Localizing A^*

Stefan Edelkamp
Institut für Informatik
Am Flughafen 17
D-79110 Freiburg
edelkamp@informatik.uni-freiburg.de

Stefan Schrödl
DaimlerChrysler Research and Technology
1510 Page Mill Road
Palo Alto, CA 94303
schroedl@rtna.daimlerchrysler.com

Abstract

Heuristic search in large problem spaces inherently calls for algorithms capable of running under restricted memory. This question has been investigated in a number of articles. However, in general the efficient usage of two-layered storage systems is not further discussed. Even if hard-disk capacity is sufficient for the problem instance at hand, the limitation of *main memory* may still represent the bottleneck for their practical applications. Since breadth-first and best-first strategies do not exhibit any locality of expansion, standard *virtual memory management* can soon result in thrashing due to excessive page faults.

In this paper we propose a new search algorithm and suitable data structures in order to minimize page faults by a local reordering of the sequence of expansions. We prove its correctness and completeness and evaluate it in a real-world scenario of searching a large road map in a commercial route planning system.

Introduction

Heuristic search algorithms are usually applied to huge problem spaces. Hence, having to cope with memory limitations is an ubiquitous issue in this domain. Since the development of the A^* algorithm (Hart, Nilsson, & Raphael 1968), the main objective has always been to develop methods to regain tractability.

The class of *memory-restricted search algorithms* has been developed under this aim. The framework imposes an absolute upper bound on the total memory the algorithm may use, regardless of the size of the problem space. Most papers do not explicitly distinguish whether this limit refers to disk space or to working memory, but frequently the latter one appears to be implicitly assumed.

IDA* explores the search space by iterative deepening and uses space linear in the solution length, but may revisit the same node again and again (Korf 1985). It does not use additionally available memory. *MREC* switches from A^* to *IDA** if the memory limit is reached (Sen & Bagchi 1989). In contrast, *SMA** (Russell 1992) reassigns the space by dynamically deleting a previously expanded node, propagating up computed f-values to the parents in order to save re-computation as far as possible. Eckerle and Schuierer improve the dynamic re-balancing of the search tree (Eckerle & Schuierer 1995). However, it remains to be shown that these algorithms in general outperform A^* or *IDA** since they impose a large administration overhead. A more recent work employs stochastic node caching and is shown to reduce the number of visited nodes compared to *MREC* (Minura & Ishida 1998).

Even if secondary storage is sufficient, limitation of *working memory* may still represent a bottleneck for practical applications. Modern operating systems provide a general-purpose mechanism for processing data larger than available main memory called *virtual memory.* Transparently to the program, *swapping* moves parts of the data back and forth from disk as needed. Usually, the virtual address space is divided up into units called *pages*; the corresponding equal-sized units in physical memory are called *page frames.* A page table maps the virtual addresses on the page frames and keeps track of their status (loaded/absent). When a *page fault* occurs, i.e., a program tries to use an unmapped page, the CPU is interrupted; the operating system picks a little-used page frame and writes its contents back to the disk. It then fetches the referenced page into the page frame just freed, changes the map, and restarts the trapped instruction. In modern computers memory management is implemented on hardware with a page size commonly fixed at 4096 Byte.

Various *paging strategies* have been explored that aim at minimizing page-faults. Belady has shown that an optimal off-line page exchange strategy deletes the page, which will not be used for the longest time (Belady 1966). Unfortunately, the system, unlike possibly the application program itself, cannot know this in advance. Several different on-line algorithms for the paging problem have been proposed, such as *Last-In-First-Out (LIFO)*, *First-In-First-Out (FIFO)*, *Least-Recently-Used (LRU)*, *Least-Frequently-Used (LFU)*, *Flush-When-Full (FWF)*, etc. (Tanenbaum 1992). Sleator and Tarjan proved that *LRU* is the best on-line algorithm for the problem achieving an optimal competitive ratio equal to the number of pages that fit into main memory (Sleator & Tarjan 1985).

Programmers can reduce the number of page faults by designing data structures that exhibit *memory locality*, such that successive operations tend to access nearby memory addresses. However, sometimes it would be desirable to have more explicit control of secondary memory manipulations. For example, fetching data structures larger than the system page size may require multiple disk operations. A file buffer can be regarded as a kind of *"software" paging* that mimics swapping on a coarser level of granularity. Generally, an application can outperform the operating system's memory management because it is well-informed to predict future memory access.

