External A*

Stefan Edelkamp[1], Shahid Jabbar[1], and Stefan Schrödl[2]

[1] Computer Science Department
Baroper Str. 301
University Dortmund
{stefan.edelkamp,shahid.jabbar}@cs.uni-dortmund.de
[2] DaimlerChrysler Research and Technology Center
1510 Page Mill Road
Palo Alto, CA 94304
schroedl@rtna.daimlerchrysler.com

Abstract. In this paper we study *External A**, a variant of the conventional (internal) A* algorithm that makes use of external memory, e.g., a hard disk. The approach applies to implicit, undirected, unweighted state space problem graphs with consistent estimates. It combines all three aspects of best-first search, frontier search and delayed duplicate detection and can still operate on very small internal memory. The complexity of the external algorithm is almost linear in external sorting time and accumulates to $O(sort(|E|) + scan(|V|))$ I/O operations, 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 contrast to the internal algorithm, we exploit memory locality to allow blockwise 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.

Different variants of breadth-first and depth-first traversal of external graphs have been proposed earlier [3, 13]. 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 for the remaining distance of a state can significantly reduce the number of nodes that need to be explored [5].

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 [9] adapts Munagala and Ranade's algorithm [14] for the case of implicit graphs, and is presented next. Then we describe *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 and Basic Primitives

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.

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-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. The simplest operation is *external 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 $\Theta(\frac{N}{B}) = \Theta(n)$.

Another important operation is *external sorting*. *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 block wise 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.

3 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 [7], 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.

3.1 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 $O(\frac{|E|}{|V|}\mathit{sort}(|V|))$.

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 algorithm of *Munagala and Ranade* [14] 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 $\mathit{Open}(i)$ from $\mathit{Open}(i-1)$ as follows: Let $A(i) = N(\mathit{Open}(i-1))$ be the multi-set of neighbor vertices of nodes in $\mathit{Open}(i-1)$; $A(i)$ is created by concatenating all adjacency lists of nodes in $\mathit{Open}(i-1)$. Since after the preprocessing step the graph is stored in adjacency-list representation, this takes $O(|\mathit{Open}(i-1)| + |N(\mathit{Open}(i-1))|/B)$ I/Os. Then the algorithm removes duplicates by external sorting followed by an external scan. Hence, duplicate elimination takes $O(\mathit{sort}(A(i)))$ I/Os. Since the resulting list $A'(i)$ is still sorted, filtering out the nodes already contained in the sorted lists $\mathit{Open}(i-1)$ or $\mathit{Open}(i-2)$ is possible by parallel scanning, therefore this step can be done using $O(\mathit{sort}(|N(\mathit{Open}(i-1))|) + \mathit{scan}(|\mathit{Open}(i-1)| + |\mathit{Open}(i-2)|))$ I/Os. This completes the generation of $\mathit{Open}(i)$. The algorithm can record the nodes' BFS-level in additional $O(|V|)$ time using an external array.

```
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
```

Fig. 1. External BFS by Munagala and Ranade

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 $\sum_i |N(\mathit{Open}(i))| = O(|E|)$ and $\sum_i |\mathit{Open}(i)| = O(|V|)$, the execution of external BFS requires $O(|V| + \mathit{sort}(|E|))$ 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 [12] consists of a preprocessing and a BFS phase, arriving at a complexity of $O(\sqrt{|V| \cdot \mathit{scan}(|V| + |E|)} + \mathit{sort}(|V| + |E|))$ I/Os.

3.2 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* [9]. Let $\mathcal{I}$ be the initial state, and N be the implicit successor generation function. The algorithm maintains BFS layers on disk. Layer $\mathit{Open}(i-1)$ 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[3].

In the next step, *external merging* is applied to unify the files into $\mathit{Open}(i)$ by a simultaneous scan. The size of the output files is chosen such that a single pass suffices.

