External A*

Stefan Edelkamp stefan.edelkamp@cs.uni-dortmund.de
Baroper Straße 301
Fachbereich Informatik
Universität Dortmund, Germany

Shahid Jabbar shahid.jabbar@cs.uni-dortmund.de
Baroper Straße 301
Fachbereich Informatik
Universität Dortmund, Germany

Stefan Schroedl schroedl@rtna.daimlerchrysler.com
DaimlerChrysler Research and Technology Center
1510 Page Mill Road
Palo Alto, CA 94304

Abstract

In this paper we study *External A**, a variant of the internal A* algorithm that employs external memory. The approach applies to implicit, undirected, unweighted state space problem graphs with consistent estimates. The complexity of the External algorithm is almost linear in external sorting time and accumulates to $O(sort(|E|) + scan(|V|))$ I/Os, where V and E are the set of nodes and edges in the explored portion of the state space graph. Given that delayed duplicate elimination has to be performed, the established bound is I/O optimal. In difference to the internal design in the construction we exploit memory locality to allow block rather than random access. The algorithmic design refers to external shortest path search in explicit graphs and extends the strategy of delayed duplicate detection recently suggested for breadth-first search to best-first search. We conduct experiments with sliding-tile puzzle instances.

1. Introduction

Often search spaces are so big that they don't fit into main memory. In this case, during the algorithm only a part of the graph can be processed at a time; the remainder is stored on a disk. However, hard disk operations are about a $10^5 - 10^6$ times slower than main memory accesses. Moreover, according to recent estimates, technological progress yields about annual rates of 40-60 percent increase in processor speeds, while disk transfers only improve by seven to ten percent. This growing disparity has led to a growing attention to the design of *I/O-efficient algorithms* in recent years.

Most modern operating systems hide secondary memory accesses from the programmer, but offer one consistent address space of *virtual memory* that can be larger than internal memory. When the program is executed, virtual addresses are translated into physical addresses. Only those portions of the program currently needed for the execution are copied into main memory. Application programs may exhibit *locality* in their pattern of memory accesses: i.e., data residing in a few pages are repeatedly referenced for a while, before the

program shifts attention to another working set. Caching and pre-fetching heuristics have been developed to reduce the number of page faults (the referenced page does not reside in the cache and has to be loaded from a higher memory level). By their nature, however, these methods are general-purpose and can not always take full advantage of locality inherent in algorithms. Algorithms that explicitly manage the memory hierarchy can lead to substantial speedups, since they are more informed to predict and adjust future memory access.

Different variants of breadth-first and depth-first traversal of external graphs have been proposed earlier (Meyer, Sanders, & Sibeyn, 2003; Chiang, Goodrich, Grove, Tamasia, Vengroff, & Vitter, 1995). In this paper we address *A* search* on secondary memory: in problems where we try to find the shortest path to a designated goal state, it has been shown that the incorporation of a heuristic estimate (lower bound) for the remaining distance of a state can significantly reduce the number of nodes that need to be explored (Hart, Nilsson, & Raphael, 1968).

The remainder of the paper is organized as follows. First we introduce the most widely used computation model, which counts I/Os in terms of transfers of blocks of records of fixed size to and from secondary memory. We describe some basic external-memory algorithms and some data structures relevant to graph search. Then we turn to the subject of external graph search that is concerned with breadth-first search in explicit graphs stored on disk. Korf's *delayed duplicate detection* algorithm (Korf, 2003) adapts Munagala and Ranade's algorithm (Munagala & Ranade, 2001) for the case of implicit graphs, and is presented next. Then we invent *External A**, which extends delayed duplicate detection to heuristic search. Internal and I/O complexities are derived followed by an optimality argument based on a lower bound for delayed duplicate detection. Finally, we address related work and draw conclusions.

2. Model of Computation

The commonly used model for comparing the performance of external algorithms consists of a single processor, a small internal memory that can hold up to M data items, and an unlimited secondary memory. The size of the input problem (in terms of the number of records) is abbreviated by N. Moreover, the *block size* B governs the bandwidth of memory transfers. It is often convenient to refer to these parameters in terms of blocks, so we define $m = M/B$ and $n = N/B$. It is usually assumed that at the beginning of the algorithm, the input data is stored in contiguous block on external memory, and the same must hold for the output. Only the number of block read and writes are counted, computations in internal memory do not incur any cost. An extension of the model considers D disks that can be accessed simultaneously. When using disks in parallel, the technique of *disk striping* can be employed to essentially increase the block size by a factor of D. Successive blocks are distributed across different disks.

Formally, this means if we enumerate the records from zero, the i-th block of the j-th disk contains items number $(iDB+jB)$ through $(iDB+(j-1)B-1)$. Usually, it is assumed that $M < N$ and $DB < M/2$. We can distinguish two general approaches to external memory algorithms: either we can devise algorithms to solve specific computational problems while explicitly controlling secondary memory access; or, we can develop general-purpose external-

Operation	Complexity	Optimality achieved by
scan(N)	$\Theta(\frac{N}{DB}) = \Theta(\frac{n}{D})$	Trivial sequential access
sort(N)	$\Theta(\frac{N}{DB} \log_{M/B} \frac{N}{B}) = \Theta(\frac{n}{D} \log_m n)$	*Merge* or *Distribution Sort*

Table 1: Primitives of external-memory algorithms.

memory data structures, such as stacks, queues, search trees, priority queues, and so on, and then use them in algorithms that are similar to their internal-memory counterparts.

3. Basic Primitives of I/O-Efficient Algorithms

It is often convenient to express the complexity of external-memory algorithms using a number of frequently occurring primitive operations. These primitives, together with their complexities, are summarized in Table 1. The simplest operation is *scanning*, which means reading a stream of records stored consecutively on secondary memory. In this case, it is trivial to exploit disk- and block-parallelism. The number of I/Os is $\Theta(\frac{N}{DB}) = \Theta(\frac{n}{D})$.

Sorting is a fundamental problem that arises in almost all areas of computer science. Besides the classical uses, it is often useful to eliminate I/O accesses in external-memory algorithms. The proposed algorithms fall into two categories: those based on the *merging* paradigm, and those based on the *distribution* paradigm.

External *mergesort* converts the input into a number of elementary sorted sequences of length M using internal-memory sorting. Subsequently, a merging step is applied repeatedly until only one run remains. A set of k sequences $S_1, \ldots, S_k$ can be merged into one run with $O(N)$ operations by reading each sequence in blockwise manner. In internal memory, k cursors p_k are maintained for each of the sequences; moreover, it contains one buffer block for each run, and one output buffer. Among the elements pointed to by the p_k, one with the smallest key, say p_i, is selected; the element is copied to the output buffer, and p_i is incremented. Whenever the output buffer reaches the block size B, it is written to disk, and emptied; similarly, whenever a cached block for an input sequences has been fully read, it is replaced with the next block of the run in external memory. When using one internal buffer block per sequence, and one output buffer, each merging phase uses $O(N/B)$ operations. The best result is achieved when k is chosen as big as possible, i.e., $k = M/B$. Then sorting can be accomplished in $O(\log_{M/B} \frac{N}{B})$ phases, resulting in the overall optimal complexity.