Particularly for search algorithms, system paging can become the major bottleneck. We experienced this problem when applying A^* to the domain of route planning. Node structures become large, compared to hardware pages; moreover, A^* does not respect locality at all; it explores nodes in the strict order of f values, regardless of their neighborhood, and hence jumps back and forth in a spatially unrelated way for only marginal differences in the estimation value.

In the following we present a new heuristic search algorithm to overcome this lack of locality. In connection with software paging strategies, it can lead to a significant speedup. The idea is to organize the graph structure for spatial locality and to expand spatial local data even if it can lead to a possible non-optimal solution. As a consequence, the algorithm cannot stop with the first solution found, but has to do the additional work of exploring all pending paths. However, the increased number of node expansions can be outweighed by the reduction in the number of page faults.

In the next section, we review traditional A^* and extend it so as to allow for node expansions in arbitrary order. We prove its correctness and completeness, and as a byproduct we fix a minor lack of accuracy in the traditional proof for A^*. Then, we describe a data structure called *Heap-Of-Heaps* that is suitable to accommodate locality and is based on a partitioning of the search space. Finally the algorithm is evaluated within a commercial route planning system.

The Algorithm

We start by characterizing the standard A^* algorithm (Hart, Nilsson, & Raphael 1968) in an unusual but concise way on the basis of Dijkstra's algorithm to find shortest paths in (positively) weighted graphs from a *start node* s to a set of *goal nodes* T (Dijkstra 1959). Dijkstra's algorithm uses a priority queue *Open* maintaining the set of currently reached yet unexplored nodes. If $f(u)$ denotes the total weight of the currently best explored path from s to some node u (also called the *merit* of u), the algorithm always selects a node from *Open* with minimum f value for expansion, updates its successors' f-values, and transfers it to the set *Closed* with established minimum cost path.

Traditional A^* = Dijkstra + Re-weighting

Algorithm A^* accommodates the information of a *heuristic* $h(u)$, which estimates the minimum cost of a path from node u to a goal node in T. It can be cast as a search through a re-weighted graph. More precisely, the edge weights w are replaced by new weights $\hat{w}$ by adding the heuristic difference: $\hat{w}(u,v) = w(u,v) - h(u) + h(v)$. At each instant of time in the re-weighted Dijkstra algorithm, the merit f of a node u is the sum of the new weights along the currently cheapest path explored by the algorithm.

By this transformation, negative weights can be introduced. Nodes that have already been expanded might be encountered on a shorter path. Thus, contrary to Dijkstra's algorithm, A^* deals with them by possibly re-inserting nodes from *Closed* into *Open*.

On every path p from s to u the accumulated weights in the two graph structures differ by $h(s)$ and $h(u)$ only, i.e., $w(p) = \hat{w}(p) - h(u) + h(s)$. Consequently, on every cycle c we have $\hat{w}(c) = w(c) \geq 0$, i.e., the re-weighting cannot lead to negatively weighted cycles so that the problem remains solvable.

Let $\delta(u,v)$ and $\hat{\delta}(u,v)$ denote the least-cost path weights between nodes u and v in the initial resp. re-weighted graphs. The heuristic h is called *consistent* if and only if $\hat{w}(u,v) \geq 0$ for all u and v. It is called *optimistic* if $h(u) \leq \min\{\delta(u,t) | t \in T\} = h^*(u)$. This is equivalent to the condition $\min\{\hat{\delta}(u,t) | t \in T\} \geq 0$.

For convenience, since in the following we are dealing only with the transformed weights, we will write w instead of $\hat{w}$.

Invariance Condition

In each iteration of the A^* algorithm, the element u with minimum f value is chosen from the set *Open* and is inserted into *Closed*. Then the set of successors $\Gamma(u)$ is generated. Each node $v \in \Gamma(u)$ is inspected and *Open* and *Closed* are adjusted according to the following procedure *Improve*.

```
Procedure Improve (Node $u$, Node $v$)
    if ($v \in$ Open)
        if ($f(u) + w(u,v) < f(v)$)
            Open.DecreaseKey($v, f(u) + w(u,v)$)
        else if ($v \in$ Closed)
            if ($f(u) + w(u,v) < f(v)$)
                Closed.Delete($v$)
                Open.Insert($v, f(u) + w(u,v)$)
    else
        Open.Insert($v, f(u) + w(u,v)$)
```