[3] 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*
$\mathit{Open}(-1) \leftarrow \emptyset, \mathit{Open}(0) \leftarrow \{\mathcal{I}\}$
$i \leftarrow 1$
while $(\mathit{Open}(i-1) \neq \emptyset)$
$A(i) \leftarrow N(\mathit{Open}(i-1))$
$A'(i) \leftarrow$ *remove duplicates from* $A(i)$
$\mathit{Open}(i) \leftarrow A'(i) \setminus (\mathit{Open}(i-1) \cup \mathit{Open}(i-2))$
$i \leftarrow i+1$

Fig. 2. Delayed duplicate detection algorithm for BFS.

Duplicates are eliminated. Since the files were presorted, the complexity is given by the scanning time of all files. One also has to eliminate $\mathit{Open}(i-1)$ and $\mathit{Open}(i-2)$ from $\mathit{Open}(i)$ 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 $\mathit{Open}(i-1)$ 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 in the Algorithm of Munagala and Ranade. Termination is not shown, but imposes no additional implementation problem.

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

The algorithm applies $\mathit{scan}(|\mathit{Open}(i-1)|+|\mathit{Open}(i-2)|)$ I/Os in each phase. Does summing these quantities in fact yield $O(\mathit{scan}(|V|))$ I/Os, as stated? In very sparse problem graphs that are simple chains, if we keep each $\mathit{Open}(i)$ in a separate file, this would accumulate to $O(|V|)$ I/Os in total. However, in this case the states in $\mathit{Open}(i)$, $\mathit{Open}(i+1)$, 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 [10]. One file for each BFS layer will be sufficient. The algorithm shares similarities with the internal *Frontier search* algorithm [8, 11] that was used for solving multiple sequence alignment problems.

4 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 only has to 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 each u and its child 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 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 it is consistent, since for two successive states u and v the the difference $h(v) - h(u)$ is either -1 or 1. Therefore, f-values of u and v are either the same or $f(v) = f(u) + 2$.

4.1 Buckets

Like external BFS, *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 [4]. 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

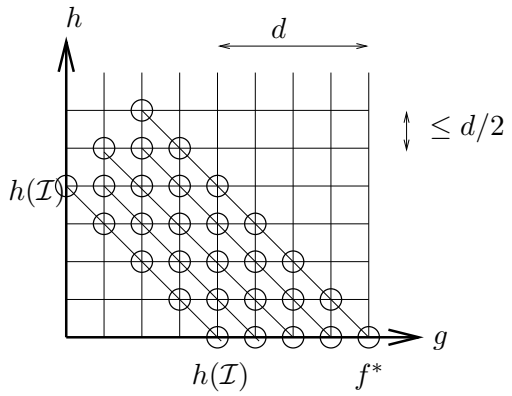

Fig. 3. The number of buckets selected in A*.

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 can be either *open* or *closed*.

For an optimal heuristic, i.e., a heuristic that computes 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 equal to zero, it considers the buckets *Open*$(0, 0), \ldots,$ *Open*$(f^*, 0)$. This leads 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 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* 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 $d = f^* - h(\mathcal{I})$. 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 $((f^*)^2 + f^* + 1)/3$ 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* Open$(i, j) \cap$ Open$(i', j') = \emptyset$.

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

4.2 The Algorithm

For ease of presentation, 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 $(f^* + 1)^2/3 = O(sort(|E|))$.

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 active bucket. 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; this will reduce the internal memory requirements to $4B$. 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.

Note that it suffices to perform the duplicate removal only for the bucket that is to be expanded next. 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 with the previously existing *Open* buckets (both residing on disk), duplicates are eliminated, leaving the set *Open*$(g_{min}, h_{\max})$, duplicate free. Moreover, bucket *Open*$(g_{\min}, h_{\max})$ is refined not to contain any state in *Open*$(g_{\min} - 1, h_{\max})$ or *Open*$(g_{\min} - 2, h_{\max})$. This can be achieved through a parallel scan of the presorted files.

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* [15].

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.*

Proof. By simulating internal A*, the delayed duplicate elimination scheme looks at each edge in the state space problem graph at most once.

Each data item is I/O efficiently written once as a successor, once for external sorting, once for expansion and scanned twice for duplicate elimination.