On the other hand, algorithms based on the *distribution* paradigm partition the input data into disjoint sets S_i, $1 \le i \le k$, such that the key of each element in S_i is smaller than that of any element in S_j, if $i < j$. In order to produce this partition, a set of *splitters* $-\infty = s_0 < s_1 < \ldots < s_k < s_{k+1} = \infty$ is chosen, and S_i is defined to be the the subset of elements $x \in S$ with $s_i < x \le s_{i+1}$. The splitting can be done I/O-efficiently by streaming the input data through an input buffer, and also using an output buffer. Then, each subset S_i is recursively sorted, unless its size allows sorting in internal memory. The final output is produced by concatenating all of the elementary sorted subsequences. Optimality can be achieved by a good choice of splitters, i.e., such that $|S_i| = O(N/k)$. It has been proposed to calculate the splitters in linear time based on the classical internal-memory *selection* algorithm to find the k-smallest element. We note that, while we will only be concerned

with the case of a single disk ($D = 1$), it is much more challenging to make optimal use of multiple disks with $\Theta(\frac{N}{DB} \log_{M/B} \frac{N}{B}) = \Theta(\frac{n}{D} \log_m n)$ I/Os. Simple disk striping does not lead to optimal external sorting. It has to be ensured that each read operation brings in $\Omega(D)$ blocks, and each write operation must store $\Omega(D)$ blocks on disk. For distribution sort, the buckets have to be hashed to the disks almost uniformly. This can be achieved using a randomized scheme.

4. External BFS

Since heuristic search algorithms are often applied to huge problem spaces, it is an ubiquitous issue in this domain to cope with internal memory limitations. A variety of *memory-restricted search algorithms* have been developed to work under this constraint. A widely used algorithm is Korf's *iterative deepening A* (IDA*)* algorithm, which requires only space linear in the solution length (Korf, 1985), in exchange for an overhead in computation time due to repeated expansion. Various schemes have been proposed to reduce this overhead by flexibly utilizing additionally available memory. The common framework usually imposes a fixed upper limit on the total memory the program may use, regardless of the size of the problem space. Most of these papers do not explicitly distinguish whether this limit refers to internal memory or to disk space, but frequently the latter one appears to be implicitly assumed. On the contrary, in this section we introduce techniques that explicitly manage a two-level memory hierarchy.

4.1 Explicit Graphs

Under *external graph algorithms*, we understand algorithms that can solve the *depth-first search (DFS)*, *breadth-first search (BFS)*, or *single-source shortest path (SSSP)* problem for explicitly specified directed or undirected graphs that are too large to fit in main memory. We can distinguish between assigning (BFS or DFS) numbers to nodes, assigning BFS levels to nodes, or computing the (BFS or DFS) tree edges. However, for BFS in undirected graphs it can be shown that all these formulations are reducible to each other up to an edge-list sorting in $O(sort(|E|)$ I/O operations.

The input is usually assumed to be an unsorted edge list stored contiguously on disk. However, frequently algorithms assume an *adjacency list representation*, which consists of two arrays, one which contains all edges sorted by the start node, and one array of size $|V|$ which stores, for each vertex, its out-degree and offset into the first array. A preprocessing step can accomplish this conversion in time $O(\frac{|E|}{|V|} sort(|V|))$.

Recall the standard internal-memory BFS algorithm: it visits each vertex $v \in V$ of the input graph $G = (V, E)$ in a one-by-one fashion, as stored in a FIFO queue Q. After a vertex v is extracted, its adjacency list (the sets of neighbors in G) is examined, and those of them that haven't been visited so far are inserted into Q in turn. Naively running the standard internal-BFS algorithm in the same way in external memory will result in $\Theta(|V|)$ I/Os for unstructured accesses to the adjacency lists, and $\Theta(|E|)$ I/Os for finding out whether neighboring nodes have already been visited. The latter task is considerably easier for *undirected graphs*, since duplicates are constrained to be located in adjacent levels. Next section presents an algorithm for this case; then, a later improvement of the algorithm is described that helps to reduce the former complexity.

```
Procedure External Breadth-First-Search
    Open(−1) ← Open(−2) ← ∅; U ← V
    i ← 0
    while (Open(i − 1) ≠ ∅ ∨ U ≠ ∅)
        if (Open(i − 1) = ∅)
            Open(i) ← {x}, where x ∈ U
        else
            A(i) ← N(Open(i − 1))
            A′(i) ← remove duplicates from A(i)
            Open(i) ← A′(i) \ (Open(i − 1) ∪ Open(i − 2))
        foreach v ∈ Open(i)
            U ← U \ {v}
        i ← i + 1
```

Figure 1: External BFS by Munagala and Ranade

The algorithm of *Munagala and Ranade* (Munagala & Ranade, 2001) improves on the latter complexity for the case of undirected graphs, in which duplicates are constrained to be located in adjacent levels. The algorithm builds $Open(i)$ from $Open(i-1)$ as follows: Let $A(i) = N(Open(i-1))$ be the multi-set of neighbor vertices of nodes in $Open(i-1)$; $A(i)$ is created by concatenating all adjacency lists of nodes in $Open(i-1)$. Since after the preprocessing step the graph is stored in adjacency-list representation, this takes $O(|Open(i-1)|+|N(Open(i-1))|/B)$ I/Os. Then the algorithm removes duplicates by external sorting followed by an external scan. Hence, duplicate elimination takes $O(sort(A(i)))$ I/Os. Since the resulting list $A'(i)$ is still sorted, filtering out the nodes already contained in the sorted lists $Open(i-1)$ or $Open(i-2)$ is possible by parallel scanning, therefore this step can be done using $O(sort(|N(Open(i-1))|) + scan(|Open(i-1)| + |Open(i-2)|))$ I/Os. This completes the generation of $Open(i)$. The algorithm can record the nodes' BFS-level in additional $O(|V|)$ time using an external array. Figure 1 provides the implementation of the algorithm of Munagala and Ranade in pseudo-code. A doubly-linked list U maintains all unvisited nodes, which is necessary when the graph is not completely connected. Since $\sum_i |N(Open(i))| = O(|E|)$ and $\sum_i |Open(i)| = O(|V|)$, the execution of external BFS requires $O(|V| + sort(|E|))$ time, where $O(|V|)$ is due to the external representation of the graph and the initial reconfiguration time to enable efficient successor generation.

The bottleneck of the algorithm are the $O(|V|)$ unstructured accesses to adjacency lists. The refined algorithm (Mehlhorn & Meyer, 2002) consists of a preprocessing and a BFS phase, arriving at a complexity of $O(\sqrt{|V| \cdot scan(|V| + |E|)} + sort(|V| + |E|))$ I/Os.