The core of the standard optimality proof of A^* published in AI-literature (Pearl 1985) consists of an invariance stating that while the algorithm is running there is always a node v in the *Open* list on an optimal path with the optimal f-value $f(v) = \delta(s,v)$. In our opinion, this reasoning is true but lacks some formal rigidness: if the child of a node with optimal f-value was already contained in *Closed* (be it with optimal f value), then it wouldn't be reopened and the invariance would be violated. It is part of the proof to show that this situation cannot occur. Thus, we strengthen the invariance condition by requiring the node not to be followed by any *Closed* node on the same optimal solution path.

Invariance I. *Let $p = (s = v_0, \ldots, v_n = t)$ be a least-cost path from the start node s to a a goal node $t \in T$. Application of* Improve *preserves the following invariance: Unless v_n is in* Closed *with $f(v_n) = \delta(s, v_n)$, there is a node v_i in* Open *such that $f(v_i) = \delta(s, v_i)$, and no $j > i$ exists such that v_j is in* Closed *with $f(v_j) = \delta(s, v_j)$.*

Proof: W.l.o.g. let i be maximal among the nodes satisfying (I). We distinguish the following cases:

1. Node u is not on p or $f(u) > \delta(s, u)$. Then node $v_i \neq u$ remains in *Open*. Since no v in $Open \cap p \cap \Gamma(u)$ with $f(v) = \delta(s, v) \leq f(u) + w(u, v)$ is changed and no other node is added to *Closed*, (I) is preserved.
2. Node u is on p and $f(u) = \delta(s, u)$. If $u = v_n$, there is nothing to show.

   First assume $u = v_i$. Then *Improve* will be called for $v = v_{i+1} \in \Gamma(u)$; for all other nodes in $\Gamma(u) \setminus \{v_{i+1}\}$, the argument of case 1 holds. According to (I), if v is in *Closed*, then $f(v) > \delta(s, v)$, and it will be reinserted into *Open* with $f(v) = \delta(s, u) + w(u, v) = \delta(s, v)$. If v is neither in *Open* or *Closed*, it is inserted into *Open* with this merit. Otherwise, the *DecreaseKey* operation will set it to $\delta(s, v)$. In either case, v guarantees the invariance (I).

   Now suppose $u \neq v_i$. By the maximality assumption of i we have $u = v_k$ with $k < i$. If $v = v_i$, no *DecreaseKey* operation can change it because v_i already has optimal merit $f(v) = \delta(s, u) + w(u, v) = \delta(s, v)$. Otherwise, v_i remains in *Open* with unchanged f-value and no other node besides u is inserted into *Closed*; thus, v_i still preserves (I). □

Note that we have not required f to be optimistic. Under this assumption, the *optimality* of A^* is implied as a corollary, i.e., the fact that a solution returned by the algorithm is indeed a shortest one. To see this, suppose that the algorithm terminates the search process with the first node t' in the set of goal nodes T and $f(t')$ is not optimal. Then $f(t') > \delta(s, u) + \min\{\delta(u, t) | t \in T\} \geq \delta(s, u) = f(u)$, since for an optimistic estimate the value $\min\{\delta(u, t) |\ t \in T\}$ is not negative. This contradicts the choice of t'. □

General-Node-Ordering A^*

Move ordering is a search optimization technique which has been explored in depth in the domain of two-player games and single-agent applications. It is well-known that substituting the priority queue by a stack or a FIFO-queue results in a depth-first resp. breadth-first traversal of the problem graph. In this case the *DeleteMin* operation is replaced by *Pop* or *Dequeue*, respectively. In the following we will assume a generic operation *DeleteSome* not imposing any restrictions on the selection criteria. The subsequent section will give an implementation that is allowed to select nodes which are "local" to to previously expanded nodes with respect to the application-dependent storage scheme, even though they do not have a minimum f value.