More precisely, we have $O(sort(|N(Open(g_{\min} - 1, h_{\max}))| + |N(Open(g_{\min} - 1, h_{\max} - 1))| + |N(Open(g_{\min} - 1, h_{\max} + 1))|))$ I/Os for eliminating duplicates in the accumulated successor lists before expanding $(Open(g_{\min}, h_{\max}))$, since this operation is based on *external sorting*. While each state is expanded at most once, this yields an amount of $O(sort(|E|))$ I/Os for the overall run time.

File subtraction requires $O(scan(|N(Open(g_{\min} - 1, h_{\max}))| + |N(Open(g_{\min} - 1, h_{\max} - 1))| + |N(Open(g_{\min} - 1, h_{\max} + 1))|) + scan(|N(Open(g_{\min} - 1, h_{\max}))|) +$

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

Fig. 4. *External A** for consistent and integral heuristics.

scan$(|N($*Open*$(g_{\min} - 2, h_{\max}))|))$ I/Os. Therefore, subtraction add $O($*scan*$(|V|) +$ *scan*$(|E|))$ I/Os to the overall run time.

All other operation are available in scanning time of all reduced buckets.

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. All operation base on batched access, we can scale the internal memory requirements down to $O(1)$, namely 2-3 states, depending on the internal memory needs for external sorting. Since each state is considered only once for expansion, the internal time requirements are $|V|$ *times* the duration t_{exp} for successor generation, plus the efforts for internal duplicate elimination and sorting, if applied. By setting new edges weight $w(u, v)$ to $h(u) - h(v) + 1$, for consistent heuristics A* is 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 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*$(|V|))$.

In practice, to save disk space when expanding bucket *Open*$(g_{\min}, h_{\max})$, we can eliminate the bucket *Open*$(g_{\min} - 2, h_{\max})$ after file subtraction. In this case, solution path has to be reconstructed by regeneration or through divide-and-conquer strategy.

4.3 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.

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

If we do not impose a bound C, 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 could become sparse and should be handled more carefully, as we would be wasting a number of I/Os in accessing the buckets having fewer than B states. If we have $O((f^*)^2 \cdot B)$ main memory space, a plausible solution in this case would be to keep all the unfilled buffers in main memory. The space requirement can be reduced to $O(C \cdot f^* \cdot B)$, i.e., saving only the C layers that change between successive active buckets. In any case, our algorithm requires at least $\Omega(C^2 \cdot B)$ main memory, to be able to store the C^2 buffers into which a successor might fall.

5 Lower Bound

Is $O($*sort*$(|V|))$ I/O-optimal? Aggarwal and Vitter [1] showed that external sorting 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 [2] 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

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 1. 15-puzzle instances used for experiments

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 preceding buckets and a branching factor $b \leq 4$ we have $N_i \leq 8$. For general consistent estimates in uniform graphs, we have $N_i \leq 3c$, with c being an upper bound on the maximal branching factor. An algorithm performs *delayed duplicate bucket elimination*, if it eliminates duplicates within a 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 $\Omega(\text{sort}(|V|))$ I/O operations.*

Proof. Since each state gives rise to at most c successors and there at most 3 preceding 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 $\Omega(sort(N) + scan(N))$.

6 Experiments

We selected 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

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 2. States inserted in the buckets for instance 1

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. In Table 1 we give the example instances that we have used for our experiments. Some of them are adopted from Korf's seminal paper [7] (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 2 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.

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

Table 3. Effects on I/O performance due to different internal buffer sizes

The impact of internal buffer size on the I/O performance is clearly observable in Table 3. 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 and time data are collected using the task manager of Windows XP.

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 4. Impact of duplicate removal and bucket subtraction on generated states

In Table 4, 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 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. 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*. 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^*} [7]	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 5. 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 5. 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.

7 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 in standard graph search, 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, through the achievement of better memory locality for access, the external design for A* seems 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.

