Finding Optimal Solutions to Atomix

Falk Hüffner[1], Stefan Edelkamp[2], Henning Fernau[1], and Rolf Niedermeier[1]

[1] Wilhelm-Schickard Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
`{hueffner,fernau,niedermr}@informatik.uni-tuebingen.de`

[2] Institut für Informatik, Universität Freiburg,
Georges-Köhler-Allee 51, D-79110 Freiburg, Fed. Rep. of Germany
`edelkamp@informatik.uni-freiburg.de`

Abstract. We present solutions of benchmark instances to the solitaire computer game Atomix found with different heuristic search methods. The problem is PSPACE-complete. An implementation of the heuristic algorithm A* is presented that needs no priority queue, thereby having very low memory overhead. The limited memory algorithm IDA* is handicapped by the fact that, due to move transpositions, duplicates appear very frequently in the problem space; several schemes of using memory to mitigate this weakness are explored, among those, "partial" schemes which trade memory savings for a small probability of not finding an optimal solution. Even though the underlying search graph is directed, backward search is shown to be viable, since the branching factor can be proven to be the same as for forward search.

1 Introduction

Atomix was invented in 1990 by Günter Krämer and first published by Thalion Software for the popular computer systems of that time. The goal is to assemble a given molecule from atoms (see Fig. 1). The player can select an atom at a time and "push" it towards one of the four directions north, south, west, and east; it will keep on moving until it hits an obstacle or another atom. The game is won when the atoms form the same constellation (the "molecule") as depicted beside the board. A concrete Atomix problem, given by the original atom positions and the goal molecule, is called a *level* of Atomix.

The original game had a time limit and did not count the moves needed; we will instead focus on the analytical aspect and try to minimize the solution length as a goal. Note that we are only interested in *optimal* solutions; in order to just find any solution fast, quite different algorithms would be necessary.

An implementation of this Atomix variation for the X Window System is available as "katomic" from `http://games.kde.org/`. A JavaScript version can be played online at `http://www.sect.mce.hw.ac.uk/~peteri/atomix`.

Our solver program written in C++ is able to solve 17 of the 30 problems from the original Atomix and 18 of the 67 problems from katomic optimally. In an appendix, we list a selection of these findings.

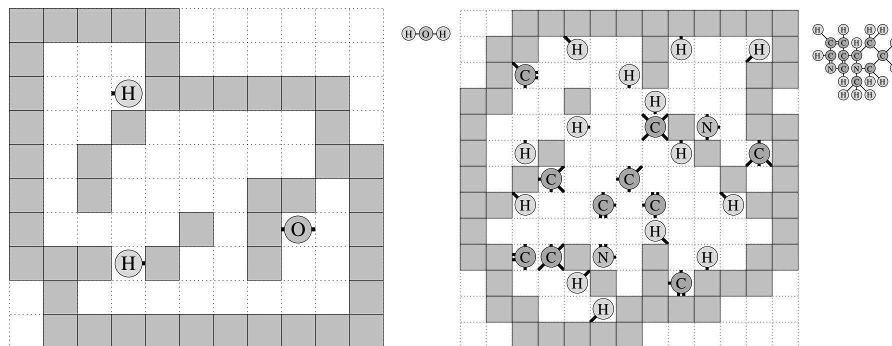

Fig. 1. Two Atomix problems. The left one—which is Atomix 01 in the list of the appendix—can be solved with the following 13 moves, where the atoms are numbered left-to-right in the molecule: 1 down left, 3 left down right up right down left down right, 2 down, 1 right. The right one—which is level number 43 from the "katomic" implementation—illustrates a more complex problem; it takes at least 66 moves to solve.

2 Heuristic Search

Many common problems and, especially, most solitaire puzzles can be formulated as a *state space search* problem: given are a start state, a set of goal states and a set of operators to transform one state into another; wanted is a sequence of operators, also simply called a *move sequence*, that transforms the start state into a goal state and that is of minimal length. A state space can be represented as a graph, with nodes representing states and (directed) edges representing moves. That way, well-known graph algorithms can be applied. To emphasize this aspect, states generated in a state space search are often called "nodes".

For hard combinatorial problems, the use of *heuristics* can often lead to dramatic improvements for a state space search. Many problems would even be unsolvable without them. For a state space search, "heuristic" has a well-defined meaning: an estimate of the moves left from the current state to a goal. Of special interest are *admissible* heuristics: they never overestimate the number of moves. The well-known algorithms A* and IDA* can be proven to always find an optimal solution when using an admissible heuristic. An admissible heuristic judges the "quality" of a state s: if $g(s)$ is the number of moves already applied, and $h(s)$ is the heuristic estimate, then $f(s) := g(s) + h(s)$ is a lower bound on the total number of moves. This number, customarily called the "f-value", can be used in two ways: to guide the search and to reduce the effective depth of the search. The first idea naturally leads to the A* algorithm: "promising" states are examined first. The second is applied in the IDA* algorithm: "hopeless" states are not examined at all.

