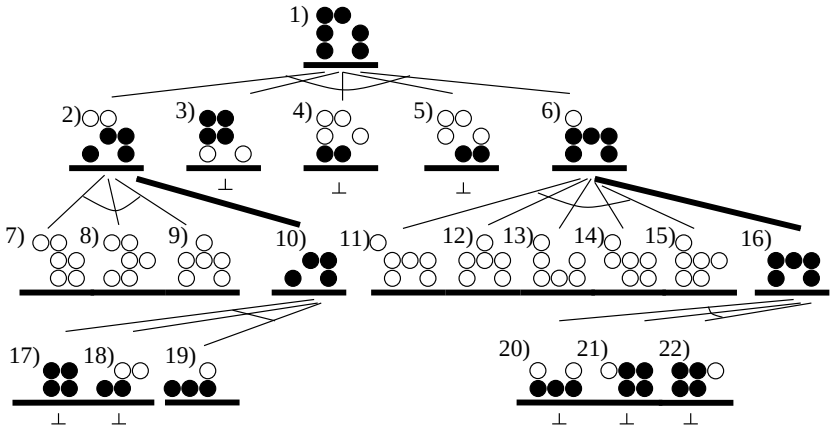


Abbildung 7.2: Ein durch Expansion und Aufteilung aufgespannter Suchbaum.

Da der Status der Positionen 2 und 6 noch nicht festgelegt ist, werden beide expandiert (Nachfolger werden über dünne Kanten erreicht) und aufgeteilt (Nachfolger werden über dicke Kanten erreicht). Sowohl bei der Expansion der Stellungen 10 als auch der Position 16 werden alle Nachfolger durch *Stuck* als Sackgasse klassifiziert. Somit können wir die gefundenen Sackgassenmuster nach 10 bzw. 16 aufwärtspropagieren, wobei wir alle Bälle in den Stellungen als verantwortlich kennzeichnen. Da die beiden Stellungen jeweils durch Dekomposition entstanden sind, können wir die schwarzen Bälle auf die

Vorgängerpositionen direkt übertragen. Nun sind alle Expansionsnachfolger von Stellung 1 als Sackgasse klassifiziert und wir finden das zugehörige Sackgassenmuster durch die Vereinigung der Färbungen gemäß der vorgestellten Regel. Das Programm 7.1 beschreibt die Aufwärtsausbreitung der Sackgasseninformation (*BuProbDead* steht für bottom up propagation of dead subpositions). Dabei bezeichnen wir den Anteil der schwarzen Bälle allgemein als minimal verantwortliche Sackgassenteilstellung (engl. minimal dead subposition, *mdsp*).

7.3 Der Teilpositionenspeicher

Da der Test auf Sackgassen bei jeder Expansion durchgeführt wird, ist eine effiziente Implementierung des *Teilpositionenspeichers* ausschlaggebend für die Gesamtlaufzeit des Suchverfahrens. In Sokoban besteht ein Muster aus einem Bit pro Gitterpunkt, das anzeigt, ob ein Ball auf ihm liegt oder nicht.

7.3.1 Zählvektoren

Eine Möglichkeit ist eine inkrementelle Lösung. Wir können ein Wörterbuch benutzen, in dem an jeder möglichen Ballposition eine Liste von Mustern angehängt wird, die einen Ball an jener Stelle haben. Zusätzlich verwalten wir eine Menge von p Zählern (für jedes Muster einen), die die Anzahl der Übereinstimmung der derzeitigen Position mit allen Mustern notiert. Da nur zwei Ballpositionen pro Zug beteiligt sind, benötigt eine solche Lösung maximal p Vergleiche. Die an den Ballpositionen anliegenden Listen haben in der Praxis jedoch eine wesentlich geringere Länge. Sei L die Summe der Bälle aller gespeicherten Bälle, so ist die mittlere Vergleichsanzahl (unter der Annahme einer uniformen Verteilung der Muster auf dem Brett) durch L/B beschränkt.

7.3.2 Bitvektoren

Ist in der iterierten Tiefensuche *IDA** (Korf 1985a) obiges Verfahren anwendbar, so benötigt der Vektor von p Werten in A^* zuviel Speicherplatz benötigt, um zu jedem Knoten des Suchraumes abgelegt zu werden.

Die einfachste, nicht inkrementelle Lösung ist es, jedes Muster als Bitvektor der Größe B abzulegen und alle Muster nacheinander mit der Stellung logisch zu vergleichen. Bei einer Implementation auf einem Rechner mit der Wortlänge w erhalten wir somit maximal $\lceil pB/w \rceil$ Vergleiche.

Wie wir gesehen haben, kann zu jedem Stellungsmuster ein Bitvektor assoziiert werden. In einer weiteren Lösung fassen wir gemäß der vorliegenenden Wortbreite w unseres Rechners jeweils w Muster zusammen. Für jede Brettposition erhalten wir somit $\lceil p/w \rceil$ Bitvektoren der Länge w.

Der Vergleich der Muster mit einer gegebenen Position kann Ballposition für Ballposition nacheinander erfolgen. Falls jedoch kein Eintrag eines Musters an einer Ballposition vorliegt, können wir bis zu dem nächsten Eintrag springen. Durch eine Verkettung dieser Einträge können wir eine große Anzahl von unnötigen Vergleichen einsparen.

Da ein Test abgeschlossen werden kann, wenn alle gespeicherten Muster widerlegt sind, können weitere Verbesserungen durch geeignetere Reihenfolgen der Abfragen erreicht werden. Eine Möglichkeit ist es, das *next* Array gemäß

der am häufigsten belegten Ballpositionen zu ordnen. Weiterhin kann – je nach Ausgabe des vorangegangenen Tests – ein Entscheidungsbaum (Langley 1996) aufgebaut werden.

7.3.3 Teilpositionen und *OBDDs*

In den vorangestellten Lösungen haben wir insbesondere die Mustereigenschaften von Sokobanstellungen genutzt. Versuchen wir nun das Problem der Teilpositionserkennung zu verallgemeinern.

Binäre Entscheidungsdiagramme (*OBDDs*, vergl. Kapitel 4) eignen sich auch zur Darstellung von Teilpositionen. Voraussetzung ist eine effiziente Codierung der Spielstellung, die in Sokoban durch die Bitmap der Kugel und die Binärdarstellung der Männchenposition gegeben ist. Eine Stellung v ist genau dann ein Teil einer Position u, wenn die charakteristische Funktion von u die charakteristischen Funktion von v impliziert. Ist z.B. $\Phi_{\{u\}} = \overline{x_1} \wedge x_2 \wedge x_3 \wedge \overline{x_4} \wedge x_5$ die Beschreibung einer Spielstellung, so ist $\Phi_{\{v\}} = x_2 \wedge x_3$ ein Beispiel für eine Teilposition, da $\Phi_{\{u\}} \Rightarrow \Phi_{\{v\}}$ eine Tautologie darstellt.

Wenn c die Länge der Zustandskodierung ist, kann der Teilpositionenspeicher durch die Disjunktion der charakteristischen Funktionen der Teilstellungen realisiert werden und Anfragen in der worst-case Zeit von c beantworten.

7.4 Der Algorithmus *ADP**

Unser Hauptprogramm *ADP** (für A^* *with decomposition and propagation*) ist eine konservative Ergänzung zum A^* Verfahren, dessen Struktur durch die unterstrichenen Zeilen hervorgehoben ist. Wie üblich werden die Horizontknoten in einer Vorrangwarteschlange *Open* verwaltet, die gemäß der Bewertung f aus der Generierungspfadlänge g und dem heuristischen Schätzwert h geordnet ist. Die Hashtabelle wird genutzt, um schon einmal erreichte Positionen aufzuspüren.

ADP* führt Expansionen und Aufteilungen parallel aus, wobei die gleiche Vorrangwarteschlange *Open* und Hashtabelle H für beide Nachfolgermengen verwendet wird. Die Lösung einer Teilstellung wird einer Expansion auf der äußersten Suchebene vorgezogen, da der heuristische Schätzwert für den optimalen Lösungspfad kleiner ist. Dabei wird die äußere, ausschließlich aus Expansionsknoten bestehende Ebene von den Knoten, die oder deren Vorgänger durch Aufteilung entstanden sind, getrennt. Letztere erhalten eine Markierung *decompose*.

Der einzige Grund für diese Markierung ist es, daß es Methoden geben kann, die die Lösbarkeit eines Knotens beurteilen, ohne daß eine Lösung explizit berechnet werden muß. Eine solche zu *Stuck* duale Funktion wollen wir *Solvable* nennen. Für diese Prozedur lassen wir einen einseitigen Fehler zu, d.h. wir können eine Sackgasse fälschlicherweise als *lebendig* charakterisieren, ohne die Korrektheit des Gesamtverfahrens zu gefährden.

Um Neuberechnungen des Status zu vermeiden, nutzen wir eine zusätzliche Hashtabelle A. Falls ein lösbarer Knoten durch die Expansion erzeugt wurde, müssen auch die Vorgängerpositionen automatisch lösbar sein. Bei einem durch Aufteilung entstandenen Knoten verhält es sich gerade gegenteilig. Die Aufwärtsverbreitung der Information, daß ein Knoten lösbar ist, wird durch die Prozedur *BuPropAlive* verwirklicht.

Programm 7.2 *ADP**: Lösung einer gerichteten Zustandsraumsuche mit Hilfe von Aufteilungen in Teilstellungen und Aufwärtspropagierung. Eingabe: Startzustand s. Ausgabe: Optimaler Lösungsweg.

```
Insert(Open(s,h(s))); Insert(H,s)          {initialisiere Strukturen}
TPS ← ∅                                    {initialisiere Teilpositionenspeicher}
while Open≠ ∅                              {solange Vorrangwarteschlange nicht leer}
  u ← DeleteMin(Open)                      {entnehme Knoten geringster Bewertung}
  if (goal(u))                             {falls Ziel erreicht}
    if (decompose(u)) BuPropAlive(u) continue {Teilstellung lebendig}
    else return path(u)                    {gesamte Stellung gelöst}
  elsif ((v ∈TPS für Teilstellung v von u) or Stuck(u))   {u Sackgasse}
    BuPropDead(u) continue                 {informiere Vorfahren}
  if not (decompose(u) and u ∈ A)          {u nicht zu expandieren}
    if (decompose(u) and Solvable(u))      {u lebendig}
      BuPropAlive(u) continue              {informiere Vorfahren}
  else                                     {expandiere u}
    Γ(u) ← Expand(u)                       {bestimme Nachfolgermenge}
    Δ(u) ← Decompose(u)                    {bestimme Aufteilungsmenge}
    for all v in Γ(u) ∪ Δ(u)               {für alle Nachfolger v}
      A*-Improve(u,v)                      {füge v in Strukturen ein}
```

Bis die Vorrangwarteschlange leer ist, wird das Element mit der geringsten Bewertung entnommen und untersucht. Falls es ein Zielknoten in der äußeren Suchebene ist, so haben wir eine Lösung gefunden und wir sind fertig. Sonst untersuchen wir den Lösbarkeitsstatus, der durch den Aufruf eines simplen Sackgassenerkenners *Stuck*, der Suchoperation innerhalb des *Teilpositionenspeichers* und der *Solvable* Prozedur ergibt. Falls ein Knoten als *lebendig* oder als Sackgasse klassifiziert ist, propagieren wir die Neuigkeiten aufwärts und wenden uns dem nächsten Horizontknoten zu. Anderenfalls bleibt der Status der Stellung undefiniert, und wir erzeugen die Nachfolgermengen $\Delta(u)$ und $\Gamma(u)$ durch Aufteilung bzw. Expansion. Die Nachfolgermengen werden wie in A^* üblich mit den gespeicherten Knoten abgeglichen und in die Strukturen *Open* und H eingefügt. Eine gefundene Lösung des Problems durch ADP^* ist korrekt, da der äußere Suchbaum dem Suchbaum von A^* entspricht, indem nur Sackgassenpositionen unterdrückt worden sind.

Abhängig von den gegebenen Ressourcen kann eine Aufteilung entweder in jedem Schritt, hier und da, oder nur in kritischen Situationen angestoßen werden. ADP^* ist als inkrementeller Lernalgorithmus konzipiert, d.h. jedes gefundene Sackgassenmuster kann unmittelbar zur Beschneidung des Suchraumes benutzt werden, um damit wiederum tiefer in den Suchraum hineinzutauchen. Dieses Verfahren wird von manchen Autoren als *Bootstrapping* bezeichnet, was soviel heißt, daß man sich an den eigenen Haaren aus dem Sumpf zieht.

7.4.1 Erste Experimente

Wir testen ADP^* in der *Pushes* Variante von Sokoban. Mit unserem Ansatz haben wir 13 der 90 Testprobeme gelöst. Jede Spielstufe wurde fünfmal aufgerufen. Jeweils für die unterschiedlichen Aufteilungsregeln *vk* und *sa* wurde ADP^* einmal gestartet. Der sich ergebende *Teilpositionenspeicher* wurde in beiden Fällen danach in einer *lernfreien* Version genutzt, um ein Ziel zu erreichen. Die Resultate wurden verglichen mit den Expansionszahlen in A^*. Bei 2 Millionen gespeicherten Zuständen haben wir die Suche abgebrochen.

		A^*	ADP^* mit vk		ADP^* mit sa	
Level	Tiefe	Expansion	Muster	Expansion	Muster	Expansion
1	97	45	141	45	134	44
2	131	4144	220	2764	127	2485
3	134	468	204	167	143	252
4	355	>691320	1001	668522	241	> 291745
5	143	>822955	65	413541	55	>370385
6	110	1807	161	69	23	88
7	88	231752	125	159589	156	156370
17	213	>1068770	841	738254	255	>1059749
38	81	826069	331	38178	161	83929
78	136	159	14	159	9	159
80	231	3498	91	706	815	2996
82	135	>93969	122	563152	81	>93218
83	194	271	124	150	22	259

Tabelle 7.1: Lösungen in Sokoban mit und ohne Beachtung von Sackgassenmustern.

Tabelle 7.1 zeigt die Anzahl der Bälle, die optimale Lösungslänge, die Anzahl der Knotenexpansionen für die unterschiedlichen Suchverfahren und die Anzahl der gelernten Sackgassenmuster im *Teilpositionenspeicher*.

Die Aufteilung mit Hilfe der verbundenen Komponenten führt sowohl zu einer größeren Vielzahl von Mustern als auch zu kleineren Mustern. Es gilt der Grundsatz: Je kleiner die Muster, desto größer ist der Effekt der Generalisierung. Daher tritt die Verringerung der Expansionszahlen gegenüber der Aufteilung durch die Abstreifung eines Balles mehr hervor. Da beide Aufteilungsstrategien sich stark unterscheiden, ist die beobachtete Performanz bei den gestellten Problemen nicht sehr homogen, so daß es naheliegt, beide Verfahren durch die Zusammenführung ihrer *Teilpositionenspeicher* miteinander zu kombinieren.

7.4.2 Verwandte Arbeiten

Kürzlich haben Junghanns und Schaeffer unabhängig eine zu dem obigen Ansatz verwandte Strategie *pattern search* zur Erzeugung von Sackgassenmustern zur Lösung von Sokoban vorgeschlagen (Junghanns & Schaeffer 1998b). Ihr Programm löst 29 der 90 Benchmarkprobleme und demonstriert die Wichtigkeit der Sackgassenerkennung in Sokoban.

In diesem Abschnitt vergleichen wir die zentralen Ideen in beiden Ansätzen. Junghanns und Schaeffer erkennen Sackgassen durch eine Strategie der iterativen Vergrößerung. Sie beginnen mit einem Einballproblem. Nach der Lösung des n-Ballproblems verfolgen sie den Weg des Männchen und des Balles, um einen Ball zu finden, der im Konflikt mit der Lösung steht. Daraufhin wird für die folgende Iteration durch Integration dieses *Störenfrieds* ein $(n + 1)$-Ballproblem generiert und gelöst. Dieses wird so lange verfolgt, bis letztendlich eine konfliktfreie Lösung gefunden, eine gesetzte Tiefenschranke überschritten oder eine Sackgasse aufgespürt wird. In letzterem Fall werden die Muster durch stete Wegnahme eines Balles minimiert, um die größtmögliche Generalisierung zu gewinnen.

Diese Strategie steht sozusagen komplementär zur *sa*-Heuristik: Im Gegensatz dazu, einen *kritischen* Ball zu einer Menge nacheinander hinzuzufügen, entnehmen wir einzelne, unabhängig voneinander lösbare Bälle. Da in unserem Ansatz in jeder Iteration nur ein Einballproblem gelöst wird, ist unsere Lösung effizienter. Auf der anderen Seite nutzen Junghanns und Schaeffer als Nebenprodukt die Verbesserung der unteren Schranke. Die Schranke ergibt sich aus der Sackgassenteilsuche, selbst wenn sich diese nicht zu einem Sackgassenmuster führt. Deshalb ist der wesentliche Erfolg ihres Ansatzes in der Verbesserung der heuristischen Bewertung zu sehen, eine zu den heuristischen Datenbanken vergleichbare Idee. Unser Interesse galt der Erkennung von reinen Sackgassenmustern, doch können die Lösungen der Teilprobleme, die sich insbesondere durch die *vk*-Dekomposition ergeben, die heuristische Bewertung in gleicher Art und Weise unterstützen.

Die Idee, mit Hilfe einer Aufwärtspropagierung weitere stärker strukturierte Muster zu finden, wurde von Junghanns und Schaeffer nicht untersucht. Da sie jedoch weit höher innerhalb des Suchbaumes auftauchen, geben sie zu einer sehr frühen Beschneidung Anlaß. Die Aufwärtspropagierung kann in Zusammenhang mit der *pattern search* Methode bei der Bewältigung ungelöster Benchmarkprobleme dienen.

Es gibt eine Vielzahl von verschiedenartigen Ansätzen, Sackgassen zu erkennen, die insgesamt zu einer großen Menge von Mustern führen. In der Ausarbeitung von Junghanns und Schaeffer wird entgegen obiger Darstellung die effiziente Implementation des Teilpositionenspeichers nicht thematisiert.

Die angegebenen Expansionszahlen können leider nicht mit Junghanns und Schaeffer verglichen werden, da sie den unterschiedlichen Basissuchanatz *IDA** verwenden (vergl. Kapitel 3). Jedoch muß anerkannt werden, daß in unseren Resultaten nicht so viele Spielstellungen gelöst wurden.

Die Suche nach Sackgassenmustern dominiert die Suchkosten. Während wir einen uniformen Suchalgorithmus, basierend auf einer Vorrangwarteschlange beschreiben, nutzen sie eine spezialisierte Prozedur mit einer vereinfachten Zielbedingung und Heuristik. Verschiedene Parameter kontrollieren die Initialisierung, den Schwellwert an maximalen Expansionen und die Mustergröße. Die Autoren selbst schreiben: “there are numerous parameter in the search, each of which can be tuned for maximal performance” (es gibt eine Vielzahl von Parametern in der Suche, die zu einer maximalen Performanz eingestellt werden können). *Rolling Stone* wurde in mehr als ein und einem halben Jahr entwickelt und *state of the art* innerhalb dieser Problemdomäne.

Ginsberg schlägt ein im Zweipersonenspiel einsetzbares Verfahren mit Namen *partition search* vor (Ginsberg 1996). Ein entscheidender Punkt in seinem

Ansatz ist im Gegensatz zur Speicherung von Einzelpositionen die Verwaltung von Positionsmengen. Da auch er in seiner Arbeit nicht auf Implementationsdetails eingeht, schlagen wir hiermit den Teilpositionenspeicher für die effiziente Realisierung von *partition search* vor.

7.5 Ausblick

Eine zentrale Herausforderung in der heuristischen Suche ist die Erkennung von Sackgassen, d.h. Positionen, von denen aus auf keinem Zweig des Suchbaumes eine Lösung gefunden werden kann. Der Algorithmus ADP^* entdeckt und generalisiert Sackgassenmuster, die zur Suchraumbeschneidung genutzt werden können.

Generalisierung bedeutet, daß der Grund für eine Sackgasse häufig an wenigen Merkmalen der Stellung festgemacht werden kann, die somit auch Aussagekraft für weitere Stellungen haben. Einige Sackgassenmuster werden durch die domänenabhängige Prozedur *Stuck* gefunden, die mit Hilfe der Aufteilung und Aufwärtspropagierung zu größeren Mustern verbunden werden können.

ADP^* kann als ein Verfahren zur maschinellen Suchraumbeschneidung angesehen werden und ist somit konzeptionell mit der Suchraumbeschneidung mit endlichen Automaten eng verwandt (vergl. Kapitel 6) . Für einen bestehenden Automat kann die Generalisierungsanfrage inkrementell mit konstantem Platz und in konstanter Zeit durchgeführt werden. Eine wichtige theoretische Fragestellung, die durch unseren Algorithmus aufgeworfen wird, ist, ob der *Teilpositionenspeicher* in diesem Sinne optimal inkrementell gestaltet werden kann.

Kapitel 8

Lernen kürzester Wege

In diesem Kapitel betrachten wir die Jagd nach beweglichen Zielen und studieren, wie die kürzesten Wege in Iteration i berechnet werden können, wenn wir die berechneten Werte aus der Iteration $i-1$ schon kennen. Das Hauptinteresse liegt darin, den Durchlauf durch die gesamte bisher betrachtete Exploration zu vermeiden. Falls der Jäger eine noch nicht bekannte Position betritt, so ist die Entscheidungszeit konstant. Der Fall, in dem der Knoten schon betreten worden ist, wird von uns genau untersucht. Das vorgestellte Verfahren zur Aktualisierung kürzester Wege kann am besten als eine dynamische Version des Algorithmus von Dijkstra aufgefaßt werden. Ein Lerneffekt des Verfahrens tritt dann ein, wenn nach dem Aufspüren eines Zieles ein weiteres, sich bewegendes Ziel innerhalb des Suchraumes ausgemacht wird.

8.1 Bewegliche Ziele

Die Jagd nach einem beweglichen Ziel ist in der Tierwelt ein lebenswichtiges Unterfangen. Dem aus dem Volksmund entliehenen *fressen und gefressen werden* geht zumeist eine, wenn auch von Instinkten gesteuerte, durchaus trickreiche, letztendlich doch unbarmherzige Verfolgungsjagd voraus. Innerhalb der Jagd müssen beide Beteiligte auf die Reaktion des jeweils anderen kurzfristig reagieren. Von einem simplen Fangspiel wie *Schnitzeljagd* oder *Räuber und Gendarm* bis hin zu einer dramatischen *Autoverfolgung* – auch im menschlichen Leben gibt es Situationen, in denen wir versuchen, ein sich bewegendes Ziel einzufangen.

8.1.1 Modell

Wir vereinfachen die dargestellten sehr komplexen Situationen der Verfolgung auf die Zustandsraumsuche, d.h. die Züge des Jägers und des Gejagten sind durch Zustandsübergänge innerhalb eines das Problem repräsentierenden ungerichteten Graphen gegeben (vergl. Abbildung 8.1). Dabei werden wir uns insbesondere auf die Aktualisierung der kürzesten Wege innerhalb der explorierten Teilgraphen konzentrieren.

Die Suche nach beweglichen Zielen ist demnach ein Ansatz der Zustandsraumsuche, um den Veränderungen der Situation innerhalb der Suche Rech-

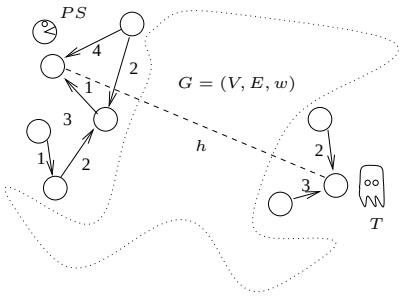


Abbildung 8.1: Die Jagd auf ein bewegliches Ziel in einem Graphen.

nung zu tragen. Wir betrachten das folgende Modell: Ein Problemlöser (engl. problem solver) befindet sich in einer ihm unbekannten Umgebung, in der er ein sich bewegendes Ziel (engl. target) jagen und einfangen soll. Um dem Löser eine Chance einzuräumen, soll er zu jeder Zeit eine optimistische Schätzung der Entfernung zum Ziel haben und sich etwas schneller als das Ziel bewegen können. Die gewonnene Information über den von ihm explorierten Suchraum darf der Problemlöser in einer geeigneten Datenstruktur verwalten. Weiterhin soll er die absolute Position des Zieles kennen, insbesondere um festzustellen, ob das gejagte Ziel in sein Territorium eingedrungen ist.

Ishida und Korf versuchen, die Schätzwerte der kürzesten Wege zwischen je zwei Zuständen innerhalb des Problemraumes (engl. all pair shortest paths) durch einen Realzeitansatz zu verbessern (Ishida & Korf 1991). Das in Kapitel 9 zur Verbesserung unterer Schranken besprochene *LRTA** Verfahren kann leicht erweitert werden: Für je zwei Zustände im Problemraum wird ein Schrankenwert dynamisch verwaltet, der den derzeitigen Schätzwert der Distanz zwischen diesen Zuständen beschreibt. Leider steigt der Speicheraufwand hierbei quadratisch im Verhältnis zum Wachstum des Suchbaumes, was für die sowieso als groß angenommenen Suchräume schnell zur Erschöpfung der Ressourcen führt.

8.1.2 Spurensuche

Deshalb liegt es nahe, die kürzesten Wege aller expandierten Knoten zum derzeitigen Standort in jedem Schritt zu aktualisieren. Innerhalb des vorgestellten Modells der Suche kann der Problemlöser die Menge der expandierten Knoten (seine eigenen und evtl. die des beweglichen Zieles) innerhalb einer sogenannten *Karte* verwalten.

Chimura und Tokoro nennen ihre, auf einer solche Repräsentation aufbauenden Jagd *trailblazer search*, was einer auf diese Karte begründeten *Spurensuche* gleichkommt (Chimura & Tokoro 1994). Die erste als *Suche* bezeichnete Phase besteht aus zwei verschiedenen, sich nicht überlappenden Spuren, die des Problemlösers und die des Zieles. Dabei versucht der Jäger, die Spur des Gejagten unter Zuhilfenahme der heuristischen Bewertungsfunktion aufzuspüren, in dem er bei der Auswahl der Nachfolgerzustände die Zustände mit geringerer heuristischer Bewertung bevorzugt. Falls sich die Spuren des Zieles und des Problemlösers kreuzen, wird die *Jagdphase* eingeleitet, in der die verwaltete Information kürzester Wege direkt genutzt wird. Da der Problemlöser

schneller als das Ziel ist, wird er somit letztendlich das Ziel einfangen.

