19th National Conference on Artificial Intelligence (AAAI-04) • San Jose, CA • July 25 - 29, 2004

Structured Duplicate Detection in External-Memory Graph Search

Rong Zhou and Eric A. Hansen
Department of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
{rzhou,hansen}@cse.msstate.edu

Abstract

We consider how to use external memory, such as disk storage, to improve the scalability of heuristic search in state-space graphs. To limit the number of slow disk I/O operations, we develop a new approach to duplicate detection in graph search that localizes memory references by partitioning the search graph based on an abstraction of the state space, and expanding the frontier nodes of the graph in an order that respects this partition. We demonstrate the effectiveness of this approach both analytically and empirically.

Introduction

Heuristic search in state-space graphs is a widely-used framework for problem solving in AI, but its scalability is limited by memory requirements. This has led to extensive research on how to use available memory efficiently in searching large state-space graphs.

The concept of "available memory" is ambiguous, since memory in most computer systems has a hierarchical structure in which fast, random-access internal memory is complemented by low-speed external memory, such as disk storage. Although external memory is vastly larger (and much cheaper) than internal memory, most heuristic search algorithms are designed to use internal memory only, and most research on memory-efficient heuristic search assumes a memory model in which access to all stored data items is equally fast. Algorithms that assume this memory model perform extremely poorly if external memory is used, since random access of external memory requires time-consuming I/O operations that are several orders of magnitude slower than random access of internal memory.

Some recent work begins to address this issue. Edelkamp and Schrödl (2000) consider use of virtual memory in A* graph search, and propose techniques that change the best-first order of node expansions in a way that improves reference locality and reduces the number of page faults in virtual memory. A more direct approach to limiting slow disk I/O is to design a search algorithm that explicitly manages access to external memory, since this approach can exploit knowledge of how the graph is structured, and is not limited by the size of virtual memory, which is usually much smaller than external memory. In the theoretical computer science community, there has been extensive research on external-memory algorithms, including development of external-memory breadth-first search algorithms for explicitly represented graphs (Munagala & Ranade 1999; Mehlhorn & Meyer 2002). In the AI community, Korf (2003a; 2003b) recently described an external-memory breadth-first search algorithm for implicitly represented graphs.

A key issue in designing an efficient external-memory graph search algorithm is *duplicate detection*. In graph search, the same node can be reached along different paths, and preventing redundant search effort requires storing already-visited nodes, so that the search algorithm can recognize when it encounters a node that has been already explored. Because duplicate detection potentially requires comparing each newly generated node to all stored nodes, it can lead to crippling disk I/O if already-explored nodes are stored on disk. All previous external-memory breadth-first search algorithms limit disk I/O by using a technique called *delayed duplicate detection* in which an entire set of nodes is expanded before performing any duplicate detection. In this paper, we introduce a new approach. Instead of delaying duplicate detection, we limit disk I/O by localizing memory references based on the structure of the search graph as revealed by a high-level abstraction of the state space. Computational results on challenging graph-search problems indicate that this approach, which we call *structured duplicate detection*, significantly limits disk I/O. For graphs with sufficient local structure, it allows graph-search algorithms to use external memory almost as efficiently as internal memory.

Background

Development of external-memory algorithms for various computational problems is an active area of research. Because disk I/O is several orders of magnitude slower than random access of internal memory, the concept of *I/O complexity* has been introduced to analyze the performance of external-memory algorithms (Aggarwal & Vitter 1988). It assumes a two-level model of memory in which internal memory of size M is supplemented by external memory, and an I/O operation moves data in blocks of size B between internal memory and external memory, where $1 < B \leq M/2$. The I/O complexity of an algorithm is defined as the number of I/O operations it performs.

External-memory breadth-first search has been considered by several researchers. In the theoretical computer sci-

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

ence community, researchers have focused on establishing bounds on its I/O complexity. A graph is assumed to be explicitly represented using adjacency lists and stored in external memory due to its large size. In searching explicit graphs, worst-case I/O complexity depends on the method of generating successor nodes as well as the method of duplicate detection, since adjacency lists must be read into internal memory to determine the successors of a node. Munagala and Ranade (1999) describe an external-memory breadth-first search algorithms that adopts an approach to successor generation that has linear complexity in the number of edges in the graph. Mehlhorn and Meyer (2002) achieve sublinear complexity for successor generation by a more sophisticated approach that involves partitioning the adjacency list based on a decomposition of the graph into connected subgraphs. Both algorithms use the same method of duplicate detection, called *delayed duplicate detection*, in which a set of nodes is expanded without performing duplicate detection, and the multi-set of generated successors is sorted and checked for duplicates in a single, more efficient operation. The algorithms alternate between these two steps.