3 Related Puzzles

The following table compares some search space properties of Atomix to other games. The results are contained in [5, 13, 15].

	24-Puzzle	Sokoban	Atomix
Branching factor	2–4	0–50	12–40
effective	2.3	10	7
Solution Length	80–112	97–674	8–120
typical	100	260	45
Search space size	10^{25}	10^{18}	10^{21}
Graph	Undirected	Directed	Directed

Table 1. Search space properties of some puzzles. The effective branching factor is the number of children of a state, after applying memory-bounded pruning methods (in particular, not utilizing transposition tables; see Sect. 5.3 for the methods applied to Atomix). For Sokoban and Atomix, the numbers are for typical puzzles from the human-made test sets; for Sokoban, those problems are about 20×20 and, for Atomix, about 16×16 squares large.

Due to its close relationships to Atomix (which will become important in the next section), we discuss the 15- and the 24-puzzle as special instances of the $(n^2 - 1)$-puzzle in more details.

The 15-puzzle consists of a square tray of size 4×4 with 15 tiles numbered 1 through 15 and one empty square. A move consists of sliding one tile adjacent to the empty square into the empty space. The goal is to obtain the usual ordering of the numbers on the tiles by some move sequence. The 15-puzzle is likely to be the most thoroughly analyzed puzzle of this kind [15]. It serves as a kind of "fruit fly" for heuristic search. It is easy to implement, has an obvious heuristic with the "Manhattan distance", and not too large a search space. The Manhattan distance heuristic can be calculated by summing up the number of turns it would take for a tile to get to its goal position if it was the only tile in the tray. This is obviously a lower bound on the actual number of turns.

Many search methods developed for the 15-puzzle can be easily adapted for Atomix. One important difference is that the underlying search graph for Atomix is directed; not every move can be undone.

Improved heuristics for the 15-puzzle make it possible to solve even the extended "24-puzzle"-variation [15]. Most of them follow the common theme of examining a sub-problem where only a few tiles are regarded and most are ignored. The "linear conflict heuristic" [9], for example, tries to find pairs of tiles in a row or column which need to pass each other to get to the goal position. In such a case, another two moves can be added to the heuristic given by the Manhattan distance, since one tile will have to move out of the way and back. The work of Culberson and Schaeffer [2] generalizes this idea to "pattern databases": Each possible distribution of the tiles 1–8 on the board is analyzed and solved, yielding a lower bound which is often better than the Manhattan heuristic with

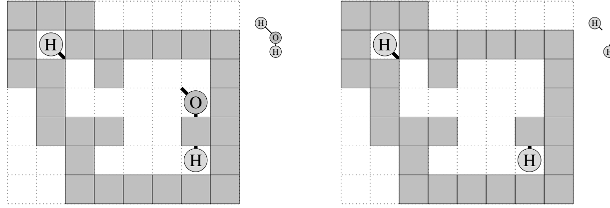

Fig. 2. The left problem can be solved in 13 moves. We cannot get a lower bound by leaving out one atom, as in the right picture; the problem even becomes unsolvable.

the linear conflict heuristic, since there are more tile interactions. The same is done for the other 7 tiles. Unfortunately, these powerful techniques cannot be directly applied to Atomix, since removing atoms from a state does not necessarily make it easier to solve; in fact, it can even become unsolvable, see Fig. 2.

4 Complexity of Atomix

4.1 Complexity of Sliding-Block Puzzles

The time complexity of sliding block puzzles was the subject of intense research in the past. Though seemingly trivial, most variations are at least NP-hard and, some, even PSPACE-complete. The following table shows some results. The table was basically taken from Demaine et al. [3], extended by the category of games where the blocks are pushed by an external agent not represented on the board, into which Atomix falls. The columns mean:

1. Are the moves performed by a robot on the board, or by an outside agent?
2. Can the robot pull as well as push?
3. Does each block occupy a unit square, or may there be larger blocks?
4. Are there fixed blocks, or are all blocks movable?
5. How many blocks can be pushed at a time?
6. Does it suffice to move the robot/a special block to a certain target location, instead of pushing *all* blocks into their goal locations?
7. Will the blocks "keep sliding" when pushed until they hit an obstacle?
8. The dimension of the puzzle: is it 2D or 3D?

Game	1. Robot	2. Pull	3. Blocks	4. Fixed	5. #	6. Path	7. Slide	8. Dim.	9. Complexity
PushPush3D	+	−	unit	−	1	+	+	3D	NP-hard
PushPush	+	−	unit	−	1	+	+	2D	NP-hard
Push-∗	+	−	unit	−	k	−	−	2D	NP-hard
Sokoban+	+	−	1×2	+	2	−	−	2D	PSPACE-compl.
Sokoban	+	−	unit	+	1	−	−	2D	PSPACE-compl. [1]
15-Puzzle	−		unit	−	1	−	−	2D	NP-compl. [17]
Rush Hour	−		1×{2,3}	−	1	+	−	2D	PSPACE-compl.
Atomix	−		unit	+	1	−	+	2D	PSPACE-compl. [12]

4.2 A Formal Definition of Atomix

We will now give a formal definition of an Atomix problem instance (*level*).