Die Verwaltungskosten der Karten innerhalb der *Spurensuche* von Chimura und Tokoro sind sehr hoch, da in jedem Schritt die kürzesten Wege aller expandierten Zustände beider Spuren zu den derzeitigen Positionen von Problemlöser und Ziel von neu auf berechnet werden. Dazu verwenden die Autoren den Algorithmus von Dijkstra in seiner ursprünglichen Form.

Die Entscheidung in Zeit $O(n^2)$ ist für die Lösung großer Realzeitprobleme zu langsam. Dahingehend wurde der Trailblazer Ansatz durch die Einführung von Teilproblemräumen und einer sogenannten *abstrakten Karte* erweitert (Sasaki, Chimura & Tokoro 1995). Die Teilproblemstruktur muß dem Problemlöser im vornherein bekannt sein oder zumindest durch eine Lernphase möglichst präzise vorausbestimmt werden. Das die Entscheidungszeit $O(n^2)$ verbessernde Resultat von $O(n^{4/3})$ läßt sich nur auf gitterähnliche Strukturen anwenden, in denen schon eine Breitensuche die kürzesten Wege in $O(n)$ bestimmt. Weiterhin ist gerade in Labyrinthen ein Zusammenziehen des unterliegenden Gitters zu einem gewichteten Graphen naheliegend.

Das Hauptargument gegen den Ansatz von Chimura und Tokoro und gegen den Ansatz von Sasaki et al. richtet sich dagegen, die kürzesten Wege in der Suchphase überhaupt zu berechnen. Da der Algorithmus von Dijkstra sowieso keine Information aus den schon berechneten Wegen zu der soeben verlassenen Position nutzt, reicht es vollkommen aus, die Berechnung erst dann anzustoßen, wenn sich die Spuren von Problemlöser und Ziel kreuzen.

8.2 Aktualisierung kürzester Wege

Wir betrachten die erste Phase innerhalb der Jagd, in der sich die beiden Spuren von Löser und Ziel noch nicht überschneiden (vergl. Abbildung 8.1). Somit können wir das Problem der Aktualisierung der kürzesten Wege von Problemlöser und Ziel unabhängig voneinander betrachten. Ohne Beschränkung der Allgemeinheit beschreiben wir demnach die Veränderungen des *kürzesten Wegebaumes*, die sich durch die Bewegung des Problemlösers ergeben. Seien u und v die Positionen vor und nach dem i-ten Standortwechsel und $G_i = (V_i, E_i)$ der Explorationsgraph der Iteration i. Wie üblich beschreiben wir mit $\Gamma(u)$ die Menge der Nachfolgerzustände von u und mit H die Hashtabelle, in der die Zustände V_i, inklusive ihren Vorgängen, im kürzesten Wegebaum abgelegt werden.

Wenn wir uns vom Knoten u zum Knoten v bewegen, dann drehen wir zuerst die Vorgängerbeziehung zwischen u und v um. Somit erhalten wir einen Baum mit Wurzel v. Falls v nicht schon vorher expandiert worden ist, ergibt sich der kürzeste Wegebaum zu v unmittelbar aus einer gerichteten Kante von u nach v. Dies ist durch die gerichtete Suche ein durchaus häufig vorkommender Fall, da wir uns bei jeder Expansion für den Problemlöser immer zu dem Knoten geringster Bewertung hinwenden, den wir noch nicht besucht haben. Damit verhindern wir oszillierende Zustände aus Abbildung 8.2. Die Vergrößerung des Suchhorizontes wird demnach der direkten Zielverfolgung vorangestellt. Erst in einer Sackgasse kehren wir so lange zu dem Vorgängerknoten zurück, bis wir einen noch nicht untersuchten Zweig innerhalb des Graphens entdecken.

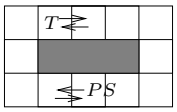


Abbildung 8.2: Eine Oszillation in der Jagd.

8.2.1 Die Durchführung eines Schrittes

Sei $\delta_i(z)$ die Länge des kürzesten Weges von z zur Wurzel von G_i. Weiterhin bezeichnen wir mit T_u den von u aufgespannten kürzesten Wegebaum und mit T_v den von v aufgespannten Teilbaum in G_{i-1}. (linke Seite in Abbildung 8.3). Die Teilbäume T_u und T_v liefern eine Partition von V_i. Seien *Closed* und *Open* die Menge der Knoten in T_v bzw. T_u, die durch eine Kante verbunden sind.

Die Aktualisierung der kürzesten Wege erfolgt durch die Exploration des durch *Open* gegebenen Horizonts in die Menge T_u hinein, bis die Länge des alten Pfades in T_u hin zu v die Länge des neuen, über *Open* angebotenen Weges, übersteigt (vergl. rechte Seite in Abbildung 8.3). Dazu senden (engl. broadcast) wir die initial erzielte Verbesserung an die anliegenden Knoten aus, bis letztendlich der Verbesserungswert verebbt.

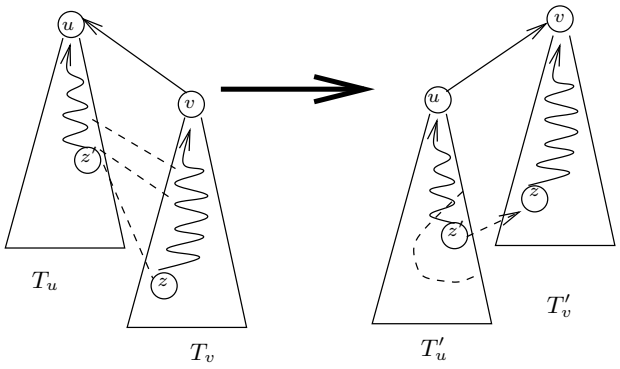


Abbildung 8.3: Die Verbreitung der Information kürzerer Wege.

Nach der Aktualisierung aller kürzesten Wege innerhalb des Graphens bestimmen wir letztendlich die neue Position der Knoten u und v.

Programm 8.1 faßt die Schritte eines Überganges von u nach v algorithmisch zusammen. Der Aufruf $\mathit{IncreaseKey}(\mathit{Open}, z', z'.\phi)$ aktualisiert den ϕ Wert von z' in *Open*, oder, falls z' noch nicht in *Open* existiert, wird z' mit diesem Wert neu eingefügt.

Die δ Werte der Knoten z und z' entsprechen $\delta_{i-1}(z) - w(u,v)$ bzw. $\delta_{i-1}(z') + w(u,v)$ und werden in einem Aufwärtsgang des kürzesten Wegebaumes berechnet (siehe Programm 8.2). Dabei stellen wir mit einer Markierung *calc* fest, ob der Wert innerhalb einer Iteration schon einmal berechnet wurde oder nicht.

Um ein Rücksetzen der Werte *calc* zu vermeiden, empfiehlt sich, die Iterationsnummer i zur Unterscheidung der Knoten verschiedener Läufe zusammen mit dem Knoten in der Hashtabelle abzulegen.

Programm 8.1 *DoMove*: Die Aktualisierung des kürzesten Wegebaumes beim Übergang von einem alten auf einen neuen Knoten. Eingabe: Alte Position u und neue Position v, kürzester Wegebaum aller generierter Knoten zu u. Ausgabe: Kürzester Wegebaum aller generierter Knoten zu v.

```
u.pred ← v; v.pred ← ∅                          {v ist die neue Wurzel}
if (v ∈ H)                                      {v wurde vorher noch nicht erreicht}
  for all e = {z', z} in T_u × T_v              {z' ist in Open und z ist in Closed}
    BuildPath(z'); BuildPath(z)                 {berechne z.δ und z'.δ}
    φ(z') ← z'.δ − (z.δ + w(z, z'))             {Verbesserung am Knoten z'}
    if (φ(z') > 0)                              {die Alternative ist besser}
      z'.pred ← z                               {ändere den Vorgänger}
      z'.δ ← z.δ + w(z, z')                     {ändere den Schätzwert}
      z'.φ ← z'.φ + φ(z')                       {ändere die Verbesserung}
      IncreaseKey(Open, z', z'.φ)               {setze oder aktualisiere z'.φ}
  Broadcast(Open)                               {propagiere die Aktualisierungen}
u ← v; v ← Choice                               {wähle neue Positionen u und v aus}
```

Wenden wir uns nun der *Broadcast* Prozedur zu. Bis die *Vorrangwarteschlange* leer wird, entnehmen wir den Knoten z mit der höchsten Verbesserungszahl ϕ und berechnen für alle Nachfolger die sich ergebende Verbesserung. Gemäß dieser Bewertung werden diese Nachfolger in der Vorrangwarteschlange *Open* eingefügt bzw. aktualisiert.

Wenn in Programm 8.3 der Nachfolger z' von z aus erstmalig erreicht wird, dann sollte der in *BuildPath* berechnete Wert $z'.\delta$ dem Wert $\delta_{i-1}(z') + w(u, v)$ entsprechen. Da jedoch der δ Wert des ersten Knotens x auf dem Pfad p zu z' sich im Verlaufe des Verfahrens ändern kann, müssen wir $\delta_{i-1}(z')$ durch die Information von $x.\phi$ und $x.\delta$ erschließen. Dies wird über die Erhöhung des Wertes $y.\delta$ durch $x.\phi$ aller y auf p gewährleistet.

Programm 8.2 *BuildPath*: Berechung des kürzesten Wegewertes der Eingabe und aller seiner Vorgänger. Eingabe: Knoten z.

```
p_1 ← z; i ← 1                                  {starte mit Knoten z}
while (x ≠ v) and not (p_i.calc)                {bis x schon bekannt oder Wurzel ist}
  p_i ← p_{i−1}.pred; i ← i + 1                 {betrachte nächsten Knoten auf dem Pfad}
if (v = p_i)                                    {die Wurzel ist erreicht}
  v.δ ← 0; v.φ ← 0 v.calc ← 1                   {kürzester Weg ist null}
for all j in {i − 1, ..., 1}                    {traversiere Pfad rückwärts}
  p_j.δ ← p_{j+1}.δ + p_{j+1}.φ + w(p_{j+1}, p_j)   {berechne δ auf dem Pfad}
  p_j.φ ← 0; p_j.calc ← 1                       {Verbesserung an p_j ist null}
```

Programm 8.3 *Broadcast*: Die Information über mögliche Abkürzungen wird in die Nachbarschaft versendet, bis sie letztendlich versiegt. Eingabe: Vorrangwarteschlange *Open*. Ausgabe: Aktualisierter Baum kürzester Wege.

```
while (Open != ∅)
  z ← DeleteMax(Open)                    {erweitere die Menge Closed um z}
  for all z' in Γ(z) ∩ H                       {für alle Nachfolgerknoten}
    BuildPath(z')                                        {berechne z'.δ}
    φ(z') ← z'.δ − (z.δ + w(z, z'))             {Verbesserung am Knoten z'}
    if (φ(z') > 0)                             {die Alternative ist besser}
      z'.pred ← z                                      {ändere den Vorgänger}
      z'.δ ← z.δ + w(z, z')                          {ändere den Schätzwert}
      z'.φ ← z'.φ + φ(z')                              {ändere Verbesserung}
      IncreaseKey(Open, z', z'.φ)                {aktualisiere z'.φ in Open}
```

8.2.2 Korrektheit

Die *Broadcast* Prozedur ist mit der Prozedur von Dijkstra verwandt, in der die Vorrangwarteschlange gemäß den δ Werten geordnet ist (vergl. Kapitel 3).

Hilfssatz 6 *Sei V_i die Knotenmenge des Explorationsgraphen G_i der Iteration i und C_i die Teilmenge von Knoten in V_i, die schon mit calc markiert sind. Weiterhin sei u der Wurzelknoten von G_{i-1} und v der Wurzelknoten von G_i. Für alle Knoten y aus C_i gilt die folgende Gleichung*

$$y.\phi = \delta_{i-1}(y) + w(u, v) - y.\delta. \tag{8.1}$$

Beweis: Die δ Werte für jeden Knoten y in C_i werden in Iteration i erstmalig in der Prozedur *BuildPath* gesetzt. Sei der $p = (x = p_i, \ldots, p_1 = z)$ der von der Prozedur *BuildPath* in der Aufwärtsbewegung erzeugte Pfad. Nutzt man das behauptete Resultat als Invariante, so gilt für den Basisknoten x die Gleichung $x.\phi + x.\delta = \delta_{i-1}(x) + w(u, v)$. unmittelbar. Für den nächsten Knoten p_{i-1} gilt $p_{i-1}.\phi = 0$ und

$$\begin{aligned} p_{i-1}.\delta &= x.\phi + x.\delta + w(x, p_{i-1}) \\ &= \delta_{i-1}(x) + w(u, v) + w(x, p_{i-1}) \\ &= \delta_{i-1}(p_{i-1}) + w(u, v). \end{aligned}$$

Demnach ist die Gleichung 8.1 für p_{i-1} erfüllt. Für alle anderen Knoten p_j für j aus $\{i-2, \ldots, 1\}$ wird die Gleichung 8.1 gleichsam vom Vorgängerknoten p_{j+1} geerbt.

Bleibt demnach zu zeigen, daß die Gleichung 8.1 nach jeder Zuweisung von δ und ϕ innerhalb der Programme *DoMove* und *Broadcast* gilt. Sei y erstmalig über den Knoten x erreicht, dann wird y in die Vorrangwarteschlange *Open* aufgenommen. Weiterhin sei p der in *BuildPath* generierte Aufwärtspfad.

Da y auf p liegt, ist der alte Wert $y.\delta$ nach der obigen Betrachtung gleich $\delta_{i-1}(y) + w(u, v)$ und der alte Wert $y.\phi$ gleich null. Der neue Wert $y.\delta$ wird

durch $x.\delta + w(x,y)$ festgelegt. Somit erhalten wir für $y.\phi$ die Beziehung

$$y.\phi = \delta_{i-1}(y) + w(u,v) - (x.\delta + w(x,y)).$$

Daraus folgt Gleichung 8.1 wie behauptet.

Nun nehmen wir an, daß y abermalig erreicht wird. Wenn $\phi(y)$ nicht größer als null ist, dann bleibt Gleichung 8.1 richtig, da keine Veränderung der Variablen durchgeführt wird. Sonst wird y über einen Knoten, sagen wir x', erreicht und $y.\delta$ wird auf $x'.\delta + w(x',y)$ gesetzt. Weiterhin wird $y.\phi$ um $\phi(y)$ erhöht. Um den alten und neuen Wert von $y.\phi$ voneinander zu unterscheiden, benutzen wir ein Apostroph im ersten Fall. Induktiv erhalten wir

$$\begin{aligned} y.\phi &= \phi(y) + y.\phi' \\ &= y.\delta' - (x'.\delta + w(x',y)) + y.\phi' \\ &= \delta_{i-1}(y) + w(u,v) - y.\delta. \end{aligned}$$

□

Hilfssatz 7 *Sei $\widehat{\phi}(z)$ für z durch $\delta_{i-1}(z) + w(u,v) - \delta_i(z)$ festgelegt und sei y derjenige Nachfolger von x, der auf dem kürzesten Weg von v nach y liegt. Dann gilt $\widehat{\phi}(x) \geq \widehat{\phi}(y)$.*

Beweis: Wendet man die Definition von $\widehat{\phi}$ auf x und y an, so erhält man

$$\widehat{\phi}(x) = \delta_{i-1}(x) + w(u,v) - \delta_i(x)$$

und

$$\widehat{\phi}(y) = \delta_{i-1}(y) + w(u,v) - \delta_i(y).$$

Angenommen $\widehat{\phi}(x)$ ist kleiner als $\widehat{\phi}(y)$. Dann haben wir

$$\begin{aligned} 0 > \widehat{\phi}(x) - \widehat{\phi}(y) &= \delta_{i-1}(x) - \delta_{i-1}(y) - \delta_i(x) + \delta_i(y) \\ &= \delta_{i-1}(x) - \delta_{i-1}(y) + w(x,y). \end{aligned}$$

Demnach ist $\delta_{i-1}(x) + w(x,y)$ echt kleiner als $\delta_{i-1}(y)$ im Widerspruch zu der Dreiecksungleichung, die für kürzeste Wege gelten muß. □

Der Wert $\widehat{\phi}$ entspricht der Gesamtverbesserung, die an den jeweiligen Knoten gemacht werden kann.

Satz 28 *(Korrektheit DoMove) Die DoMove Prozedur ist korrekt, d.h. sie berechnet den kürzesten Wegebaum aller expandierten Knoten zur neu betretenen Wurzel.*

Beweis: Wir haben schon angemerkt, daß der kürzeste Wegebaum zu einem Knoten u die Eigenschaft an einen erstmalig betretenen Knoten v vererbt, da v von allen Knoten eben nur via u erreicht werden kann.

Den verbleibenden Beweis teilen wir in zwei Teile auf. Auf der einen Seite zeigen wir, daß der kürzeste Wegebaum für alle expandierten Knoten in T_u korrekt ist. Auf der anderen Seite beweisen wir die Korrektheit des kürzesten Wegebaumes für den Rest der Knoten in T_u.

Betrachten wir den ersten Fall. Es genügt zu zeigen, daß zu der Zeit, zu der der Knoten u expandiert wird, die kürzesten Wege (gegeben durch die Vorgängerbeziehung) korrekt sind, da sie danach nicht mehr verändert werden. Angenommen, dies wäre nicht der Fall. Dann gibt es einen Pfad p in dem resultierenden kürzesten Wegebaum mit $q.\delta$ ungleich $\delta(v,q)$. Zu der Zeit, in der q expandiert wird, existiert ein x in *Closed* und ein y in *Open* auf p. Sei (x,y) minimal in dieser Eigenschaft. Da alle Elemente in der Vorrangwarteschlange mit *calc* markiert sind, können wir Hilfssatz 7 für diese Knoten anwenden.

$$\begin{aligned} y.\phi &= \delta_{i-1}(y) + w(u,v) - y.\delta \\ &= \delta_{i-1}(y) + w(u,v) - (\delta_i(x) + w(x,y)) \\ &= \widehat{\phi}(y) \geq \ldots \geq \widehat{\phi}(q) \\ &= \delta_{i-1}(q) + w(u,v) - \delta_i(q) \\ &> \delta_{i-1}(q) + w(u,v) - q.\delta = q.\phi \end{aligned}$$

Da q und nicht y expandiert worden ist, übersteigt der Wert $y.\phi$ nicht $q.\phi$. Dies ist jedoch ein Widerspruch zu der angeführten Gleichungskette. Somit ist $q.\delta$ gleich $\delta(v,q)$.

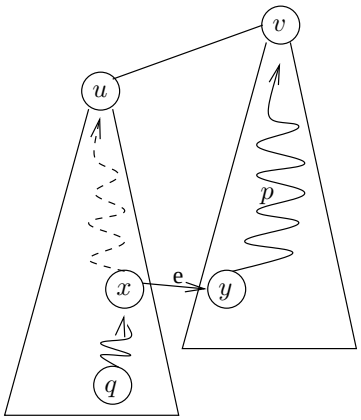


Abbildung 8.4: Die Beziehungen der beiden Mengen $T_u - \mathit{Closed}$ und $T_v \cup \mathit{Closed}$ untereinander.

Uns verbleibt der Beweis, daß die kürzesten Wege auch für die Knoten aus $T_u - \mathit{Closed}$ korrekt berechnet werden. Nehmen wir auch hier an, dies wäre für den Knoten q innerhalb des Verfahrens erstmalig nicht der Fall.

Dann existiert ein kürzester Weg p von q zu v, der die Kante $\{u,v\}$ nicht beinhaltet. Somit verbindet p die Mengen $T_u - \mathit{Closed}$ und *Closed* über eine Kante $e = \{x,y\}$ miteinander (vergl. Abbildung 8.4). Zu der Zeit, als y expandiert worden ist, war der Wert $y.\delta$ schon korrekt. Die Zuweisung von $x.\delta$ auf $y.\delta + w(u,v)$ wurde nicht ausgeführt, da der Vorgänger von x nicht gleich y ist. Damit war $x.\delta$ schon vorher richtig. Somit muß ein anderer Pfad p' von q nach v existieren, der die Kante e und die Kante $\{u,v\}$ nicht betritt. Nacheinander werden nun alle Verbindungen zwischen *Closed* und $T_u - \mathit{Closed}$ ausgeschlossen, was uns zu einem Widerspruch führt. Damit sind auch die kürzesten Wege

in der Menge $T_u - Closed$ korrekt. □

Die Argumentation im ersten Teil des Beweises ist ähnlich zu dem Korrektheitsbeweis des Verfahrens von Dijkstra (vergl. Kapitel 3). Wir haben hierbei Hilfssatz 7 und Hilfssatz 6 genutzt, um die ϕ Werte mit den δ Werten zu verbinden.

Wir kommen nun zu einem in Abbildung 8.5 dargebotenen worst case Beispiel. Da die Vorgängerrelation aller Knoten sich beim Wechsel von u nach v ändert, müssen wir den kürzesten Wegebaum des gesamten in der Hashtabelle abgelegten Graphen traversieren.

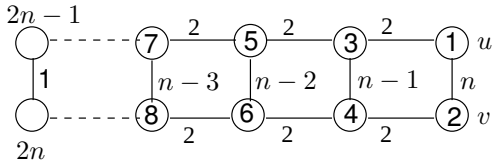


Abbildung 8.5: Ein ganzer Baum muß umstrukturiert werden.

8.2.3 Effizienz

Wir nehmen an, daß die initiale Aufteilung der Knoten in *Open* und *Closed* vollzogen ist und G' der Teilgraph von G mit n' Knoten und e' Kanten ist, dessen Knoten innerhalb einer *DoMove* Operation jemals in der Vorrangwarteschlange verwaltet wurden. Wie wir empirisch belegen werden, ist n' klein im Verhältnis zu n.

Die *BuildPath* Prozedur berechnet die alten kürzesten Wegewerte innerhalb des Teilbaumes $S_v(G')$ von G, in dem alle Vorgänger zu Knoten aus G' liegen. Durch den Gebrauch von Fibonacci heaps benötigt *InreaseKey* konstante amortisierte Zeit, so daß wir insgesamt eine Laufzeit $O(n' \log n' + e' + |S_v(G')|)$ für *DoMove* erhalten.

Die Menge der erreichten Zustände G' ist im Problemgraphen miteinander verbunden, und so sind die Wege von den Blättern zu der Wurzel innerhalb des Baumes $S_v(G')$ nicht wesentlich voneinander verschieden. Demnach können wir annehmen, daß fast alle innere Knoten in $S_v(G')$ mehrere Nachfolger in $S_v(G')$ haben und daß die Größe von $S_v(G')$ somit durch $O(n')$ beschränkt ist. Unter dieser Annahme ist die Laufzeit im ungünstigsten Fall durch $O(n' \log n' + e')$ beschränkt.

Wenn $\phi_{\max} = \max\{z'.\phi | z' \in Open\}$ klein und die Kantengewichte ganzzahlig sind, dann kann die Vorrangwarteschlange durch $\phi_{\max}$ verschiedene Schubladen (engl. buckets) realisiert werden. Die Idee ist innerhalb der Zustandsraumsuche bekannt, doch gibt es hier zwei Besonderheiten zu betrachten. Erstens ist $\phi_{\max}$ wesentlich kleiner als $\delta_{\max} = \max\{z'.\delta | z' \in Closed\}$. Zweitens kann $\phi_{\max}$ schon bei der ersten Einfügung eines Elementes in *Open* ermittelt und somit die Obergrenze an benötigten Schubladen genau bestimmt werden.

Man mag fragen, ob die Verwendung einer Vorrangwarteschlange notwendig ist, oder ob man mit einer simplen Tiefensuche auskommt. Abbildung 8.6 zeigt ein Beispiel auf, in der die durch die Vorrangwarteschlange vorgegebene

Ordnung der Knotenbetrachtungen für die Korrektheit des Verfahrens entscheidend ist. Eine einfacher Tiefensuchdurchlauf zur Propagierung der Verbesserung ϕ, beginnend am Knoten z, führt zu inkorrekten kürzesten Wegen. Wenn c und b vor a expandiert werden, dann wird c fälschlicherweise der Vorgänger im kürzesten Wegebaum zu v.

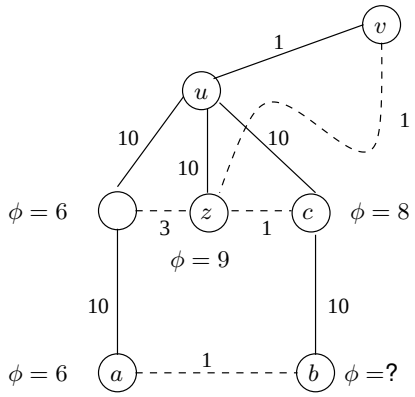


Abbildung 8.6: Die Reihenfolge der Knotenbetrachtungen ist wichtig.

Der Algorithmus kann weiter verbessert werden, indem wir die Berechnung der Verbesserungszahl an *Artikulationsknoten* a von G_i (Knoten, die durch ihre Entnahme den Graphen in mehrere Zusammenhangskomponenten aufteilen) stoppen. Innerhalb des kürzesten Wegebaumes müssen alle kürzesten Wege unterhalb des Flaschenhalses a korrekt sein, da sie sich nicht über einen anderen Weg verändern können. Die Menge der Artikulationsknoten kann dynamisch berechnet werden, indem alle Knoten innerhalb von aufgedeckten Zyklen aus dieser Menge entnommen werden können.

Es gibt drei Möglichkeiten, die Menge der Kanten zwischen T_u und T_v aufzuspüren. Als erstes können wir alle Kanten $e = \{z, z'\}$ in E_{i-1} betrachten und die Wurzeln von z und z' berechnen. Damit traversieren wir allerdings den gesamten Graphen G. Günstiger ist ein Start am Knoten v und eine Exploration des Baumes T_v, bis die Knoten aufgefunden werden, die nicht selbst in T_v liegen. Dieser auf die Vorgängerbeziehung im kürzesten Wegebaum beruhende Ansatz läßt offensichtlich in Zeit $O(|T_v|)$ durchführen. Die letzte Variante ist die analoge Untersuchung des Baumes T_u. Wir haben uns für die zweite Variante entschieden, da wir T_u durch den gerichteten Suchprozeß als sehr groß betrachten müssen.

Es ist empfehlenswert, die Menge $T_u \times T_v$ dynamisch zu verwalten. In diesem Ansatz werden die Veränderungen von $T_u \times T_v$, die sich durch die Umstrukturierungen bei der Bewegung von einem zum anderen Knoten ergeben, in jedem Schritt aktualisiert. Hierbei muß jedoch darauf geachtet werden, daß diese Verwaltung der Zwischenknoten nicht die Gesamtlaufzeit dominiert.