1. *successor generation*, in which the algorithm generates successors for a set of nodes on the search frontier and appends these successors to a file (or files) in the order in which they are generated, without performing any duplicate detection, and
2. *delayed duplicate detection*, in which the file(s) of successor nodes are sorted (usually by an external-memory sort algorithm) based on their indices or state encodings, followed by a scan and compaction stage in which duplicate nodes in the sorted file(s) are eliminated.

In the AI community, Korf (2003a; 2003b) recently described an external-memory breadth-first search algorithm that uses the same method of delayed duplicate detection. His algorithm differs in two ways. First, he considers search in implicit graphs, in keeping with the standard AI approach to state-space search. An *implicit graph* is a compact representation of a graph in the form of a start node, a node expansion function that generates the immediate successors of a node, and a predicate that tests whether a node is a goal node. In searching implicit graphs, successor generation does not require disk I/O, and duplicate detection is the only source of I/O complexity.

A second difference is that Korf builds his external-memory breadth-first search algorithm on top of *frontier search*, a memory-efficient search algorithm that does not need to store a Closed list (Korf & Zhang 2000).

Structured duplicate detection

We propose an alternative approach to duplicate detection in external-memory graph search that has some significant advantages over delayed duplicate detection. It can also be combined with delayed duplicate detection in some cases.

The central idea of our approach is to exploit the structure of a state-space graph in order to localize memory references during duplicate detection, and so we call the approach *structured duplicate detection*. As an example, consider the $(n^2 - 1)$ sliding-tile puzzle. When a new node is generated, it can only differ from its parent node with respect to the position of the "blank" by either one row or one column. Therefore, in checking for duplicates, it is not necessary to check stored nodes for which the position of the "blank" differs from that of the parent node by more than one row or one column. If we partition the set of stored nodes in this way, we can significantly limit the number of stored nodes that need to be checked to guarantee that all duplicates are found. We continue to use this motivating example in the following, to help explain the idea.

Projection function and abstract state-space graph

Our approach is based on using a state-space projection function to decompose a state-space graph, and create an abstract state-space graph that reveals the local structure of the original graph at a high-level. State abstraction in heuristic search is well studied, and is used to create admissible heuristics and to organize hierarchical search (Holte *et al.* 1996). We use an approach to state abstraction that is very similar to previous approaches, but has a different purpose: localizing memory references in duplicate detection.

A state-space projection function is a many-to-one mapping from the original state space to an abstract state space, in which each abstract state corresponds to a set of states in the original state space. If a state x is mapped to an abstract state y, then y is called the *image* of x, and x is called the *pre-image* of y. There are many ways to define a state-space projection function. A common approach is to ignore some state variables in the encoding of the problem. For example, a simple state-space projection function for the $(n^2 - 1)$ sliding-tile puzzle can be defined by ignoring the positions of all tiles and considering only the position of the "blank." In this case, an abstract state corresponds to all states with the same position of the "blank," and there are n^2 abstract states compared to $n^2!/2$ states in the original state space. State-space projection functions can be defined in a similar way for other search problems (Klein & Manning 2003).

Given a state-space graph and state-space projection function, an *abstract state-space graph* is constructed as follows.

1. The set of nodes, called *abstract nodes*, in the abstract state-space graph corresponds to the set of abstract states.
2. An abstract node y' is a successor of an abstract node y iff there exist two states x' and x, such that
    a. x' is a successor of x, and
    b. x' and x are preimages of y' and y, respectively.

   If $y = y'$, it means that the abstract node y has a self loop.

For example, Figure 1(a) shows the nine possible positions of the "blank" in the Eight-puzzle. Figure 1(b) shows the abstract state-space graph created by the simple state-space projection function that maps a state into an abstract state based only on the position of the "blank." Each abstract node B_i in Figure 1(b) corresponds to the set of states with the "blank" located at position i in Figure 1(a).