Definition 1. *An Atomix problem instance consists of:*

- *A finite set A of so-called* atom types.
- *A game* board $B = \{0, \ldots, w-1\} \times \{0, \ldots, h-1\}$.
- *A bit matrix $O = (O[p] \in \{0,1\} \mid p \in B)$ of size $w \times h$ (the* obstacles*). A* position *is simply a tuple $p = (p_x, p_y) \in B$. A* state *s is defined as a subset of $A \times B$. An element of s is also called an* atom*. Note that the same atom type might appear several times in a state.*
  A position $p = (p_x, p_y)$ is said to be empty *for a state s if $O[p] = 0$ and there is no $a \in A$ with $(a, (p_x, p_y)) \in s$.*
  Positions outside of B areassumed not to be empty.
- *A state S (the* start state*), which satisfies that, for all $(a, p) \in S$, $O[p] = 0$.*
- *A state G (the* goal state*). For the problem to be solvable, for all $(a, p) \in G$, $O[p] = 0$ and there must be a bijection between S and G where each atom in S maps onto an atom in G with the same atom type.*

A direction *(d_x, d_y) is a tuple of x and y offsets, i. e., one of $(0, -1)$, $(1, 0)$, $(0, 1)$ and $(-1, 0)$. A* move *is a tuple of a position p and a direction d. For a state s, a move (p, d) is only legal if there is an atom (a, p) in s, and $(p_x + d_x, p_y + d_y)$ is empty.*

Applying a move (p, d) to a state s will yield another state s' in which every atom has the same position, except the atom (a, p): it will be replaced by (a, p') with $p' = (p_x + \delta d_x, p_y + \delta d_y)$, where $(p_x + \delta' d_x, p_y + \delta' d_y)$ is empty for all $0 < \delta' \leq \delta$, and $(p_x + (\delta + 1)d_x, p_y + (\delta + 1)d_y)$ is not empty. A solution *is a sequence of moves which, incrementally applied to the start state, yields the goal state.*

The main difference between this formal definition and the informal introduction is that the goal positions of the atoms are given explicitly. The reason is that this makes the puzzle both easier to analyze and to implement. Since the number of goal positions is linear in the board size, this difference does not affect the time complexity significantly. Our implementation handles different possible goal positions by imposing a move limit and trying all possible goal positions with that limit, and repeating with an incremented move limit until a solution is found.[1]

4.3 The Hardness of Atomix

Proposition 1. *Atomix on an $n \times n$ board is NP-hard.*

[1] As explained later, this incremental approach is already inherent to IDA*, and can be applied to A* with reasonable overhead.

Proof. Any $(n^2 - 1)$-puzzle instance can be transformed into an Atomix instance by replacing the numbered tiles with atoms of unique atom types. For the $(n^2 - 1)$-puzzle, a legal move consists of sliding a tile into the empty space. In the reduction, those are also the only legal moves, since all atoms not adjacent to the empty square cannot satisfy the move legality condition, and those adjacent to the empty square can only take its place as a move. As shown by Ratner and Warmuth, the $(n^2 - 1)$-puzzle is NP-complete [17], so Atomix is NP-hard. □

Proposition 2. *Atomix on an $n \times n$ board is in PSPACE.*

Proof. A nondeterministic Turing-machine can solve Atomix by repeatedly applying a legal move from the start state encoded on its tape until a goal is reached. The number of possible Atomix states is limited by $n^2!$; hence, the machine can announce that the puzzle is unsolvable after having applied more moves without finding a solution. Since an encoding of an Atomix state needs only polynomial space, it follows that Atomix is in NPSPACE = PSPACE. □

Very recently, Holzer and Schwoon [12] showed by reduction from *non-empty intersection of finite automata* that Atomix is even PSPACE-complete. They also provide a level with an exponentially long optimal solution.

5 Searching the State Space of Atomix

Much progress has been made in the area of heuristic search. This is due to: faster machines with more memory, better heuristics, and better search methods. Of these three, by far, the largest improvements come from better heuristics.

5.1 Heuristics for Atomix

As is often the case, a heuristic for Atomix can be devised by examining a model with relaxed restrictions. We drop the condition that an atom slides as far as possible: it may stop at any closer position. These moves are called *generalized moves.*[2] In order to obtain an easily computable heuristic, we also allow that an atom may also pass through other atoms or share a place with another atom. The goal distance in this model can be summed up for all atoms to yield an admissible heuristic for the original problem.

The following properties are immediate consequences of the definition.

Property 1. The heuristic is admissible.

Property 2. The h-values of child states can only differ from that of the parent state by 0, +1 or −1.

Property 3. The heuristic is *monotone* (consistent), i. e., the f-value of a child state cannot be lower than the f-value of the parent state.

[2] The variant of Atomix which uses generalized moves has an undirected search graph. Atomix with generalized moves on an $n \times n$ board is also NP-hard but is in PSPACE.