8.3 Experimentelle Resultate

In den Experimenten betrachten wir die Verfolgungsjagd innerhalb eines Labyrinths, das auf einem 100 mal 100 großen Gitter aufbaut.

Die Wahl des Nachbars ist abhängig von der Strategie des sich bewegenden Objektes. Sie wird *schüchtern* genannt, falls sie die heuristische Funktion zu vergrößern sucht. Falls das Objekt, wie der Problemlöser in unserem Fall, den Schätzwert jeweils verkleinert, bezeichnen wir sie als *aggressiv*. Eine interessante Beobachtung innerhalb von Labyrinthen ist es, daß ein schüchternes Ziel oft besser eingefangen werden kann als ein aggressives, da es sich schneller in Sackgassen verfängt. Eine *zufällige* Strategie würfelt aus, welcher Nachfolger beschritten werden soll.

In Abbildung 8.7 vergleichen wir die Anzahl von Knotenexpansionen in unserem dynamischen *DoMove* Verfahren mit der Anzahl der Knotenexpansionen in der Hashtabelle, d.h. in den gewählten Bezeichnern vergleichen wir n' und n. Das Kürzel a steht für eine aggressive Strategie während r die Knotenexpansionen einer zufälligen (engl. *random*) Strategie bezeichnet. Die gewählten Präfixe a und na merken an, ob das *DoMove* Verfahren mit oder ohne die Sonderbehandlung an Artikulationsknoten gestartet wurde. Eine Iteration gründet sich auf hundert *DoMove* Aufrufe, und wir haben 15 unabhängige Iterationen gestartet, um die Streuung der Werte zu veranschaulichen.

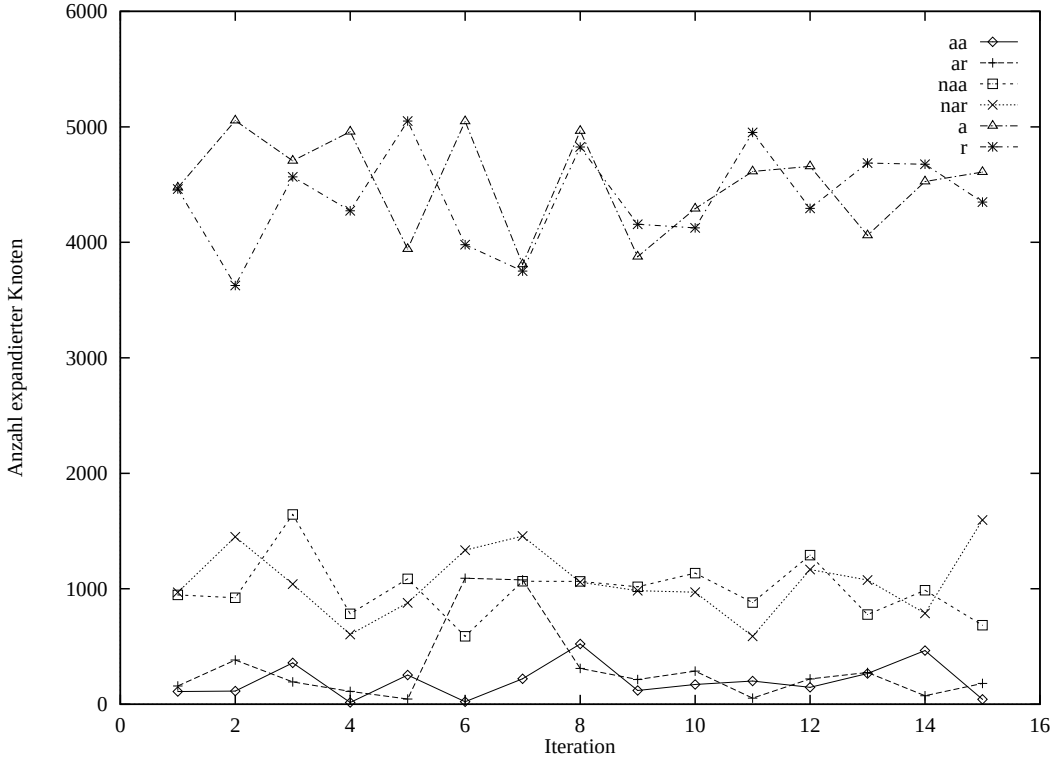


Abbildung 8.7: Experimentelle Resultate für Aktualisierung kürzester Wege.

Wenn sich der Problemlöser aggressiv verhält und wir das Abbrechen der Aussendung an Artikulationen nutzen, dann expandiert unser Verfahren zwischen einen Prozent (Iteration 4,6 und 15) und zehn Prozent (Iteration 8) der in der Hashtabelle gespeicherten und somit der im Algorithmus von Dijkstra betrachteten Knoten. Ohne Artikulationsknoten gewinnen wir ca. drei Viertel an Knotenexpansionen im Vergleich zu Chimura and Tokoro.

Der über die 15 Iterationen maximale betrachtete ϕ Wert ist 10.5 bei einer zufälligen Strategie und 8.5 bei einer aggressiven Strategie, während die maximalen δ Werte hingegen bei 36 bzw. bei 40.3 liegen.

8.4 Ausblick

Der Algorithmus von Dijkstra ist einer der berühmtesten Verfahren in der Informatik. Die Lücke (engl. gap) zwischen der besten unteren Schranke, die durch die Eingabelänge $O(n+e)$ gegeben ist, und des schnellsten Algorithmus ist - mit Ausnahme spezieller Graphklassen, wie z.B. azyklisch gerichtete oder uniform gewichtete Graphen - bis heute nicht geschlossen worden.

Eine offene Frage ist, ob die hier präsentierte dynamische Variante des Verfahrens zur Berechnung kürzester Wege in diesem Sinne optimal gestaltet werden kann. Immerhin besitzen wir die Zusatzinformation über die kürzesten Wege zu einem Nachbarknoten.

Unser Verfahren verhält sich im Mittel sehr gut, hat jedoch im ungünstigsten Fall starke Ausreißer. Interessant ist eine weitere Studie über eine zurückhaltende Restrukturierungsstrategie, die zu einer kürzeren Entscheidungszeit und damit zu einem besseren *Realzeitverhalten* führt.

Kapitel 9

Lernen der Heuristik

In diesem Kapitel beschäftigen wir uns mit der Verbesserung des Schätzwertes für die noch zu bewältigende Zieldistanz in der durch eine konstante Entscheidungszeit eingeschränkten Suche. Ausgehend von einer Fallstudie zum (n^2-1) Puzzle stellen wir zuerst die grundlegenden Verfahren RTA* und LRTA* und ihre enge Verwandtschaft untereinander vor. Im direkten Anschluß erörtern wir die leichten Verbesserungen forward updating and backward updating für LRTA*. Durch die Einbeziehung von richtungsweisenden Schildern an den Kanten des Graphen ergibt sich, daß das Minimum aller anliegenden Richtungsinformationen eine untere Schranke für die Lösung des Problems liefert. Die Idee von CRTA* ist es, diese Werte durch einen Rückwärtslauf innerhalb des Expansionspfades zu gewinnen, während in SLRTA* diese Berechnungen auf den Zeitpunkt verschoben werden, an dem wir den jeweiligen Knoten neu besuchen.

9.1 Realzeitsuche

In den meisten Situationen des alltäglichen Lebens lassen sich durchgeführte zeitkritische Entscheidungen nicht mehr oder nur schwerlich korrigieren. Gerade diese Momente geringer Information und Zeit erfordern bekannterweise gelerntes Wissen bzw. Erfahrung. Damit sind die wesentlichen Merkmale der sogenannten *Realzeitsuche* angesprochen. Sie verlangt demnach, Aktionen explizit durchzuführen, bevor die vollständigen, sich daraus ergebenden, Konsequenzen bekannt sind. Die Ursachen der Realzeitsuche sind vielfältig: Unzureichende Information über die Domäne, ein für eine optimale Entscheidungsfindung zu großer Suchraum oder eine aktive Veränderung des Suchraumes selbst, z.B. durch Interaktion mit der Umgebung.

9.1.1 Direkte Problemlösung

Ian Parberry schlägt ein Realzeitverfahren zur Lösung des (n^2-1)–Puzzles vor, das im ungünstigsten Fall $\Theta(n^3)$ viele Schritte benötigt (Parberry 1995).

Die untere Schranke wird dadurch einsichtig, da es Konfigurationen gibt, deren Manhattendistanz in $\Omega(n^3)$ liegt. Dazu wird ein Spielbrett (für gerades

n) in vier Quadranten eingeteilt und diese dann diagonal miteinander vertauscht (vergl. linke Seite in Abbildung 9.1). Da jeder der (n^2-1) Steine nun eine Distanz von n zu seinem *Heimatfeld* hat, erhalten wir eine Manhattandistanz von $n(n^2-1)$.

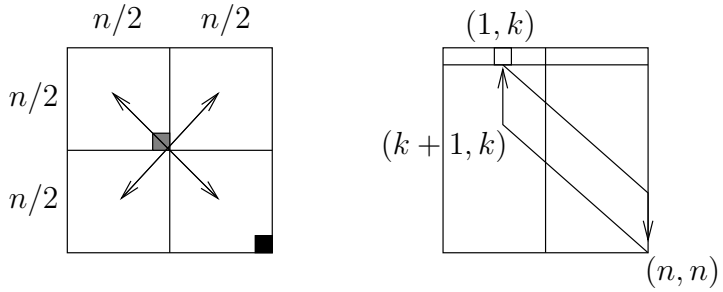


Abbildung 9.1: Untere und obere Schranke für ein Realzeitverfahren im (n^2-1)-Puzzle.

Die obere Schranke (rechte Seite in Abbildung 9.1) wird durch einen einfachen rekursiven Ansatz gelöst, in dem jeweils die obere Reihe und rechte Spalte vervollständigt werden. Die worst case Laufzeit $T(n)$ ist durch $T(n-1)+15n^2$ für n größer als zwei beschränkt. Sie ergibt sich aus folgender Betrachtung: Um $(1,k)$ für ein k zwischen und $n/2$ zu plazieren, müssen schlimmstenfalls $8n-2k-7$ Züge für den ungünstigst auf (n,n) plazierten Zielstein gemacht werden: Max. $2n-k-1$ Züge zum Erreichen des Steines, max. $6(n-k-1)$ Züge, um dieses Quadrat dann diagonal nach $(k+1,k)$ zu befördern, und $5k$ Züge, um es hinauf zum Ziel zu bewegen. Somit ergeben sich maximal $\sum_{k=1}^{n/2}(8n-2k-7)$ bzw. $15n^2/4-4n$ Züge für die erste Hälfte der oberen Reihe. Der Mehraufwand zur Integration des letzten Spielsteines in der Reihe ist kleiner als $8n$. Für die gesamte obere Reihe sind demnach weniger als $15n^2/2$ Züge notwendig. Die Auflösung der Rekursionsformel ist leicht als kubisch nachzuweisen.

Dieser Realzeitalgorithmus kann als obere Schranke in einem *branch and bound* Ansatz (vergl. Kapitel 3) genutzt werden. Er ist in seiner schichtweisen Strategie dem menschlichen Problemlösen prinzipiell sehr nahe. Das zentrale Problem des vorgestellten Ansatzes ist es jedoch, daß die Lösungsqualität weder im vornherein noch in aufeinanderfolgenden Versuchen verbessert wird. Doch gerade dann, wenn die Zeit knapp ist, wollen wir uns mit gelerntem Wissen in einer vorgegebenen zeitkritischen Situation bewähren.

9.1.2 Makroproblemlöser

Wir schauen uns als nächstes ein Realzeit-Verfahren mit vorgeschaltetem Lernmechanismus an und nehmen als Beispiel das *Acht*-Puzzle. Die Idee im *Makroproblemlöser* ist es, eine Stellung nach und nach zu komplettieren, indem aus einer Tabelle der jeweils nächste Makrozug ausgelesen wird (Korf 1985b). Der Eintrag aus Zeile r (engl. row) und Spalte c (engl. column) in Tabelle 9.1 bezeichnet die Zugsequenz, um den auf Position r befindlichen Spielstein in die Position c zu befördern, so daß sich die schon richtig postierten Spielsteine 1 bis $r-1$ nach Durchführung des Makros wieder auf ihren Plätzen befinden.

	0	1	2	3	4	5	6
0							
1	DR						
2	D	LURD					
3	DL	URDLLURD	URDL				
4	L	RULDLURD	RULD	LURRDLULDR			
5	UL	DRULDLURULDR	RDLURULD	RULDRDLUURDL	RDLU		
6	U	DLURULDR	DRUULD	DLUURDRULLDR	DRUL	LURRDDLURULLDR	
7	UR	LDRUULDR	ULDDRULURD	LDRULURDRULLDR	DLURDRUL	DRULDLURRDLU	DLUR
8	R	ULDR	LDRRUULD	LURDRULLDR	LDRRUL	DRULLDRURDLU	LDRU

Tabelle 9.1: Makrotabelle für das *Acht*-Puzzle.

Abbildung 9.2 zeigt auf, wie eine Stellung nach und nach durch die Anwendung der Makrooperatoren aus Tabelle 9.1 in die Zielstellung überführt werden kann. Für den Spielstein mit der Nummer i wird für ein von null bis sechs ansteigendes i die derzeitige Position c und die Zielposition r ausgemacht und der entsprechende Makrozug in der Tabelle angewendet, bis letztendlich die Zielstellung erreicht ist. Die Positionen der letzten beiden Steine sind durch die ersten sechs festgelegt (vergl. Kapitel 2).

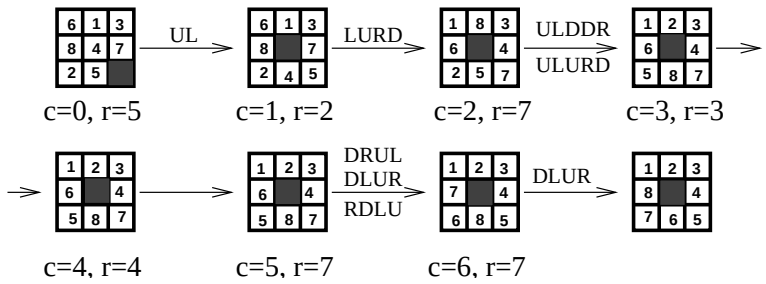


Abbildung 9.2: Makrooperatoren in der Lösung des *Acht*-Puzzles.

An der *Makrotabelle* kann man sehr gut die worst case Lösungslänge ablesen, in dem man die Spaltenmaxima aufsummiert. Für das *Acht*-Puzzle erhalten wir eine maximale Lösungslänge von

$$2 + 12 + 10 + 14 + 8 + 14 + 4 = 64.$$

Als sehr gute Schätzung für die mittlere Lösungslänge erweist sich die Summe der Spaltenmittel, also im Beispiel

$$12/9 + 52/8 + 40/7 + 58/6 + 22/5 + 38/4 + 8/3 = 39.78.$$

Die Makrotabelle kann durch eine Tiefen- oder Breitensuche vom Zielzustand aus konstruiert werden. Dazu benötigen wir Rückwärtszüge, die jedoch nicht unbedingt selbst wieder legale Züge im Spiel sein müssen. Wir wollen das zu m inverse Makro mit m^{-1} bezeichnen. Gegeben sei eine Vektordarstellung der derzeitigen Position $p = (p_0, \ldots, p_k)$, die durch m^{-1} von der Zielposition $p' = (p'_0, \ldots, p'_k)$ aus erreicht wird. Die Spalte $c(m)$ des Makros m, das also p in p' überführt, ist gegeben durch die Länge des längsten gemeinsamen Anfangsstücks von p und p', also formal durch

$$c(m) = \min\{i \in \{0, \ldots, k-1\} \mid p_i \neq p'_i\}.$$

Die Zeile $r(m)$ des Makros m entspricht der Position, auf der sich das Element $p_{c(m)}$ befindet, das im nächsten Zug auf $c(m)$ befördert werden soll. Nehmen wir zum Beispiel das Makro $m^{-1} =$**LDRU**, das die Zielposition $p' = (0, 1, 2, 3, 4, 5, 6, 7, 8)$ in die Position $p = (0, 1, 2, 3, 4, 5, 8, 6, 7)$ überführt. Das inverse Makro m ist demnach **DLUR**. Damit entspricht $c(m)$ dem Wert sechs und $r(m)$ dem Wert sieben in Übereinstimmung mit dem letzten Makrozug in Abbildung 9.2. Für größere Probleme führt eine Breitensuche zur Erschöpfung der Speicherplatzressourcen. In alternativen Suchverfahren lassen sich die Einträge innerhalb der Makrotabelle jedoch dynamisch verbessern, indem innerhalb der Suche immer das kürzeste Makro festgehalten wird.

Für das *Fünfzehn*-Puzzle findet Korf 119 Makros, die eine worst case Lösungslänge von 214 Zügen und eine mittlere Lösungslänge von 139.4 Zügen voraussagen. Für den Zauberwürfel ergeben sich 238 Makros, die eine maximale Zuganzahl von 134 und eine mittlere Zuganzahl von 86.38 ergeben. Die Lösung ist durch das Aneinanderhängen der Operatoren durchweg länger als das Optimum, doch ist der Ansatz sehr ressourcenfreundlich.

9.2 Lernen der Knotenbewertung

Angenommen, wir suchen nach einem Ziel innerhalb eines Labyrinths. An jeder Kreuzung dürfen wir obere und untere Schranken der erwarteten Lösungslänge notieren. Informationen anderer Schilder können wir nur im engsten Umkreis einsehen. Demnach müssen wir uns entscheiden, welche Richtung wir aktiv wählen. Erreichen wir erneut die Kreuzung, so können wir unseren Informationsgewinn über die Struktur des Labyrinths durch eine Verfeinerung der Beschriftungen auf den Schildern konkretisieren und unser Lösungsverhalten über die Zeit hin verbessern. Korf schlägt zum Lernen der Schätzfunktion den Algorithmus *LRTA** vor (Korf 1990). Aus pädagogischen Gründen beschreiben wir jedoch den Algorithmus Realzeit A^* – *RTA** – zuerst.

9.2.1 Realzeit A^*

Gegeben sei ein mit w gewichteter Graph G und eine untere Schranke h. Wird der Knoten u in *RTA** expandiert, so werden die folgenden zwei Schritte hintereinander ausgeführt, bis letztendlich ein Zielknoten gefunden wird.

1. Der heuristische Wert $h(u)$ wird auf den zweitbesten Wert $h(v)+w(u,v)$ aller Nachfolger v aus $\Gamma(u)$ gesetzt. Existieren keine zwei Nachfolger, so wird $h(u)$ auf unendlich gesetzt.

2. Der Algorithmus bestätigt den Übergang zu v^*, der den besten Wert $h(v) + w(u, v)$ aller v aus $\Gamma(u)$ hat.

Korf hat gezeigt, daß *RTA** lokal optimal auf Bäumen agiert, d.h. sich immer in Richtung des Horizontknotens mit geringster Bewertung wendet. Zusätzlich beweist er, daß *RTA** vollständig ist, d.h. einen Lösungspfad findet, sofern er existiert. Die Idee des ersten Beweises ist, auf dem Weg von v nach u die Bewertung $h(v)$ als das Minimum aller Horizontbewertungen $h(w) + g(v, w)$ im Teilbaum $T(u, v)$ zu erkennen, dessen Wurzel durch die Kante (u, v) beschrieben wird. Im zweiten Beweis wird aufgezeigt, daß das *RTA** Verfahren in einem endlichen Graphen aus jedem Zyklus irgendwann ausbrechen muß.

Der schlechte Teil der Nachricht wird offenbar, wenn ein Zielknoten gefunden ist. Einige der Knoten u überschätzen die Distanz zum Ziel, da ihr h-Wert immer noch auf dem zweitbesten Wert $h(v) + w(u, v)$ der sie umgebenden Knoten v aus $\Gamma(u)$ gesetzt sind. Ein erneuter Aufruf von *RTA** ist mit den veränderten Schätzwerten unter diesen Umständen nicht möglich.

Ein Weg bessere untere (oder auch obere) Schranken zu finden ist es, eine konstante Anzahl von Schritten in die Tiefe zu gehen und das Resultat von dem Horizont dieser Teilsuche aufwärts zu reichen. Dies erinnert an das *Minimax*-Verfahren aus dem Suchbereich der Zwei-Personenspiele und wird von Korf *Minimin* Schema genannt.

Tabelle 9.2 zeigt auf, daß man mit *RTA** in sehr großen Problemräumen, wie dem Fünfundzwanzig-Puzzle, zu ansprechenden Lösungen kommen kann. Wir haben das Problem (17, 1, 20, 9, 16, 2, 22, 19, 14, 5, 15, 21, 0, 3, 24, 23, 18, 13, 12, 7, 10, 8, 6, 4, 11) mit wachsendem Horizont der Teilsuche aufgerufen. Es hat eine Lösungslänge von 100 Zügen. Dabei wurde die Länge des Zielpfades und die Anzahl der insgesamt generierten Knoten gezählt.

Teilsuchtiefe	1	2	5	10	20	30
Länge Zielpfad	48392	6854	1126	752	414	304
betrachtete Knoten	160204	37634	14736	125382	938245	14242649

Tabelle 9.2: *Vierundzwanzig*-Puzzle mit *RTA** und *Minimin* Schema.

Die Laufzeit des Programms im Rahmen des *HSF-Light* Systems (vergl. Anhang B) betrug für alle Instanzen unter 10 Sekunden (Sun Ultra).

9.2.2 Lernendes Realzeit *A**

Wenn wir einen Algorithmus mehrere Male hintereinander zur Verbesserung seines Lösungsverhaltens aufrufen, sprechen wir davon, daß dieser Algorithmus *lernt*. In unserem Fall wollen wir die heuristische Bewertungsfunktion lernen.

Eine Iteration in der erweiterten lernenden Variante von *RTA** – *LRTA** – unterscheidet sich vom obigen Ansatz nur marginal:

1. Der heuristische Wert $h(u)$ wird auf $\min_{v \in \Gamma(u)}\{h(v) + w(u, v)\}$ gesetzt.
2. Der Algorithmus bestätigt den Übergang zu v^*, der den besten Wert $h(v) + w(u, v)$ aller v aus $\Gamma(u)$ hat.

Korf zeigt, daß die heuristischen Bewertungen in *LRTA** letztendlich gegen die Längen der kürzesten Wege konvergieren. In Abbildung 9.3 geben wir ein einfaches Beispiel für beide Verfahren an. Der Startknoten ist a und die Zielknoten sind d und g. Tabelle 9.3 stellt die unterschiedlichen Abläufe von *RTA** und *LRTA** einander gegenüber.

Die Abbildung 9.4 illustriert, daß *LRTA** im Gegensatz zu *RTA** beliebig schlecht werden kann. Der uniform gewichtete Problemgraph G besitzt einen Pfad v_0 bis v_4. Als heuristische Bewertung legen wir $h(v_0)$ und $h(v_3)$ auf k fest, während $h(v_1)$ und $h(v_2)$ auf Null gesetzt werden. Keiner der Knoten ist ein Zielknoten. Wenn wir die Suche bei v_1 oder bei v_2 starten, so braucht das Verfahren *LRTA** $O(k)$ Expansionen, bis der Rest des Graphens außerhalb

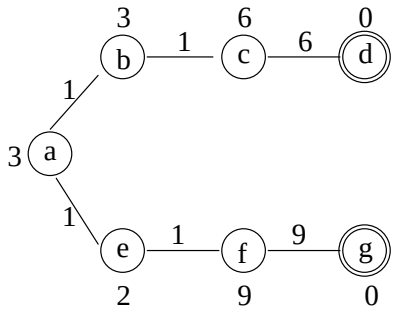


Abbildung 9.3: Ein Beispiel für die Realzeitsuche.

*RTA**		*LRTA**	
Knoten	Aktualisierung	Knoten	Aktualisierung
a	$h(a) \leftarrow h(b) + w(a,b) = 4$	a	$h(a) \leftarrow h(e) + w(a,e) = 3$
e	$h(e) \leftarrow h(f) + w(e,f) = 10$	e	$h(e) \leftarrow h(a) + w(e,a) = 4$
a	$h(a) \leftarrow h(e) + w(a,e) = 11$	a	$h(a) \leftarrow h(b) + w(a,b) = 4$
b	$h(b) \leftarrow h(a) + w(b,a) = 12$	b	$h(b) \leftarrow h(a) + w(b,a) = 5$
c	$h(c) \leftarrow h(b) + w(c,b) = 13$	a	$h(a) \leftarrow h(e) + w(a,e) = 5$
		e	$h(e) \leftarrow h(a) + w(e,a) = 6$
		a	$h(a) \leftarrow h(b) + w(a,b) = 6$
		b	$h(b) \leftarrow h(a) + w(b,a) = 7$
		a	$h(a) \leftarrow h(e) + w(a,e) = 7$
		e	$h(e) \leftarrow h(a) + w(e,a) = 8$
		a	$h(a) \leftarrow h(b) + w(a,b) = 8$
		b	$h(b) \leftarrow h(c) + w(b,c) = 7$
		c	$h(c) \leftarrow h(d) + w(c,d) = 6$

Tabelle 9.3: Beispiellauf von *RTA** und *LRTA**.

von v_0 und v_3 untersucht werden kann. Dieses Verhalten bezeichnen wir in anschaulicher Weise als *Ping–Pong–Effekt*. Der Algorithmus *RTA** findet hingegen den Ausweg sofort, da der zweitbeste Wert $h(v)+w(u,v)$ für die Knoten v_1 bzw. v_2 gleich $k+1$ ist.

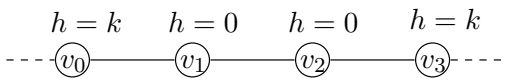


Abbildung 9.4: Ein Beispiel für den *Ping–Pong–Effekt*.

9.2.3 Erste Verbesserungen

Da die Konvergenz recht langsam ist, sind verschiedene Wege zur Verbesserung von *LRTA** eingeschlagen worden. Barto et al. untersuchen einen Ansatz, der auf dynamischer Programmierung basiert (Barto, Bradtke & Singh 1995).

Sie zeigen auf, daß *LRTA** als ein Teil in die Theorie des Reinforcementlernens (Sutton & Barto 1998) eingebettet werden kann. Ishida und Shimbo führen die ϵ bzw. die δ Suchalgorithmen ein, die *LRTA** durch obere und untere Abschneidegrenzen erweitern (Ishida & Shimbo 1996). Die folgenden beiden Ansätze *forward updating* und *backward updating* erlauben, die Heuristik h aller an einer Aktualisierung beteiligten Knoten zu verbessern. Die zu untersuchende Invariante (I) ist, daß h eine optimistische Schätzung ist, d.h. es gilt $h(u) \leq h^*(u)$ für alle Knoten u.