Duplicate-detection scope

Although transforming a state-space graph into an abstract state-space graph can result in exponential reduction in the

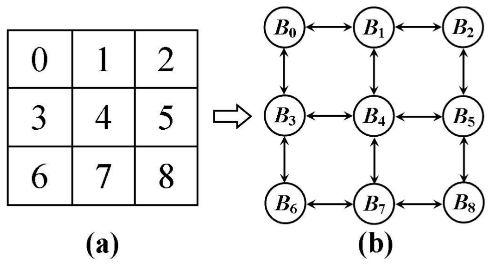

Figure 1: Panel (a) shows all possible positions of the "blank" for the Eight-puzzle. Panel (b) shows an example of an abstract state-space graph for the Eight-puzzle. This definition of the state-space projection function is based on the position of the "blank" only.

size of the graph, the local structure of the original state-space graph can be largely preserved. For example, abstract node B_0 in Figure 1(b) has only two successors; abstract nodes B_1 and B_3. This captures the fact that a single move of a tile can only change the position of the "blank" by either one column ($B_0 \rightarrow B_1$ in this case) or one row ($B_0 \rightarrow B_3$ in this case).

Definition 1 *An abstract state-space graph has* local structure *if it has bounded out-degree.*

For the $(n^2 - 1)$ sliding-tile puzzle using this simple state-space projection function, the out-degree of the abstract state-space graph is bounded by 4, for every n.

The local structure of an abstract state-space graph can be used to restrict the scope of duplicate detection so that only a fraction of stored nodes needs to be checked for duplicates, while still guaranteeing that all duplicates are found. For example, when generating successors for nodes that are pre-images of abstract node B_0 in Figure 2, the algorithm only needs to check for duplicates against stored nodes that are pre-images of abstract node B_1 or B_3.

Let an abstract node $y = p(x)$ be the image of a node x under a state-space projection function $p(\cdot)$ and let $successors(y)$ be the set of successor abstract nodes of y in the abstract state-space graph.

Definition 2 *The* duplicate-detection scope *of a node x under a state-space projection function $p(\cdot)$ corresponds to the union of sets of stored nodes that are pre-images of an abstract node y' such that $y' \in successors(p(x))$, that is,*

$$\bigcup_{y' \in \, successors(p(x))} p^{-1}(y')$$

where $p^{-1}(y')$ is the set of stored nodes that are pre-images of y'.

Theorem 1 *The duplicate-detection scope of a node contains all stored duplicates of the successors of the node.*

Proof: Suppose that a node x has a previously-generated successor node x' that is not included in the duplicate-detection scope of node x. Let $y' = p(x')$ be the image of x' under the state-space projection function $p(\cdot)$. According to Definition 2, we have $y' \notin successors(p(x))$. But according to the definition of an abstract state-space graph, if x' is a successor of x, then $p(x')$ must be a successor of $p(x)$. In other words, $y' \in successors(p(x))$. Since this leads to a contradiction, Theorem 1 must hold. □

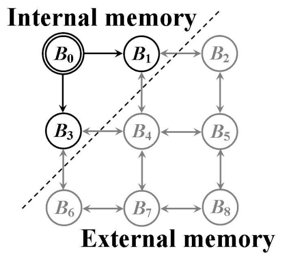

Figure 2: The duplicate-detection scope of nodes that are pre-images of abstract node B_0 includes nodes that are pre-images of abstract node B_1 or B_3. When expanding nodes that are pre-images of abstract node B_0, the search algorithm can use internal memory to store nodes that are pre-images of abstract node B_0, B_1, or B_3, and use external memory to store the rest of the nodes.

The concept of duplicate-detection scope can be very useful in external-memory graph search, because it suggests that a search algorithm use internal memory to store nodes within the duplicate-detection scope of a set of expanding nodes, and use external memory to store other nodes, when internal memory is full. Figure 2 illustrates this idea.

We use the term *nblock* to refer to a set (or "block") of nodes in the original state space that correspond to (i.e., are pre-images of) an abstract node. If an abstract state-space graph has local structure, then the duplicate-detection scope of any node consists of a bounded number of *n*blocks, and the largest duplicate detection scope can be a small fraction of the overall number of stored nodes. Note that the largest duplicate detection scope establishes a minimum internal-memory requirement for structured duplicate detection.

Search using structured duplicate detection

We now describe how to integrate structured duplicate detection into a state-space search algorithm.