The preprocessing stage partitions the graph into K disjoint subgraphs $\{S_i \mid 1 \le i \le K\}$ with small internal shortest-path distances; the adjacency lists are accordingly partitioned into consecutively stored sets $\{F_i | 1 \le i \le K\}$ as well. The partitions are created by choosing *seed nodes* independently with uniform probability μ. Then K BFS are run in parallel, starting from the seed nodes, until all nodes of the graph have been assigned to a subgraph. In each round, the active adjacency lists of nodes lying on the boundary of their partition are

scanned; the requested target nodes are labelled with the partition identifier, and are sorted (Ties between partitions are arbitrarily broken). Then, a parallel scan of the sorted requests and the graph representation can extract the unvisited part of the graph, as well as label the new boundary nodes and generate the active adjacency lists for the next round. The expected I/O-bound for the graph partitioning is $O((|V|+|E|)/\mu DB + sort(|V|+|E|))$; the expected shortest-path distance between any two nodes within a subgraph is $O(\frac{1}{\mu})$. The main idea of the second phase is to replace the node-wise access to adjacency lists by a scanning operation on a file H that contains all F_i in sorted order such that the current BFS level has at least one node in S_i. All subgraph adjacency lists in F_i are merged with H completely, not node-by node. Since the shortest path within a partition is of order $O(\frac{1}{\mu})$, each F_i stays in H accordingly for at most $O(\frac{1}{\mu})$ levels. The second phase uses $O(\mu|V| + (|V|+|E|)/\mu DB + sort(|V|+|E|))$ I/Os in total; choosing $\mu = \min\{1, \sqrt{(|V|+|E|)/\mu DB}\}$, we arrive at a complexity of $O(\sqrt{|V| \cdot scan(|V|+|E|)} + sort(|V|+|E|))$ I/Os. An alternative to the randomized strategy of generating the partition described here is a deterministic variant using an Euler tour around a minimum spanning tree. Thus, the obtained bound also holds in the worst case.

4.2 Implicit Graphs

An *implicit graph* is a graph that is not residing on disk but generated by successively applying a set of operators to states selected from the search horizon. The advantage in implicit search is that the graph is generated by a set of rules, and hence no disk accesses for the adjacency lists are required.

A variant of Munagala and Ranade's algorithm for BFS-search in implicit graphs has been coined with the term *delayed duplicate detection for frontier search* (Korf, 2003). Let $\mathcal{I}$ be the initial state, and N be the implicit successor generation function. The algorithm maintains BFS layers on disk. Layer $Open(i-1)$ is scanned and the set of successors are put into a buffer of size close to the main memory capacity. If the buffer becomes full, internal sorting followed by a duplicate elimination phase generates a sorted duplicate-free state sequence in the buffer that is flushed to disk[1]. The outcome of this phase are k pre-sorted files.

In the next step, *external merging* is applied to unify the files into $Open(i)$ by a simultaneous scan. The size of the output files is chosen such that a single pass suffices. Duplicates are eliminated. Since the files were presorted, the complexity is given by the scanning time of all files. One also has to eliminate $Open(i-1)$ and $Open(i-2)$ from $Open(i)$ to avoid re-computations; that is, nodes extracted from the external queue are not immediately deleted, but kept until after the layer has been completely generated and sorted, at which point duplicates can be eliminated using a parallel scan. The process is repeated until $Open(i-1)$ becomes empty, or the goal has been found.

The corresponding pseudo-code is shown in Figure 2. Note that the explicit partition of the set of successors into blocks is implicit. Termination is not shown, but imposes no additional implementation problem.

1. Delayed internal duplicate elimination can be improved by using hash tables for the blocks before flushed to disk. Since the state set in the hash table has to be stored anyway, the savings by early duplicate detection are small.

Procedure *Delayed-Duplicate-Detection-Frontier-Search*
$Open(-1) \leftarrow \emptyset$, $Open(0) \leftarrow \{\mathcal{I}\}$
$i \leftarrow 1$
while $(Open(i-1) \neq \emptyset)$
$A(i) \leftarrow N(Open(i-1))$
$A'(i) \leftarrow$ *remove duplicates from* $A(i)$
$Open(i) \leftarrow A'(i) \setminus (Open(i-1) \cup Open(i-2))$
$i \leftarrow i+1$

Figure 2: Delayed duplicate detection algorithm for BFS.

As with the algorithm of Munagala and Ranade, delayed duplicate detection applies $O(sort(|N(Open(i-1))|)+scan(|Open(i-1)|+|Open(i-2)|))$ I/Os. However, since no explicit access to the adjacency list is needed, by $\sum_i |N(Open(i))| = O(|E|)$ and $\sum_i |Open(i)| = O(|V|)$, the total execution time is $O(sort(|E|) + scan(|V|))$ I/Os.

In exploration problems where the branching factor is bounded, we have $|E| = O(|V|)$, and thus the complexity for implicit external BFS reduces to $O(sort(|V|))$ I/Os.

The algorithm applies $scan(|Open(i-1)|+|Open(i-2)|)$ I/Os in each phase. Does summing these quantities in fact yield $O(scan(|V|))$ I/Os, as stated? In very sparse problem graphs that are simple chains, if we keep each $Open(i)$ in a separate file. this would accumulate to $O(|V|)$ I/Os in total. However, in this case the states in $Open(i)$, $Open(i+1)$, and so forth are stored consecutively in internal memory. Therefore, I/O is only needed if a level has $\Omega(B)$ states, which can happen only for $O(|V|/B)$ levels.

Delayed duplicate detection was used to generate the first complete breadth-first search of the 2×7 sliding tile puzzle, and the Towers of Hanoi puzzle with 4 pegs and 18 disks. It can also be used to generate large pattern databases that exceed main memory capacity (Korf & Felner, 2002). One file for each BFS layer will be sufficient. The algorithm shares similarities with the internal *Frontier search* algorithm (Korf, 1999; Korf & Zhang, 2000) that was used for solving multiple sequence alignment problem.

5. External A*

In the following we study how to extend external breadth-first-exploration in implicit graphs to *best-first* search. The main advantage of A* with respect to BFS is that, due to the use of a lower bound on the goal distance, it must only traverse a smaller part of the search space to establish an optimal solution.

In A*, the merit for state u is $f(u) = g(u) + h(u)$, with g being the cost of the path from the initial state to u and $h(u)$ being the estimate of the remaining costs from u to the goal. In each step, a node u with minimum f-value is removed from *Open*, and the new value $f(v)$ of a successor v of u is updated to the minimum of its current value and $f(v) = g(v) + h(v) = g(u) + w(u,v) + h(v) = f(u) + w(u,v) - h(u) + h(v)$; in this case, it is inserted into *Open* itself.

In our algorithm, we assume a *consistent* heuristic, where for all u and v we have $w(u,v) \geq h(u) - h(v)$, and a uniformly weighted undirected state space problem graph. These conditions are often met in practice, since many problem graphs in single agent search are uniformly weighted and undirected and many heuristics are consistent. BFS can be seen as a special case of A* in uniform graphs with a trivial heuristic h that evaluates to zero for each state.

Under these assumptions, we have $h(u) \leq h(v) + 1$ for every state u and every successor v of u. Since the problem graph is undirected this implies $|h(u) - h(v)| \leq 1$ and $h(v) - h(u) \in \{-1, 0, 1\}$. If the heuristic is consistent, then on each search path, the evaluation function f is non-decreasing. No successor will have a smaller f-value than the current one. Therefore, the A* algorithm, which traverses the state set in f-order, expands each node at most once.