Hilfssatz 8 *(Forward updating) Das Setzen von $h(v)$ auf das Maximum von $h(v)$ und $h(u) - w(u,v)$ für alle $v \in \Gamma(u)$ erhält die Invarianz (I).*

Beweis: Falls $h(v)$ nicht kleiner als $h(u,v) - w(u,v)$ ist, bleibt nichts zu zeigen. Andernfalls wird $h(v)$ auf $h(u) - w(u,v)$ gesetzt. Wir nehmen an, daß $h(v)$ größer als $h^*(v)$ ist, und bestimmen mit p den kürzesten Weg von v zum Ziel. Dann läßt sich p zu p' durch das Einfügen einer Kante $\{u,v\}$ zu einem Zielpfad für u verlängern. Es gilt

$$w(p') = h^*(v) + w(u,v) > h(v) + w(u,v) = h(u),$$

was ein Widerspruch zur Invarianz (I) darstellt. □

Hilfssatz 9 *(Backward updating) Das Setzen von $h(u)$ auf das Maximum von $h(u)$ und $\min_{v\in\Gamma(u)}\{h(v) + w(u,v)\}$ erhält die Invarianz (I).*

Beweis: Wenn $h(u)$ nicht kleiner ist als $\min_{v\in\Gamma(u)}\{h(v) + w(u,v)\}$, ist nichts zu zeigen. Jeder Pfad p von u nach F muß einen von u's Nachfolgerknoten v aus $\Gamma(u)$ passieren, so daß gilt

$$w(p) = w(p') + w(u,v) \geq h^*(v) + w(u,v) \geq h(v) + w(u,v).$$

Dies gilt insbesondere, falls p der kürzeste Pfad mit Länge $h^*(u)$ ist. Damit haben wir die Ungleichung $h^*(u) \geq \min_{v\in\Gamma(u)}\{h(v) + w(u,v)\}$, deren rechte Seite $h(u)$ zugewiesen wird. □

Der Beweis von Hilfssatz 9 belegt die Korrektheit von *LRTA**. Die neue Idee verbirgt sich darin, $h(u)$ selbst mit in die Berechnung einzubeziehen. Zusammenfassend haben wir somit folgenden Satz gezeigt.

Satz 29 *(Updating) In LRTA* mit forward und backward updating bleibt die heuristische Funktion optimistisch.*

Wir wissen bereits, daß h als untere Schranke für den kürzesten Weg zum Ziel aufgefaßt werden kann. Demnach läßt sich auch eine obere Schranke h' für den kürzesten Weg zu einem Zielknoten durch $h'(v) \leftarrow \min\{h'(v), h'(u) - w(u,v)\}$ bzw. durch $h'(u) \leftarrow \min\{h'(u), max_{v\in\Gamma(u)}\ \{h'(v) + w(u,v)\}\}$ festlegen. Analog zu den oberen Betrachtungen finden wir, daß h' innerhalb dieser Operationen invariant als obere Schranke Bestand hat.

9.3 Lernen der Kantenbewertung

Die Idee einer weiteren Verbesserung von *LRTA** basiert auf Schildern, die an jeder Kante innerhalb eines gerichteten Graphens angebracht sind. Ungerichtete Graphen sind Spezialfälle von gerichteten Graphen, in denen zu jeder Kante (u, v) auch die Rückwärtkante (v, u) existiert.

Jedes Schild ist als untere Schranke für den kürzesten Weg anzusehen, der diese Kante passiert. Da die kürzesten Wege zum Ziel keine Zyklen enthalten, braucht diese Schranke nur für alle zyklenfreien Wege zum Ziel zu gelten.

Wir legen den *Expansionspfad* p durch den ausgeführten bestätigten Weg vom Startknoten zu einem erreichten Zielknoten fest.

9.3.1 *SRTA**

Das folgende *SRTA** Verfahren ist eine direkte Erweiterung von *RTA**. Es sei mit einem Knoten u aufgerufen, der kein Zielknoten ist.

1. Für alle Nachfolger v aus $\Gamma(u)$ setze Wert $f(u, v)$ auf $h(v) + w(u, v)$.
2. Setze $h(u)$ auf den zweitbesten Wert von $f(u, v)$. Existiert kein weiterer Knoten, so wird $h(u)$ auf unendlich gesetzt.
3. Bestätige den Übergang zu v^* mit $h(u) = \min_{v \in \Gamma(u)}\{h(v) + w(u, v)\}$.

Satz 30 *(Optimistische Kantenbewertung) Sei $G = (V, E)$ ein uniform gewichteter (die Kantenbewertungen sind alle eins) endlicher Graph. Der Wert $f(u, v)$ überschätzt in SRTA* keinen der Wege von u über v zum Ziel, die die Kante von v nach u nicht enthalten.*

Beweis: Die Beweisführung beruht auf Induktion, d.h. wir nutzen die Aussage des Satzes als Invariante. Sei u der expandierte Knoten. Falls ein Nachbar v von u noch nicht besucht wurde, ist der initiale h-Wert optimistisch. Somit erfüllt $f(u, v)$ die Behauptung.

Sei demnach v schon einmal expandiert und $f(u, v)$ auf $h(v) + w(u, v)$ gesetzt. Der Wert $h(v)$ entspricht dem zweitbesten Wert $h(w) + w(v, w)$ aller w aus der Nachfolgermenge $\Gamma(v)$. Falls der beste Wert durch $h(u) + w(v, u)$ gegeben ist, so ist $h(v)$ durch das Minimum der Werte $f(v, w)$ mit w aus $\Gamma(v) - \{u\}$ festgelegt. Nach Induktionsannahme unterschätzt $f(v, w)$ alle Wege zum Ziel, die die Kante (w, v) nicht enthalten. Demnach ist durch das Minimum der Werte $f(v, w)$ auch $f(u, v)$ eine obere Schranke für alle Zielpfade, die (v, u) nicht enthalten.

Somit ist der Nachfolger w, auf dem v verlassen wurde, nicht u. Sei $C = (u = p_0, v = p_1, w = p_2, \ldots, p_n = u)$ der Zyklus, auf dem u erneut erreicht wird. Wir betrachten die Differenz d zwischen dem zweitbesten und dem besten f-Wert an v. Wir werden zeigen, daß diese Differenz allein durch das Erreichen des Knotens u aufgezehrt wird, d.h. wir bei der Expansion von u den Wert $f(u, v)$ ohne Schwierigkeiten auf den zweitbesten f-Wert an v plus $w(u, v)$ festlegen können, da der beste f-Wert nicht mehr aktuell sein kann.

Wir definieren $d(p_i)$ für i größer drei rekursiv durch $d(p_i) = d(p_{i-1}) + h(p_{i-1}) - h(p_i) - w(p_{i-1}, p_i)$. Dabei legen wir $d(p_2)$ als die Differenz d fest. Der Wert $d(p_i)$ gibt an, wieviel der ursprünglichen Differenz am Knoten p_i noch besteht, d.h. wie sich durch das Begehen des Pfades von w nach u, der beste

f-Wert an v verschätzt. Bei der Berechnung von $h(p_n)$ heben sich die Glieder $h(p_{i-1}) - h(p_i)$ teleskopartig auf. Wir erhalten

$$d(p_n) = d(p_2) + h(p_2) - h(p_n) - w(p_2, \ldots, p_n).$$

Der beste f-Wert an v ist $f(v, w)$ bzw. $h(p_2) + w(p_1, p_2)$. Der zweitbeste f-Wert an v ist durch $h(p_n) + w(p_1, p_n)$ beschränkt. Also ist $d(p_n)$ höchstens gleich $w(p_1, p_2) - w(p_1, \ldots, p_n)$. Damit ist $d(p_n)$ nur dann echt größer null, wenn $w(p_1, p_2)$ größer als $w(p_1, \ldots, p_n)$ ist. Dies ist in einem uniform gewichteten Graphen jedoch nicht möglich. □

Folgerung 7 *Sei h' für alle Knoten u durch $h'(u) \leftarrow \min_{v \in \Gamma(u)}\{f(u, v)\}$ definiert. Dann ist $h'(u)$ optimistisch.*

Folgerung 7 gibt Anlaß zu zwei neuen Lernalgorithmen *CRTA** (*RTA** mit Aufräumstrategie) und *SLRTA** (*LRTA** mit Schildern).

9.3.2 *CRTA**

Wir beschreiben den *CRTA**-Ansatz zuerst. Das Verfahren durchläuft nach dem Fund des Zielknotens t den Expansionspfad $p = \langle s = v_0, \ldots, v_k = t \rangle$ rückwärts, um Überschätzungen von h durch h' zu ersetzen. Eine weitere Verbesserung kann dadurch erreicht werden, daß die f-Werte in der Rückwärtsbewegung mit einbezogen werden, d.h. wir setzen erst $f(v_i, v_{i+1})$ auf den neuen Wert $h(v_{i+1})$ plus $w(v_i, v_{i+1})$ und berechnen dann den neuen Wert $h(v_i)$ als das Minimum der neuen f-Werte. Wiederum ist gewährleistet, daß h die tatsächliche Lösungslänge nicht überschätzt. Um die explizite Speicherung der f-Werte zu vermeiden, läßt sich auch der Generierungspfad von *RTA** mit der Aktualisierung $h(v_i) = \min\{h(v_i), h(v_{i+1}) + w(v_i, v_{i+1})\}$ rückwärts traversieren.

Der Algorithmus *CRTA** mag dahingehend kritisiert werden, daß er den gesamten Expansionspfad zurückverfolgt. Da der Pfad nun zweimal durchlaufen wird, macht sich diese Idee weniger an der Gesamtkomplexität des Verfahrens bemerkbar als an der Realzeitanforderung von einer konstanten Entscheidungszeit an jedem Knoten. Als Ausweg könnten wir uns mit einer Änderung der Aufgabenstellung anfreunden, in der der Roboter die Kunde vom Erreichen des Zieles der Person am Startpunkt berichten muß.

9.3.3 *SLRTA**

Da dieser Kunstgriff unbefriedigend ist, wenden wir uns dem zweiten Lernalgorithmus *SLRTA** zu, in dem wir den Aktualisierungstribut erst bei dem abermaligen Besuch eines Knotens zollen. Das Verfahren *SLRTA** benützt zusätzlich eine Iterationsnummer für jeden Knoten, die angibt, in welcher Runde der Knoten expandiert wurde. Offensichtlich dient diese Zahl der Unterscheidung der Knoten in der aktuellen Iteration von den Knoten in der vorangegangenen.

Die Idee ist einfach. Anstatt die Schätzwerte direkt nach dem Erreichen des Zielknotens auf einen optimistischen Wert zu drücken, verschieben wir die Berechnung auf den Zeitpunkt, an dem der Knoten neu besucht wird. Der Aufforderung, den h-Wert auf das Minimum der f-Werte zu setzen, kann in der abermaligen Generierung der jeweiligen Knoten nachgekommen werden. Daraufhin werden die f Werte für die aktuelle Iteration gesetzt. *SLRTA** ist

korrekt, da die von einem Knoten ausgehende f Bewertung an jedem Knoten nur bei der Expansion eben dieses Knotens verändert wird.

Die Qualität von *SLRTA** ist etwas schlechter als die bei *CRTA**, da die Einbeziehung der veränderten f-Werte in der Rückwärtsbewegung nicht möglich ist. In beiden Algorithmen konvergieren die Schätzwerte gegen die Länge der kürzesten Wege, da das Argument von Korf zum Beweis der Konvergenz von *LRTA** auch hier greift: Vom Ziel ausgehend werden für all diejenigen Knoten, deren Nachfolger schon die optimale Weglänge schätzen, durch die Bildung des Minimums die eigenen Schätzwerte exakt.

9.3.4 Experimentelle Resultate

Wir untersuchen zuerst den Algorithmus *CRTA** in der verbesserten Version, d.h. wir nehmen für den Expansionspfad $p = (v_0, \ldots, v_k)$ und aktualisieren die Werte $h(v_i)$ durch $\min\{h(v_i), h(v_{i+1}) + w(v_i, v_{i+1})\}$.

Der Problemraum ist gegeben durch ein festes Labyrinth auf einem unterliegenden Gitter der Größe 30 mal 30. Eine Mauer wird mit einer Wahrscheinlichkeit von 35 Prozent an einer Stelle (i, j), $1 \leq i, j \leq 30$ errichtet. Wenn die heuristischen Werte einen Schwellwert von Hundert überschreiten, so nehmen wir an, daß keine Lösung existiert und brechen die Suche ab.

Abbildung 9.5 illustriert die Verbesserung von *CRTA** im Vergleich mit *LRTA**. Da die ersten drei Werte jeweils sehr groß waren (214,166 und 2094 für *CRTA**und 364,260 und 4848 für *LRTA**), wurden sie für eine detaillierte Darstellung des Lernvorganges nicht mit aufgezeichnet.

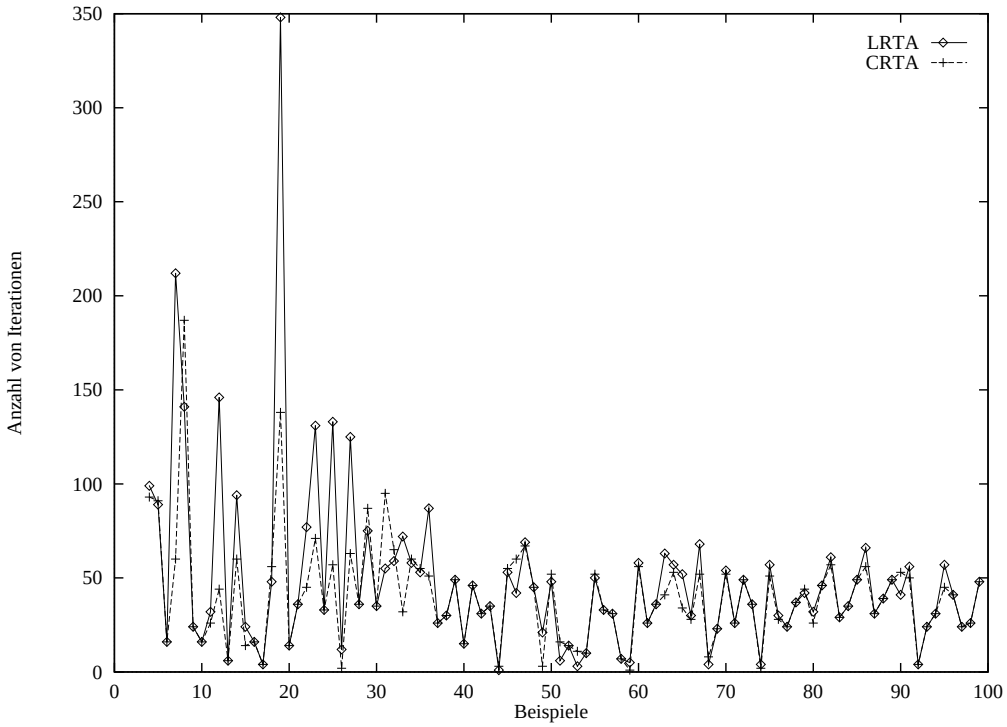


Abbildung 9.5: Vergleich von *LRTA** und *CRTA** in einem Labyrinth.

Offensichtlich ist am Anfang des Lernvorganges die Anzahl der expandier-

ten Knoten von *CRTA** wesentlich geringer als die von *LRTA**(3339 gegenüber 6801 Knoten für die ersten zwanzig Beispiele), danach jedoch schwingen sich beide Werte recht schnell auf eine sehr gute Approximation der kürzesten Wege ein, in der sie sich nicht allzu sehr unterscheiden.

Als nächstes Beispiel untersuchen wir *SLRTA** bei der Lösung des Acht-Puzzles. Die optimale Lösungslänge liegt im Schnitt bei 21.97 Zügen (Reinefeld 1993). Um einen von den unterschiedlichen Lösungslängen der Einzelinstanzen unabhängigen Vergleichswert zu erhalten, fassen wir immer tausend Iterationen zur Bildung eines Mittelwertes zusammen. Abbildung 9.6 stellt die Ergebnisse des *RTA**-, des *LRTA**- und des *SLRTA**-Verfahrens der gemittelten optimalen Lösungslänge gegenüber.

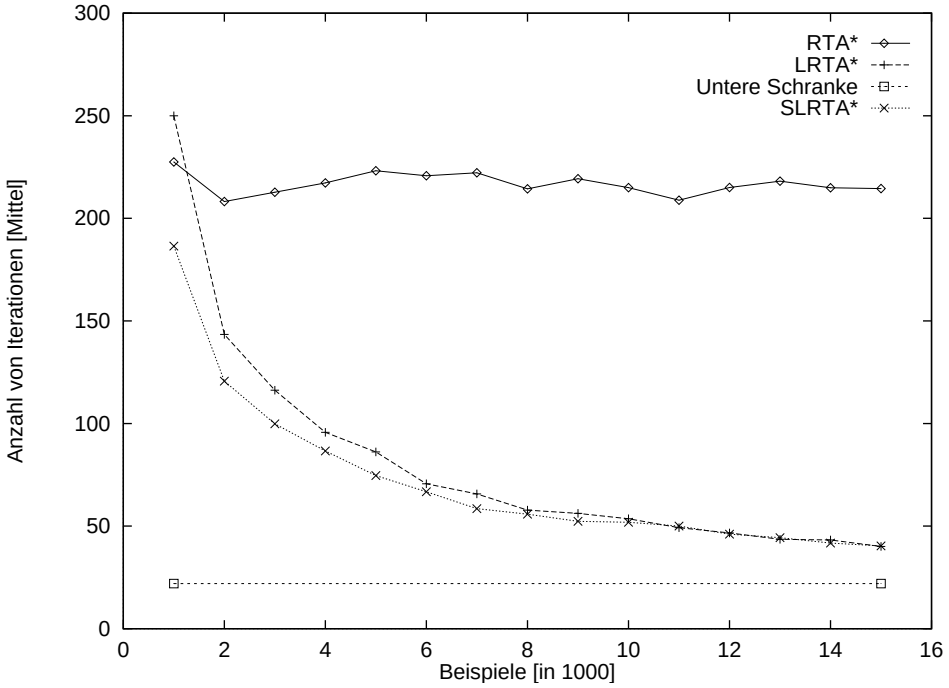


Abbildung 9.6: Vergleich von *RTA**, *LRTA** und *SLRTA** bei der Lösung verschiedener Acht-Puzzle Instanzen.

Es gibt einen direkten Zusammenhang zwischen der Qualität der heuristischen Bewertung und der Anzahl der expandierten Knoten. Wenn der Schätzwert gut ist, so sind die erreichten Lösungslängen kurz.

In beiden Fällen ist auffällig, daß *LRTA**, gemessen in der Anzahl der Beispiele, nach einiger Zeit mit den vorgeschlagenen Verbesserungen gleichwertig ist. Wir erklären dies durch den hohen Explorationsaufwand in den ersten Iterationen von *LRTA**.

9.3.5 *ELRTA** und *HLRTA**

Unabhängig von den obigen Studien haben die beiden Studenten Paul Morris und Paul Eaton Thorpe zwei Verfahren *ELRTA** und *HLRTA** zur Verbesserung von *LRTA** entwickelt, die sich trotz ihrer unterschiedlichen Beschreibung beide als äquivalent herausstellen (Thorpe 1994). Wir stellen demnach nur das

*HLRTA** Verfahren vor. Sei u der zu expandierende Knoten, $b(v)$ der beste und $s(v)$ der zweitbeste f-Wert an v. Initial ist $b(v)$ gleich $s(v)$ gleich $h(v)$.

1. Für alle Nachfolger v aus $\Gamma(u)$ setze $f(u,v)$ auf $w(u,v) + s(v)$, falls u der bestbewertete Knoten von v ist, sonst auf $w(u,v) + b(v)$.
2. Setze $b(u)$ bzw. $s(u)$ auf den besten bzw. zweitbesten Wert von $f(u,v)$. Existiert kein weiterer Knoten, so wird $s(u)$ auf unendlich gesetzt.
3. Bestätige den Übergang zu v^* mit $b(u) = f(u,v^*)$.

Bei der erneuten Generierung eines Knotens wird demnach nur dann der zweitbeste Wert gewählt, wenn dieser Knoten auch auf der Kante verlassen wurde. Wir wollen an einem kleinen Beispiel belegen, warum unser Verfahren vorteilhafter als *HLRTA** ist. Betrachten wir den Zyklus aus Abbildung 9.7. Startpunkt ist der Knoten b, das Ziel liegt bei f. Wir wir sehen, besucht *SRTA** die Knoten in der Reichenfolge b, c, e, d, b, a und f. *HLRTA** durchläuft den Zyklus hingegen zweimal: b, c, e, d, b, c, e, d, b, a und f.

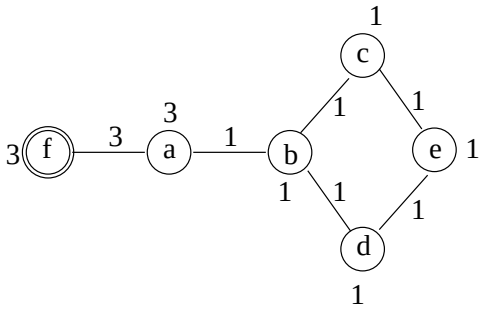


Abbildung 9.7: Zyklen in *SRTA** und *HLRTA**.

9.4 Ausblick

Eine Menge von neuen Aspekten für die Verbesserung der Effizienz beim Lernen der heuristischen Bewertungsfunktion sind in diesem Kapitel analysiert worden. Alle Algorithmen operieren lokal in der Umgebung von unmittelbar benachbarten Knoten. Die Idee von Schildern an den Kanten hat eine bessere Einsicht in die Problemstruktur der grundlegenden Realzeitsuchalgorithmen gebracht, indem sie die Beweisbarkeit der unteren Schranke von auf *RTA** aufbauenden Verfahren liefert. Die beiden Direktiven der weiterhin optimistisch zu gestaltenden Bewertungsfunktion und des möglichst schnellen Erreichens des Zieles werden voneinander trennt.

In *LRTA** und in den hier präsentierten Alternativen wird erwartet, daß zu jedem Knoten des Suchraumes eine heuristische Bewertung abgelegt werden kann. Speicherplatzfreundlichere Lösungen für dieses Problem sind nicht erforscht. Eine weitere Aufgabe ist es, inkrementelle mit nicht inkrementellen Lernverfahren zu vergleichen. Zum Beispiel mag ein Roboter in einer Lernphase erst ein wenig seine Umgebung erkunden, bevor er sich auf die Suche nach einer Lösung begibt. In dem Sinne fragen wir nach einem geeigneten Mittelweg zwischen der Kurzzeit- und Langzeitperformanz des Verfahrens.

Kapitel 10

Berechnung des Verzweigungsgrads

Viele Probleme, wie z.B. Schiebepuzzle, erzeugen Suchbäume, in denen unterschiedliche Knoten eine unterschiedliche Anzahl von Kindern haben. Wir beschreiben die Berechnung des asymptotischen Verzweigungsgrads und der exakten Anzahl von Knoten zu einer gegebenen Tiefe. Diese Information erweist sich als wichtig, um die Komplexität verschiedener Suchalgorithmen zu beschreiben. Zusätzlich zu den Schiebepuzzles wenden wir unseren Ansatz auf den Zauberwürfel an. Obwohl die Techniken sehr einleuchtend erscheinen, ist es sehr leicht, sich in den Fehler der vielen inkorrekten Methoden zu verfangen. Wir schließen das Kapitel mit der Berechnung des asymptotischen Verzweigungsgrads in Suchräumen, in denen ein endlicher Automat zur Suchraumbeschneidung eingesetzt wird.

10.1 Verzweigungsgrade

In vielen, auf die (iterierte) Tiefensuche aufbauenden, heuristischen Suchverfahren durchforsten wir den Zustandssuchbaum, während die unterliegende Problemstruktur zumeist jedoch Graphen sind und Zyklen enthalten. In Kapitel 3 haben wir gesehen, daß dadurch der Suchbaum exponentiell größer als der unterliegende Graph werden kann und selbst für einen endlichen Graphen mitunter unendlich groß ist.

Die Zeitkomplexität eines Baumsuchverfahrens hängt hauptsächlich vom *Verzweigungsgrad b* und der *Lösungstiefe d* ab. Die Lösungstiefe ist die Länge des kürzesten Lösungspfades und somit abhängig von der Probleminstanz, während der Verzweigungsgrad typischerweise auf einem konstanten Wert für den gesamten Suchraum konvergiert. Damit ist der Verzweigungsgrad für ein gegebenes Problem ein essenzieller Parameter für die Bestimmung der Komplexität eines Suchverfahrens und kann bei der Auswahl von alternativen Repräsentationen des gleichen Problems genutzt werden.

Der Verzweigungsgrad eines Knotens ist die Anzahl der Kinder, die dieser Knoten hat. In einem Baum, in dem jeder Knoten den gleichen Verzweigungsgrad besitzt, ist dieses somit auch der Verzweigungsgrad des Baumes. Die

Schwierigkeit tritt auf, wenn verschiedene Knoten in derselben Ebene innerhalb des Baumes einen unterschiedlichen Verzweigungsgrad haben. In diesem Fall definieren wir mit dem Verzweigungsgrad für eine gegebene Tiefe das Verhältnis der Knoten in dieser Ebene gegenüber der Anzahl der Knoten in der darüberliegenden Ebene. Falls wir die Tiefe immer weiter anwachsen lassen, konvergiert der Verzweigungsgrad in vielen Fällen gegen einen Grenzwert, dem sogenannten *asymptotischen Verzweigungsgrad*, der das beste Maß für die Größe des Baumes darstellt.

In diesem Kapitel werden wir auf verlockende Irrwege, aber auch auf die korrekte Berechnung des asymptotischen Verzweigungsgrads anhand einfacher Beispiele eingehen. Wir werden das Problem als eine Menge simultan zu lösender Rekursionsgleichungen formulieren und aufzeigen, wie wir diese Aufgabe in ein Nullstellenproblem umformen können. Alternativ wird ein numerisches Verfahren besprochen, das den Verzweigungsgrad eines Problems schnell und präzise bestimmt. Die asymptotischen Verzweigungsgrade für einige Zustandsräume werden angegeben.