Apart from this somewhat obvious heuristic, it proved to be pretty hard to make any improvements. Two ideas were considered, but not implemented due to their limited applicability:

If an atom needs a "stopper" at a certain position to make a turn for each optimal path, but no optimal path of any atom has an intermediate position at the stopper position, h can be incremented by one.

If an atom is alone in a "cave", for some positions, one or two moves can be added to the heuristic (see the example below). A "cave" is an area that contains no goal position and has only one entry; if an atom is alone in there, it cannot use any stoppers unless another atom leaves its optimal path. This heuristic has a greater potential, since it can be added up admissibly for each cave. Unfortunately, only a few levels from our test set contain caves which could yield improved heuristics.

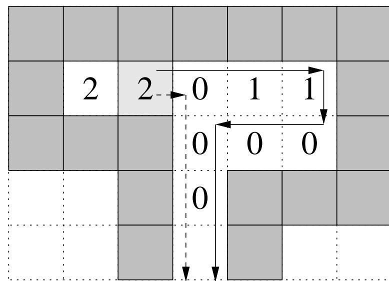

Fig. 3. An example for the "cave"-heuristic: if only one atom is in the cave, the number denoted on its square can be added to the heuristic estimate. For example, an atom on the light grey square has to take the path marked with a solid line, instead of the optimal path of generalized moves marked with a dashed line, which is two moves shorter.

5.2 A*

A* is one of the oldest heuristic search algorithms [10]. It is very time-efficient, but needs an exponential amount of memory. A* remembers all states ever encountered, which is the reason for its exponential space complexity. A priority queue holds all states that have not yet been expanded. It is sorted by the f-value of the states. Nodes are popped from the queue and expanded afterwards. The children are inserted into the queue or discarded if they were already encountered. Sometimes, the same state is reached with a lower g-value; in that case, its entry in the state table has to be updated and it will be re-inserted into the queue. With an admissible heuristic, A* will always find an optimal solution.

The state table is usually implemented as a hash table for fast access and low memory overhead. The priority queue can be implemented with a bucket for each f-value, containing all open states with that f-value. In Sect. 6.2, we present an alternative implementation that only needs the state table and does without a priority queue.

5.3 IDA*

Iterative Deepening A* (IDA*) (see [14]) was the first algorithm that allowed finding optimal solutions to the 15-puzzle. IDA* performs a series of depth-first

searches, with an increasing move limit. The heuristic is used to prune subtrees where it is known that the bound will be exceeded, since the f-value is larger than the bound. Each iteration will visit all nodes encountered in the previous iteration again; but, since the majority of nodes will be generated in the last iteration, this does not affect the time complexity.

IDA* uses no memory except for the stack, so its memory use is linear in the search depth. Also, since it needs no intricate data structures, it can be implemented very efficiently. But of course, this comes at a price: IDA* does not detect *transpositions* in the search graph. If a state is encountered that has already been expanded and dismissed, it will be expanded again, possibly resulting in the re-evaluation of a huge subtree. There are two approaches to lessen this weakness: use of problem specific knowledge and use of memory.

Pruning the Search Space. Several techniques are known for pruning, e. g., predecessor elimination, which disallows to take back moves immediately. For games with undirected underlying graphs like the 15-puzzle, this is an obvious optimization. For Atomix, it can still be applied, since pushing an atom into the opposite direction immediately after a move always yields the same state as pushing it in that direction in the first place.

Move Pruning. When examining a solution move sequence for an Atomix level, one notices that many, though not all moves could be interchanged. Interchanging moves is not possible in four cases, as is explained in Fig. 4.

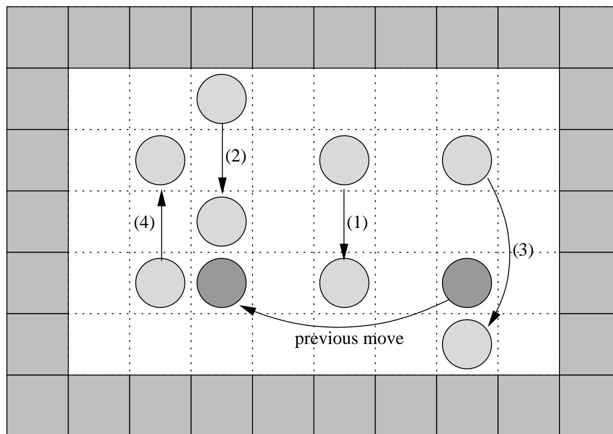

Fig. 4. There are four cases where two moves are dependent, i. e., their order cannot be interchanged: (1) The current atom would have stopped the previously moved atom earlier. (2) The current atom uses the previously moved atom as a stopper. (3) The current atom would stop earlier if the previously moved atom had not been moved. (4) The current atom was the stopper of the previously moved atom.

The idea is to check if a generated move is independent of the previous move (i. e., applying them in reversed order would yield the same state) and, if they are independent, to impose an arbitrary order (the atom with the lower number must move first). This scheme has proven to be very efficient in avoiding transpositions, reducing running time by several orders of magnitudes.