Take for example sliding tile puzzles, where numbered tiles on a rectangular grid have to be brought into a defined goal state by successively sliding tiles into one empty square. The *Manhattan distance* is defined as the sum of the horizontal and vertical differences between actual and goal configurations, for all tiles. It is easy to see that the Manhattan distance is consistent, since for two successive states u and v the difference of the according estimate evaluations $h(v) - h(u)$ is either -1 or 1. Therefore, f-values of u and v are either the same or $f(v) = f(u) + 2$.

5.1 Buckets

As above, *External A** maintains the search horizon on disk, possibly partitioned into main-memory-sized sequences. In fact, the disk files correspond to an external representation of Dial's implementation of a priority queue data structure that is represented as an array of buckets (Dial, 1969). In the course of the algorithm, each bucket addressed with index i will contain all states u in the set *Open* that have priority $f(u) = i$. An external representation of this data structure will memorize each bucket in a different file.

We introduce a refinement of the data structure that distinguishes between states of different g-values, and designates bucket $Open(i,j)$ to all states u with path length $g(u) = i$ and heuristic estimate $h(u) = j$.

As with the description of external BFS, we do not change the identifier *Open* to separate *generated* from *expanded* states (traditionally denoted as the *Closed* list). During the execution of A*, bucket $Open(i,j)$ may refer to elements that are in the current search horizon or belong to the set of expanded nodes. During the exploration process, only nodes from one currently *active bucket* $Open(i,j)$ with $i + j = f_{\min}$ are expanded, up to its exhaustion. Buckets are selected in lexicographic order for (i,j); then, the buckets $Open(i',j')$ with $i' < i$ and $i' + j' = f_{\min}$ are *closed*, whereas the buckets $Open(i',j')$ with $i' + j' > f_{\min}$ or with $i' > i$ and $i' + j' = f_{\min}$ are *open*. For states in the active bucket the status is either *open* or *closed*.

For an optimal heuristic, i.e., a heuristic that estimates the shortest path distance f^*, A* will consider the buckets $Open(0, f^*), \ldots, Open(f^*, 0)$. On the other hand, if the heuristic is *trivial* it considers the buckets $Open(0,0), \ldots, Open(f^*, 0)$. This might lead to the hypothesis that the A* looks at f^* buckets. Unfortunately, this is not true.

Consider Figure 3, in which the g-values are plotted with respect to the h-values, such that states with the same $f = g + h$ value are located on the same diagonal. For states

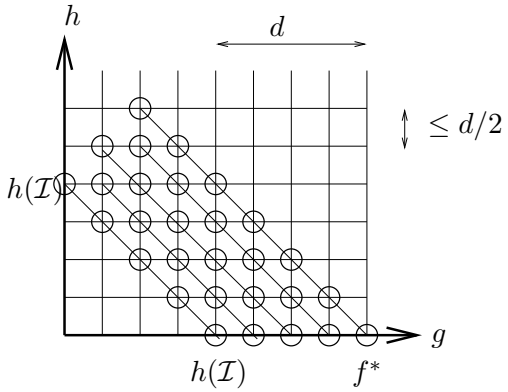

Figure 3: The number of buckets selected in A*.

that are expanded in $Open(g, h)$ the successors fall into $Open(g+1, h-1)$ $Open(g+1, h)$, or $Open(g+1, h+1)$. The number of naughts for each diagonal is an upper bound on the number buckets that are needed. It is trivial to see that the number is bounded by $f^*(f^*+1)/2$, since naughts only appear in the triangle bounded by the f^*-diagonal. We can, however, achieve a slightly tighter bound.

Lemma 1 *The number of buckets* $\mathrm{Open}(i, j)$ *that are considered by A* in a uniform state space problem graph with a consistent heuristic is bounded by* $(f^*+1)^2/3$.

Proof: Let d be the distance $f^* - h(\mathcal{I})$. Below $h(\mathcal{I})$ there are at most $d \cdot h(\mathcal{I}) + h(\mathcal{I})$ nodes. The *roof* above $h(\mathcal{I})$ has at most $1 + 3 + \ldots + 2(d/2) - 1$ nodes (counted from top to bottom). Since the sum evaluates to $d^2/4$ we need at most $d \cdot h(\mathcal{I}) + h(\mathcal{I}) + d^2/4$ buckets altogether. The maximal number of bucket $((f^*)^2+f^*+1)/3$ is reached for $h(\mathcal{I}) = (f^*+2)/3$.

By the restriction for f-values in the sliding-tile puzzles only about half the number of buckets have to be allocated. Note that f^* is not known in advance, so that we have to construct and maintain the files on the fly.

As in the algorithm of Munagala and Ranade, we can exploit the observation that in undirected state space graph structure, duplicates of a state with BFS-level i can at most occur in levels i, $i-1$ and $i-2$. In addition, since h is a total function, we have $h(u) = h(v)$ if $u = v$. This implies the following result.

Lemma 2 *During the course of executing A*, for all* i, i', j, j' *with* $j \neq j'$ *we have that* $\mathrm{Open}(i, j) \cap \mathrm{Open}(i', j') = \emptyset$.

Lemma 2 allows to restrict duplicate detection to buckets of the same h-value.

5.2 The Algorithm

For ease of describing the algorithm, we consider each bucket for the *Open* list as a different file. By Lemma 1 this accumulates to at most $(f^*+1)^2/3$ files. For the following we

therefore generally assume $(f^*+1)^2/3 = O(scan(|V|))$ and $(f^*+1)^2/3 = O(sort(|E|))$. As with the algorithm of Munagala and Ranade, however, it is not difficult to implement the algorithm to handle sparser graphs, too.

Figure 4 depicts the pseudo-code of the *External A** algorithm for consistent estimates and uniform graphs. The algorithm maintains the two values $g_{\min}$ and $f_{\min}$ to address the currently considered buckets. The buckets of $f_{\min}$ are traversed for increasing $g_{\min}$ up to $f_{\min}$. According to their different h-values, successors are arranged into three different horizon lists $A(f_{\min})$, $A(f_{\min}+1)$, and $A(f_{\min}+2)$; hence, at each instance only four buckets have to be accessed by I/O operations. For each of them, we keep a separate buffer of size $B/4$; this will reduce the internal memory requirements to B. If a buffer becomes full then it is flushed to disk. As in BFS it is practical to presort buffers in one bucket immediately by an efficient internal algorithm to ease merging, but we could equivalently sort the unsorted buffers for one buckets externally.

There can be two cases that can give rise to duplicate nodes within an active bucket: two different nodes of the *same* predecessor bucket generating similar nodes, and two nodes belonging to *different* predecessor buckets generating similar nodes. These two cases can be dealt with by merging all the pre-sorted buffers corresponding to the same bucket, resulting in one sorted file. This file can then be scanned to remove the duplicate nodes from it. In fact, both the merging and duplicates removal can be done simultaneously.

Another special case of the duplicate nodes exists when the nodes that have already been evaluated in the upper layers are generated again. These duplicate nodes have to be removed by a file subtraction process for the next active bucket $Open(g_{min}+1, h_{\max}-1)$ by removing any node that has appeared in $Open(g_{min}, h_{\max}-1)$ and $Open(g_{min}-1, h_{\max}-1)$. This file subtraction can be done by a mere parallel scan of the presorted files and by using a temporary file in which the intermediate result is stored.