Insbesondere gehen wir auf Verbindungen der Berechnung des asymptotischen Verzweigungsgrads zu der automatischen Codierung einer Abschneideregel durch einen endlichen Automaten ein, wie sie in Kapitel 6 vorgestellt wurde. Ein Beschneiden des Suchraumes führt immer zu einer Verkleinerung des Verzweigungsgrads. Wir werden zeigen, daß die Graphstruktur des endlichen Automaten sich direkt in der Menge der Rekursionsgleichungen widerspiegelt, und der veränderte Verzweigungsgrad sich somit mit Hilfe der gleichen Methode berechnen läßt.

10.2 Irrwege

Unser erstes Beispiel ist das *Fünf*-Puzzle, die 2 mal 3 Version des in Kapitel 2 beschriebenen und von Samuel Loyd erfundenen Schiebepuzzles (siehe Abbildung 10.1). Es gibt fünf quadratische Spielsteine und ein Leerfeld. Jeder an ein Leerfeld angrenzender Spielstein kann horizontal oder vertikal in die Position des Leerfeldes gezogen werden. Die Aufgabe ist es, die Spielsteine so umzuordnen, daß der Zielzustand, der in dem rechten Teil der Abbildung dargestellt ist, erreicht wird.

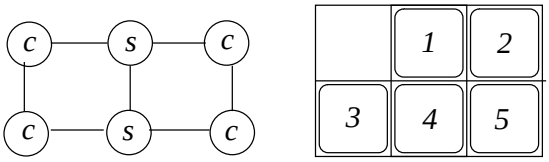


Abbildung 10.1: Seiten und Ecken im *Fünf*-Puzzle.

Der Verzweigungsgrad eines Knotens in diesem Problem hängt allein von der Position des Leerfeldes ab. Wie Abbildung 10.1 vergegenwärtigt, gibt es zwei verschiedene Zustände in dem Puzzle, *Seiten* bzw. s Zustände und *Ecken* bzw. c Zustände (engl. corner). Wir werden dementsprechend Knoten innerhalb des Suchbaumes, deren Leerfeld sich in einem s bzw. c Zustand befindet, s bzw. c Knoten nennen. Unter der Annahme, daß der Vorgänger auch als Nachfolger eines Knotens generiert wird, erhalten wir für einen s Knoten einen Verzweigungsgrad von drei und für einen c Knoten einen Verzweigungsgrad von

zwei. Demnach wird sich der asymptotische Verzweigungsgrad zwischen zwei und drei einpendeln.

Der exakte Wert des Verzweigungsgrades hängt von dem Anteil f_s an s Knoten innerhalb einer Ebene ab. Analog ist f_c als der Anteil der c Knoten einer Ebene und somit durch $1 - f_s$ gegeben. Für eine gegebene Tiefe wird f_s davon abhängen, ob der initiale Zustand ein s oder ein c Knoten ist. Da wir jedoch nur an dem Grenzwert dieses Verhältnisses interessiert sind, können wir vom Anfangszustand abstrahieren.

10.2.1 Gleichverteilung

Die einfachste Annahme ist es, daß f_s gleich 2/6 bzw. 1/3 ist, da zwei von sechs möglichen Positionen Seiten sind. Somit ergibt sich ein asymptotischer Verzweigungsgrad von $3 \cdot 1/3 + 2 \cdot 2/3 = 2.333$. Leider ist die Annahme, daß die Position des Leerfeldes gleichmäßig auf den sechs Zuständen verteilt ist, falsch. Intuitiv ist dies dadurch zu begründen, daß die Seiten in der Mitte liegen und somit häufiger von dem Leerfeld besucht werden und folglich im Suchbaum überrepräsentativ vorhanden sind.

10.2.2 Random Walk Modell

Eine bessere Annahme ist die folgende: Betrachten wir den Graphen aus sechs Knoten auf der linken Seite in Abbildung 10.1. In einem Zufallsweg (engl. random walk) des Leerfeldes innerhalb des Graphen konvergiert der Anteil der Zeit, die ein Leerfeld an einem Knoten verweilt, letztendlich gegen einen Grenzwert.

Wenn wir die sechs Positionen in zwei Mengen mit gerader oder ungerader Zustandsnummer unterteilen, muß ein Zug das Leerfeld immer von der einen in die andere Menge verschieben.

Die Grenzverteilung läßt sich wie folgt ermitteln. Da s Knoten den Grad drei und c Knoten den Grad zwei haben, muß der Anteil der Zeit in einem s Knoten gegenüber der Zeit in einem c Knoten dem Verhältnis drei zu zwei entsprechen (Motwani & Raghavan 1995). Damit werden s Knoten in 2/3 der Zeit und c Knoten in 1/3 der Zeit besucht. Demnach erhalten wir einen Verzweigungsgrad von $3 \cdot 2/3 + 2 \cdot 1/3 = 2.6666$ im Unterschied zum oben ermittelten Wert von 2.3333.

Leider ist diese Berechnung auch falsch. Auch wenn der Zufallslauf die Wahrscheinlichkeit des Aufenthaltes in einem Zustand für einen langen Weg innerhalb eines Baumes mit nicht uniformem Verzweigungsgrad gut voraussagt, entspricht diese Zahl doch nicht den relativen Anteilen der Zustände in der erreichten Ebene im Baum. Betrachte als Beispiel das Baumfragment aus Abbildung 10.2. Wenn wir zufällig eines der drei Blätter wählen, besitzt jedes Blatt die gleiche Wahrscheinlichkeit, von uns betrachtet zu werden. In einem Zufallslauf wird der linke Blattknoten hingegen mit einer Wahrscheinlichkeit von 50 Prozent besucht, während die anderen beiden Knoten nur zu 25 Prozent erreicht werden.

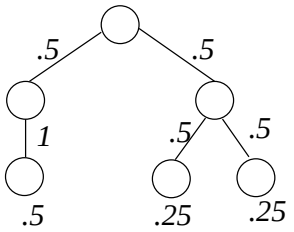


Abbildung 10.2: Baum mit nicht uniformem Verzweigungsgrad.

10.3 Die richtige Antwort

Wir kommen nun zu der korrekten Art und Weise, den Grenzanteil f_s zu berechnen. Ein c Knoten in der einen Ebene erzeugt einen s Knoten und einen c Knoten in der nächsten Schicht. Genauso erzeugt ein s Knoten einen weiteren s Knoten und zwei c Knoten in der nächsten Ebene.

Damit entspricht die Anzahl der c Knoten innerhalb einer Ebene der doppelten Anzahl von s Knoten plus der Anzahl der c Knoten der vorhergehenden Schicht, und die Anzahl der s Knoten ergibt sich aus der Summe aus der Anzahl der c und der s Knoten der vorhergehenden Ebene. Angenommen wir haben nf_s viele s Knoten und nf_c viele c Knoten in einer gegebenen Ebene, dann erhalten wir genau $2nf_s+nf_c$ viele c und nf_s+nf_c viele s Knoten in der darauf folgenden Schicht. Nun nehmen wir an, daß die Anteile f_s und f_c gegen einen konstanten Grenzwert konvergieren, der auch für die darauf folgende Ebene gelten muß. Damit ist

$$f_s = \frac{nf_s + nf_c}{nf_s + nf_c + 2nf_s + nf_c} = \frac{f_s + 1 - f_s}{f_s + 1 - f_s + 2f_s + 1 - f_s} = \frac{1}{f_s + 2}.$$

Eine Multiplikation mit f_s+2 ergibt eine quadratische Gleichung $f_s^2+2f_s-1=0$ mit der positiven Nullstelle $\sqrt{2}-1 \approx 0.4142$. Somit erhalten wir einen asymptotischen Verzweigungsgrad von $3f_s+2(1-f_s) = 3(\sqrt{2}-1)+2(2-\sqrt{2}) = \sqrt{2}+1 \approx 2.4142$.

Die Annahme, die wir gemacht haben, ist die, daß der Vorgänger eines Knotens abermals generiert wird. In der Praxis ist dieses jedoch unsinnig, da der optimale Lösungsweg keine Zyklen enthält. Demnach kann der Verzweigungsgrad durch Vorgängerelimination um rund einen Punkt gesenkt werden. Dabei ist es wichtig zu notieren, daß der Verzweigungsgrad nicht genau um eins reduziert wird, da eine Beschneidung des Suchbaumes sich auch auf die Grenzanteile der s und c Knoten auswirkt. So ist der Verzweigungsgrad des *Fünf*-Puzzles mit Vorgängerelimination, wie wir in Abschnitt 10.5 sehen werden, gleich 1.3532.

10.4 Der Zauberwürfel

Ein anderes Beispiel ist der Zauberwürfel, den wir bereits in Kapitel 2 vorgestellt haben. Da es sechs verschiedene Flächen gibt, und wir eine Fläche entweder um 90, 180 oder 270 Grad drehen können, ist der Verzweigungsgrad 18 für alle Knoten des Suchraumes. Es scheint jedoch angebracht, keine Seite zweimal hintereinander zu drehen, da dasselbe Resultat durch einmaliges

Drehen erreicht werden kann. Somit verringern wir den Verzweigungsgrad auf 15 nach dem ersten Zug. Die nächste Beobachtung ist es, daß das Drehen gegenüberliegender Seiten unabhängig voneinander ist. Deshalb können wir die Reihenfolge der Drehung in eine Ordnung zwingen. Dazu bezeichnen wir für jedes Paar gegenüberliegender Seiten eine Seite als *primär* (z.B. *links*, *oben* und *vorn*) und die andere Seite (z.B. *rechts*, *unten* und *hinten*) als *sekundär*.

Nachdem eine primäre Seite gedreht worden ist, gibt es drei verschiedene Möglichkeiten, die übrigen Seiten zu drehen. Dies liefert uns einen Verzweigungsgrad von 15. Nachdem eine sekundäre Seite gedreht worden ist, verbleiben uns drei mögliche Drehungen von nur vier Seiten. Somit erhalten wir in diesem Fall einen Verzweigungsgrad von 12. Demnach liegt der asymptotische Verzweigungsgrad irgendwo zwischen 12 und 15.

Um den Wert exakt zu berechnen, benötigen wir die Grenzanteile der primären (f, engl. first) und sekundären (s) Knoten innerhalb einer Ebene, wobei ein primärer Knoten durch den Zug einer primären Seite definiert ist. Jeder f Knoten erzeugt sechs f Knoten und neun s Knoten, da keine Seite zweimal hintereinander gedreht wird. Jeder s Knoten erzeugt sowohl sechs f als auch sechs s Knoten, da sowohl die gleiche als auch die gegenüberliegende Seite unterdrückt werden. Seien f_f und $f_s = 1 - f_f$ die Grenzanteile an f bzw. s Knoten innerhalb einer Ebene. Da wir eine Konvergenz dieser Anteile voraussetzen, erhalten wir für den Anteil an f Knoten den folgenden Selbstbezug

$$f_f = \frac{6f_f + 6f_s}{6f_f + 6f_s + 9f_f + 6f_s} = \frac{6f_f + 6(1 - f_f)}{15f_f + 12(1 - f_f)} = \frac{6}{3f_f + 12} = \frac{2}{f_f + 4}.$$

Die Multiplikation von $f_f + 4$ liefert uns die quadratische Gleichung $f_f^2 + 4f_f - 2 = 0$, die eine positive Nullstelle bei $f_f = \sqrt{6} - 2 \approx .44949$ hat. Folglich erhalten wir einen asymptotischen Verzweigungsgrad von $15f_f + 12(1 - f_f) \approx 13.34847$.

10.5 Das Rekursionsgleichungssystem

Die obigen Beispiele erfordern nur die Lösung einer einfachen quadratischen Gleichung. Allgemein wird ein System von Rekursionsgleichungen generiert. Als ein mehr repräsentatives Beispiel wählen wir das *Fünf*-Puzzle mit Vorgängerelimination. Um das Inverse der letzten Operation zu unterdrücken, müssen wir die letzten zwei Positionen des Leerfeldes verfolgen. Sei cs ein Zustand oder Knoten, in dem die aktuelle Leerfeldposition eine Seite ist und unmittelbar davor eine anliegende Ecke war. Wir legen ss, sc und cc analog fest.

Abbildung 10.3 zeigt die verschiedenen Zustandstypen auf, während die Pfeile auf den Typ der erzeugten Kinder im Suchbaum weisen. Zum Beispiel bedeutet der Doppelpfeil von ss nach sc, daß jeder ss Knoten zwei sc Knoten generiert.

Sei $n(t, d)$ die Anzahl der Knoten vom Typ t in der Tiefe d des Suchbaumes. Dann können wir die folgenden Rekursionsgleichungen direkt von dem Graphen in Abbildung 10.3 ablesen:

$$n(cc, d + 1) \quad = \quad n(sc, d)$$

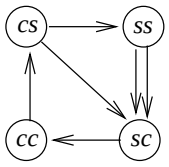


Abbildung 10.3: Der Graph des *Fünf*-Puzzles mit Vorgängerelimination.

$$\begin{aligned} n(cs, d+1) &= n(cc, d) \\ n(ss, d+1) &= n(cs, d) \\ n(sc, d+1) &= 2n(ss, d) + n(cs, d). \end{aligned}$$

Zum Beispiel ergibt sich die letzte Gleichung aus dem Vorhandensein zweier Pfeile von ss nach sc und eines Pfeils von cs nach sc.

Es sei angemerkt, daß wir die Anfangsbedingung nicht explizit aufgeführt haben. Der erste Zug wird entweder einen ss Knoten und zwei sc Knoten oder einen cs Knoten und einen cc Knoten erzeugen, je nachdem, ob das Leerfeld anfänglich an einer Seite oder in einer Ecke steht. Die nächste Frage ist, wie ein solches System von Gleichungen gelöst werden kann.

10.5.1 Numerische Lösung

Der einfachste Weg ist es, die Rekursionsgleichungen nacheinander auszuwerten, bis die relativen Anteile konvergieren. Für eine gegebene Suchtiefe sei f_{cc}, f_{cs}, f_{ss} und f_{sc} die Anzahl der Knoten des gegebenen Typs dividiert durch die Gesamtzahl der Knoten innerhalb dieser Schicht. Den Verzweigungsgrad für die erreichte Tiefe erhält man durch das Verhältnis der Gesamtanzahl der Knoten innerhalb zweier aufeinanderfolgender Ebenen. Nach ungefähr 100 Iterationen nähern sich die Anteile mit $f_{cc} = 0.274854$, $f_{cs} = 0.203113$, $f_{ss} = 0.150097$ und $f_{sc} = 0.371936$ ihrem Grenzwert. Da der Verzweigungsgrad der ss und der cs Knoten gleich zwei und aller anderen Zustände gleich eins ist, erhalten wir den asymptotischen Verzweigungsgrad von $1 \cdot f_{cc} + 2 \cdot f_{cs} + 2 \cdot f_{ss} + 1 \cdot f_{sc} = 0.274854 + 0.406226 + 0.300194 + 0.371936 = 1.35321$. Wenn q die Anzahl der verschiedenen Zustandstypen ist und d die erreichte Tiefe der Iteration, so liegt die Laufzeit dieses Verfahrens in $O(dq)$.

10.5.2 Analytische Lösung

Um die analytisch exakte Lösung für den asymptotischen Verzweigungsgrad zu erhalten, nehmen wir an, daß die Anteile f_{cc}, f_{cs}, f_{ss} und f_{sc} konvergieren. Damit erhalten wir eine Menge von Gleichungen, in denen die Anteile der Knoten bestimmten Typs auf beiden Seiten erscheint. Sei b der zu berechnende asymptotische Verzweigungsgrad. Wenn wir zum Beispiel f_{cc} als die normalisierte Anzahl von cc nodes in Tiefe d betrachten, dann verändert sich diese Zahl auf bf_{cs} für die Tiefe $d + 1$.

Dies ermöglicht uns, die Rekursionsgleichungen direkt in die folgende Menge von Gleichungen zu überführen, wobei die letzte Gleichung heraushebt, daß die Summe der normalisierten Anteile eins ergeben muß:

$$bf_{cc} = f_{sc}$$

$$\begin{aligned} bf_{cs} &= f_{cc} \\ bf_{ss} &= f_{cs} \\ bf_{sc} &= 2f_{ss} + f_{cs} \\ 1 &= f_{cc} + f_{cs} + f_{ss} + f_{sc}. \end{aligned}$$

Wir haben fünf Gleichungen mit fünf Unbekannten. Bei dem Versuch diese Gleichungen durch wiederholte Einsetzung zu lösen, erreichen wir einen immer weiter ansteigenden Exponenten für b. Letztendlich können wir dieses System auf das Nullstellenproblem $b^4 - b - 2 = 0$, also eines Polynoms vierten Grades in b, reduzieren. Es ist leicht zu überprüfen, daß $b \approx 1.35321$ eine Lösung dieser Gleichung ist. Nullstellenprobleme vierten Grades können immer explizit gelöst werden, doch für Gleichungen fünften und höheren Grades existiert keine allgemeine Lösungsformel. Für das *Fünfzehn*-Puzzle erhalten wir jedoch fünf verschiedene Zustandstypen, die somit zu einer Gleichung fünften Grades führen.

10.5.3 Allgemeine Formulierung

In diesem Abschnitt wollen wir von den gegebenen Beispielen abstrahieren, um die Struktur des Rekursionsgleichungssystems zu analysieren. Wir beginnen mit einer Repräsentation des zugrunde liegenden Graphen G als Adjazenzmatrix P. Für Abbildung 10.3 erhalten wir als Zeilen P_j von P für j aus $V = \{cc, cs, ss, sc\}$ die folgenden Vektoren $P_{cc} = (0, 1, 0, 0)$, $P_{cs} = (0, 0, 1, 1)$, $P_{ss} = (0, 0, 0, 2)$ und $P_{sc} = (1, 0, 0, 0)$. Die Spalten von P werden mit P^j bezeichnet. Wir repräsentieren die Anteile der Zustandstypen innerhalb einer Ebene als Vektor F. In unserem Beispiel ist diese Verteilung F gegeben durch $F = (f_{cc}, f_{cs}, f_{ss}, f_{sc})$. Ohne Beschränkung der Allgemeinheit numerieren wir die Knoten V mit den ersten Zahlen $0, \ldots, |V|-1$ und nehmen an, daß die Verteilung letztendlich gegen eine Grenzverteilung konvergiert, was zu der folgenden Fixpunktgleichung führt: $bF = FP$ mit b als dem asymptotischen Verzweigungsgrad. Zusätzlich haben wir die Bedingung $\sum_{i \in V} f_i = 1$. Damit erhalten wir insgesamt $|V| + 1$ Gleichungen in $|V| + 1$ Unbekannten.

Das unterliegende mathematische Problem ist das der Eigenwertberechnung.

Satz 31 *Unter der Annahme, daß eine Grenzverteilung F der Knotentypen innerhalb einer Schicht bei steigender Tiefe existiert, ist der asymptotische Verzweigungsgrad b ein positiver Eigenwert von P.*

Beweis: Um dies einzusehen, formt man $bF = FP$ äquivalent zu $0 = F(bI - P)$ um, wobei I die Identitätsmatrix darstellt. Die Lösungen für b sind durch die Determinante der charakteristischen Gleichung $\det(bI - P) = 0$ gegeben. □

Im *Fünf*-Puzzle mit Vorgängerelimination müssen wir demnach

$$\det \begin{pmatrix} b & -1 & 0 & 0 \\ 0 & b & -1 & -1 \\ 0 & 0 & b & -2 \\ -1 & 0 & 0 & b \end{pmatrix} = 0$$

berechnen. Diese Gleichung vereinfacht sich erwartungsgemäß zu $b^4 - b - 2 = 0$ für den asymptotischen Verzweigungsgrad.

Es sei an dieser Stelle bemerkt, daß die Konvergenz der Verteilung und des asymptotischen Verzweigungsgrads nicht immer gegeben ist. Im Beispiel des *Acht*-Puzzles alterniert die Folge zwischen zwei Häufungspunkten, wie wir später sehen werden. Darum ist es von entscheidender Bedeutung, den Konvergenzprozeß im Detail zu untersuchen. Sei n_i^d die Anzahl der Knoten vom Typ i in der Tiefe d des Suchbaumes und n^d die Gesamtzahl der Knoten in Tiefe d. Weiterhin definiere mit N^d den Vektor $(n_0^d, n_1^d, ..., n_{|V|-1}^d)$. Gleichfalls sei mit f_i^d der Anteil der Knoten vom Typ i in Tiefe d gegeben und mit F^d die Verteilung $(f_0^d, f_1^d, ..., f_{|V|-1}^d)$ dieser Anteile. In anderen Worten gilt f_i^d gleich n_i^d/n^d für alle i aus V. Sei der Knotenverzweigungsgrad b_k als die Anzahl der Kinder eines Knotens vom Typ k festgelegt und B durch den Vektor $(b_0, b_1, ..., b_{|V|-1})$ gegeben. In der Formulierung mit der Adjazenzmatrix P gleicht b_k der Summe der Matrixeinträge der k-ten Zeile, d.h. $b_k = \sum_{j \in V} p_{k,j}$.

Satz 32 *(Iterationsformel) Es gilt die Gleichung $F^d = F^{d-1}P/F^{d-1}B$, um aus der Verteilung F^{d-1} der Knoten der Tiefe $d-1$ die Verteilung F^d der Tiefe d zu gewinnen.*

Beweis:

$$\begin{aligned}
f_i^d &= n_i^d/n^d \\
&= \frac{N^{d-1}P^i}{\sum_{j \in V} N^{d-1}P^j} \\
&= \frac{\sum_{j \in V} n_j^{d-1} p_{j,i}}{\sum_{j \in V} \sum_{k \in V} n_k^{d-1} p_{k,j}} \\
&= \frac{\sum_{j \in V} f_j^{d-1} n^{d-1} p_{j,i}}{\sum_{k \in V} \sum_{j \in V} f_k^{d-1} n^{d-1} p_{k,j}} \\
&= \frac{F^{d-1}P^i}{\sum_{k \in V} f_k^{d-1} \sum_{j \in V} p_{k,j}} \\
&= F^{d-1}P^i/F^{d-1}B.
\end{aligned}$$

□

Satz 33 *(Verzweigungsgrad) Der durchschnittliche Verzweigungsgrad in Tiefe $d+1$ ergibt sich aus F^dB.*

Beweis:

$$\begin{aligned}
F^dB &= \sum_{i \in V} f_i^d b_i \\
&= \sum_{i \in V} n_i^d b_i/n^d \\
&= \sum_{i \in V} n_i^d \sum_{j \in V} p_{i,j}/n^d \\
&= \sum_{j \in V} \sum_{i \in V} n_i^d p_{i,j}/n^d \\
&= \sum_{j \in V} n_j^{d+1}/n^d \\
&= n^{d+1}/n^d = b^{d+1}.
\end{aligned}$$

□

Wenn somit durch die Iterationsformel eine Grenzverteilung erreicht wird, ist mit FB der asymptotische Verzweigungsgrad festgelegt, und wir kommen zu der Gleichung $bF = PF$ zurück.

Die Konvergenzuntersuchung der Iterationsformel erweisen sich als recht schwierig. Wir unternehmen dazu einen kurzen Abstecher in das Gebiet der Markovprozesse und Markovketten.

Eine Markovkette ist ein mathematisches Modell für zufällige zeitabhängige Phänomene (siehe Norris (1997)). Markovketten sind entweder endlich oder unendlich, diskret oder kontinuierlich. Wir interessieren uns für den endlichen und diskreten Fall. Auch in der Theorie der Markovprozesse dient ein Zustandsgraph als Ausgangspunkt. Die sogenannten Markovzustände lassen sich gemäß den starken Zusammenhangskomponenten des Graphens in eine Klassenstruktur eingliedern. Dabei existiert für jeden Zustand innerhalb einer starken Zusammenhangskomponenten ein Weg zu allen anderen Zuständen der Komponente. Eine Zusammenhangskomponente wird *geschlossen* genannt, wenn es keine Kante gibt, die aus der Klasse führt. Ein Zustand i heißt *rekurent*, falls die Wahrscheinlichkeit, i im weiteren Verlauf unendlich oft zu treffen, eins ist. Bei flüchtigen (*transienten*) Zuständen ist diese Wahrscheinlichkeit null. Wenn alle einen rekurenten Zustand enthaltenden Schleifen einen größten gemeinsamen Teiler von p haben, dann wird der Markovprozeß sich für i auf p Häufungspunkte einschwingen. Alle Zustände innerhalb einer starken Zusammenhangskomponente sind entweder rekurent oder transient. Die Grenzverteilung für transiente Zustände ist null. Für eine stochastische Matrix Q (die Zeilensummen addieren sich zu eins) ergibt sich die folgende Iterationsformel: $F^d = QF^{d-1}$. Dabei ist der Matrixeintrag $q_{i,j}$ ein von der Vergangenheit unabhängiger Übergangswahrscheinlichkeitswert, um von Zustand i in den Zustand j zu gelangen.

In unserem Fall liefert die Adjazenzmatrix P eine recht ähnliche Formel $F^d = PF^{d-1}/BF^{d-1}$. Dennoch verhindert der Nenner $F^{d-1}B$ die Interpretation unseres Problems als homogenen Markovprozeß.

10.6 Experimente

Hier wenden wir unsere Technik zur Berechnung des Verzweigungsgrads für quadratische Schiebepuzzles bis zur Größe 10×10 an. Tabelle 10.1 zeigt die geraden und ungeraden asymptotischen Verzweigungsgrade im (n^2-1)-Puzzle mit Vorgängerelimination auf. Die Einträge in den Spalten konvergieren gegen drei, dem Verzweigungsgrad eines unbegrenzt großen Schiebepuzzles mit Vorgängerelimination.

Um diesen Effekt der unterschiedlichen Konvergenz innerhalb des Puzzles zu verstehen, betrachten wir das *Acht*-Puzzle, das in Abbildung 10.4 dargestellt ist. In jeder zweiten Ebene des Suchbaumes wird das Leerfeld an einer Seite anliegen. Die restlichen Ebenen werden aus einer Mischung von Eck- und Mittelknoten bestehen. Die analytische Betrachtung des *Acht*-Puzzles kann dem Leser als Übung dienen.

Wenn wir das (n^2-1)-Puzzle wie ein Damebrett einfärben, wird das Leerfeld sich jeweils von einer Farbe auf eine andere bewegen. Für das *Acht*-Puzzle entsprechen die s Knoten einer Farbe und alle anderen der einer anderen.

n	n^2-1	gerade Tiefe	ungerade Tiefe
3	8	1.5	2
4	15	2.1304	2.1304
5	24	2.30278	2.43426
6	35	2.51964	2.51964
7	48	2.59927	2.64649
8	63	2.6959	2.6959
9	80	2.73922	2.76008
10	99	2.79026	2.79026

Tabelle 10.1: Der asymptotische Verzweigungsgrad für das (n^2-1)-Puzzle.