5.4 Partial IDA*

Analogously to the two-player game search, a *transposition table* can be used to avoid re-expanding states [18]. States are inserted into a hash table together with their g-value as they are generated. Then, for each newly generated state, it is looked up whether it has already been expanded with the same or a lower g-value so it can be pruned. If memory was unlimited, this would avoid all possible transpositions. Many schemes have been proposed for proper management of the transposition table with limited memory [4]; our implementation simply refuses to insert states into an exhausted table.

A lot of memory can be saved with *Partial IDA** [6, 7]. This idea originates in the field of *protocol verification*, where the objective is to generate all reachable states and check if they fulfill a certain criterion. A hash table is used to avoid re-expanding states. Just as for a single-agent search, memory is the limiting resource. Therefore, Holzmann suggested *bitstate hashing* [11]: instead of storing the complete state, only a single bit corresponding to the hash value is set, indicating that this state has been visited before. Because of the possibility of hash collisions, states might get pruned erroneously, so this method can give false positives. When applied to IDA*, states on optimal paths could get pruned, so the method looses admissibility, but is still useful to determine upper bounds and likely lower bounds.

For Atomix, initial experiments with Partial IDA* rarely found optimal solutions. The reason is that just knowing a state has been encountered before is not sufficient, because if we encounter it with a lower g-value than previously, it needs to be expanded again. To achieve this, we include g into the hash value and look up with g and $g-1$. This means transpositions with better g will not be found in the table and expanded, as desired. Transpositions with g worse by 2 or more will also not be detected; experiments showed that they are rare and the resulting subtrees are shallow, though.

By probing twice (with g and $g-1$), we increase the likelihood of hash collisions. For example, if we declare the table to be full if every 8th bit is set, we have an effective memory usage of 1 byte per state, and a collision probability of $1-\left(\frac{7}{8}\right)^2 = 23\%$. To improve the collision resistance, one can calculate a second hash value and always set and check two bits, effectively doubling memory usage but lowering collision probability to $1-\left(\frac{63}{64}\right)^2 = 3\%$.

A related scheme with better memory efficiency and collision resistance is *hash compaction* [20]. It utilizes a hash table where, instead of the complete state, only a hash signature is saved. In our implementation, we use 1 byte for the signature, and probe for g and $g-1$. This way, we have a collision probability of $1-\left(\frac{255}{256}\right)^2 = 0.8\%$, so even if there is only a single possible solution of length 30, the probability of finding it is $\left(\left(\frac{255}{256}\right)^2\right)^{30} = 79\%$; and in fact, all 47 solutions found this way were optimal.

Different policies are possible in the case of a hash collision detected by differing signatures. Usual hash table techniques like chaining or open addressing can be applied. We tried a much simpler scheme: the old entry gets overwritten.

This can be seen as a special case of the *t-limited scheme* proposed by Stern and Dill [19] with $t = 1$. One disadvantage of this scheme is that entries will already get overwritten before the table is completely full. Since for the "interesting" (difficult) cases, the state table will fill up soon anyway, this effect is limited.

5.5 Backward Search

Many puzzles are *symmetric*, i. e., the set of children of a state equals the set of possible parents. This is equivalent to the state space graph being undirected. As already mentioned, this is the case for the 15-puzzle, but not for Sokoban or Atomix. For Atomix, it is simple to find all potential parent states, though: they can be found by applying all legal *backward moves*. In a backward move, an atom being pushed may stop moving at any position, but it can only be pushed in a direction if it is adjacent to an obstacle in the *opposite* direction.

Formally defined, a backward move is a triple of a position p, a direction d, and a distance δ. It is legal for a state s if there is an atom (a, p) in s, and $(p_x - d_x, p_y - d_y)$ is *not* empty, and $(p_x + \delta' d_x, p_y + \delta' d_y)$ is empty for all $0 < \delta' \leq \delta$. Applying a backward move is analogous to applying a forward move.

Expanding states for backward Atomix is about as easy as for forward Atomix, and the same heuristic can be used, since the generalized moves from Sect. 5.1 comprise backward moves. Hence, the crucial point is the branching factor.

Lemma 1. *The sum of possible forward moves and the sum of possible backward moves of all states of a level are identical and, therefore, the average number of children for backwards expansion is exactly the same as for forward expansion.*

Proof. We first show the equality for a single atom by structural induction. On a board with no empty squares, the equation is trivially true. We show it also remains true when removing an obstacle. The change in the number of moves depends on the pattern of empty squares around the obstacle being removed; we examine all possible patterns (up to symmetry, and omitting the trivial case of 4 obstacles), as illustrated in Fig. 5, with a, b, c and d being the number of empty squares in each direction.