Similar to delayed duplicate detection, structured duplicate detection is used with a search algorithm that expands a set of nodes at a time, such that the underlying search strategy is consistent with expanding the nodes in this set in any order. This is straightforward in breadth-first search, where a set of nodes corresponds to all nodes in the same layer of the breadth-first search graph. (We later explain how it can apply to best-first search.) Given this set of nodes, structured duplicate detection determines an order of expansion that minimizes I/O complexity, by exploiting local structure revealed by the abstract state-space graph.

To support structured duplicate detection, the set of stored nodes (i.e., nodes in the Open and Closed lists) is partitioned based on an abstract state-space graph. Each *n*block in the partition consists of all stored nodes that are pre-images of the same abstract node (i.e., each *n*block corresponds to an abstract node). This local structure is exploited as follows.

First, nodes in the same *n*block are expanded together, i.e., consecutively. This improves locality of memory reference because all nodes in the same *n*block have the same duplicate-detection scope. But it leaves undetermined the order in which to consider *n*blocks. In general, if the duplicate-detection scopes of two *n*blocks are almost the same, they should be expanded close to each other, in order to minimize the number of possible I/O operations. *N*blocks correspond to abstract nodes, and abstract nodes that are neighbors in the abstract state-space graph tend to have similar duplicate-detection scopes. Therefore, we choose to expand *n*blocks in an order that reflects neighbor relations in the abstract state-space space graph. A simple way to do this is to expand the *n*blocks in an order that reflects a breadth-first traversal of the abstract state-space graph.

When internal memory is full, the search algorithm must remove from internal memory one or more *n*blocks that do not belong to the duplicate-detection scope of nodes currently being expanded. Immediately before expanding nodes in a different *n*block, it must check if some *n*blocks in their duplicate-detection scope are missing from internal memory, and if so, read them from external memory. Because reading an *n*block into internal memory often results in writing another *n*block to external memory, we refer to these pairs of read and write operations as *nblock replacements*.

Except for *n*blocks that are part of the duplicate-detection scope of nodes being expanded, any *n*block can potentially be moved from internal memory to external memory. Deciding which *n*blocks to remove is identical to the page-replacement strategy of virtual memory, except that now the unit of replacement is an *n*block instead of a page. Thus, we call it an *nblock-replacement strategy*. It is well-known that the optimal replacement strategy always removes the page (or *n*block) that will not be used for the longest time (Belady 1966). Implementing such a strategy is generally impossible, as it requires future knowledge of the order in which pages (*n*blocks) will be needed, and *least-recently-used* is often the best strategy in practice (Sleator & Tarjan 1985). However, it is possible to compute an optimal *n*block-replacement strategy in our case, since the order in which *n*blocks are expanded is given by the breadth-first traversal order of an abstract state-space graph.

Thus, given a set of nodes to expand, structured duplicate detection expands them in an order such that (1) nodes in the same *n*block are expanded together, (2) *n*blocks are considered in an order that reflects a breadth-first traversal of the abstract state-space graph, and, (3) when internal memory becomes full, selection of which *n*blocks to write to disk is based on a pre-computed optimal *n*block-replacement strategy, or else, the least-recently used strategy.

It it straightforward to use structured duplicate detection with breadth-first search, where the set of nodes being expanded is a layer of the breadth-first search graph. It is also possible to use it with a best-first search algorithm such as A*. In solving a search problem with many ties, the order in which A* expands nodes with the same f-value is nondeterministic, and structured duplicate detection can determine an order that minimizes I/O complexity. The (n^2-1) sliding-tile puzzle is an example of a domain with many ties.

For search problems in which there are not many ties, it is possible to use structured duplicate detection as part of a slightly-modified version of A*, in which A* selects a set of nodes to expand containing nodes with almost the same f-value – for example, the best k nodes on the Open list, where k is a suitably large number. Structured duplicate detection can determine the order in which to expand this set of nodes. Although it may expand nodes in an order that is not strictly best-first, improved performance from structured duplicate detection may outweigh an increase in the number of nodes expanded. This modification of the best-first expansion order of A* to improve locality of memory references was previously proposed by Edelkamp and Schrödl (2000) as a way of improving use of virtual memory.