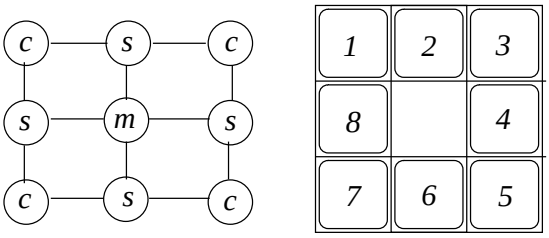


Abbildung 10.4: Seiten, Ecken und die Mitte innerhalb des *Acht*-Puzzles.

Wenn die zwei unterschiedlichen Mengen der Puzzles übereinstimmen, wird ein eindeutiger Grenzwert existieren. Falls, wie im *Acht*-Puzzle, diese Mengen unterschiedlich sind, so erhalten wir zwei Häufungspunkte.

10.7 Verallgemeinerte Beschneidungen

Bis jetzt haben wir Duplikate in den Schiebepuzzles dadurch erkannt, daß wir das Inverse des letzten Operators unterdrückt haben. Doch kann dieser Ansatz auf die in Kapitel 6 erläuterte Suchbaumbeschneidung mittels endlichen Automaten erweitert werden.

10.7.1 Restriktionsfreie Suchräume

Der das Rekursionsgleichungssystem begründende Graph ist durch die Struktur des endlichen Automaten gegeben. Die Anzahl der Zustände kann, wie in Kapitel 6 beschrieben, mehrere hunderttausend Zustände beinhalten und somit eine analytischen Lösung unmöglich machen.

Deshalb simulieren wir analog zur numerischen Lösung das Wachstum des Suchbaumes innerhalb der Struktur. Mit jedem Automatzustand verbinden wir eine Zahl n_q, die die Anzahl der Suchbaumzustände angibt, die sich in dem Zustand q befinden. Initial befindet sich ein einzelner Suchbaumknoten in dem Startzustand. In jeder Iteration werden die Suchbaumknoten entlang der möglichen Übergänge geleitet und aufaddiert.

Sei Q die Menge der Automatzustände Der Zählalgorithmus läßt sich für ein gegebenes (restriktionsfreies) Suchproblem für eine Iteration d wie folgt zusammenfassen.

1. Initialisiere alle n_q^{d+1} mit Null.

2. Für alle Zustände q aus Q und Übergänge a aus dem Zugalphabet Σ addiere den Wert n_q^d zu $n_{\delta(q,a)}^{d+1}$.

Die Laufzeit pro Iteration ist $O(|Q||\Sigma|)$. Der Speicherplatzverbrauch liegt bei $O(|Q|)$, da die n Werte in jeder zweiten Iteration neu überschrieben werden können.

Als Beispiel für den Algorithmus wählen wir den im *Gitter* (vergl. Abbildung 6.5 auf Seite 98) eingesetzten Automat. Abbildung 10.5 gibt die ersten vier Iterationen für das beschriebene Verfahren an.

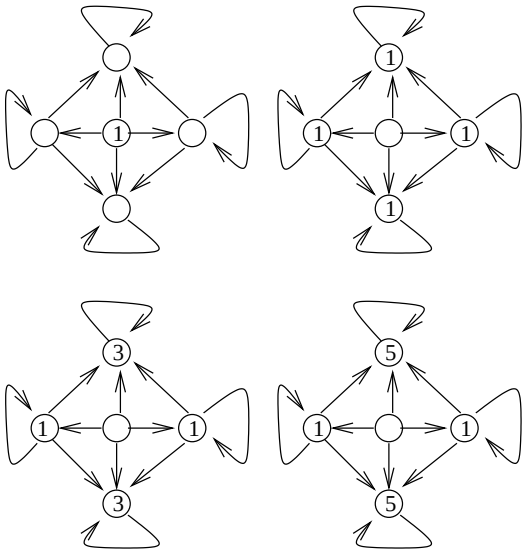


Abbildung 10.5: Zählen innerhalb des Automaten.

10.7.2 Restringierte Suchräume

Nicht immer lassen sich alle Übergänge innerhalb des Suchraumes auch wirklich durchführen. Befindet sich das z.B. das Leerfeld in dem $(n^2 - 1)$–Puzzle am Brettrand, so ist ein Zug in eben diese Richtung nicht möglich.

Wir wollen die Graphen zur Linken der Abbildung 10.1 *Zugmöglichkeitsgraph* nennen. Er ergibt sich durch die Variation einiger weniger Parameter des Suchraumes, wie hier des Leerfeldes. Durch die implizite Beschreibung des Suchraumes ist der Zugmöglichkeitsgraph im allgemeinen viel kleiner als der Suchraum selber.

Der zur automatischen Suchraumbeschneidung eingesetzte endliche Automat kann durch die Kombination mit der Postion im Zugmöglichkeitsgraph zur Ermittlung des Verzweigungsgrads in einem Zählalgorithmus genutzt werden. Gibt G' den durch den Automat beschriebenen Graphen an und G'' den Zugmöglichkeitsgraph, so bestimmen wir den Produktgraphen $G = G' \times G''$.

Als Beispiel wählen wir das Fünf-Puzzle mit Vorgängerelimination. Der Produktgraph aus dem Zugmöglichkeitsgraph des Fünf-Puzzles und dem Automat zur Vorgängerelimination (Abbildung 6.3 auf Seite 97) ist in Abbildung 10.6 dargestellt. Dabei wurde der Startzustand auf die obere linke Ecke

festgesetzt und die aus diesem Zustand erreichbaren Zustände und Übergänge aufgezeichnet.

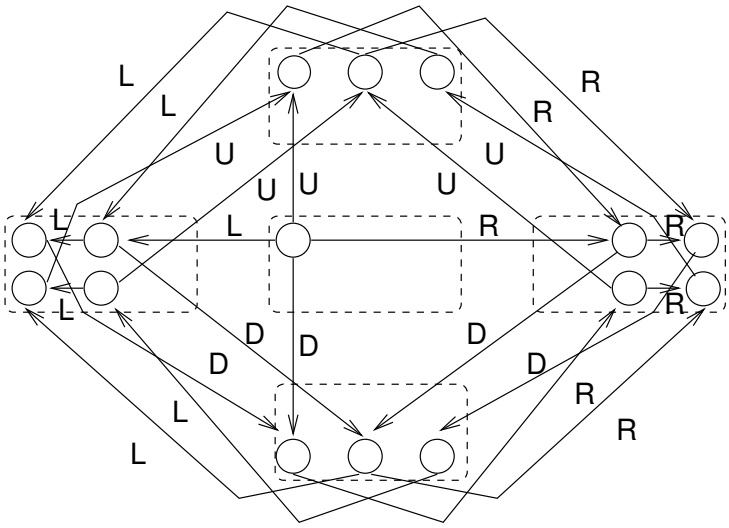


Abbildung 10.6: Produktgraph im *Fünf*-Puzzle mit Vorgängerelimination.

Innerhalb des Produktgraphen läßt sich nun der Zählalgorithmus zur Berechnung der Suchbaumgröße und des Verzweigungsgrads anwenden.

10.8 Ausblick

Wir haben gezeigt, wie der asymptotische Verzweigungsgrad eines Suchbaumes berechnet werden kann, in dem unterschiedliche Knoten eine unterschiedliche Anzahl von Nachfolgern hat. Existiert ein eindeutiger Verzweigungsgrad, so ist er ein Eigenwert einer bestimmten Matrix.

Die einzige offene Frage ist, die Konvergenzeigenschaften des Rekursionsgleichungssystems genau zu analysieren. Zwar wurde die Verbindung zu den Markov-Prozessen hergeleitet, doch steht eine vollständige Betrachtung des sich in diesem Fall ergebenden inhomogenen Prozesses noch aus.

In der heuristischen Suche wird der heuristische Verzweigungsgrad als der Quotient der Knotenexpansionszahlen von *IDA** in zwei aufeinanderfolgenden Iterationen festgelegt. Korf zeigt auf, daß unter vier Annahmen der heuristische Verzweigungsgrad gegen den hier dargestellten asymptotischen (bruteforce) Verzweigsgrad konvergiert (Korf 1997a): Erstens müssen die Schätzwerte ganzzahlig sein, zweitens dürfen die Bewertungen aller Nachfolgerknoten nicht um mehr als eins von der Bewertung ihres Vorgängers abweichen, drittens muß die aktuelle Suchtiefe d den maximalen Schätzwert $h_{\max}$ überschreiten und viertens sollte die Verteilung der heuristischen Werte von Knoten in der Tiefe $d + 1 - h_{\max}$ der des gesamten Suchraumes entsprechen. Durch Zählen und Aufsummieren der generierten Knoten in der Suchbaumtiefe i, $i \in \{1, \ldots, d+1\}$, findet man, daß die Anzahl generierter Knoten in *IDA** zur Tiefenschranke d unter diesen Annahmen gleich $\sum_{i=1}^{d+1} b^i \; Prob(d - i + 1)$ ist. Dabei ist $Prob(i)$ die Wahrscheinlichkeit, daß ein Knoten einen Schätzwert von höchstens i hat.

Literaturverzeichnis

Aho, A. V. and Corasick, M. J. 1975. Efficient string matching: An aid to bibliographic search. *Communications of the ACM* 18(6):333–340.

Amir, A. and Farach, M. 1991. Adaptive dictionary matching. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, 760–766. IEEE Computer Society Press.

Amir, A., Farach, M., Galil, Z., Giancarlo, R. and Park, K. 1994. Dynamic dictionary matching. *Journal of Computer and System Sciences* 49(2):208–222.

Amir, A., Farach, M., Idury, R. M., Poutré, J. A. L. and Schäffer, A. 1995. Improved dynamic dictionary matching. *Information and Computation* 119(2):258–282.

Amir, Farach and Matias. 1992. Efficient randomized dictionary matching algorithms. In *CPM: 3rd Symposium on Combinatorial Pattern Matching*.

Anderson, J. R. 1995. *Kognitive Psychologie*. Spektrum Akademischer Verlag.

Aoe, J. I. 1994. *Computer Algorithms: String Pattern Matching Strategies*. IEEE Computer Society Press.

Barr, A. and Feigenbaum, E., eds. 1981. *The Handbook of Artificial Intelligence*, volume 1. William Kaufmann.

Barrett, C., Jacob, R. and Marathe, M. 1998. Formal language constrained path problem. *Scandinavian Conference on Algorithm Theory*. To appear.

Barto, A. G., Bradtke, S. J. and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72(1):81–138.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Berlekamp, E. R., Conway, J. H. and Guy, R. K. 1982. *Winning ways (book)*. Academic Press, vol.I and II, 850 pages.

Biere, A. 1997. μcke - efficient μ-calculus model checking. In *Computer Aided Verification*, volume 1254 of *LNCS*, 468–471.

Blum, A. and Furst, M. 1995. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1636–1642.

Bollig and Wegener. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEETC: IEEE Transactions on Computers* 45.

Bonet, B., Loerincs, G. and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *Proceedings of the 14th National Conference on Artificial Intelligence*, 714–719. AAAI Press.

Brüngger, A. 1996. *Solving Large Combinatorial Optimization Problems in Parallel: Two Case Studies*. Ph.D. Dissertation, Department of Computer Science, Eidgenössische Technische Hochschule Zürich.

Bryant, R. E. 1985. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, 688–694. IEEE Computer Society Press.

Bryant, R. E. 1991. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers* 40(2):205–213.

Bundy, A., ed. 1997. *Artificial Intelligence Techniques, A Comprehensive Catalogue*, volume 1. Springer.

Carbonell, J., Blythe, J., Etzione, O., Gil, Y., Joseph, R., Kahn, D., Knoblock, C. and Minton, S. 1992. Prodigy 4.0: The manual and tutorial. Technical Report CMU-CS-92-150, Carnegie Mellon University.

Chang, W. I. and Lawler, E. L. 1994. Sublinear approximate string matching and biological applications. *Algorithmica* 12(4/5):327–344.

Chen and Seiferas. 1985. Efficient and elegant subword-tree construction. In *Alberto Apostolico and Zvi Galil, Combinatorial Algorithms on Words*. NATO ISI Series, Springer-Verlag.

Chimura, F. and Tokoro, M. 1994. The trailblazer search: A new method for searching and capturing moving targets. In *Proceedings of the 12th National Conference on Artificial Intelligence*, 1347–1352. AAAI Press.

Choi and Lam. 1996. Two-dimensional dynamic dictionary matching. In *ISAAC: 7th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms)*.

Commentz-Walter, B. 1979. A string matching algorithm that is fast on the average. In *Automata, Languages and Programming*, volume 72 of *Lecture Notes in Computer Science*. Springer. 118–132.

Cook, S. A. 1971. The complexity of theorem proving procedures. *ACM Symposium on Theory of Computing*.

Cormen, T. H., Leiserson, C. E. and Rivest, R. L. 1990. *Introduction to Algorithms*. The MIT Press.

Culberson, J. C. and Schaeffer, J. 1996. Searching with pattern databases. In *Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, volume 1081 of *LNAI*, 402–416. Springer.

Culbersone, J. 1997. Sokoban is PSPACE-complete. Technical Report TR-97-02, Departement of Computing Science, University of Alberta.

Dietz, P. and Sleator, D. 1987. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 365–372. ACM Press.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Dillenburg, J. F. and Nelson, P. C. 1994. Perimeter search. *Artificial Intelligence* 65(1):165–178.

Doppelhamer, J. and Lehnert, J. K. 1998. Optimal solution for sokoban using OBDDs. University of Trier, available from the authors.

Dutton, R. D. 1992. The weak-heap data structure. Technical report, University of Central Florida, Orlando, FL 32816.

Dutton, R. D. 1993. Weak-heap sort. *BIT* 33:372–381.

Eckerle, J. and Lais, T. 1998. Limits and possibilities of sequential hashing with supertrace. In *Conference FORTE 11/ PSTV 13*. To appear.

Eckerle, J. and Schuierer, S. 1994. Effiziente speicherplatzbeschränkte graph-such-algorithmen. Technical Report TR-57, University of Freiburg.

Eckerle, J. 1996. BDBIDA: A new approach for space-limited bidirectional heuristic graph search. In *12th European Conference on Artificial Intelligence*, 370–374.

Eckerle, J. 1997. *Heuristic search with restricted memory (Heuristische Suche unter Speicherbeschränkung)*. Ph.D. Dissertation, Institut für angewandte Wissenschaften, Universität Freiburg. DISKI, Infix, Band 185.

Edelkamp, S. and Eckerle, J. 1997. New strategies in real-time heuristic search. Technical Report WS-97-10, To be ordered from the AAAI Press.

Edelkamp, S. and Korf, R. E. 1998. The branching factor of regular search spaces. In *Proceedings of the 15th National Conference on Artificial Intelligence*. 299–304.

Edelkamp, S. and Reffel, F. 1998. OBDDs in heuristic search. In *KI-98: Advances in Artificial Intelligence*, Lecture Notes in Artificial Intelligence, 81–92.

Edelkamp, S. and Schrödl, S. 1998. Learning dead ends in sokoban. Technical Report CSR-98-01, Fakultät für Informatik, Technische Universität Chemnitz. http://www.informatik.uni-trier.de/GI/TheorieTag.

Edelkamp, S. 1996. Weak-heapsort, ein schnelles Sortierverfahren. Master's thesis, Fachbereich Informatik, Universität Dortmund.

Edelkamp, S. 1997. Suffix tree automata in state space search. In *KI-97: Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Artificial Intelligence*, 381–385.

Edelkamp, S. 1998. Updating shortest paths. In *13th European Conference on Artificial Intelligence*, 655–659.

Eidswick, J. A. 1986. Cubelike puzzles - what are they and how do you solve them. *American Mathematical Monthly* 93(3):157–176.

Fan, J.-J. and Su, K.-Y. 1993. An efficient algorithm for matching multiple patterns. *IEEE Transactions on Knowledge and Data Engineering* 5(2):339–351.

Ferragina and Luccio. 1996. On the parallel dynamic dictionary matching problem: New results with application. In *ESA: Annual European Symposium on Algorithms*.

Fikes, R. E. and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(1-4):189–208.

Fredman, M. L. and Tarjan, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithm. *Journal of the ACM, JACM* 34(3):596–615.

Friedman, S. J. and Supowit, K. J. 1990. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers* 39(5):710–713.

Gardner, M. 1959. *The Mathematical Games of Sam Loyd*. Dover Publications.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractibility, A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

Gasser, R. 1993. *Harnessing Computational Resources for Efficient Exhaustive Search*. Ph.D. Dissertation, Department of Computer Science, Eidgenössische Technische Hochschule Zürich.

Ghosh, S., Mahanti, A. and Nau, D. S. 1994. ITS: An efficient limited-memory heuristic tree search algorithm. In *Proceedings of the 12th National Conference on Artificial Intelligence*, 1353–1358.

Ginsberg, M. 1996. Partition search. In *Proceedings of the 13th National Conference on Artificial Intelligence*, 228–233.

Greeno, J. G. 1974. Hobbits and orcs: Acquisition of a sequential concept. *Cognitive Psychology* 6:270–292.

Hart, P. E., Nilsson, N. J. and Raphael, B. 1968. A formal basis for heuristic determination of minimum path cost. *IEEE Trans. on SSC* 4:100.

Holzmann, G. J. and Puri, A. 1998. A minimized automaton representation of reachable states. Bell Laboratories, Murray Hill, NJ. Available from the authors.

Holzmann, G. J. 1987. On limits and possibilities of automated protocol analysis. In *Proceedings of the 7th IFIP WG 6.1 International Workshop on Protocol Specification*, 137–161. North-Holland Publications.

Holzmann, G. J. 1988. An improved protocol reachability analysis technique. *Software, Practice and Experience* 18(2):137–161.

Holzmann, G. J. 1991. *Design and Validation of Computer Protocols*. Prentice Hall software series.

Huffman, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40(9):1098–1101.

Idury, R. M. and Schaeffer, A. A. 1994. Dynamic dictionary matching with failure functions. *Theoretical Computer Science* 131(2):295–310.

Ishida, T. and Korf, R. E. 1991. Moving target search. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 204–211. Morgan Kaufmann.

Ishida, T. and Shimbo, M. 1996. Improving the learning efficiencies of realtime search. In *Proceedings of the 13th National Conference on Artificial Intelligence*, 305–310.

Jeffries, R. P., Polson, P. G., Razran, L. and Atwood, M. E. 1977. A process model for missionaries-cannibals and other river-crossing problems. *Cognitive Psychology* 9:412–440.

Johnson, W. W. 1879. Notes on the 15 puzzle 1. *American Journal of Mathematics* 2:397–399.

Junghanns, A. and Schaeffer, J. 1998a. Single agent search in the presence of deadlocks. Proceedings of the 15th National Conference on Artificial Intelligence.

Junghanns, A. and Schaeffer, J. 1998b. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. Proceedings CSCSI-98,Vancouver, Canada, LNCS, Springer.

Knuth, D. E. 1973. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, Reading.

Koehler, W. 1927. *Intelligenzprüfung an Menschenaffen*. Springer.

Korf, R. E. and Taylor, L. A. 1996. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the 13th National Conference on Artificial Intelligence*, 1202–1207. AAAI Press.

Korf, R. E. 1985a. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. E. 1985b. Macro-operators: A weak method for learning. *Artificial Intelligence, 1985* 26:35–77.

Korf, R. E. 1988. Search: A survey of recent results. In *Exploring Artificial Intelligence*, 197–237. Morgan Kaufman.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2-3):189–211.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Korf, R. E. 1997a. Analysis of time complexity of heuristic search. Personal communications.

Korf, R. E. 1997b. Finding optimal solutions to rubik's cube using pattern databases. In *Proceedings of the 14th National Conference on Artificial Intelligence*, 700–705. AAAI Press.

Langley, P. 1996. *Elements of Machine Learning*. Morgan Kaufmann.

Larsen, M. E. 1985. Rubik's revenge: The group theoretical solution. *American Mathematical Monthly* 92(6):381–390.

McAllester, D. and Rosenblitt, D. 1996. Systematic nonlinear planning. In *Proceedings of the 10th National Conference on Artificial Intelligence*, 634–639.

McCreight, E. M. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM* 23(2):262–272.

McDermott, D. 1996. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Plannng Systems*.

McDiarmid, C. J. H. and Reed, B. A. 1989. Building heaps fast. *Journal of Algorithms* 10:352–365.

McMillan, K. 1993. *Symbolic Model Checking*. Kluwer Academic Press.

Mehlhorn, K. 1984. *Data Structures and Algorithms, volume 2 : NP Completeness and Graph Algorithms*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

Meinel, C. and Stangier, C. 1997. OBDD-based verification of communication protocols - Methods for the verification of data link protocols. Technical report, Universität Trier.

Meyer, B. 1985. Incremental string matching. *Information Processing Letters* 21:219–227.

Morrison, D. R. 1968. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM* 15(4):514–534.

Motwani, R. and Raghavan, P. 1995. *Randomized Algorithms*. Cambridge University Press.

Newell, A. and Simon, H. A. 1963. GPS: A program that simulates human thought. In *Computers and Thought*. McGraw-Hill.

Nilsson, N. J. 1982. *Principles of Artificial Intelligence*. Symbolic Computation. Springer.

Ottmann, T. and Widmayer, P. 1993. *Algorithmen und Datenstrukturen*. BI, 2 edition.

Parberry, I. 1995. A real-time algorithm for the (n^2-1)-puzzle. *Information Processing Letters* 56(1):23–28.

Pearl, J. 1985. *Heuristics*. Addison-Wesley.

Penberty, J. and Weld, D. 1992. UCPOP: A sound, complete partial order planner for ADP. In *3rd International Conference on Principles of Knowledge Presentation and Reasoning*, 189–197.

Ratner, D. and Warmuth, M. 1990. The (n^2-1)-puzzle and related relocation problems. *Journal of Symbolic Computation* 10(2):111–136.

Reinefeld, A. and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Reinefeld, A. 1993. Complete solution of the eight-puzzle and the benefit of node ordering in IDA*. In *13th International Joint Conference on Artificial Intelligence*, 248–253.

Rubik, E., Varga, T., Ummy, D. and Ummy, D. 1987. *Rubik's Cubic Compendium*. Oxford University Press.

Russell, S. J. and Norvig, P. 1995. *Artificial Intelligence. A Modern Approach*. Prentice-Hall.

Russell, S. 1992. Efficient memory-bounded search methods. In *Proceedings of the European Conference on Artificial Intelligence*, 1–5. Wiley.

S. Minato, N. Ishiura and S. Yajima. 1990. Shared Binary Decision Diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 52–57. ACM/IEEE.

Sasaki, T., Chimura, F. and Tokoro, M. 1995. The trailblazer search with a hierarchical abstract map. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 259–265. Morgan Kaufmann.

Schofield, P. D. A. 1967. Complete solution of the eight puzzle. In *Machine Intelligence 2*. Elsevier/North-Holland. 125–133.

Schöning, U. 1995. *Theoretische Informatik, kurz gefaßt*. Spektrum.

Schöning, U. 1997. *Algorithmen, kurz gefasst*. Spektrum.

Sieling, D. 1994. *Algorithmen und untere Schranken für allgemeine OBDDs*. Ph.D. Dissertation, Fachbereich Informatik, Universität Dortmund.

Simon, H. U. 1992. Effiziente algorithmen. Vorlesungsskript, Universität Dortmund.

Slisenko. 1983. Detection of periodicities and string-matching in real time. *JSM: Journal of Mathematical Sciences (formerly Journal of Soviet Mathematics)* 22.

Stangier, C. 1996. Speicherverwaltung bei der BDD-Synthese. Master's thesis, Fachbereich Informatik, Universität Dortmund.

Stephen, G. A. 1994. *String Searching Algorithms*. Lecture-Notes-Series-on-Computing. World-Scientific-Publishing.

Storey, W. 1879. Notes on the 15 puzzle 2. *American Journal of Mathematics* 2:399–404.

Sutton, R. S. and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. MIT Press.

Tani, Hamaguchi and Yajima. 1993. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *ISAAC: 4th International Symposium on Algorithms and Computation*.

Taylor, L. A. and Korf, R. E. 1993. Pruning duplicate nodes in depth-first search. In *Proceedings of the 11th National Conference on Artificial Intelligence*, 756–761. AAAI Press.

Taylor, L. A. 1997. *Pruning duplicate nodes in depth-first search*. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles.

Thorpe, P. E. 1994. Learning real time search algorithms. Master's thesis, Computer Science Department, University of California, Los Angeles.

Turner, E. C. and Gold, K. F. 1985. Rubik's groups. *American Mathematical Monthly* 92(9):617–629.

Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica* 14(3):249–260.

Wegener, I. 1992. The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP HEAPSORT is less than $n \log n + 1.1n$. *Information and Computation* 97(1):86–96.

Wegener, I. 1993a. The size of reduced OBDDs and optimal read-once branching programs for almost all boolean functions. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93)*.

Wegener, I. 1993b. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT, beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science* 118:81–98.

Wegener, I. 1993c. *Theoretische Informatik*. B.G. Teubner.