In contrast to A^*, reaching the first goal node will no longer guarantee optimality of the found solution path. Hence, the algorithm has to continue until the *Open* list runs empty. By storing and updating the current best solution path length as a global lower bound value α, we give an anytime extension to A^*that improves the solution quality over time. The concept can be compared to the linear best first algorithm *Depth-First-Branch-and-Bound* (Korf 1993).

```
Function General-Node-Ordering A*
      Open.Insert(s, h(s))
      $\alpha \leftarrow \infty$
      bestSolution $\leftarrow \emptyset$
      while not (Open.IsEmpty())
            $u \leftarrow$ Open.DeleteSome()
            Closed.Insert(u)
(*)         if ($f(u) > \alpha$) continue
            if ($u \in T \wedge f(u) < \alpha$)
                  $\alpha \leftarrow f(u)$
                  bestSolution $\leftarrow$ retrieved path to $u$
            else $\Gamma(u) \leftarrow$ Expand(u)
                  for all $v$ in $\Gamma(u)$
                        Improve(u, v)
      return bestSolution
```

Theorem 1 *If the heuristic estimate h is* optimistic, General-Node-Ordering A* *is optimal.*

Proof: Upon termination, each node inserted into *Open* must have been selected at least once. Suppose that invariance (I) is preserved in each loop, i.e., that there is always a node v in the *Open* list on an optimal path with $f(v) = \delta(s, v)$. Thus the algorithm cannot terminate without eventually selecting the goal node on this path, and since by definition it is not more expensive than any found solution path and *bestSolution* maintains the currently shortest path, an optimal solution will be returned. It remains to show that the invariance (I) holds in each iteration. If the extracted node u is not equal to v there is nothing to show. Otherwise $f(u) = \delta(s, u)$. The bound α denotes the currently best solution length. If $f(u) \leq \alpha$ the condition in (*) is not fulfilled and no pruning takes place. On the other hand $f(u) > \alpha$ leads to a contradiction since $\alpha \geq \delta(s, u) + \min\{\delta(u, t) | t \in T\} \geq \delta(s, u) = f(u)$ (the latter inequality is justified by h being optimistic). □

Theorem 2 *Algorithm* General-Node-Ordering A* *is complete, i.e., terminates on finite graphs.*

Proof: For each successor generation, *General-Node-Ordering A^** adds new links to its traversal tree. Moreover, the algorithm only reopens a node in *Closed* when it finds a *strictly* cheaper path to it and, as said above, re-weighting of positively weighted graphs keeps weights of cycles positive. Hence, the algorithm considers at most the number of acyclic path of the underlying finite graph. This number is finite and, therefore, the algorithm terminates. □

The Heap-Of-Heaps Data Structures

Let us briefly review the usual A^* implementation in terms of data structures. The set *Open* is realized as a priority queue (heap) supporting the operations *IsEmpty*, *Min*, *Insert*, *DecreaseKey* *DeleteMin*. The membership tests $v \in \mathit{Open}$ resp. $v \in \mathit{Closed}$ in procedure *Improve* are implemented using a hash table T. This makes explicit storage of the *Closed* set obsolete, since it is equal to $T \setminus \mathit{Open}$.

For large node structures, it is inefficient to move them physically around; rather, they are maintained in an auxiliary data structure D containing all graph information. D can also contain the links related to the heap and to the hashing chains maximizing *memory locality* with respect to node operations. If the graph is entirely stored, the hash table collapses with D. In some cases there is even no other option than explicit storage, e.g. in the domain of route planning.

Our approach to achieve memory locality is to find a suitable partition of the search space and of all associated data structures into a set of (software) pages $P_1, \ldots, P_k$. We assume a function $\phi : \mathit{Node} \to \{1, \ldots, k\}$ which maps each node to the corresponding page it is contained in.

The data structure *Heap-Of-Heaps* represents the *Open* set. It consists of a collection of k priority queues $H_1, \ldots, H_k$, one for each page. At any instant, one of the heaps, H_{active}, is designated as being *active*. One additional priority queue $\mathcal{H}$ keeps track of the root nodes of all H_i with $i \neq \mathit{active}$; It is used to quickly find the overall minimum across all of these heaps.