(a) 3 adjacent obstacles: $1 - b + b + 1 = 1 + 1 = 2$.
(b) 2 adjacent obstacles, where the obstacles are diagonally adjacent:
$1 - b + b + d + 2 - d + 1 = 1 + 2 + 1 = 4$.
(c) 2 adjacent obstacles, where the obstacles are opposite:
$c + 2 - b + 0 - c + b + 2 = 1 + 2 + 1 = 4$.
(d) 1 adjacent obstacle: $c + 2 - b + d + 1 - c + b + 2 - d + 1 = 1 + 3 + 1 + 1 = 6$.
(e) no adjacent obstacles:
$d + 2 - a + c + 2 - b + 0 - c + b + 2 - d + a + 2 = 1 + 1 + 4 + 1 + 1 = 8$.

Now, let us consider the contribution of one atom to the possible moves. Each possible distribution of the other atoms can be considered as a pattern of obstacles. With the observation just made, the sum of possible forward and backward moves is the same when summing up over all possible positions of the

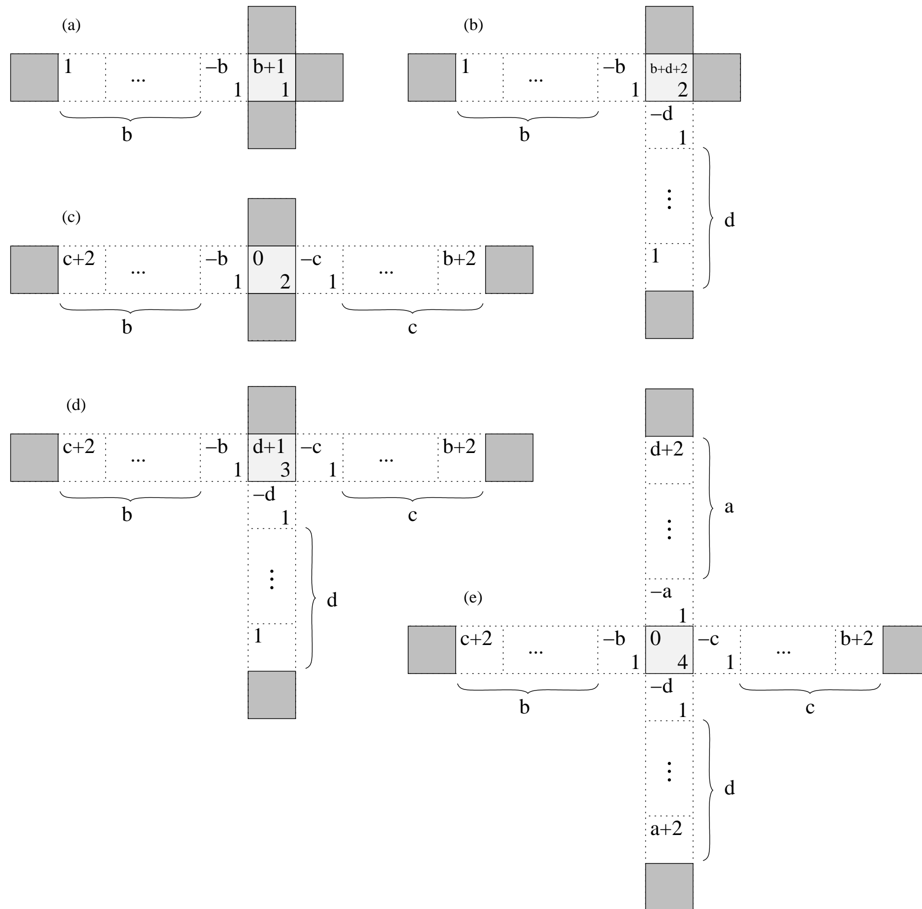

Fig. 5. The light grey obstacle in the center is being removed. The upper left corner of each square denotes the number of backward moves that are lost or gained by this change for an atom on this square. The lower right corner denotes the number of new forward moves. Squares which are skipped in the sketches (denoted by dots) have zero gain with respect to both forward and backward moves.

considered atom; so the sum over all possible distributions of the other atoms is also identical and, since this equality holds for each atom, the lemma is true. □

In practice, the branching factors can differ substantially, since the generated states are not random; the move operators make certain states more likely than others, and states close to the goal where (by convention) all atoms are close together are much more likely. In our experiments, we observed differences up to 30% in forward and backward branching factors.

6 Implementation

6.1 Identical Atoms

The presence of undistinguishable atoms (i. e., atoms with identical atom types) poses problems for an implementation: The heuristic cannot simply perform a table lookup to find a lower bound for an atom, since it is not clear which atom should go to which goal position. To find a good lower bound, a *minimum cost perfect matching* has to be done for each set of identical atoms to find the cheapest assignment of atoms to goal positions. Minimum cost perfect matching for a bipartite graph can be solved using minimum cost augmentation in time quadratic in the number of identical atoms [16].

6.2 A*

An implementation of A* needs the following operations: check if a state has been encountered before and with which g-value, find an open state with optimal f-value, mark an open state as closed, and update the g-value of a saved state to a lower value.

This is usually implemented with a hash table and a priority queue which stores all open states. We will show that if the heuristic is monotone, no priority queue is actually needed: an optimal open state can be found efficiently without any additional data structures. Our algorithm is easy to implement and time and space efficient.