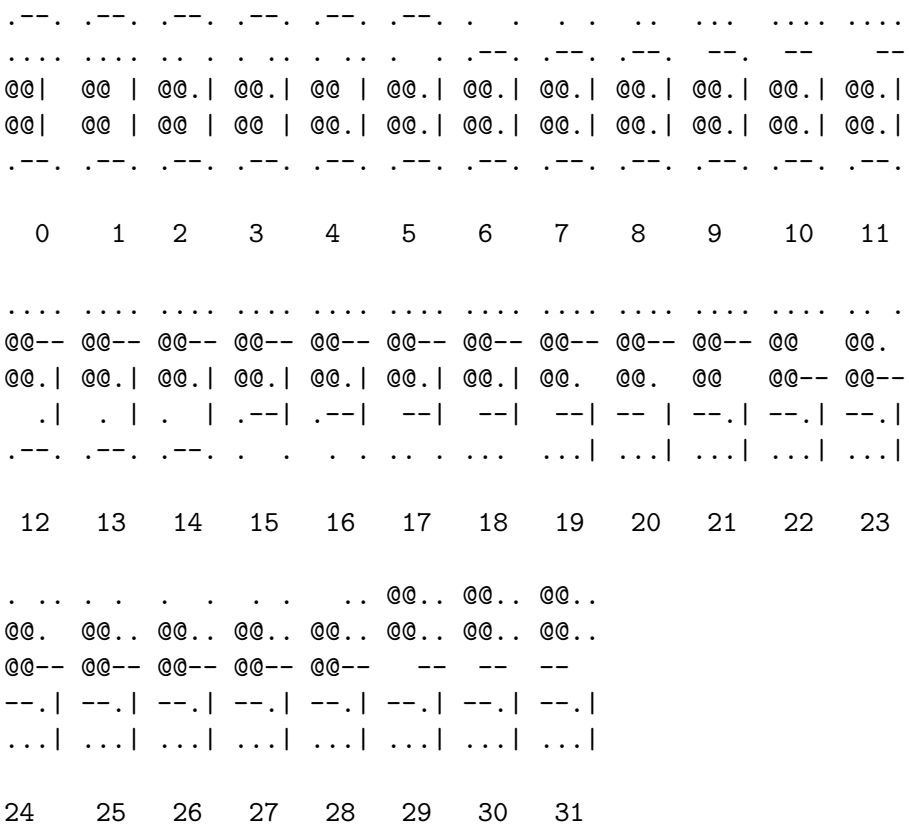
Weiner, P. 1973. Linear pattern matching algorithms. In *Conference Record, IEEE* 14^{th} *Annual Symposium on Switching and Automata Theory*, 1–11.

Wilson, R. 1974. Graph puzzle, homotopy and the alternating group. *Journal of Combinatorical Theory* B 16:86–96.

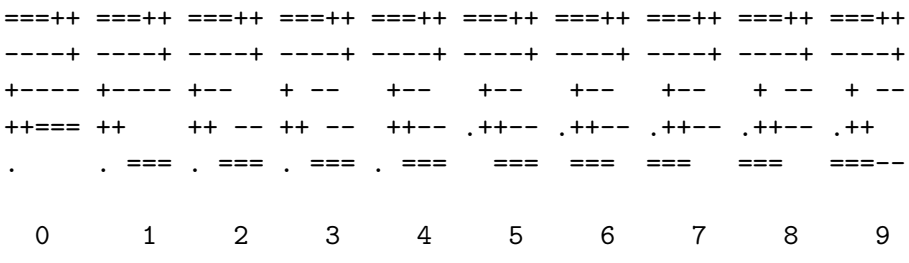
Anhang A

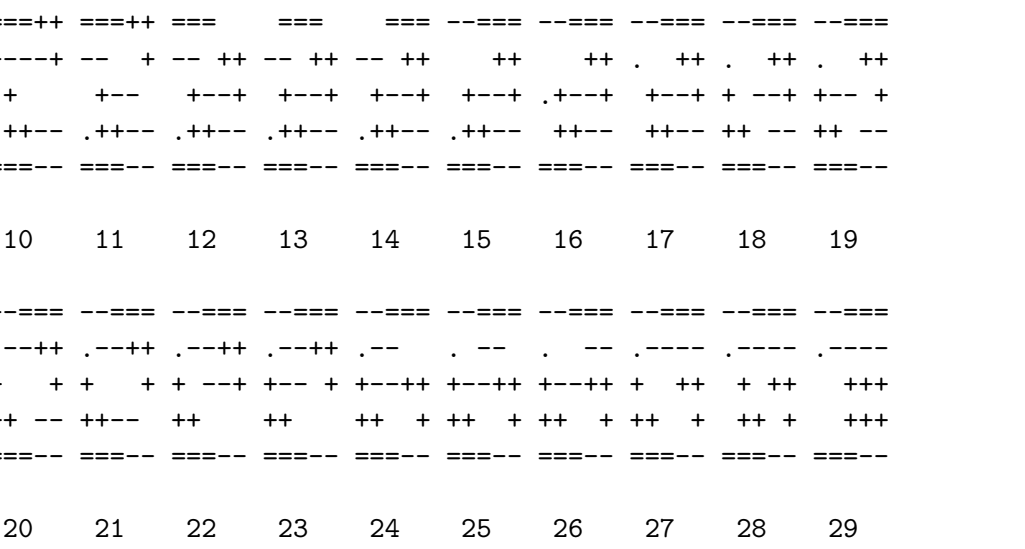
Lösungen

A.1 Vereinfachtes *Man and Bottle*-Puzzle

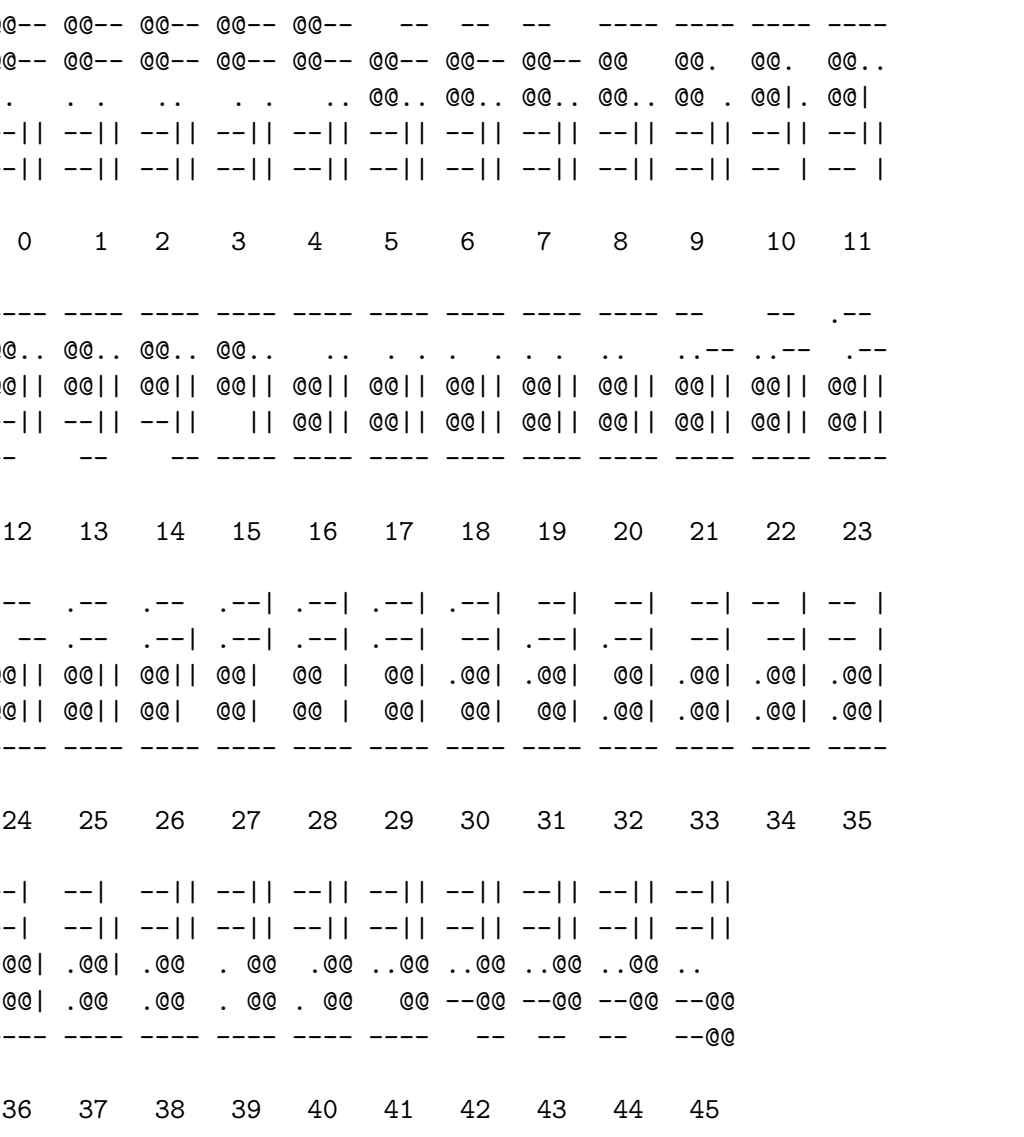


A.2 *Harlekin*-Puzzle

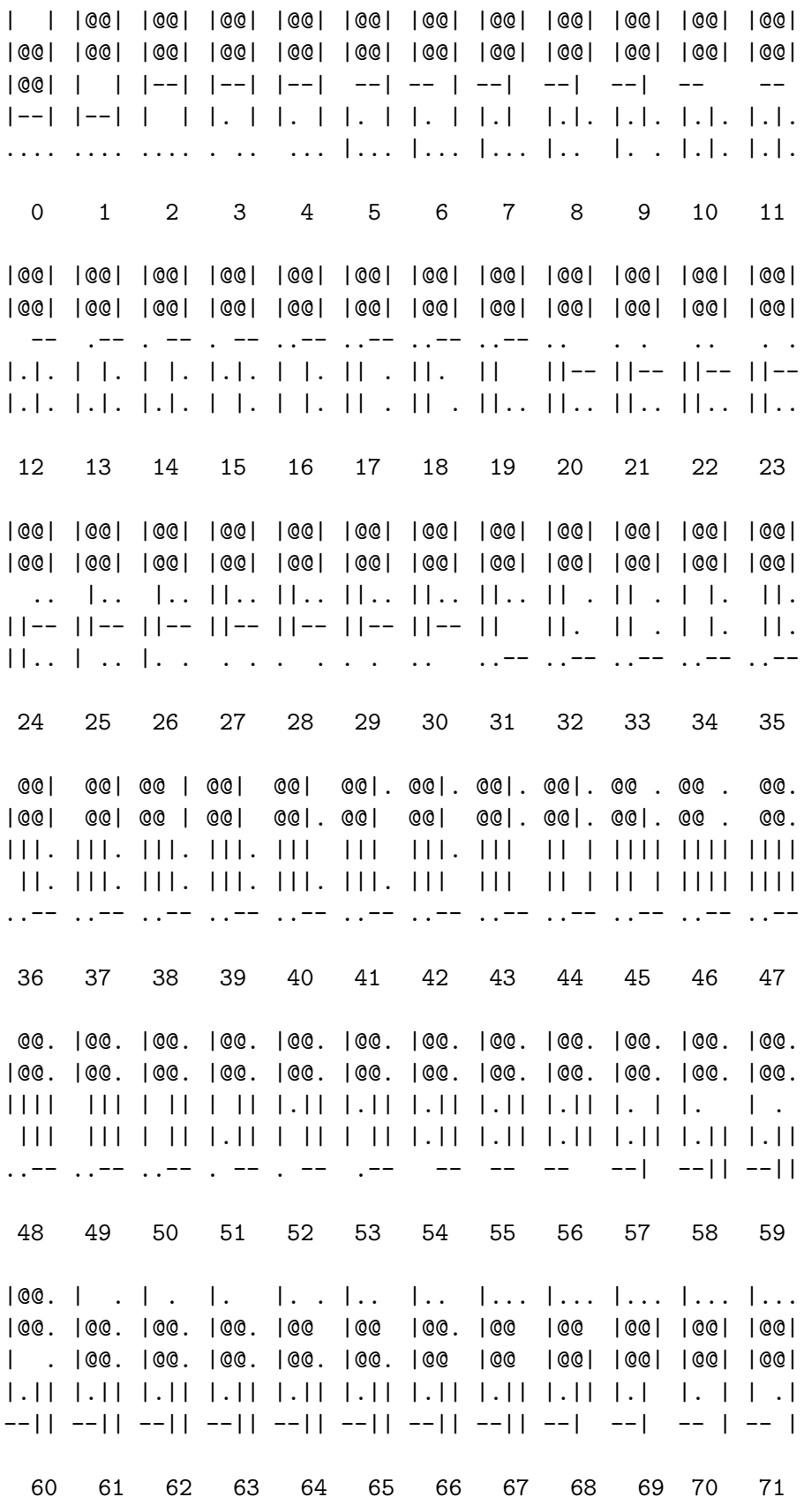


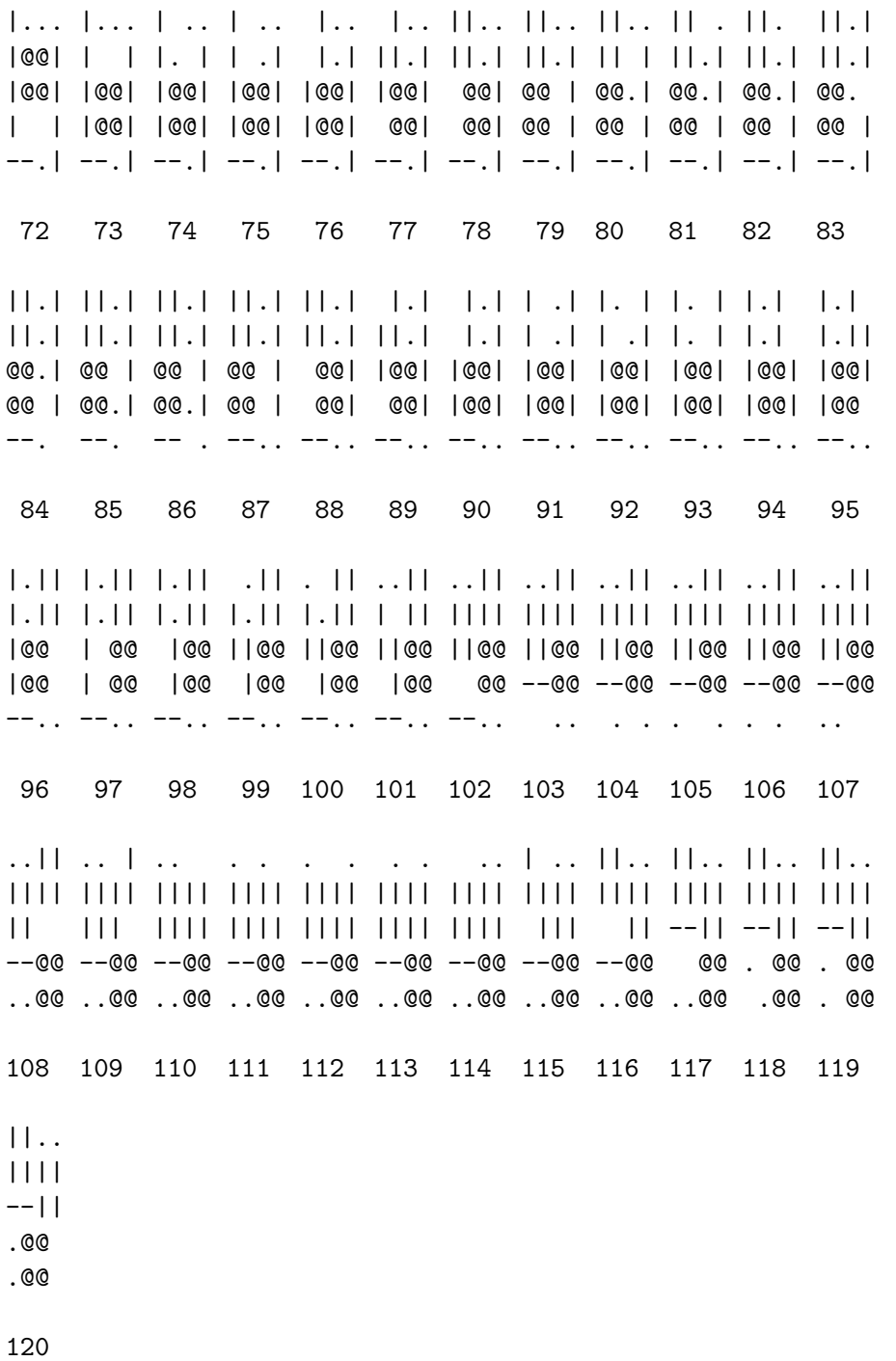


A.3 *Dad*-Puzzle

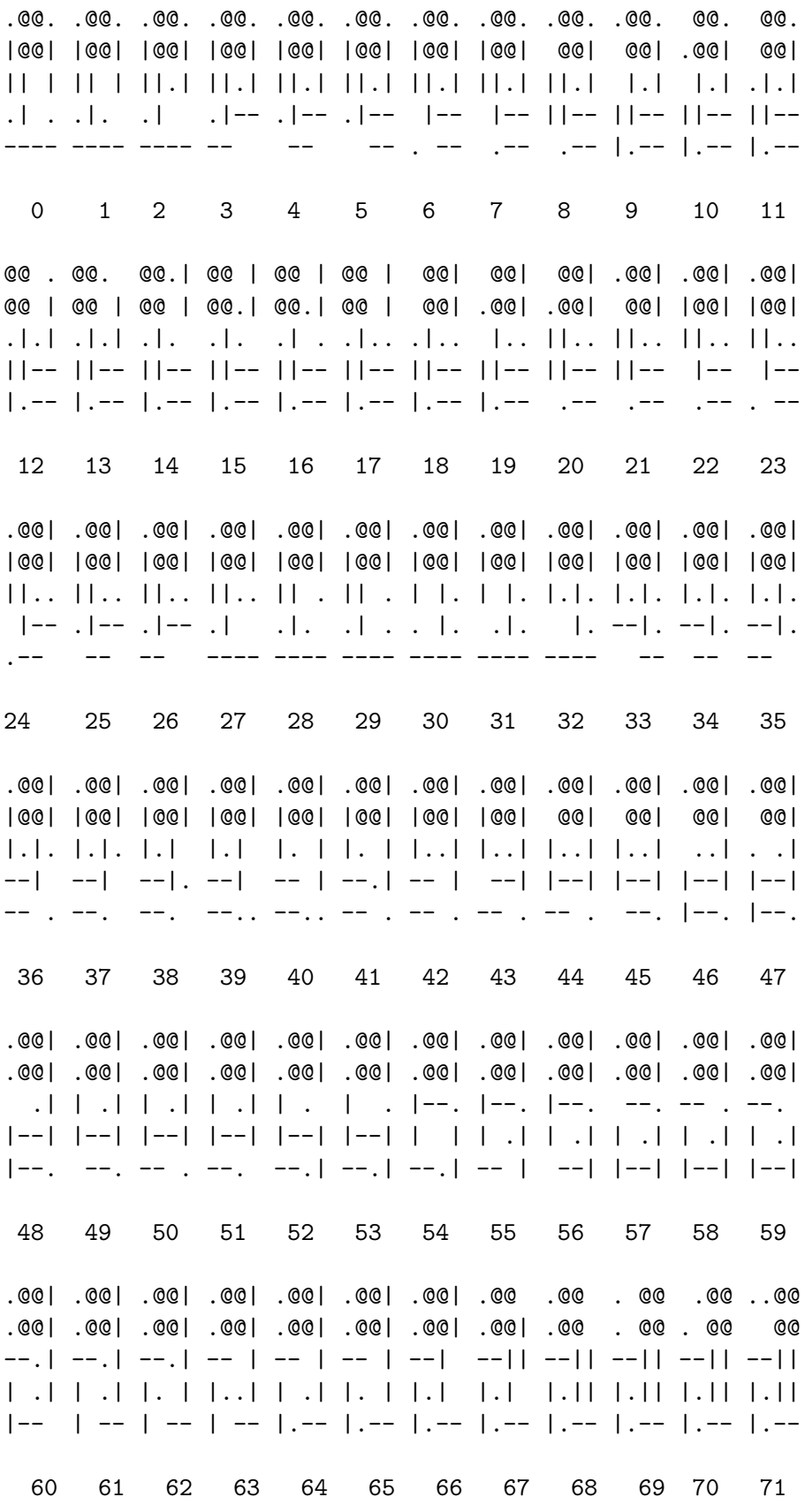


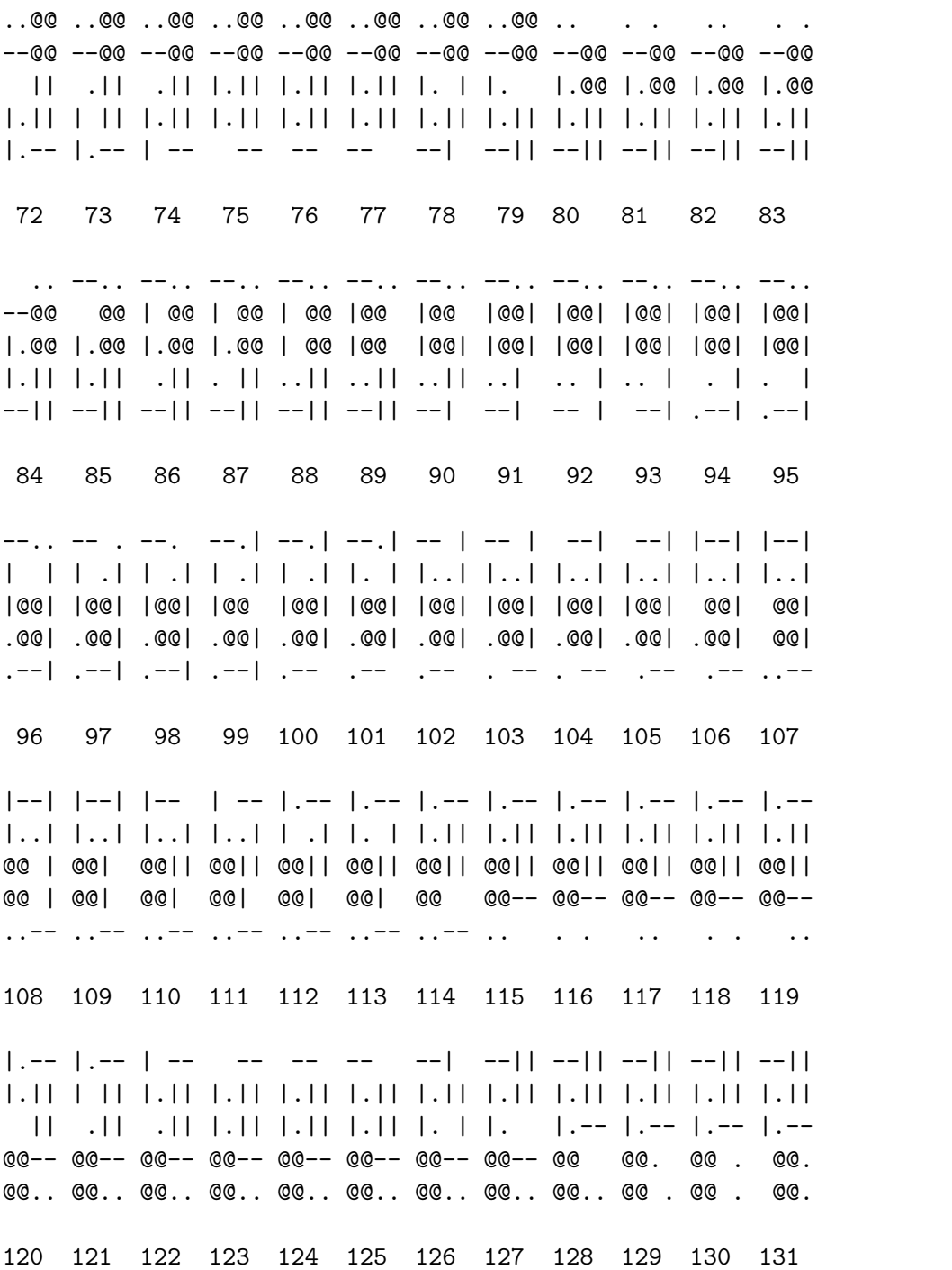
A.4 *Esel*-Puzzle



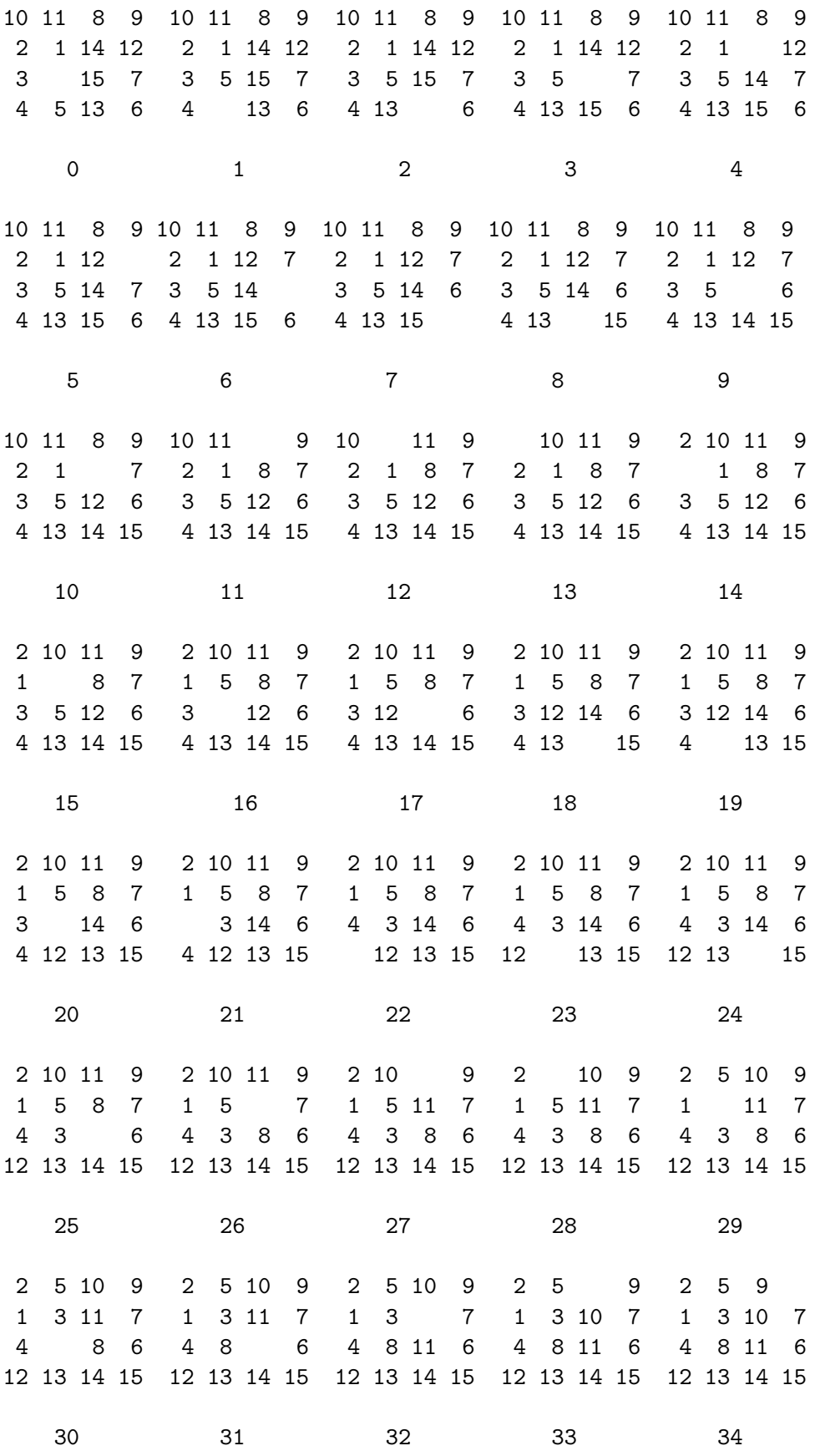

72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99 100 101 102 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119
120