The following operations are delegated to the member priority queues H_i in the straightforward way. Whenever necessary, $\mathcal{H}$ is updated accordingly.

```
Function $\mathit{IsEmpty}()$
    return $\bigwedge_{i=1}^{k} H_i.\mathit{IsEmpty}()$
```

```
Procedure $\mathit{Insert}(\mathit{Node}\ u,\ \mathit{Merit}\ f(u))$
    if $(\phi(u) \neq \mathit{active}\ \wedge\ f(u) < f(H_{\phi(u)}.\mathit{Min}()))$
        $\mathcal{H}.\mathit{DecreaseKey}(H_{\phi(u)}, f(u))$
    $H_{\phi(u)}.\mathit{Insert}(u,\ f(u))$
```

```
Procedure $\mathit{DecreaseKey}(\mathit{Node}\ u,\ \mathit{Merit}\ f(u))$
    if $(\phi(u) \neq \mathit{active}\ \wedge\ f(u) < f(H_{\phi(u)}.\mathit{Min}()))$
        $\mathcal{H}.\mathit{DecreaseKey}(H_{\phi(u)}, f(u))$
    $H_{\phi(u)}.\mathit{DecreaseKey}(u, f(u))$
```

Operation *DeleteSome* performs *DeleteMin* on the active heap.

```
Function $\mathit{DeleteSome}()$
    $\mathit{CheckActive}()$
    return $H_{\mathit{active}}.\mathit{DeleteMin}()$
```

The *Insert* and *DecreaseKey* operations can affect all heaps. However, the hope is that the number of adjacent pages of the active page is small and that they are already in memory or have to be loaded only once; all other pages and priority queues remain unchanged and do not have to reside in main memory.

As the aim is to minimize the number of switches between pages, the algorithm favors the *active* page by continuing to expand its nodes although the minimum f value might already exceed the minimum of all remaining priority queues. There are two control parameters: An *activeness bonus* Δ and an estimate Λ for the cost of an optimum solution.

```
Procedure $\mathit{CheckActive}()$
    if $(H_{\mathit{active}}.\mathit{IsEmpty}() \vee$
            $(f(H_{\mathit{active}}.\mathit{Min}()) - f(\mathcal{H}.\mathit{Min}().\mathit{Min}()) > \Delta$
                $\wedge\ f(H_{\mathit{active}}.\mathit{Min}()) > \Lambda))$
            $\mathcal{H}.\mathit{Insert}(H_{\mathit{active}}, f(H_{\mathit{active}}.\mathit{Min}()))$
            $H_{\mathit{active}} \leftarrow \mathcal{H}.\mathit{Min}()$
            $\mathcal{H}.\mathit{Remove}(H_{\mathit{active}})$
```

If the minimum f-value of the active heap is larger than that of the remaining heaps plus the *activeness bonus* Δ, the algorithm may switch to the priority queue satisfying the minimum root f value. Thus, Δ discourages page switches by determining the proportion of a page to be explored. As it increases to large values, in the limit each activated page is searched to completion.

However the active page still remains valid, unless Λ is exceeded. The rationale behind this second heuristic is that one can often provide a heuristic for the total least cost path which is, on the average, more accurate than that obtained from h, but which might be overestimating in some cases.

With this implementation, algorithm *General-Node-Ordering* A^* itself remains almost unchanged, i.e., the data structure and page handling is transparent to the algorithm. Traditional A^* arises as a special case for $\Delta = 0$ and $\Lambda < h^*(s)$, where $h^*(s)$ denotes the actual minimum cost between the start node and a goal node.

Optimality is guaranteed, since we leave the heuristic estimates unaffected by the heap prioritization scheme, and since each node inserted into the *Heap-of-Heaps* structure is eventually returned by *DeleteMin*.

Experiments

In our experiments we incorporated our algorithm into a commercially available route planning system running on Windows platforms. The system covers an area of approximately 800 × 400 km at a high level of detail, and comprises approximately 910,000 nodes (road junctions) linked by 2,500,000 edges (road elements). The entire graph structure, together with the members needed for the search algorithm, results in a total memory size of 40 MByte, which already exceeds the advertized minimum main memory hardware requirement of 32 MByte.

For long-distance routes, conventional A^* expands the nodes in a spatially uncorrelated way, jumping to a node as far apart as some 100 km, but possibly returning to the successor of the previous one in the next step. Therefore, the working set gets extremely large, and the virtual memory management of the operating system leads to excessive paging and is the main burden on the computation time.