I/O complexity

I/O complexity is the primary factor that affects the speed of external-memory algorithms. The I/O complexity of search in an implicit graph depends entirely on the I/O complexity of duplicate detection. The worst-case I/O complexity of delayed duplicate detection is $O(\frac{n+m}{B}\log_{\frac{M}{B}}(n+m))$, where n is the number of nodes in the state-space graph, m is the number of edges, B is the size of a disk block, M is the size of internal memory, and $O(\frac{x}{B}\log_{\frac{M}{B}} x)$ is the worst-case I/O complexity of externally sorting x elements (Munagala & Ranade 1999).

For structured duplicate detection, we have this result.

Theorem 2 *If every duplicate-detection scope fits in internal memory, the worst-case I/O complexity of structured duplicate detection is* $O(\frac{n}{B}\cdot E)$.

In this result, E is the number of directed edges in the abstract state-space graph. (To keep the analysis simple, we consider an undirected edge as two reciprocally directed edges.) The theorem follows from the fact that E is the worst-case number of *n*block replacements that must be done in traversing the abstract state-space graph, and the worst-case number of I/O operations needed per *n*block replacement is the size of the largest *n*block divided by the disk block size, where n bounds the size of the largest *n*block.

The factor $\frac{1}{B}$ is the same in both analyses. Since an abstract state-space graph is typically exponentially smaller than the original state-space graph, the factor E for structured duplicate detection is comparable to $\log_{\frac{M}{B}}(n+m)$ for delayed duplicated detection. The difference in I/O complexity is the difference between the factor $n+m$ for delayed duplicate detection, and the factor n for structured duplicate detection. The presence of $n+m$ in the complexity analysis for delayed duplicate detection, in contrast to n for structured duplicate detection, can be interpreted as a penalty for *delaying* duplicate detection, since it bounds the cardinality of the multi-set of successors that is generated by expanding a set of nodes *before* performing duplicate detection.

In a highly-connected graph, m is *much* larger than n, and the penalty for delaying duplicate detection can be severe. It follows that delayed duplicate detection works best in sparse graphs. However, many important search problems do not have sparse graphs. An example is multiple sequence alignment, a motivating problem for frontier search (Korf & Zhang 2000). An advantage of structured duplicate detection is that it works equally well in sparse and highly-connected graphs.

#	Sol	Int Mem	Ext Mem	Exp	Secs
17	66	564,628	16,389,872	279,167,411	801
49	59	1,382,504	20,557,691	345,487,863	898
53	64	481,533	12,588,843	224,545,853	641
56	55	228,334	12,989,362	208,969,445	665
59	57	414,775	13,829,299	228,945,351	671
60	66	1,738,022	56,436,901	978,819,646	3,069
66	61	1,355,264	20,699,891	368,181,735	973
82	62	1,410,292	46,329,201	765,608,989	2,389
88	65	1,808,591	77,711,235	1,360,544,093	4,456
92	57	371,779	12,505,737	213,445,215	642

Table 1: Performance on the 10 most difficult instances of Korf's 100 random instances of the 15-puzzle. Columns show the instance number (#); solution length (Sol); peak number of nodes stored in internal memory (Int Mem); peak number of nodes stored in external memory (Ext Mem), number of node expansions (Exp), and running time in CPU seconds (Secs).

Name	Cost	Int Mem	Ext Mem	Exp	Secs
1aboA	8,483	8K	130K	4,207K	45
1amk	33,960	29K	555K	31,704K	373
1csy	14,160	6K	52K	1,943K	25
1ezm	40,733	17K	154K	6,186K	73
1gtr	58,010	188K	7,468K	1,094,936K	22,381
1idy	7,888	19K	415K	10,246K	110
1pfc	15,718	12K	153K	6,600K	71
1tis	37,581	24K	383K	29,778K	357
1wit	13,979	53K	1,513K	80,591K	1,192
actin	52,117	102K	2,783K	240,168K	3,972

Table 2: Performance in aligning groups of 5 protein sequences from reference set 1 of BAliBASE (Thompson, Plewniak, & Poch 1999). Columns show name of instance, cost of optimal alignment, peak number of nodes stored in internal memory (in thousands), peak number of nodes stored in external memory (in thousands), number of node expansions (in thousands), and CPU seconds.