Note that it suffices to perform the duplicate removal only for the bucket that is to be expanded next, i.e., $Open(g_{min}+1, h_{\max}-1)$. The other buckets might not have been fully generated and hence we can save the redundant scanning of the files for every iteration of the inner most *while* loop.

When merging the presorted sets A' with the previously existing *Open* buckets (both residing on disk), duplicates are eliminated, leaving the sets $Open(g_{min}+1, h_{\max}-1)$, $Open(g_{min}+1, h_{\max})$ and $Open(g_{\min}+1, h_{\max}-1)$ duplicate-free. Then the next active bucket $Open(g_{\min}+1, h_{\max}-1)$ is refined not to contain any state in $Open(g_{\min}-1, h_{\max}-1)$ or $Open(g_{\min}, h_{\max}-1)$. This can be achieved through a parallel scan of the presorted files and by using a temporary file in which the intermediate result is stored, before $Open(g_{\min}+1, h_{\max}-1)$ is updated. It suffices to perform file subtraction lazily only for the bucket that is expanded next.

Since *External A** simulates A* and changes only the order of elements to be expanded that have the same f-value, completeness and optimality are inherited from the properties shown for A* (Pearl, 1985).

Theorem 1 (I/O performance of External A*) *The complexity for External A* in an implicit unweighted and undirected graph with a consistent estimate is bounded by* $O(\text{sort}(|E|)+\text{scan}(|V|))$ *I/Os.*

Procedure *External A**
  $Open(0, h(\mathcal{I})) \leftarrow \{\mathcal{I}\}$
  $f_{\min} \leftarrow h(\mathcal{I})$
  while $(f_{\min} \neq \infty)$
    $g_{\min} \leftarrow \min\{i \mid Open(i, f_{\min} - i) \neq \emptyset\}$
    while $(g_{\min} \leq f_{\min})$
      $h_{\max} \leftarrow f_{\min} - g_{\min}$
      $A(f_{\min}), A(f_{\min}+1), A(f_{\min}+2) \leftarrow N(Open(g_{\min}, h_{\max}))$
      $Open(g_{\min}+1, h_{\max}+1) \leftarrow A(f_{\min}+2)$
      $Open(g_{\min}+1, h_{\max}) \leftarrow A(f_{\min}+1) \cup Open(g_{\min}+1, h_{\max})$
      $Open(g_{\min}+1, h_{\max}-1) \leftarrow A(f_{\min}) \cup Open(g_{\min}+1, h_{\max}-1)$
      $Open(g_{\min}+1, h_{\max}-1) \leftarrow$ *remove duplicates from* $Open(g_{\min}+1, h_{\max}-1)$
      $Open(g_{\min}+1, h_{\max}-1) \leftarrow Open(g_{\min}+1, h_{\max}-1) \setminus$
        $(Open(g_{\min}, h_{\max}-1) \cup Open(g_{\min}-1, h_{\max}-1))$
      $g_{\min} \leftarrow g_{\min} + 1$
    $f_{\min} \leftarrow \min\{i+j > f_{\min} \mid Open(i,j) \neq \emptyset\} \cup \{\infty\}$

Figure 4: *External A** for consistent and integral heuristics.

Proof: By simulating internal A*, the delayed duplicate elimination per serves that each edge in the state space problem graph is looked at at most once.

Similar to the analysis for external implicit BFS $O(sort(|N(Open(g_{\min}+1, h_{\max}-1))|)$ I/Os are needed to eliminate duplicates in the successor lists. Since each state is expanded at most once, this yields $O(sort(|E|))$ I/Os for the overall run time. Filtering, evaluating states, and merging lists is available in scanning time of all buckets in consideration. During the exploration, each bucket *Open* will be referred to at most six times, once for expansion, at most three times as a successor bucket and at most two times for duplicate elimination as a predecessor of the same h-value as the currently active bucket. Therefore, evaluating, merging and file subtraction add $O(scan(|V|) + scan(|E|))$ I/Os to the overall run time. Hence, the total execution time is $O(sort(|E|) + scan(|V|))$ I/Os.

As a corollary, if we additionally have $|E| = O(|V|)$, the complexity reduces to $O(sort(|V|))$ I/Os.

Internal costs have been neglected in the above analysis. Since each state is considered only once for expansion, the internal costs are $|V|$ times the time t_{exp} for successor generation, plus the efforts for internal duplicate elimination and sorting. By setting new edges weight $w(u, v)$ to $h(u) - h(v) + 1$, for consistent heuristics A* can be cast as a variant of Dijkstra's algorithm that requires internal costs of $O(C \cdot |V|)$, $C = \max\{w(u, v) \mid v$ successor of $u\}$ on a Dial. Due to consistency we have $C \leq 2$, so that, given $|E| = O(|V|)$, internal costs are bounded by $O(|V| \cdot (t_{exp} + \log |V|))$, where $O(|V| \log |V|))$ refers to the total internal sorting efforts.

To reconstruct a solution path, we could store predecessor information with each state on disk, and apply backward chaining, starting with the target state. However, this is not

strictly necessary: For a state in depth g, we intersect the set of possible predecessors with the buckets of depth $g-1$. Any state that is in the intersection is reachable on an optimal solution path, so that we can recur. The time complexity is bounded by the scanning time of all buckets in consideration and surely in $O(scan(|V|))$.

5.3 Non-Uniformly Weighted Graphs

Up to this point, we have made the assumption of uniformly weighted graphs; in this section, we generalize the algorithm to small integer weights in $\{1,\ldots,C\}$. Due to consistency of the heuristic, it holds for every state u and every successor v of u that $h(v) \geq h(u)-w(u,v)$. Moreover, since the graph is undirected, we equally have $h(u) \geq h(v)-w(u,v)$, or $h(v) \leq h(u)+w(u,v)$; hence, $|h(u)-h(v)| \leq w(u,v)$. This means that the successors of the nodes in the active bucket are no longer spread across three, but over $3+5+\ldots+2C+1 = C\cdot(C+2)$ buckets.

For duplicate reduction, we have to subtract the $2C$ buckets $Open(i-1,j)$,..., $Open(i-2C,j)$ from the active bucket $Open(i,j)$ prior to its nodes' expansion. It can be shown by induction over $f=i+j$ that no duplicates exist in smaller buckets. The claim is trivially true for $f \leq 2C$. In the induction step, assume to the contrary that for some node $v \in Open(i,j)$, $Open(i',j)$ contains a duplicate v' with $i' < i-2C$; let $u \in Open(i-w(u,v),j_u)$ be the predecessor of v. Then, by the undirectedness, there must be a duplicate $u' \in Open(i'+w(u,v),j_u)$. But since $f(u') = i'+w(u,v)+j_u \leq i'+C+j_u < i-C+j_u \leq i-w(u,v)+j_u = f(u)$, this is a contradiction to the induction hypothesis.

