

## Edge Partitioning in External-Memory Graph Search

**Rong Zhou**

Palo Alto Research Center  
3333 Coyote Hill Road  
Palo Alto, CA 94304  
rzhou@parc.com

**Eric A. Hansen**

Dept. of Computer Science and Eng.  
Mississippi State University  
Mississippi State, MS 39762  
hansen@cse.msstate.edu

### Abstract

There is currently much interest in using external memory, such as disk storage, to scale up graph-search algorithms. Recent work shows that the local structure of a graph can be leveraged to substantially improve the efficiency of external-memory graph search. This paper introduces a technique, called edge partitioning, which exploits a form of local structure that has not been considered in previous work. The new technique improves the scalability of structured approaches to external-memory graph search, and also guarantees the applicability of these approaches to any graph-search problem. We show its effectiveness in an external-memory graph-search algorithm for domain-independent STRIPS planning.

### 1 Introduction

Breadth-first search,  $A^*$ , and related graph-search algorithms store generated nodes in memory in order to be able to detect duplicates and prevent node regeneration. The scalability of these graph-search algorithms can be dramatically increased by storing nodes in external memory, such as disk storage. Because random access of disk is several orders of magnitude slower than internal memory, external-memory graph-search algorithms use duplicate-detection strategies that serialize disk access in a way that minimizes disk I/O.

A widely-used approach to external-memory graph search is *delayed duplicate detection* [Stern and Dill, 1998; Korf, 2004; Edelkamp *et al.*, 2004]. In its original and simplest form, delayed duplicate detection expands a set of nodes (e.g., the nodes on the search frontier) without checking for duplicates, stores the generated nodes (including duplicates) in a disk file, and eventually removes the duplicates by sorting the file. In keeping with its use by theoretical computer scientists in analyzing the complexity of external-memory graph search [Munagala and Ranade, 1999], delayed duplicate detection makes no assumptions about the structure of the search graph (except that it is undirected and unweighted).

Recent work shows that the performance of external-memory graph search can be significantly improved by exploiting the structure of a graph in order to localize memory references. Zhou and Hansen (2004; 2005; 2006b) pro-

pose a technique called *structured duplicate detection* that exploits local structure that is captured in an abstract representation of the graph. For graphs with sufficient local structure, structured duplicate detection outperforms delayed duplicate detection because it never generates duplicates, even temporarily, and thus has lower overhead and reduced complexity. Korf and Schulze (2005) show how to improve the performance of delayed duplicate detection by using similar local structure to reduce the delay between generation of duplicates and their eventual detection and removal.

Although approaches that exploit a graph's local structure are very effective, they depend on a graph-search problem having the appropriate kind of local structure, and a sufficient amount of it, in order to be effective. Thus one can legitimately question whether these approaches will be equally effective for all graphs, or effective at all for some graphs. In this paper, we introduce a technique that exploits a different kind of local structure than is exploited in previous work. The kind of local structure exploited by this technique is in some sense *created* by the way the algorithm expands nodes, in particular, by use of a novel form of incremental node expansion. The new technique, called *edge partitioning*, can localize memory references in any graph, no matter what its structure – even a fully-connected graph. Moreover, the way this new technique localizes memory references complements the kind of local graph structure that is exploited by previous approaches. This allows the new technique to be combined with previously-developed approaches in order to create a more powerful external-memory graph-search algorithm. In this paper, we focus on how to use edge partitioning to improve the performance of structured duplicate detection. We implement it in an external-memory graph-search algorithm for domain-independent STRIPS planning that uses structured duplicate detection, and show that it greatly improves scalability. At the close of the paper, we discuss how edge partitioning can also be used to exploit local structure in delayed duplicate detection.

### 2 Structured duplicate detection

Structured duplicate detection (SDD) [Zhou and Hansen, 2004] is an approach to external-memory graph search that leverages local structure in a graph to partition stored nodes between internal memory and disk in such a way that duplicate detection can be performed immediately, during node ex-

pansion, and no duplicates are ever generated.

The local structure that is leveraged by this approach is revealed by a state-space projection function that 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  $x$ . One way to create a state-space projection function is by ignoring the value of some state variables. For example, if we ignore the positions of all tiles in the Eight Puzzle and consider only the position of the “blank,” we get an abstract state space that has only nine abstract states, one corresponding to each possible position of the blank.