There is a tight connection between the exploration of externally stored sets of states, and an efficient *symbolic* representation for sets of states with *Binary Decision Diagrams (BDDs)*. The design of existing symbolic heuristic search algorithms seems to be strongly influenced by the delayed duplication and external set manipulation. Another related research area are internal memory-restricted algorithms, that are mainly interested in oan 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* [16]. 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. In near future we expect a practical relevant outcome of this research in application domains especially AI planning, model checking and route planning.

Very recently, there are two related but independent research results, considering external best-first exploration. On the one hand, Korf [6] has successfully extended delayed duplicate detection to best-first search and also considered omission of the visited list as proposed in *frontier search*. It turned out that any 2 of the 3 options were compatible: Breadth-first frontier search with delayed duplicate detection, best-first frontier search, and best-first with external but non-reduced visited list. For the latter Korf simulates the buffered traversal in a Dial priority queue. With respect to this work, we contribute an algorithm that can deal with all three approaches. As an additional feature, Korf showed how external sorting can be avoided, by a selection of hash functions that split larger files into smaller pieces which fit into main memory. As with the h-value in our case a state and its duplicate will have the same hash address.

Zhou and Hansen [17] incorporated a projection function that maps states into an *abstract* state space; this reduces the successor scope of states that have to be kept in main memory. Projections are state space homomorphisms, such that for each pair of consecutive abstract states there exist an original pair of consecutive original states. In the running example of the 15-puzzle, the projection was based on states that have the same blank position. Unfortunately, this state-space abstraction also preserves the additional property that the successor set and the expansion sets are disjoint, yielding no self-loops in the abstract state space graph. For this case a reduction similar to the 3-layer idea of Munagala and Ranade applies to the reduced graph. For multiple-sequence alignment the authors could define an abstract graph structure that works well together with the *Sweep-A** algorithm. The method is crucially dependent on the availability of suitable partition functions. If the remaining duplicate elimination scope fits into main memory, the authors provide an improved worst case bound of $O(n \cdot |E|)$ I/Os. By the

additional assumption this does not contradict the lower bound provided. In contrast, we do not rely on any partitioning beside the h function and we do not require the duplicate scope to fit in main memory.

It seems that the other two approaches are quite compatible with our approach; e.g., by introducing the abstract state space concept, the spatial locality of the states can be further improved. Also, duplicate detection using external hashing within each of our buckets of the priority queue might result in better run-time of our algorithm, in practice. In summary, all three approaches have independent contributions and the future will show how they cooperate.

Acknowledgments The work is supported by *Deutsche Forschungsgemeinschaft* (DFG) in the projects *Heuristic Search* (Ed 74/3) and *Directed Model Checking* (Ed 74/2).

References

1. A. Aggarwal and J. S. Vitter. Complexity of sorting and related problems. In *International Colloquim on Automata, Languages and Programming (ICALP)*, number 267 in LNCS, pages 467–478, 1987.
2. L. Arge, M. Knudsen, and K. Larsen. Sorting multisets and vectors in-place. In *Workshop on Algorithms and Data Structures (WADS)*, LNCS, pages 83–94, 1993.
3. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 139–149, 1995.
4. R. B. Dial. Shortest-path forest with topological ordering. *Communication of the ACM*, 12(11):632–633, 1969.
5. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Trans. on on Systems Science and Cybernetics*, 4:100–107, 1968.
6. R. Korf. Best-first frontier search with delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, 2004. To appear.
7. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
8. R. E. Korf. Divide-and-conquer bidirectional frontier search: First results. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1184–1191, 1999.
9. R. E. Korf. Delayed duplicate detection. In *IJCAI-Workshop on Model Checking and Artificial Intelligence (MoChart)*, 2003.
10. R. E. Korf and A. Felner. *Chips Challenging Champions: Games, Computers and Artificial Intelligence*, chapter Disjoint Pattern Database Heuristics, pages 13–26. Elsevier, 2002.
11. R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pages 910–916, 2000.
12. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, 2002.
13. U. Meyer, P. Sanders, and J. Sibeyn. *Memory Hierarchies*. Springer, 2003.
14. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 87–88, 2001.
15. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
16. R. Zhou and E. Hansen. Breadth-first heuristic search. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 92–100, 2004.
17. R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI)*, 2004. To appear.