The derivation of the I/O complexity is similar to the uniform case; the difference is that each bucket is referred to at most $2C+1$ times for bucket subtraction and expansion. Therefore, each edge in the problem graph is considered at most once. Also, we need $O(C^2)$ I/Os for accessing the files, which in fact can be eliminated by a similar strategy as in BFS algorithm by Munagala and Ranade. The following theorem bounds the I/O complexity of the external A* algorithm for non-uniform graphs.

Theorem 2 (I/O performance of External A* for non-uniform graphs) *The I/O complexity for External A* in an implicit unweighted and undirected graph, where the weights are in $\{1,\ldots,C\}$, with a consistent estimate, is bounded by* $O(\mathrm{sort}(|E|) + C \cdot \mathrm{scan}(|V|))$.

If we do not impose a bound C on the maximum integer weight, or if we allow directed graphs, the run time increases to $O(sort(|E|)+f^*\cdot scan(|V|))$ I/Os. For larger edge weights and f^*-values, buckets become sparse and should be handled more carefully.

6. Lower Bound

Is $O(sort(|V|))$ I/O-optimal? To devise lower bounds for delayed duplicate elimination, the following definition for *big-oh* is appropriate:

$$\begin{aligned} O(f(n,M,B)) \;=\; \{g \mid\; & \exists c \in \mathbb{R}^+ \; \forall M,B \in \mathbb{N} \\ & \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 : g(n,M,B) \leq f(n,M,B)\}. \end{aligned}$$

The classes Θ and Ω are defined analogously. The intuition for universally quantifying M and B is that the adversary first chooses the machine, and then we, as the *good guys*, evaluate the bound. Aggarwal and Vitter (Aggarwal & Vitter, 1987) showed that external sorting in this model has the above-mentioned I/O complexity of $\Omega\left(N \log \frac{N}{B} / B \log \frac{M}{B}\right)$ and provide two algorithms that are asymptotically optimal. As internal *set inequality*, *set inclusion* and *set disjointness* require at least $N \log N - O(N)$ comparisons, the lower bound on the number of I/Os for these problems is also bounded by $\Omega(sort(N))$.

Arge, Knudsen and Larsen (Arge, Knudsen, & Larsen, 1993) considered the duplicate elimination problem. A lower bound on the number of comparisons needed is $N \log N - \sum_{i=1}^{k} N_i \log N_i - O(N)$ where N_i is the multiplicity of record i. The authors argue in detail that after the duplicate removal, the total order of the remaining records is known. This corresponds to an I/O complexity of at most

$$\Omega\left(\max\left\{\frac{N \log \frac{N}{B} - \sum_{i=1}^{k} N_i \log N_i}{B \log \frac{M}{B}}, N/B\right\}\right).$$

The authors also give an involved algorithm based on mergesort that matches this bound. For the sliding tile puzzle with two predecessing buckets and a branching factor $b \leq 4$ the value of N_i is less than or equal to 8. For general consistent estimates in uniform graphs, we have $N_i \leq 3c$, with c being an upper bound on the maximal branching factor. A search algorithm performs *delayed duplicate bucket elimination*, if it eliminates duplicates within one bucket and with respect to adjacent buckets that are duplicate free.

Theorem 3 (I/O Performance Optimality for External A*)) *If $|E| = \Theta(|V|)$, delayed duplicate bucket elimination in an implicit unweighted and undirected graph A* search with consistent estimates needs at least $\Omega(\mathrm{sort}(|V|))$ I/O operations.*

Proof: Since each state gives rise to at most c successors and there at most 3 predecessing buckets in A* search with consistent estimates in an uniformly weighted graph, given that previous buckets are mutually duplicate free, we have at most $3c$ states that are the same. Therefore, all sets N_i are bounded by $3c$. Since k is bounded by N we have that $\sum_{i=1}^{k} N_i \log N_i$ is bounded by $k \cdot 3c \log 3c = O(N)$. Therefore, the lower bound for duplicate elimination for N states is $\Omega(sort(N) + scan(N))$.

A related lower bound also applicable to the multiple disk model (Munagala & Ranade, 2001), establishes that solving the duplicate elimination problem with N elements having P different values needs at least $\Omega(\frac{N}{P} sort(P))$ I/Os, since the depth of any decision tree for the duplicate elimination problem is at least $N \log(P/2)$. For state space search with consistent estimates and bounded branching factor, we assume to have $P = \Theta(N) = \Theta(|E|) = \Theta(|V|)$, so that the I/O complexity reduces to $O(sort|V|)$.

7. Experiments

We implemented our approach in *ANSI C*, using simple file access operations `fopen`, `fclose`, `fgetc` and `fputc`. Files contain fixed-width records of binary-encoded states. We selected

S. No.	Initial State	Initial Estimate	Solution Length
1	(0 2 1 3 5 4 6 7 8 9 10 11 12 13 14 15)	4	16
2	(0 1 2 3 5 4 7 6 8 9 10 11 12 13 14 15)	4	24
3	(0 2 1 3 5 4 7 6 8 9 13 11 12 10 14 15)	10	30
4 {12}	(14 1 9 6 4 8 12 5 7 2 3 0 10 11 13 15)	35	45
5 {16}	(1 3 2 5 10 9 15 6 8 14 13 11 12 4 7 0)	24	42
6 {14}	(7 6 8 1 11 5 14 10 3 4 9 13 15 2 0 12)	41	59
7 {60}	(11 14 13 1 2 3 12 4 15 7 9 5 10 6 8 0)	48	66
8 {88}	(15 2 12 11 14 13 9 5 1 3 8 7 0 10 6 4)	43	65

Table 2: 15-puzzle instances used for experiments

15-Puzzle problem instances. Many instances cannot be solved internally with A* and the Manhattan distance. Each state is packed into 8 bytes.

Internal sorting is done by the built-in *quicksort* routine. External merge is performed by maintaining the file pointers for every flushed buffer and merging them into a single sorted file. Since we have a simultaneous file pointers capacity bound imposed by the operating system, we implemented two-phase merging. Duplicate removal and bucket subtraction are performed on single passes through the bucket file. The implementation differs a little from the algorithm presented in this paper in that the duplicate removal within one bucket, as well as the bucket subtraction are delayed until the bucket is selected for expansion. The program utilizes an implicit priority queue. For sliding tile puzzles, during expansion, the successor's f value differs from the parent state by exactly 2. This implies that in case of an empty diagonal, the program terminates.

We performed our experiments on a mobile AMD Athlon XP 1.5 GHz processor with 512 MB RAM, running MS Windows XP operating system. In Table 2 we give the example instances that we have used for our experiments. Some of them are adopted from Korf's seminal paper (Korf, 1985) (original numbers given in brackets). We chose some of the simplest and hardest instances for our experiments. The harder problems cannot be solved internally and were cited as the core reasons for the need of IDA*.

In Table 3 we show the diagonal pattern of states that is developed during the exploration for problem instance 1. The entry $x+y$ in the cell (i, j) implies that x and y number of states are generated from the expansion of $Open(i-1, j-1)$ and $Open(i-1, j+1)$, respectively.