Given a state-space graph and state-space projection function, an *abstract state-space graph* is constructed as follows. The set of nodes in the abstract graph, called the *abstract nodes*, corresponds to the set of abstract states. An abstract node  $y'$  is a successor of an abstract node  $y$  if and only if there exist two states  $x'$  and  $x$  in the original state space, such that (1)  $x'$  is a successor of  $x$ , and (2)  $x'$  and  $x$  map to  $y'$  and  $y$ , respectively, under the state-space projection function.

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).

In structured duplicate detection, stored nodes in the original search graph are divided into “*nblocks*” with each *nblock* corresponding to a set of nodes that maps to the same abstract node. Given this partition of stored nodes, structured duplicate detection uses the concept of *duplicate-detection scope* to localize memory references in duplicate detection. The *duplicate-detection scope* of a node  $x$  in the original search graph is defined as all stored nodes (or equivalently, all *nblocks*) that map to the successors of the abstract node  $y$  that is the image of node  $x$  under the projection function. In the Eight Puzzle example, the duplicate-detection scope of nodes that map to abstract node  $B_0$  consists of nodes that map to abstract node  $B_1$  and those that map to abstract node  $B_3$ .

The concept of duplicate-detection scope allows a search algorithm to check duplicates against a fraction of stored nodes, and still guarantee that all duplicates are found. An external-memory graph search algorithm can use RAM to store *nblocks* within the current duplicate-detection scope, and use disk to store other *nblocks* when RAM is full.

SDD is designed to be used with a search algorithm that expands a set of nodes at a time, such as breadth-first search, where the order in which nodes in the set are expanded can be adjusted to minimize disk I/O. SDD’s strategy for minimizing disk I/O is to order node expansions such that changes of duplicate-detection scope occur as infrequently as possible, and, when they occur, they involve change of as few *nblocks* as possible. When RAM is full, *nblocks* outside the current duplicate-detection scope are flushed to disk. Writing the *least-recently used nblocks* to disk is one way to select which *nblocks* to write to disk. When expanding nodes in a different *nblock*, any *nblocks* in its duplicate-detection scope that are stored on disk are swapped into RAM.

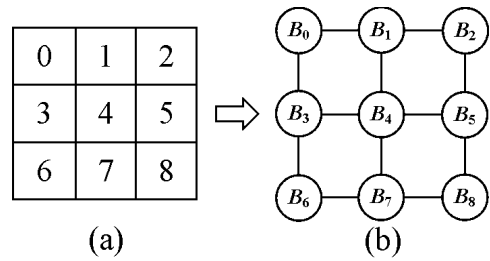


Figure 1: Panel (a) shows all possible positions of the blank for the Eight Puzzle. Panel (b) shows an abstract state-space graph that is created by the state-space projection function that considers the position of the “blank” only.

### 3 Edge Partitioning

The kind of local structure that is exploited by SDD is captured in an abstract state-space graph when the maximum outdegree of any abstract node is small relative to the total number of abstract nodes. The largest outdegree of any abstract node reflects the largest duplicate-detection scope, and this in turn determines the internal-memory requirements of the search algorithm. Experiments in several domains suggest that this form of local structure is present in many search problems [Zhou and Hansen, 2004; 2005; 2006b]. However it is present in varying degrees, and there is no guarantee that it is present in every search problem. Moreover, the degree to which it is present can affect the degree of scalability that is possible.

This motivates the development of a technique that makes SDD effective regardless of whether, and to what degree, this kind of local structure is present. In fact, the new technique we introduce is effective even if the abstract state-space graph is fully-connected, and thus captures no local structure at all. The idea behind this technique is that the local structure exploited by SDD can be *created*, in some sense, by expanding nodes incrementally, which means generating only some of the successors of a node at a time, and generating the other successors later. Incremental node expansion makes it possible to partition the original duplicate-detection scope into smaller duplicate-detection scopes, and this can significantly reduce the internal memory requirements of the algorithm.

In the original form of SDD, the duplicate-detection scope of a node being expanded is defined as all stored nodes that map to any abstract node that is a successor of the abstract node that is the image of the node being expanded. Thus the largest duplicate-detection scope reflects the largest number of successors of a node in the abstract graph. But this assumes that all successors are generated at the same time. If nodes are expanded incrementally, it is possible to subdivide this duplicate-duplicate scope into smaller scopes, one for each pair of abstract node and successor abstract node, or, equivalently, one for each outgoing abstract edge.