Unlike delayed duplicate detection, structured duplicate detection has a minimum internal-memory requirement. Theorem 2 holds only if every duplicate detection scope fits in internal memory. So, whether it holds depends partly on the size of internal memory, and partly on the locality of the graph, and how well it is captured in an abstract state-space graph. If the largest duplicate detection scope does not fit in internal memory, it is possible to combine structured duplicate detection with delayed duplicate detection. Given a set of nodes to expand, the set can be partitioned based on the abstract state-space graph, and delayed duplicate detection can be performed separately on the nodes in each *n*block, as a way of leveraging local structure to improve performance.

Computational results

How efficiently the underlying graph-search algorithm uses internal memory has an effect on disk I/O, since it affects how much total memory is needed. For this reason, Korf (2003a; 2003b) built his external-memory graph search algorithm on top of frontier search, a very memory-efficient graph-search algorithm. In our experiments, we also combine structured duplicate detection with memory-efficient graph-search algorithms. A difference is that we consider heuristic search, whereas previous work on external-memory graph search considers breadth-first search only.

For the results in Tables 1 and 2, we ran the search algorithms in a mode that minimizes use of internal memory. So the peak amount of internal memory used corresponds to the minimum internal-memory requirements of the algorithms, using structured duplicate detection and a given abstraction of the state space. Ordinarily, the algorithms would use all available RAM before using disk. The algorithms were run on a 2.4GHz Intel Pentium with 2 gigabytes of RAM and a 7200RPM Seagate disk with 120 gigabytes of storage.

Fifteen-puzzle

To solve the Fifteen puzzle, we use structured duplicate detection together with a breadth-first heuristic search algorithm that uses divide-and-conquer solution reconstruction, called Breadth-First Iterative-Deepening A* (Zhou & Hansen 2004). Unlike brute-force breadth-first search, this algorithm uses upper and lower bounds to prune nodes that cannot be part of an optimal solution. Table 1 shows how the algorithm performs on the 10 most difficult instances of Korf's 100 random instances of the 15-puzzle (Korf 1985).

We used a state-space projection function that groups together states based on the positions of tiles 15 and 8, plus the position of the "blank," creating an abstract state-space graph with $16 \cdot 15 \cdot 14 = 3360$ nodes. Based on this partition of the state space, structured duplicate detection reduces the internal-memory requirement of the search algorithm by a factor of between 16 and 58 times. In exchange, it increases running time by only about 24% in solving these examples. Using both structured duplicate detection and divide-and-conquer solution reconstruction, all 100 instances of the 15-puzzle are solved using no more than 42 megabytes of RAM, without ever re-expanding a node in the same iteration. Based on the number of distinct nodes expanded, A* would need 28 gigabytes of RAM just to store the Closed list in solving instance 88.

Multiple sequence alignment

The multiple sequence alignment problem is an example of a search problem that cannot be solved efficiently using delayed duplicate detection because its search graph is very highly-connected. The search graph for the multiple sequence alignment problem is an n-dimensional hyperlattice, where n is the number of sequences being aligned.

We implemented structured duplicate detection on top of a search algorithm called Sweep A*, which uses divide-and-conquer solution reconstruction to limit use of memory (Zhou & Hansen 2003).[1] We tested the resulting algorithm on a selection of difficult multiple sequence alignment problems from reference set 1 of BAliBASE, a widely-used benchmark (Thompson, Plewniak, & Poch 1999). All

[1]Sweep A* is a specialized search algorithm for multiple sequence alignment that expands nodes on a layer-by-layer basis, which makes it a good fit for structured duplicate detection. It is also very memory-efficient. For comparison, frontier search (Korf & Zhang 2000) cannot solve instances 1gtr, 1wit, or actin in Table 2 within 2 gigabytes of RAM, and uses an average of 25 times more memory than Sweep A* in solving the other instances.

problems involved aligning 5 protein sequences. Our cost function was a Dayhoff substitution matrix with linear gap penalty of 8, and we used a pairwise alignment admissible heuristic. To create an abstract state-space graph, we used a state-space projection function that ignores all but 3 sequences. This creates an abstract state space with $O(l^3)$ abstract nodes, where l is the average length of a sequence, compared to $O(l^5)$ nodes in the original state-space graph.

Table 2 shows that structured duplicate detection reduces the internal-memory requirements of Sweep A* by a factor of between 10 and 40 times. Using disk, the algorithm needs only 20 megabytes of RAM to solve all instances in Table 2, including the space for the pairwise heuristic. Interestingly, the external-memory version of Sweep A* runs faster than an internal-memory version of Sweep A* that does not use structured duplicate detection, by an average of 72%! The reason is that structured duplicate detection runs faster in internal memory than un-structured duplicate detection due to locality of memory references, and this speedup outweighs the extra time for disk I/O. The speedup is more evident in multiple sequence alignment than the sliding-tile puzzle because the multiple sequence alignment search graph is much more highly-connected, which significantly increases the number of duplicates generated per node expansion.