The pattern of duplicate states in each bucket is shown in Table 4. An entry $u+v$ in the cell (i, j) implies that u states are the duplicate states *within* a bucket and v states are the duplicates that are found due to the *subtraction* of the states of the bucket $(i-2, j)$. The actual number of states remaining in a bucket after duplicate removal and subtraction can be obtained by subtracting the (i, j)th entry of Table 4 from the (i, j)th entry of Table 3.

The impact of internal buffer size on the I/O performance is clearly observable in Table 5. We show the I/O performance of two instances by varying the internal buffer size B. A larger buffer implies fewer flushes during writing, fewer block reads during expansion and fewer processing time due to internally sorting larger but fewer buffers. This I/O information and time information is collected using the task manager feature of MS Windows XP.

In Table 6, we show the impact of duplicate removal and bucket subtraction. Note that we do not employ any pruning technique like hashing or predecessor elimination. As

g/h	1	2	3	4	5	6	7	8	9	10	11
0	-	-	-	1+0	-	-	-	-	-	-	-
1	-	-	-	-	2+0	-	-	-	-	-	-
2	-	-	-	0+4	-	2+0	-	-	-	-	-
3	-	-	-	-	7+3	-	4+0	-	-	-	-
4	-	-	-	0+7	-	13+4	-	10+0	-	-	-
5	-	-	-	-	5+15	-	24+10	-	24+0	-	-
6	-	-	-	0+6	-	12+26	-	46+28	-	44+0	-
7	-	-	-	-	9+10	-	20+51	-	99+57	-	76+0
8	-	-	-	0+8	-	15+25	-	48+137	-	195+0	-
9	-	-	-	-	4+17	-	45+52	-	203+0	-	-
10	-	-	-	0+3	-	13+49	-	92+0	-	-	-
11	-	-	-	-	2+19	-	46+0	-	-	-	-
12	-	-	-	0+5	-	31+0	-	-	-	-	-
13	-	-	0+2	-	10+0	-	-	-	-	-	-
14	-	0+2	-	5+0	-	-	-	-	-	-	-
15	0+2	-	5+0	-	-	-	-	-	-	-	-

Table 3: States inserted in the buckets for instance 1

g/h	1	2	3	4	5	6	7	8	9	10	11
0	-	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	1+1	-	-	-	-	-	-	-
3	-	-	-	-	2+2	-	-	-	-	-	-
4	-	-	-	3+2	-	3+2	-	-	-	-	-
5	-	-	-	-	8+6	-	6+4	-	-	-	-
6	-	-	-	1+2	-	16+12	-	14+10	-	-	-
7	-	-	-	-	6+6	-	24+24	-	26+24	-	-
8	-	-	-	3+10	-	10+10	-	52+50	-	-	-
9	-	-	-	-	9+7	-	29+23	-	-	-	-
10	-	-	-	0+2	-	21+20	-	-	-	-	-
11	-	-	-	-	6+5	-	-	-	-	-	-
12	-	-	-	0+1	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-
15	1+0	-	-	-	-	-	-	-	-	-	-

Table 4: Duplicate states within a bucket + Duplicate states in top layers for instance 1

observable from the fourth entry, the gain is about 99% when duplicate removal and bucket subtraction are used. In the latter cases, we had to stop the experiment because of the limited hard disk capacity.

It should be noted here that these states are the number of states that are *generated* during the run and do not represent the total number of states that are actually *expanded*.

Initial State	B	I/O Reads	I/O Writes	Time (sec)
2	10	5,214	6,525	2
	25	3,086	3,016	1
	50	2,371	1,843	< 1
	100	2,022	1,265	< 1
3	50	7,312	8,463	4
	75	6,093	6,377	3
	100	5,481	5,329	3
	150	4,873	4,287	3

Table 5: Effects on I/O performance due to different internal buffer sizes

Initial State	N	N_{dr}	N_{dr+sub}
1	530,401	2,800	1,654
2	$>$ 50,000,000	126,741	58,617
3	$>$ 50,000,000	492,123	314,487
4	71,751,166	611,116	493,990
5	*<out of disk space>*	7,532,113	5,180,710
6	*<out of disk space>*	*<out of disk space>*	297,583,236
7	*<out of disk space>*	*<out of disk space>*	2,269,240,000
8	*<out of disk space>*	*<out of disk space>*	2,956,384,330

Table 6: Impact of duplicate removal and bucket subtraction on generated states

The number of expanded states differs largely from the generated states because of the removal of duplicate states and generation of states of $(f^* + 2)$ diagonal.

Initial State	N_{IDA^*} (Korf, 1985)	N_{ExA*}	S_{ExA*} *(GB)*	% gain
4	546,344	493,990	0.003	9.58
5	17,984,051	5,180,710	0.039	71.2
6	1,369,596,778	297,583,236	2.2	78.3
7	3,337,690,331	2,269,240,000	16.91	32
8	6,009,130,748	2,956,384,330	22	50.8

Table 7: Comparison of space requirement by IDA* and External A*

Finally, we compare the node count of our algorithm to the node count of IDA* in Table 7. As is noticeable in the table that the problem instances 6,7, and 8 can not be solved internally, especially 7 and 8 whose memory requirements surpass even the address limits of current PC hardware.

One interesting feature of our approach from a practical point of view is the ability to pause and resume the program execution. For large problem instances, this is a desirable feature in case we reach the system bounds of secondary storage and after upgrading the system want to resume the execution.

8. Related Work

The single disk model for external algorithms has been invented by Aggarwal and Vitter. A detailed survey on current techniques for the design of external memory algorithms can be established in (Meyer et al., 2003).

External priority queues for general weights are involved. An I/O-efficient algorithm for *single-source shortest path* simulates Dijkstra's algorithm by replacing the priority queue with the *tournament tree* data structure (Kumar & Schwabe, 1996). It is a priority queue data structure that was developed with the application to graph algorithms in mind; it is similar to an external heap, but it holds additional information. The tree stores pairs (x, y), where $x \in \{1, \ldots, N\}$ identifies the element, and y is called the *key*. The tournament tree is a complete binary tree, except for possibly some rightmost leaves missing. It has N/M leaves. There is a fixed mapping of elements to the leaves, namely, IDs in the range from $(i-1)M+1$ through iM map to the i-th leaf. Each element occurs exactly once in the tree, either in its leaf or in some ancestor thereof. Each node has an associated list of elements of size between $M/2$ and M, which are the smallest ones among all descendants. Additionally, it has an associated buffer of size M. Using an amortization argument, it can be shown that a sequence of k *update*, *delete*, or *deleteMin* operations on a tournament tree containing N elements requires at most $O(\frac{k}{B} \log_2 \frac{N}{B})$ accesses to external memory.

The *buffered repository tree* is a variant of the tournament tree that provides two operations: *insert(x,y)* inserts element x under key y, where several elements can have the same key. *extractAll(y)* returns and removes all elements that have key y.

As in tournament trees, keys come from a key set $\{1, \ldots, N\}$, and the leaves in the static height-balanced binary tree are associated with the key ranges in the same fixed way. Each internal node stores signals in a buffer of size B, which is recursively distributed to its two children when it becomes full. Thus, an *insert* operation needs $O(\frac{1}{B} \log_2 |V|)$ I/O amortized operations. An *extractAll* operation requires $O(\log_2 |V| + \frac{x}{B})$ accesses to secondary memory, where the first term corresponds to reading all buffers on the path from the root to the correct leaf, and the second term reflects reading the x reported elements from the leaf.