A.5 *Jahrhundert*-Puzzle




72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99 100 101 102 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131

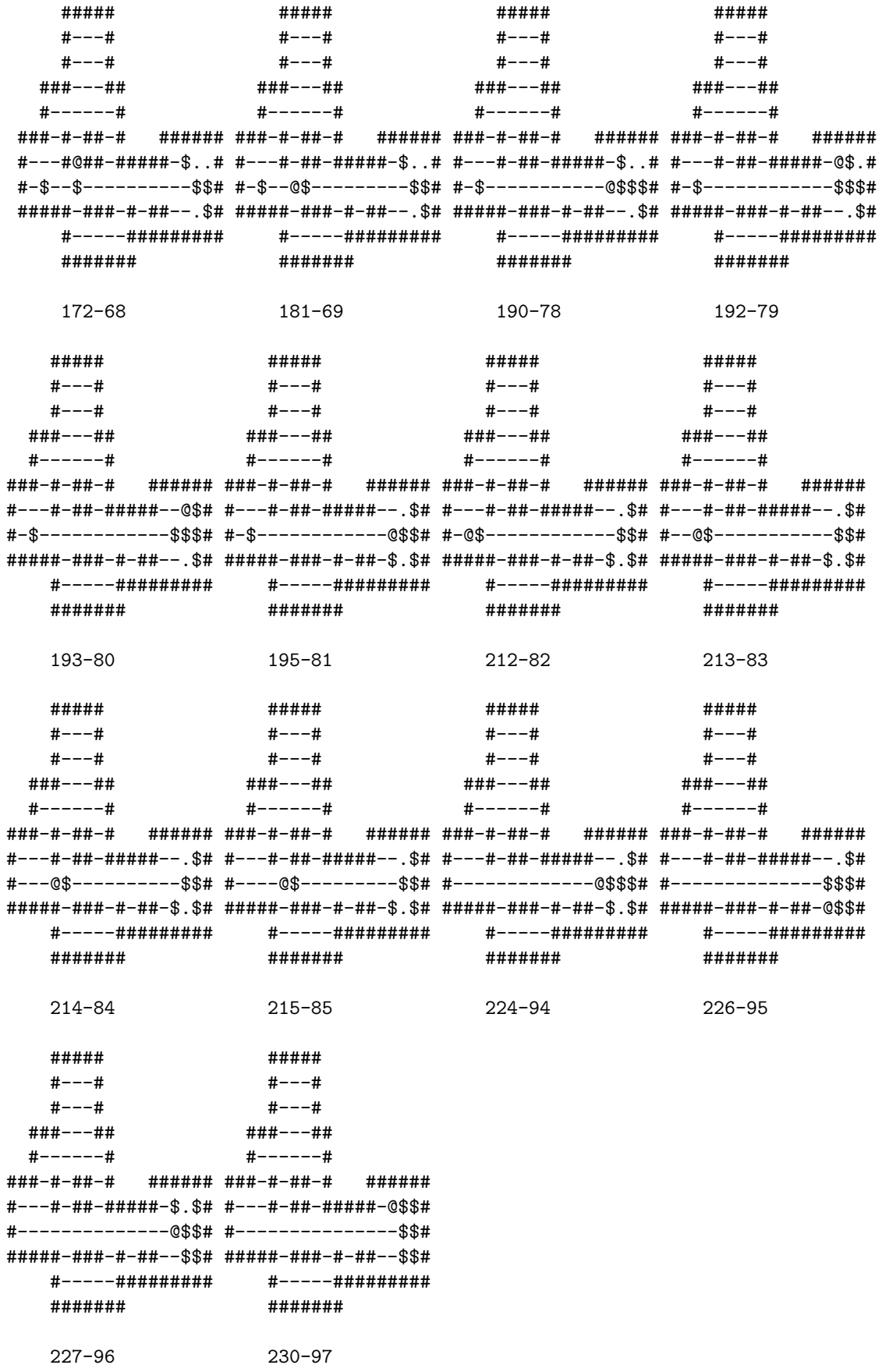
A.6 *Fünfzehn*-Puzzle



```
 2  5  9  7    2  5  9  7    2  5  9  7    2  5  9  7    2  5  9  7
 1  3 10       1  3 10  6    1  3 10  6    1  3     6    1     3  6
 4  8 11  6    4  8 11       4  8     11    4  8 10 11    4  8 10 11
12 13 14 15   12 13 14 15   12 13 14 15   12 13 14 15   12 13 14 15

    35            36            37            38            39

 2     9  7    2  9     7    2  9  3  7    2  9  3  7    2     3  7
 1  5  3  6    1  5  3  6    1  5     6    1     5  6    1  9  5  6
 4  8 10 11    4  8 10 11    4  8 10 11    4  8 10 11    4  8 10 11
12 13 14 15   12 13 14 15   12 13 14 15   12 13 14 15   12 13 14 15

    40            41            42            43            44

    2  3  7    1  2  3  7    1  2  3  7    1  2  3  7    1  2  3  7
 1  9  5  6       9  5  6    4  9  5  6    4  9  5  6    4     5  6
 4  8 10 11    4  8 10 11       8 10 11    8    10 11    8  9 10 11
12 13 14 15   12 13 14 15   12 13 14 15   12 13 14 15   12 13 14 15

    45            46            47            48            49

 1  2  3  7    1  2  3  7    1  2  3       1  2     3    1     2  3
 4  5     6    4  5  6       4  5  6  7    4  5  6  7    4  5  6  7
 8  9 10 11    8  9 10 11    8  9 10 11    8  9 10 11    8  9 10 11
12 13 14 15   12 13 14 15   12 13 14 15   12 13 14 15   12 13 14 15

     50            51            52            53            54


    1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15

    54
```

A.7 *Sokoban*-Spiel


90-34
99-35
108-44
111-45
122-46
123-47
126-48
128-50
137-51
146-60
158-61
149-62
151-63
168-64
169-65
170-66



172-68 181-69 190-78 192-79

193-80 195-81 212-82 213-83

214-84 215-85 224-94 226-95

227-96 230-97

A.8 *Seemannsspiel*-Spiel

```
WWW   WWW   WWW   WWW   WWW   WWW   WWW   WWW   WWW   WWW
WWW   WWW   WWW   WWW   WWW   WWW   WWW   WWW   WWW   WWW
WW BB W WBB WBW B WBWB  WB BW  BWBW B WBW BBW W BBWBW BBWBW
  BBB   BBB   BBB   BBB   BBB   BBB   BBB   BBB   BBB   BBB
  BBB   BBB   BBB   BBB   BBB   BBB   BBB   BBB   B B   BB

  0     1     2     3     4     5     6     7     8     9

WWW   WWW   WWW   WWW   WWW   WWW   WWW   WW    WW    BWW
WWW   WWW   WWW   WW     WW   BWW   BWW   BWW   BWW   BWW
BBWB  BB BW BBBBW BBBBW BBBBW  BBBW BB BW BBWBW BBWBW  BWBW
  BBB   BBB    BB   WBB   WBB   WBB   WBB   WBB   WBB   WBB
  BBW   BBW   BBW   BBW   BBW   BBW   BBW   BBW   BBW   BBW

 10    11    12    13    14    15    16    17    18    19


BWW   BWW   BWW   BWW   BWW   BWW   BWW   BWW   BWW   BWW
BWW   BWW   BWW   BWW   BWW   BWW   BWW   BW    BWB   BWB
B WBW BBW W BBWBW BBWBW BBWBW BBWB  BB BW BBWBW BBWBW BBWBW
  WBB   WBB   W B    WB    BW    BWW   BWW   BWW    WW   BWW
  BBW   BBW   BBW   BBW   BBW   BBW   BBW   BBW   BBW    BW

 20    21    22    23    24    25    26    27    28    29


BWW   BW    B W   BBW   BBW   BBW   BBW   BBW   BB    BBB
BWB   BWB   BWB   BWB   BWB   BWB   BWB   BWB   BWB   BW
BB BW BBWBW BBWBW B WBW BBW W BBWBW BBWBW BB BW BBWBW BBWBW
  BWW   BWW   BWW   BWW   BWW   BWW   BWW   BWW   BWW   BWW
  WBW   WBW   WBW   WBW   WBW   W W    WW   WWW   WWW   WWW

 30    31    32    33    34    35    36    37    38    39


BBB   BBB   BBB   BBB   BBB   BBB   BBB
B W   BBW   BBW   BBW   BBW   BB    BBB
BBWBW B WBW BBW W BB WW BBBWW BBBWW BB WW
  BWW   BWW   BWW   BWW    WW   WWW   WWW
  WWW   WWW   WWW   WWW   WWW   WWW   WWW

 40    41    42    43    44    45    46
```

Anhang B

Projekte

Die Darstellung der während der Promotion erstellten Software-Pakete konzentriert sich auf die Verwirklichung der wesentlichen Ideen.

B.1 Das Programmpaket *HSF-Light*

In diesem Abschnitt stellen wir die Architektur der in der Programmiersprache *c++* verwirklichten Werkbank *HSF-Light* vor. *HSF-Light* dient sowohl der Lösung von Zustandsraumproblemen als auch der Programmierung neuer Suchverfahren. Weiterhin verbindet *HSF-Light* verschiedene Lernansätze, die in dieser Dissertation angesprochen werden, wie z.B. die automatische Erkennung von Duplikatsequenzen. *HSF-Light* ist als offenes Projekt konzipiert, in das immer wieder neue Zustandsraumprobleme und Suchalgorithmen hinzugefügt werden. Zum Zeitpunkt der Abgabe bestand *HSF-Light* insgesamt aus ca. 10000 Programmzeilen.

B.1.1 Suchprobleme und Spiele

Als Standartsuchverfahren sind die in Kapitel 3 besprochenen Algorithmen A^*, IDA^*(mit oder ohne Transpositionstabelle), $MEIDA^*$ und *branch and bound* implementiert. In der Duplikatselimination wird die Stringerkennung sowohl über den Ansatz von Aho und Corasick für ein nicht-inkrementelles Lernen als auch mit Multi-Suffixbäumen für den inkrementellen Lernprozeß ILA^* realisiert. Zum Lernen der heuristischen Bewertungsfunktion sind das Verfahren $LRTA^*$ und die neuen Ansätze $CRTA^*$ und $SLRTA^*$ implementiert. Das Erkennen von Sackgassen mit dem ADP^* Algorithmus sowie das Jagen von beweglichen Zielen wird ermöglicht. Die Beispieldomänen sind die in dieser Arbeit vorgestellten Einpersonenspiele: das (n^2-1)-Puzzle, das *allgemeine Schiebepuzzle*, das *Sokoban*-Puzzle, der *Zauberwürfel*, das *Seemannsspiel*, das *Problem des Handlungsreisenden*, das *Tower-of-Hanoi*-Problem und einige *Gitter*- und *Labyrinthprobleme*. *HSF-Light* verbindet zwei Ideen miteinander.

Die am Institut für Informatik der Universität Freiburg von Eckerle und Müller für die Lehre erstellte Werkbank *HSF* ermöglicht es, ein aufgetragenes Problem innerhalb kurzer Zeit zu programmieren (Eckerle 1997). Dabei ist die Schnittstelle für die Expansion, für die Zielabfrage und für die heuristische Bewertung durch *virtueller Funktionen* vorspezifiziert. Die Anbindung an das Projekt erfolgt mit Hilfe der sogenannten *dynamischen Bindung*. Die Implementation einer Domäne gliedert sich in einen graphischen und einen

algorithmischen Teil, die über eine kleine Schnittstelle miteinander kommunizieren (Austausch von spielbeschreibenden Ganzzahlensequenzen).

Zusätzlich werden in *HSF-Light* Effizienzaspekte aus Korfs Implementationen von A^* und IDA^* berücksichtigt: Eine eigenständige Speicherverwaltung, eine ganzzahlig adressierte Vorrangwarteschlange, eine offene Hashtabelle als auch das Packen und Entpacken der Zustände. Der Speicherplatz wird statisch verwaltet und nur zur Beginn des Suchverfahrens angefordert.

B.1.2 Datenstrukturen

Speicherverwaltung

Der benötigte Speicherplatz wird in *HSF-Light* nur einmalig zum Beginn des Programms angefordert. Dies ist von Vorteil, da innerhalb der Suche kein zusätzlicher Speicher vom unterliegenden Betriebssystem angefordert wird. Eine Ausnahme sind die Multi-Suffixbäume, die dynamisch verwaltet werden.

Ein gespeicherter Zustand ist ein 8-Tupel (u,a,g,h,*parent*,*succ*,*pred*,q). Dabei ist u eine (komprimierte) Zustandsbeschreibung, a der ausgeführte Zug, g und h die Generierungspfadlänge bzw. der Schätzwert für die noch zu bewältigende Weglänge zum Ziel, *parent* der Verweis auf den Vorgänger im Suchbaum und *succ* bzw. *pred* zwei Zeiger, die die doppelte Verkettung einerseits in der Freispeicherliste und andererseits in der Hashtabelle ermöglichen. Die eindeutige Zustandsbeschreibung hat eine vordefinierte feste Länge. Für ein Tiefensuchverfahren wie IDA^* kann der Speicher als ein expliziter Rekursionsstapel aufgefaßt werden. Der aktuelle Zustand des parallel zur Suche genutzten endlichen Automaten wird durch einen weiteren Eintrag q gewährleistet.

Die Hashtabelle

Die Hashtabelle wird über ein sogenanntes offenes Hashing verwaltet, d.h. Knoten gleichen Hashwertes werden miteinander verkettet. Die Hashfunktion bildet einen gegebenen (komprimierten) Zustand auf dem mit dem Knoten verbundenen Speicherindex ab.

In einigen Domänen wie dem allgemeinen Schiebepuzzle sind nicht alle gespeicherten Informationen unbedingt zur Ermittlung der Hashadresse einzubeziehen. Die Position der Leerfelder ergibt sich zwar aus der Position der übrigen Steine, doch ist eine Extraktion dieser Information aus der Position der Spielsteine zu zeitaufwendig. Die Hashtabelle kann als Cache verwendet werden, in dem aufgrund der Speicherplatzbeschränkung einige Zustände überschrieben werden.

Die Vorrangwarteschlange

In vielen Suchräumen sind die Gewichte der Kanten sowie die heuristischen Schätzwerte ganzzahlig und durch eine nicht übermäßig große Konstante beschränkt. Dies ermöglicht uns, die Zustände in Fächer (engl. buckets) gleicher Bewertung einzuteilen und die Elemente innerhalb eines Fachs miteinander zu verbinden. Die Vorrangwarteschlangenoperation *DeleteMin* entnimmt somit aus dem ersten nicht freien Fach das *oberste* Element. Eine doppelte Verkettung erlaubt es, in konstanter Zeit einen Zustand aus einem Fach zu entfernen. Im Fall allgemeiner Kantenbewertungen greifen wir in *HSF-Light* auf die mit

Weakheaps realisierte und beidseitig zu verwendende Vorrangwarteschlange zurück.

Zeichenkettenwörterbuch

Im *HSF-Light* Programm gibt es zwei verschiedene Möglichkeiten, Duplikatssequenzen zu lernen (siehe Kapitel 5). Mit dem Algorithmus von Aho und Corasick läßt sich aus der Menge der Duplikate ein endlicher Automat konstruieren, der genau dann akzeptiert, wenn ein Anfragestring eines dieser Duplikate als Teilstring enthält. Auf der anderen Seite steht die dynamische Verwaltung der Duplikate in dem auf Multi-Suffixbäumen basierenden Zeichenkettenwörterbuch.

Wird eine Zeichenkette durch den Automaten akzeptiert, ist die in dem Lernprozeß gefundene und nun generalisierend vorgeschlagene Abkürzung zugreifbar, um in restringierten Räumen die Anwendbarkeit dieser Sequenz an der gegebenen Stelle im Suchraum zu prüfen.

Knotenhierarchie

Die Knoten innerhalb des Automaten sind statisch als Eintrag aus einer 2-dimensionalen Tabelle oder im Fall der Multi-Suffixbäume über eine hierarchische Struktur realisiert. Im zweiten Fall läßt sich die Knotenart über den Templatemechanismus in *c++* variieren, so daß wahlweise der Zugriff auf die Nachfolger durch ein Array, durch eine verkettete Liste oder durch die zusammengefügten Teile eines Huffmanbaumes (siehe Kapitel 6) erfolgt. Damit kann man sich zwischen einer geeigneten zeit- und einer platzeffizienten Darstellung des Automaten hin- und herbewegen.

B.1.3 Prozeduren

Packen und Entpacken von Zuständen

Die zur Expansion genutzte Zustandsbeschreibung ist zumeist nicht sehr speicherplatzsparend. Da eine große Menge an Zuständen (bis zu mehreren Millionen) der verschiedenen Verfahren im Hauptspeicher gehalten werden sollen, ist eine äußerst komprimierte Darstellung wünschenswert. Demnach empfiehlt es sich, eine Pack- und eine Entpackfunktion anzubieten, die die Darstellungen ineinander überführen können. Eine Konfiguration im *Fünfzehnpuzzle* kann z.B. in zwei Ganzzahlwerten (8 Byte) abgelegt werden.

Eine gute Packroutine erweist sich auch für die schnelle Berechnung der Hashadressen als nützlich, da eine kürzere Zustandsbeschreibung im allgemeinen auch zu einer schnelleren Adressierung führt. Weiterhin ermöglicht das Packen eine größere Unabhängigkeit und damit bessere Streuung der Hashfunktion von der explizit gegebenen Zustandsbeschreibung.

Direkte und inkrementelle Heuristiken

Der heuristische Schätzwert eines gegebenen unkomprimierten Zustandes kann direkt oder inkrementell geschehen. Bei der direkten Berechnung wird die untere Schranke ohne Hinzunahme der Bewertung im vorangegangenen Schritt allein durch die Betrachtung der Zustandsbeschreibung ermittelt. Bei einer

inkrementellen Berechnung nutzt man hingegen dieses Wissen zur Effizienzsteigerung aus. Als Beispiel betrachten wir das (n^2-1)-Puzzle mit der Manhattendistanz als untere Schranke. Durch die Bewegung des Leerfeldes ändert sich dieser Schätzwert um einen Wert mit Absolutbetrag eins, der sich durch den gezogenen Spielstein ergibt (entweder bewegt er sich in oder entgegen seines erstrebten Zielfeldes). Diese Verbesserungen können pro Spielstein, gegebener Position und ausgewählter Richtung in einer Tabelle abgelegt und in konstanter Zeit ausgelesen werden. Der heuristische Schätzwert des Vorgängers wird über den Zeiger *parent* erschlossen.

Vorwärts- und Rückwärtszüge

In der Zustandsraumsuche empfiehlt es sich, insbesondere für kompliziertere Puzzles zwischen der Zugmöglichkeit und der Zugausführung zu unterscheiden. In vielen in dieser Arbeit angesprochenen Themen, wie der bidirektionalen Suche, des Lernens von Makrooperatoren, der Generierung und automatischen Elimination von Duplikaten, der Aufwärtspropagierung nicht zur Lösung führender Teilstellungen und dem *CRTA** Verfahren ist außerdem die Anwendung eines Zuges in der Rückrichtung erforderlich.

Anwendbarkeitstest

Bei der automatischen Elimination von Duplikaten ist die Anwendbarkeit einer ganzen Zugsequenz zu prüfen, da die zu einem Duplikat assoziierte Abkürzung wie bei einer Mauer im Labyrinth nicht immer greift.

Die Expansion eines Knotens

Das zur Auswertung von Suchverfahren genutzte Komplexitätsmaß ist das der Knotenexpansion. Wenn man eine gute Schätzung der Anzahl von Knotenexpansionen pro Sekunde hat, läßt sich diese Zahl auf die Gesamtlaufzeit des Verfahrens hochrechnen.

Die Programmierung der wie das Zielprädikat essenziell notwendigen Expansionsfunktion bildet den Kern eines Zustandsraumproblems. Hier werden die Nachfolger eines Knotens bestimmt und an das die Expansionsfunktion aufrufende Suchverfahren weitergeleitet.

Ein zur Expansion beigegebener Zustand wird entpackt und die Werte g, h, a und *parent* der durch die Übergangsregeln festgelegten Nachfolger ermittelt. Die Nachfolgezustände werden untereinander verkettet.

B.1.4 Ein- und Ausgabe

In *HSF-Light* steht keine graphische Benutzeroberfläche zu Verfügung. Dies macht das Programm plattformunabhängig und somit vielseitig einsetzbar. Initial gibt das Programmpaket *HSF-Light* eine Übersicht über den durch die einzelnen Strukturen verbrauchten Speicherplatz, um dem Benutzer die Auslastung seines Systems vor Augen zu halten.

Der Lösungspfad wird beim erstmaligen Erreichen eines Zielknotens rekursiv aus den Vorgängerverweisen erstellt. Dabei werden sowohl die Beschriftung der Kanten als auch die Speicherindizes aufgesammelt. Dadurch können die

gesamte Zugsequenz und die erreichten Zwischenzustände auf dem Bildschirm oder in ein File abgelegt werden.

HSF-Light berechnet eine Menge von Protokolldaten: Die Anzahl der generierten und expandierten Knoten, den maximal benötigte Speicherplatz, die Anzahl der Hashoperationen und die Zahl der abgeschnittenen Suchbaumzweige innerhalb des Suchbaumes. Für A^* ähnliche Verfahren werden zusätzlich die Anzahl wiedergeöffneter Knoten angegeben, während in IDA^* die Statistiken für jeden neuen Schwellwert angegeben werden. Damit läßt sich der (heuristische) Verzweigungsfaktor innerhalb der Suche direkt ablesen.

B.2 StaticBdd

Es gibt sehr effiziente und ausgereifte *OBDD* Pakete, wie z.B. das *CUDD* Paket von Fabricio Somenzi (Department of Electrical and Computer Engineering, University of Colorado at Boulder). Dennoch ist mit *StaticBdd* ein weiterer Akzent in Richtung einer noch effizienteren *OBDD* Verwaltung gelegt. Das Projekt realisiert in ca. 1000 Zeilen die wesentlichen *OBDD* Algorithmen und eröffnet somit dem an Implementierungsdetails interessierten Leser die algorithmischen Kerne der einzelnen Verfahren. Wie das *CUDD* Paket, verwaltet *StaticBdd* mehrere *OBDDs* innerhalb einer Datenstruktur (S. Minato, N. Ishiura & S. Yajima 1990). In dieser Struktur ist der Äquivalenztest zweier Funktionen durch einen einfachen Zeigervergleich in konstanter Zeit möglich.

B.2.1 Prozeduren

Da die Syntheseoperation und Existenz-Quantifizierung schon in Kapitel 4 besprochen wurde, konzentrieren wir uns hier auf die Beseitigung nach einer solchen Operation nicht mehr benötigter Knoten.

Garbage Collection

Die zu entfernenden Knoten werden markiert und in einer sogenannten *garbage collection* aufgesammelt und gelöscht. Diese Operation ist zeitintensiv und wird nur bei absoluter Speicherknappheit aufgerufen.

Im *CUDD* Paket wird ein Referenzzähler verwaltet, der die Anzahl der eingehenden Kanten zählt. In *StaticBdd* nutzen wir hingegen ein einzelnes Bit, das aufzeigt, ob der Knoten sich innerhalb eines gelöschten *OBDDs* befindet oder nicht. Dies führt zu einer zeitintensiveren *garbage collection*, in der wegen der Überschneidung gelöschter und ungelöschter *OBDD* Knoten alle noch genutzten *OBDDs* traversiert werden müssen. Für den Tiefensuchlauf innerhalb eines *OBDDs* ist für jeden Knoten zusätzlich ein Bit als *Besucht*–Markierung vorgesehen.

Eine doppelte Verkettung der Knoten in der Kollisions- oder Freispeicherliste würde eine *garbage collection* vermeiden, wurde jedoch aus Speicherplatzgründen nicht vorgesehen.

B.2.2 Datenstrukturen

Wie im *HSF-Light* Projekt werden auch in *StaticBdd* alle Variablen statisch verwaltet. Dieses erweist sich als starker Effizienzvorteil, da nur beim Aufruf des Programms Speicherplatz angefordert wird.

Hashtabellen

Die Transpositionstabelle (vergl. Kapitel 4) ist durch einen Cache der Tiefe eins (vergl. Seite 46) und die Eindeutigkeitstabelle durch offenes Hashing realisiert. Die Verkettung der Listen ist einfach. So ist ein Zeiger an jedem Knoten vorgesehen, der alternativ für die Kollisionsbehandlung und für die Freispeicherliste genutzt wird.

Knoten

Im *CUDD*-Paket werden insgesamt 16 Bytes für jeden Knoten vorgesehen: acht Bytes für den Hinweis auf die Nachfolgerknoten, jeweils zwei Bytes für den Index und Referenzzähler des Knotens, vier Bytes für den Zeiger in der Eindeutigkeitstabelle. Ein Eintrag in der Transpositionstabelle beinhaltet die zwei Knotenindizes der Ursprungs-*OBDDs*. Auch hier wird ein Nachfolgehinweis für die offene Adressierung vorgesehen. Nimmt man an, daß die Größe der Transpositions- und der Eindeutigkeitstabelle bei ca. 1/4 der maximalen *OBDD* Knotenanzahl liegt, ergibt sich pro Knoten ein zusätzlicher Speicherbedarf von ca. fünf weiteren Bytes. Bei einer Hauptspeichergröße von ca. 100 MByte lassen sich mit *CUDD* somit *OBDDs* mit ca. fünf Millionen Knoten erzeugen.

Eine zentrale Idee in *StaticBdd* ist es, den maximalen Knotenindex auf eine 24 Bitzahl zu beschränken, und den zur Wortbreite 32 freigewordenen Platz zu nutzen, um den Knotenindex, das *garbage collection*- und das *Besucht*–Bit zu speichern. Eine 24 Bitzahl reicht aus, um 16 Millionen Knoten zu adressieren. Somit sinkt der Speicheraufwand in *StaticBdd* im Vergleich zum *CUDD*-Paket im Schnitt von ca. 21 auf ca. 17 Bytes.

Index