As a remedy, we achieve memory locality of the search algorithm by exploiting the underlying spatial relation of connected nodes. Nodes are geographically

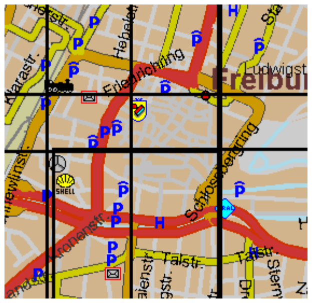

Figure 1: The granularity of the partition (lines indicate bounding rectangles of pages).

sorted according to their coordinates in such a way that neighboring nodes also tend to appear close to each other. A page consists of a constant number of successive nodes (together with the outgoing edges) according to this order. Thus, pages in densely populated regions tend to cover a smaller area than those representing rural regions. For not too small sizes, the connectivity within a page will be high, and only a comparably low fraction of road elements cross the boundaries to adjacent pages. Fig. 1 shows some bounding rectangles of nodes belonging to the same page.

There are three parameters controlling the behavior of the algorithm with respect to secondary memory, the algorithm parameters Δ and Λ, and the (software) page size. The latter one should be adjusted so that the active page and its adjacent pages together roughly fit into available main memory. The optimum solution estimate Λ is obtained by calculating the Euclidean distance between the start and the goal and adding a fixed percentage.

Fig. 2 opposes the number of page faults to the number of node expansions for varying page size and Δ. We observe that the rapid decrease of page faults compensates the increase of expansions (note the logarithmic scale). Using an activeness bonus of about 2 km suffices to decrease the value by more than one magnitude for all page sizes. At the same time the number of expanded nodes increases by less than ten percent.

Fig. 3 depicts the corresponding influence of Λ. In this case the reduction of page faults by more than a magnitude can be achieved by investing less than 50 percent extra node expansions for Λ equal to 1.25 times the Euclidean distance. The effect is almost independent of the page size.

Unfortunately, the convincing decrease in page faults did not translate proportionally to execution time; the maximum reduction amounted to about 30 percent. We suspect that the reason is that we could not totally control the operating system's hardware paging still working besides and on top of our software paging technique. Hence, more inquiry into the platform-dependent implementation is still required.

We conclude that there is a trade-off between the growth of node expansions and the savings of page

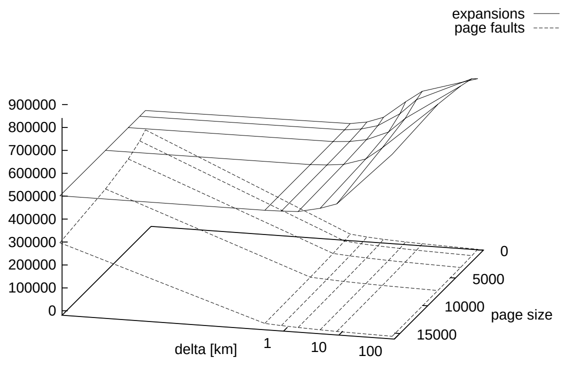

Figure 2: Number of page-faults and node expansions for varying page size and activeness bonus Δ.

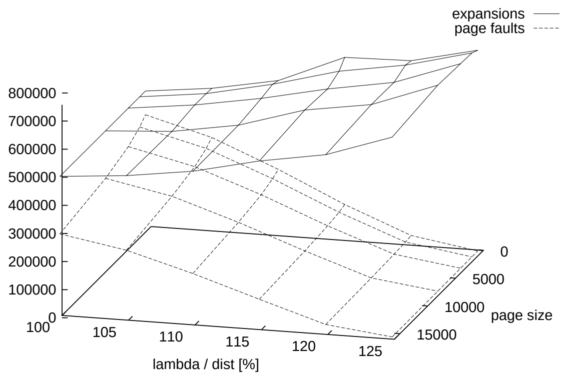

Figure 3: Number of page-faults and node expansions for varying page size and the ratio (in percent) of solution length approximation Λ and the Euclidean distance *dist* between start and goal.

faults that has to be resolved by tuning the parameters to improve the overall efficiency for best performance.

Related Work

A couple of dynamic data structures have been proposed which take into account secondary memory structures. Major representatives are tree structured indices, such as B-Trees invented by Bayer and McCreight (Bayer & McCreight 1970) and dynamic hashing variants (Larson 1978), such as extendible hashing (Fagin *et al.* 1970) and virtual hashing (Litwin 1978). External sorting algorithms (Knuth 1973) are special-tailored for handling sequences on disk storage that do not fit into working memory. An extension for the LEDA (Mehlhorn & Näher 1999) C++ library project to secondary storage systems, LEDA-SM for short, is being developed by Crauser and Mehlhorn at

MPI/Saarbrücken.