This is the key idea of edge partitioning. It considers a set of nodes that map to a particular abstract node, and a particular outgoing abstract edge, and only applies the operators that correspond to the abstract edge, in order to generate only the successor nodes that correspond to the successor abstract node along that edge. As a result, in edge partitioning,

the duplicate-detection scope consists of only a single  $n$ block corresponding to the single abstract node that is the successor of an abstract edge. At a later point in the algorithm, a different outgoing abstract edge is considered, and a different set of operators are applied to the same set of nodes, in order to generate additional successor nodes with potential duplicates in a different  $n$ block. Eventually, all operators are applied to a set of nodes and they become fully expanded. Note that full expansion of a node now requires a sequence of incremental expansions.

### 3.1 Operator grouping

The state-space projection function and abstract state space graph used by SDD with edge partitioning are the same as for SDD in its original form. But since nodes are expanded incrementally, and only one abstract edge is considered at a time, it is now important to identify which operators are associated with each abstract edge, in order to know which successors to generate. We refer to this annotation of the edges of the abstract graph as *operator grouping*. We point out that an “operator” here refers to an *instantiated* (or *grounded*) operator. For example, the Eight Puzzle has a total of 192 grounded operators, even though there are only four (left, right, up, and down) operators prior to instantiation.

Operator grouping is built on top of state abstraction. Let  $\mathcal{O}$  be the set of all instantiated operators of a search problem. An operator  $o \in \mathcal{O}$  is applicable to an abstract node  $y$  if and only if there exists a state  $x$  in the original state space, such that (a)  $o$  is applicable to  $x$ , and (b)  $x$  maps to  $y$ . Consider the Eight Puzzle. There are  $2 \times 8 = 16$  operators that are applicable to abstract node  $B_0$ , because the blank, when located at the top-left corner of the puzzle board, can move either right ( $B_0 \rightarrow B_1$ ) or down ( $B_0 \rightarrow B_3$ ), and each move has 8 different instantiations, depending on which tile of the Eight Puzzle is moved into the blank position. Similarly, each of the abstract nodes  $B_2, B_6$ , and  $B_8$  has 16 applicable operators. Abstract nodes  $B_1, B_3, B_5$ , and  $B_7$  each have  $3 \times 8 = 24$  applicable operators, and abstract node  $B_4$  has  $4 \times 8 = 32$  applicable operators.

Once the set of applicable operators for each abstract node is determined, operator grouping identifies, for each applicable operator, the abstract edge it is associated with. An *abstract edge*  $(y, y')$  is an edge in the abstract graph that connects a pair of abstract nodes  $y$  and  $y'$ , if and only if  $y'$  is a successor of  $y$ . We refer to  $y$  ( $y'$ ) as the *source* (*destination*) of abstract edge  $(y, y')$ .

Let  $\mathcal{O}_y$  be the set of operators applicable to abstract node  $y$ . An operator  $o \in \mathcal{O}_y$  is associated with an abstract edge  $(y, y')$  if and only if there exists two states  $x$  and  $x'$  in the original state space, such that (1)  $o$  is applicable to  $x$ , (2)  $x'$  is the resulting state after applying  $o$  to  $x$ , and (3)  $x$  and  $x'$  map to  $y$  and  $y'$ , respectively. For operators with deterministic effects, it is easy to see that for every  $o \in \mathcal{O}_y$ , there is a unique abstract edge  $(y, y')$  that  $o$  is associated with. Essentially, there is a many-to-one mapping from the operator space to the abstract-edge space.

To exploit local structure in the operator space, edge partitioning uses operator grouping to divide the set of applicable operators  $\mathcal{O}_y$  for abstract node  $y$  into operator groups, one

for each successor of  $y$  in the abstract graph. An *operator group*  $\mathcal{O}_{y,y'}$  is a subset of  $\mathcal{O}_y$  that consists of all the operators that are associated with abstract edge  $(y, y')$ . Note that  $\mathcal{O}_{y,y'} \cap \mathcal{O}_{y,y''} = \emptyset$  for all  $y' \neq y''$ , and

$$\bigcup_{y' \in \text{successors}(y)} \mathcal{O}_{y,y'} = \mathcal{O}_y,$$

where  $\text{successors}(y)$  is the set of successors of  $y$  in the abstract graph.