Speeding up internal-memory search

We emphasize that for both the Fifteen puzzle and multiple sequence alignment, structured duplicate detection improves the performance of internal-memory graph search, in addition to using external memory efficiently. This is because each time the search algorithm checks for duplicates, it only needs to check a small subset of the stored nodes. For the Fifteen puzzle, this leads to a 16% speedup in internal-memory performance. For multiple sequence alignment, it cuts the running time of internal-memory search in half. Again, the speedup is much greater for multiple sequence alignment because in a very highly-connected graph, many more duplicates are generated. This underscores a benefit of structured duplicate detection. Unlike delayed duplicate detection, it is more effective in highly-connected graphs, where the problem of duplicate detection is more crucial.

Conclusion

We have introduced structured duplicate detection, a novel approach to localizing memory references in duplicate detection, and showed that it can reduce the I/O complexity of external-memory graph search, as well as increase the speed of duplicate detection in internal-memory search.

Structured duplicate detection has some advantages over delayed duplicate detection, the only previous approach to external-memory graph search. In particular, it is more effective in highly-connected graphs, and it can exploit local graph structure that is not considered by delayed duplicate detection. But structured duplicate detection and delayed duplicate detection can also be viewed as complementary approaches to reducing I/O complexity, and can be used together for difficult search problems. If a search algorithm can leverage enough locality in a state-space graph so that every duplicate-detection scope fits in internal memory, it is unnecessary to ever delay duplicate detection, and structured duplicate detection can manage external memory by itself. But if any duplicate-detection scope does not fit in internal memory (perhaps due to lack of sufficient local structure in the graph), structured duplicate detection by itself is not sufficient. Since delayed duplicate detection does not have a minimum memory requirement, it can be used in this case. Structured duplicate detection can be used together with delayed duplicate detection to improve its performance, by leveraging graph locality as much as possible.

Acknowledgments We thank the anonymous reviewers for helpful comments. This work was supported in part by NSF grant IIS-9984952 and NASA grant NAG-2-1463.

References

Aggarwal, A., and Vitter, J. 1988. The input/output complexity of sorting and related problems. *Comm. of the ACM* 31(9):1116–27.

Belady, L. 1966. A study of replacement algorithms for virtual storage. *IBM Systems Journal* 5:78–101.

Edelkamp, S., and Schrödl, S. 2000. Localizing A*. In *Proc. of the 17th National Conference on Artificial Intelligence*, 885–890.

Holte, R.; Mkadmi, T.; Zimmer, R.; and MacDonald, A. 1996. Speeding up problem solving by abstraction: A graph-oriented approach. *Artificial Intelligence* 85(1–2):321–361.

Klein, D., and Manning, C. 2003. Factored A* search for models over sequences and trees. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 1246–1251.

Korf, R., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the 17th National Conference on Artificial Intelligence*, 910–916.

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

Korf, R. 2003a. Breadth-first frontier search with delayed duplicate detection. In *Proceedings of the Workshop on Model Checking and Artificial Intelligence at IJCAI-03*, 87–92.

Korf, R. 2003b. Delayed duplicate detection: Extended abstract. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 1539–1541.

Mehlhorn, K., and Meyer, U. 2002. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms*, 723–735.

Munagala, K., and Ranade, A. 1999. I/O-complexity of graph algorithms. In *Proceedings of the 10th Symposium on discrete algorithms*, 687–694. ACM-SIAM.

Sleator, D., and Tarjan, R. 1985. Amortized efficiency of list update and paging rules. *Communications of the ACM* 28:202–8.

Thompson, J.; Plewniak, F.; and Poch, O. 1999. BAliBASE: A benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics* 15(1):87–88.

Zhou, R., and Hansen, E. 2003. Sweep A*: Space-efficient heuristic search in partially ordered graphs. In *Proc. of 15th IEEE International Conf. on Tools with Artificial Intelligence*, 427–434.

Zhou, R., and Hansen, E. 2004. Breadth-first heuristic search. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling*.