One currently deeply investigated area in which the advantage of memory locality pays off is the breadth-first synthesis of binary decision diagrams (BDDs) (Bryant 1985). The idea is to construct the diagram structure in a level-wise traversal (Hu & Dill 1993). Since there is a trade-off between low memory overhead and memory access locality, hybrid approaches based on context switches are currently being explored (Yang *et al.* 1998).

Since each page is explored independently, the algorithms easily lends itself to parallelization by allowing for more than one active page at a time. In fact, a commonly used method for duplicate pruning uses a hash function similar to ϕ defined above to associate with each node of the search space a distinct subspace with a dedicated processor. In (Mahapatra & Dutt 1997), the notion of locality is important to reduce communication between processors and it is implemented as the neighborhood on a hypercube.

There are some related approaches to the re-weighting technique used in our optimality proof. Searching negatively weighted graphs has been intensively studied in literature, cf (Cormen, Leiserson, & Rivest 1990). An $O(|V||E|)$ algorithm for the single-source shortest path problem has been separately proposed by Bellman and Ford. The algorithm has been improved by Yen. The all-pair shortest path problem has been solved by Floyd based on a theorem of Warshall. It has been extended by Johnson for sparse and possibly negatively weighted graphs by re-weighting. All these algorithms do not apply to the scenario of implicitly given graphs with additional heuristic information.

Conclusion

We have presented an approach to relax the order of node expansions in traditional A^*. Its admissibility is shown using a refined invariance condition based on Dijkstra's algorithm and re-weighted graphs. The re-ordering is used to make the search algorithm take into account memory locality for the price of an increased number of expansions. However, this is offset by the minimization of secondary memory access in a two-layered storage system, which is a major bottleneck for the traditional algorithm. To this end, the data structure *Heap-of-Heaps* has been developed which partitions the underlying graph into pages; two heuristic threshold values discourage page switches and can be tuned for best performance. The count of page switches from an evaluation within a commercially available route planning system supports this view.

References

Bayer, R., and McCreight, E. 1970. Organization and maintenanace of large ordered indexes. *Acta Informatica* 13(7):427–436.

Belady, L. 1966. A study of replacement algorithms for virtual storage computers. *IBM Syst. J* 5:18–101.

Bryant, R. E. 1985. Symbolic manipulation of boolean functions using a graphical representation. In *Design Automation*, 688–694.

Cormen, T. H.; Leiserson, C. E.; and Rivest, R. L. 1990. *Introduction to Algorithms.* The MIT Press.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Eckerle, J., and Schuierer, S. 1995. Efficient memory-limited graph search. In *KI*, 101–112.

Fagin, R.; Nievergelt, J.; Pippenger, N.; and Strong, H. R. 1970. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst.* 4(3):315–344.

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.

Hu, A. J., and Dill, D. L. 1993. Reducing BDD size by exploiting functional dependencies. In *Design Automation*, 266–271.

Knuth, D. E. 1973. *The Art of Computer Programming, Vol 3: Sorting and Searching.* Addison-Wesley.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Larson, P.-A. 1978. Dynamic hashing. *BIT* 18(2):184–201.

Litwin, W. 1978. Virtual hashing: a dynamically changing hashing. In *Very Large Databases*, 517–523.

Mahapatra, N. R., and Dutt, S. 1997. Scalable global and local hashing strategies for duplicate pruning in parallel A* graph search. *IEEE Transactions on Parallel and Distributed Systems* 8(7):738–756.

Mehlhorn, K., and Näher, S. 1999. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press.

Minura, T., and Ishida, T. 1998. Stochastic node caching for efficient memory-bounded search. In *AAAI*, 450–459.

Pearl, J. 1985. *Heuristics.* Addison-Wesley.

Russell, S. 1992. Efficient memory-bounded search methods. In *ECAI-92*, 1–5.

Sen, A. K., and Bagchi, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, 297–302.

Sleator, D., and Tarjan, R. 1985. Amortized efficiency of list update and paging rules. *Communications of the ACM* 28:202–208.

Tanenbaum, A. S. 1992. *Modern Operating Systems.* New Jersey: Prentice Hall.

Yang, B.; Chen, Y.-A.; Bryang, R. E.; and Hallaron, D. R. 1998. Space- and time-efficient BDD construction via working set control. In *Asia and South Pacific Design Automation*, 423–432.