Although the technique of operator grouping is presented here in the context of searching implicitly-represented graphs (i.e., graphs represented by a start state and a set of operators for generating successors), it should be clear that the same technique applies with little modification to searching explicitly-represented graphs (i.e., graphs represented by a set of vertices and a set of edges).

### 3.2 Edge-partitioned duplicate-detection scope

The idea of edge partitioning for SDD is to subdivide the duplicate-detection scope into smaller scopes, one for each abstract edge, and use only the operator group that is associated with the abstract edge to generate successors at a time. This leads to the concept of duplicate-detection scope for an abstract edge, which is defined as follows.

**Definition 1** *The duplicate-detection scope for an abstract edge is the set of stored nodes that map to the destination of the abstract edge.*

The duplicate-detection scope for an abstract edge is guaranteed to contain only nodes that map to a *single* abstract node, regardless of the structure of the abstract graph. The following theorem follows from the definition.

**Theorem 1** *The duplicate-detection scope for an abstract edge contains all stored duplicates of the successors generated by applying its corresponding operator group to the set of nodes that map to the source of the abstract edge.*

Theorem 1 guarantees that edge partitioning only needs to store a single  $n$ block in RAM, in order to catch all the duplicates that could be generated, even in the worst case. This works in the following way. For each abstract edge  $(y, y')$ , edge partitioning uses operators  $o \in \mathcal{O}_{y,y'}$  to generate successors for nodes that map to abstract node  $y$ . After edge partitioning has expanded these nodes using one operator group, it uses a different operator group  $\mathcal{O}_{y,y''}$  to generate successors for the same  $n$ block, until all operator groups have been used in generating successors for the  $n$ block. Then it chooses a different  $n$ block to expand next.

Because not all successors are generated by edge partitioning when a node is expanded, we call a node expansion in edge partitioning an *incremental expansion*. Nodes eventually become fully expanded, once all operators are applied.

### 3.3 Example

We use the following example to illustrate how edge partitioning works in SDD. Let  $x_i$  be a search node that represents a state of the Eight Puzzle with the blank located at position  $i$  as shown in Figure 1(a). Suppose there are only two stored

nodes  $\{a_0, b_0\}$  that map to abstract node  $B_0$  shown in Figure 1(b). Let  $\{a_1, a_3\}$  and  $\{b_1, b_3\}$  be the successors of  $a_0$  and  $b_0$ , respectively. (The subscript encodes the position of the blank.) When edge partitioning expands nodes  $a_0$  and  $b_0$ , it first uses operators  $o \in \mathcal{O}_{B_0, B_1}$ . This corresponds to moving the blank to the right, to generate the first two successor nodes  $a_1$  and  $b_1$ . Note that only nodes that map to abstract node  $B_1$  need to be stored in RAM when  $a_1$  or  $b_1$  is being generated. Next, edge partitioning uses operators  $o \in \mathcal{O}_{B_0, B_3}$ , which correspond to moving the blank down, to generate the third and fourth successor nodes  $a_3$  and  $b_3$ . This time, only nodes that map to abstract node  $B_3$  need to be stored in RAM.

## 4 Implementation

Before discussing some strategies for implementing SDD with edge partitioning in an efficient way, we review the key steps in implementing SDD.

First, before the search begins, SDD uses a state-state projection function to create an abstract graph that (hopefully) captures local structure in the original search graph. The state-space projection function partitions stored nodes into  $n$ blocks (one for each node in the abstract graph) that can be moved between RAM and disk, and so each  $n$ block must be able to fit in RAM. The state-space projection function can be hand-crafted or automatically generated, as described in [Zhou and Hansen, 2006b].

Like delayed duplicate detection, SDD is designed to be used as part of a search algorithm that expands a set of nodes at a time, such as the frontier nodes in breadth-first search. The idea of SDD is to expand these nodes in an order that minimizes disk I/O. This is accomplished by expanding nodes with the same duplicate-detection scope consecutively, and minimizing changes of duplicate-detection scope during expansion of all nodes. A simple and effective heuristic is to expand nodes in order of  $n$ block, with  $n$ blocks ordered according to a breadth-first traversal of the abstract graph. When RAM is full, SDD needs to decide which  $n$ blocks to move from RAM to disk. A simple and effective heuristic is to write the *least-recently used*  $n$ blocks to disk.