Initially, the available memory is allocated for two tables: the *state table* and the *hash table*. As states are generated, they are appended to the end of the state table; states never get deleted. The states are tagged with an *open*-bit and with the g-value. The hash table stores a pointer into the state table at the position corresponding to the hash value of the state; this allows a quick lookup of states. A linear displacement scheme is used to resolve hash collisions. The monotonicity of the heuristic implies that f_{opt}, the currently optimal f-value of an open state, is also monotone over the run of A*. To find an optimal open state, a linear search on the state table is performed until an open state with $f = f_{\mathrm{opt}}$ is found. The following proposition shows that this can be done efficiently:

Proposition 3. *In A* with a monotone heuristic with a hash table and no additional data structure, a state with optimal f-value can be found in amortized time O(branching factor).*

Proof. To achieve this, we need to ensure that, for each f_{opt}-value, when we reach the end of the state table, we have expanded all states with $f = f_{\text{opt}}$, so we don't have to go through the table again. This can be ensured by not upgrading a state in place if it is re-encountered with lower g, but to append it at the end like new states. States with $f < f_{\text{opt}}$ will never be reopenened [8], so this suffices to ensure the desired property.

Two kinds of states will be skipped because their f-value differs from f_{opt}:

- Closed states with $f < f_{\text{opt}}$. We keep a pointer to the very first open state, so only closed states with $f = f_{\text{opt}} - 1$ or $f = f_{\text{opt}} - 2$ have to be skipped; for any branching factor greater than 1, this can be at most twice as many as states with $f = f_{\text{opt}}$ and, with a higher branching factor, their number even becomes negligible.
- Open states with $f > f_{\text{opt}}$. They must have been generated by states with $f = f_{\text{opt}}$ or $f = f_{\text{opt}} - 1$, so their number is linear in the number of states with $f = f_{\text{opt}}$ and the branching factor. □

Our implementation with this scheme is several times faster than a naïve implementation using the C++ STL `priority_queue` and `set`, which are based on heaps, resp., binary trees, with a memory overhead of about 30 bytes per state. On a Pentium III with 500 MHz, it can generate around a million states per second.

A disadvantage of this scheme is that it is not possible to further discriminate among optimal states. A common idea to speed up A* is to sort among states with equal f-values those closer to the top that are further advanced.

To trade time for memory, the A* implementation works iteratively: similarly to IDA*, an artificial upper bound on the number of moves is applied and, if the f-value of a generated state exceeds this bound, it is pruned. If then the search fails, it is restarted with the bound increased by one. This also allows us to take multiple goal positions into account. Due to the exponential behavior, this slows down the search only by a constant factor.

7 Conclusions

Atomix proved itself to be a challenging puzzle; this is corroborated by the recent PSPACE-completeness proof. The classic algorithms A* and IDA* have been implemented and adapted to the problem domain; we have found optimal solutions for many problems from our benchmark set. Our A* implementation with a single data structure for the *open* and *closed* set can solve "smaller" puzzles very efficiently. With Partial IDA* based on hash compaction, we have presented a memory-bounded scheme that makes excellent use of the available memory and has low runtime overhead; improved bounds on the error probability would be useful, though. Further progress is likely to come from improved heuristics rather than from better search methods, since our current heuristic is rather uninformed. We have shown that while the search graph is directed, the backward branching factor does not differ from the forward branching factor; this makes Atomix an interesting testbed for bidirectional algorithms.

References

1. J. C. Culberson. Sokoban is PSPACE-complete. In E. Lodi, L. Pagli, and N. Santoro, editors, *Proc. FUN-98*, pp. 65–76. Carleton Scientific, Waterloo, 1998.
2. J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
3. E. D. Demaine, M. L. Demaine, and J. O'Rourke. PushPush and Push-1 are NP-hard in 2D. In *Proc. 12th Canadian Conf. Computational Geometry*, pp. 211–219, Fredericton, 2000.
4. J. Eckerle and S. Schuierer. Efficient memory-limited graph search. In Proc. *KI-95*, vol. 981 of *LNCS/LNAI*, pp. 101–112. Springer, Berlin, 1995.
5. S. Edelkamp and R. E. Korf. The branching factor of regular search spaces. In *Proc. AAAI-98/IAAI-98*, pp. 299–304. AAAI Press, Menlo Park, 1998.
6. S. Edelkamp, A. L. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pp. 75–83, 2001.
7. S. Edelkamp and U. Meyer. Theory and Practice of Time-Space Trade-Offs in Memory Limited Search. This volume.
8. S. Edelkamp and S. Schrödl. Localizing A*. In *Proc. AAAI-00/IAAI-00*, pp. 885–890. AAAI Press, Menlo Park, 2000.
9. O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, 1992.
10. P. E. Hart, N. J. Nilsson and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on on Systems Science and Cybernetics*, 4:100–107, 1968.
11. G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proc. International Conf. Protocol Specification, Testing, and Verification*, pp. 339–346, Zürich, 1987. North-Holland, Amsterdam.
12. M. Holzer and S. Schwoon. Assembling Molecules in Atomix is Hard. Technical Report 0101, Institut für Informatik, Technische Universität München, May 2001.
13. A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.
14. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
15. R. E. Korf and L. A. Taylor. Finding optimal solutions to the Twenty-Four Puzzle. In *Proc. AAAI-96/IAAI-96*, pp. 1202–1207. AAAI Press, Menlo Park, 1996.
16. H. W. Kuhn. The Hungarian Method for the Assignment Problem. In *Naval Res. Logist. Quart*, pp. 83–98, 1955.
17. D. Ratner and M. K. Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990.
18. A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.
19. U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, Berichte aus der Informatik, pp. 81–90, 1996. Shaker, Aachen.
20. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. CAV-93*, pp. 59–70. Springer, Berlin, 1993.