Moreover, a buffered repository tree T is used to remember nodes that were encountered earlier. When v is extracted from H, each incoming edge (u, v) is inserted into T under key u. If at some later point u is extracted, then *extractAll*(u) on T yields a list of edges that should not be traversed because they would lead to duplicates. Before the expansion of u, these edges are removed from $P(u)$. The algorithm takes $O(|V| + |E|/B)$ I/Os to access adjacency lists. The $O(|E|)$ operations on the priority queues change between different $P(v)$ at most $O(|V|)$ times, leading to a cost of $O(|V| + \mathit{sort}(|E|))$. Additionally, there are $O(|E|)$ *insert* and $O(|V|)$ *extractAll* operations on T, which add up to $O((|V| + |E|/B) \cdot \log_2 |V|$; this term also dominates the overall complexity of the algorithm.

More efficient algorithms can be developed by exploiting properties of particular classes of graphs. In the case of *directed acyclic graphs (DAGs)* that are e.g. apparent in multiple DNA-sequence alignment problems, we can apply the general technique of *time-forward processing* for solving the SSSP-problem. We assume a topological order of G; i.e., for each edge (u, v), the index of u is smaller than that of v. The start node has index 0. Nodes are processed in this order. Due to the fixed ordering, we can access all adjacency lists in

$O(scan(|E|))$ time. Since this procedure involves $O(|V| + |E|)$ priority queue operations, the overall complexity is $O(sort(|V| + |E|))$.

It has been shown that the BFS and SSSP problem can be solved with $O(sort(|V|))$ I/Os for the many subclasses of *sparse* graphs, e.g

Planar Graphs that can be drawn in a plane in the natural way without having edges cross between nodes. For example, many route planning graphs are planar.

Outerplanar Graphs are planar graphs such that one of its faces has all nodes on its boundary.

Grid Graphs are graphs where the nodes are a subset of the vertices of a regular two-dimensional grid, and edges can only connect neighbor vertices (whose coordinates can differ by at most one).

Graphs of Bounded Treewidth A *tree decomposition* is a tree of subsets of the graph nodes such that both endpoints of each edge are contained in some subset, and such that on any path between two subsets in the tree decomposition that contain the same node x, x must also be contained in all of the connecting nodes. The *tree width* of a graph is the minimum size of a subset in any of its tree decompositions.

Most algorithms are based on *graph separation* techniques.

The only other approach that applies A* to secondary memory is referenced as *Localizing A** (Edelkamp & Schrödl, 2000). It presents a heuristic search algorithm to overcome the lack of memory locality. In connection with *software paging*, this has lead to a significant speedup. The basic idea is to organize the graph structure such that node locality is preserved as much as possible, and to prefer to some degree local expansions over those with globally minimum f value. As a consequence, the algorithm cannot stop with the first solution found. However, the overhead in the increased number of expansions can be significantly outweighed by the reduction in the number of page faults.

The application area of the algorithm is *route planning*, where a map is partitioned according to the two dimensional physical layout, and store it as in a tile-wise fashion. Ideally, the tiles should roughly have a size such that a few of them (as explained shortly) fit into main memory. The *Open* list is represented by a new data structure, called *Heap-Of-Heaps*. It consists of a collection of k priority queues one for each page. At any instant, only one of the heaps is designated as being *active*. One additional priority queue keeps track of the root nodes inactive priority queues. It is used to quickly find the overall minimum across all of these heaps.

9. Conclusion

In this work, we present an extension of external undirected BFS graph search to external A* search which can exploit a goal-distance heuristics. Contrary to some previous works, we are concerned with implicitly represented graphs. The key issue to efficiently solve the problem is a file-based priority queue matrix as a refinement to Dial's priority queue data structure. For consistent estimates in uniform graphs we show that we achieve optimal I/O complexity.

On the other side of the memory hierarchy, by the achievement of better memory locality for access, the external design for A* seem likely to increase cache performance.

Different from delayed duplicate detection, we start with the external BFS exploration scheme of Munagala and Ranade to give complexity results measured in the number of I/O operations that the algorithm executes. To the best of our knowledge, the strategy of delayed duplicate detection has not been previously generalized to best-first search.

There is a tight connection between the exploration of sets of states externally and an efficient symbolic representation for sets of states with BDDs. The design of existing symbolic heuristic search algorithms seem to be strongly influenced by the delayed duplication and external set manipulation.

The other research area that is affected are internal memory-restricted algorithms, that are mainly interested in objective of an early removal of states from the main memory. The larger space-efficiency of a breadth-first traversal ordering in heuristic search has lead to improved memory consumption for internal algorithms, with new algorithms entitled *breadth-first heuristic search* and *breadth-first iterative-deepening* (Zhou & Hansen, 2004).

We expect a practical relevant outcome of this research in application domains like AI planning, multiple sequence alignment, model checking and route planning problems.

References

Aggarwal, A., & Vitter, J. S. (1987). Complexity of sorting and related problems. In *International Colloquim on Automata, Languages and Programming (ICALP)*, No. 267 in LNCS, pp. 467–478.

Arge, L., Knudsen, M., & Larsen, K. (1993). Sorting multisets and vectors in-place. In *Workshop on Algorithms and Data Structures (WADS)*, LNCS, pp. 83–94.

Chiang, Y.-J., Goodrich, M. T., Grove, E. F., Tamasia, R., Vengroff, D. E., & Vitter, J. S. (1995). External memory graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pp. 139–149.

Dial, R. B. (1969). Shortest-path forest with topological ordering. *Communication of the ACM*, *12*(11), 632–633.

Edelkamp, S., & Schrödl, S. (2000). Localizing A*. In *National Conference on Artificial Intelligence (AAAI)*, pp. 885–890.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for heuristic determination of minimum path cost. *IEEE Trans. on on Systems Science and Cybernetics*, *4*, 100–107.

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

Korf, R. E. (1999). Divide-and-conquer bidirectional search: First results. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pp. 1184–1191.

Korf, R. E. (2003). Delayed duplicate detection. In *IJCAI-Workshop on Model Checking and Artificial Intelligence (MoChart)*.

Korf, R. E., & Felner, A. (2002). *Chips Challenging Champions: Games, Computers and Artificial Intelligence*, chap. Disjoint Pattern Database Heuristics, pp. 13–26. Elsevier.

Korf, R. E., & Zhang, W. (2000). Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pp. 910–916.

Kumar, V., & Schwabe, E. J. (1996). Improved algorithms and data structures for solving graph problems in external memory. In *Sympostium on Parallel and Distributed Processing*, pp. 169–177. IEEE.

Mehlhorn, K., & Meyer, U. (2002). External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*.

Meyer, U., Sanders, P., & Sibeyn, J. (2003). *Memory Hierarchies*. Springer.

Munagala, K., & Ranade, A. (2001). I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pp. 87–88.

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

Zhou, R., & Hansen, E. (2004). Breadth-first heuristic search. In *International Conference on Automated Planning and Scheduling (ICAPS)*. To appear.