A Experimental Results

The experiments were performed on a Pentium III with 500 MHz, utilizing 128 MB of main memory and imposing a time limit of one hour. The source can be found at `http://www-fs.informatik.uni-tuebingen.de/~hueffner`.

Man Best result found by participants of an online game
IDA*-tt IDA* with transposition table
IDA*-r IDA* backward search with transposition table
PIDA* Partial IDA* with hash compaction to 1 byte

Level	Atoms	Goals	Man	A*	IDA*	IDA*-tt	IDA*-r	PIDA*
Atomix 01	3	17		$= 13$	$= 13$	$= 13$	$= 13$	$= 13$
Atomix 02	5	6		$= 21$	$= 21$	$= 21$	$= 21$	$= 21$
Atomix 03	6	4		$= 16$	$= 16$	$= 16$	$= 16$	$= 16$
Atomix 04	6	2		≥ 23	≥ 22	$= 23$	$= 23$	$= 23$
Atomix 05	9	2		≥ 34	≥ 34	≥ 35	≥ 35	≥ 37
Atomix 06	8	4		$= 13$	$= 13$	$= 13$	$= 13$	$= 13$
Atomix 07	9	1		≥ 25	≥ 26	$= 27$	≥ 25	$= 27$
Atomix 09	7	1		$= 20$	$= 20$	$= 20$	$= 20$	$= 20$
Atomix 10	10	2		≥ 28	≥ 28	≥ 28	≥ 27	≥ 30
Atomix 11	5	14		$= 14$	$= 14$	$= 14$	$= 14$	$= 14$
Atomix 12	9	4		$= 14$	$= 14$	$= 14$	$= 14$	$= 14$
Atomix 13	8	1		$= 28$	$= 28$	$= 28$	$= 28$	$= 28$
Atomix 15	12	1		≥ 35	≥ 36	≥ 37	≥ 37	≥ 37
Atomix 16	9	2		≥ 26	≥ 26	≥ 27	≥ 25	≥ 28
Atomix 18	8	4		$= 13$	$= 13$	$= 13$	$= 13$	$= 13$
Atomix 22	8	3		≥ 24	≥ 24	≥ 25	≥ 23	≥ 27
Atomix 23	4	20		$= 10$	$= 10$	$= 10$	$= 10$	$= 10$
Atomix 26	4	17		$= 14$	$= 14$	$= 14$	$= 14$	$= 14$
Atomix 28	10	1		≥ 28	≥ 29	≥ 29	≥ 26	≥ 29
Atomix 29	8	2		$= 22$	$= 22$	$= 22$	$= 22$	$= 22$
Atomix 30	8	4		$= 13$	$= 13$	$= 13$	$= 13$	$= 13$
Unitopia 01	3	41	11	$= 11$	$= 11$	$= 11$	$= 11$	$= 11$
Unitopia 02	4	5	22	$= 22$	$= 22$	$= 22$	$= 22$	$= 22$
Unitopia 03	5	12	16	$= 16$	$= 16$	$= 16$	$= 16$	$= 16$
Unitopia 04	6	5	20	$= 20$	$= 20$	$= 20$	$= 20$	$= 20$
Unitopia 05	6	7	21	$= 20$	$= 20$	$= 20$	$= 20$	$= 20$
Unitopia 06	9	2	33	≥ 29	≥ 30	≥ 30	≥ 30	≥ 31
Unitopia 07	10	1	36	≥ 33	≥ 33	≥ 34	≥ 32	≥ 35
Unitopia 08	7	4	25	$= 23$	$= 23$	$= 23$	$= 23$	$= 23$
Unitopia 10	8	2	41	≥ 36	≥ 36	≥ 37	≥ 38	≥ 40

Time performance. A* runs out of memory usually much before a runtime of one hour and, so, can establish less stringent bounds. The advantage of using a transposition table for IDA* outweighs its runtime overhead and yields better results in all cases. Reverse search performs similar to forward search, as founded by the theroretical findings. Partial IDA* consistently beats IDA* with conventional hash tables because of better memory utilization and less runtime overhead. Note that most of these differences are expected to be more significant if the time limit is increased.