SDD with edge partitioning uses a similar strategy of trying to minimize changes of duplicate-detection scope during expansion of a set of nodes. The difference is that it considers the duplicate-detection scope for an abstract edge, and this requires incremental node expansion. A simple and effective heuristic is to apply operators to nodes in order of  $n$ block, and, for each  $n$ block, in order of outgoing abstract edge.

We next consider some ways to improve performance. To reduce the overhead of operator grouping, our implementation uses a lazy approach in which operator grouping for an abstract node is only computed immediately before the first time a node that maps to it is expanded. Because there could be a number of abstract nodes that do not have any nodes that map to them during search, this approach avoids the overhead of operator grouping for these abstract nodes. We have observed that the effectiveness of this approach tends to increase with the size of the abstract graph.

Our implementation also uses a lazy approach to reading

nodes from disk. Upon switching to a duplicate-detection scope that consists of nodes stored on disk, our implementation does not read these nodes from disk immediately. Instead, it waits until the first time a node is generated. The reason for this is that when a single operator group is used to generate successors for nodes in an  $n$ block, it may not generate any successor node, if (a) the nodes to which the operators in the group are applicable have not yet been generated, or (b) the generated successor nodes have an  $f$ -cost greater than an upper bound used in branch-and-bound search. The lazy approach avoids the overhead of reading nodes from disk (in order to setup the duplicate-detection scope in RAM) if no successors are generated by an operator group for an  $n$ block.

As previously discussed, SDD needs to decide which  $n$ blocks to move from RAM to disk, when RAM is full. Except for the  $n$ blocks that make up the current duplicate-detection scope, any  $n$ blocks can potentially be flushed to disk. But this means if an  $n$ block does not include itself as part of its own duplicate-detection scope, it may be flushed to disk even when its nodes are being expanded. While this is allowed in our implementation, it should be avoided as much as possible for efficiency reasons. We make two simple modifications to the least-recently used algorithm to ensure this. First, instead of updating the time stamp of an  $n$ block every time it is accessed, its time stamp is only updated when (a) the current duplicate-detection scope changes and (b) the  $n$ block is the next to be expanded or is part of the new scope. This also simplifies the maintenance of the clock, which needs no updates until the duplicate-detection scope changes. The second modification is that instead of moving forward the clock by one clock tick when the duplicate-detection scope changes, our algorithm advances it by two clock ticks. Then the time stamp of the to-be-expanded  $n$ block is set to one clock tick earlier than the new clock time. Finally, the time stamps of all the  $n$ blocks within the new duplicate-detection scope are updated to the new clock time. As a result, if the  $n$ block to be expanded next does not belong to the new duplicate-detection scope, it is the last to be flushed to disk, since its time stamp is more recent than any other flushable  $n$ block and earlier than any non-flushable  $n$ block, which has a time stamp equal to the current clock time.

Finally, recall that edge partitioning expands nodes in a single  $n$ block multiple times, one for each operator group. This affects the strategy with which to remove nodes stored on disk for the currently-expanding  $n$ block. While the simplest strategy is to remove these nodes from disk as soon as they are swapped into RAM, it may incur extra overhead if these nodes must be written back to disk shortly after, in order to make room for newly-generated nodes. Note that nodes in the currently-expanding  $n$ block do not change as long as the operator group used to generate the successors is not associated with an abstract edge whose source and destination are the same (i.e., a “self loop”). Because self loops are easy to detect, our implementation postpones the removal of nodes stored on disk for the currently-expanding  $n$ block until a “self loop” operator group, which (if any) is always the last operator group applied to an  $n$ block in our implementation, is used to expand the  $n$ block.

Problem	SDD				SDD + Edge Partitioning			
	RAM	Disk	Exp	Secs	RAM	Disk	Increment Exp	Secs
depots-7	2,662,253	9,524,314	16,801,412	342	742,988	10,705,324	191,263,008	460
blocks-16	3,194,703	5,527,227	18,075,779	387	1,069,901	6,997,695	395,738,702	823
trucks-9	7,085,621	20,888,173	54,348,820	3,995	953,642	26,106,623	590,454,354	5,982
storage-12	5,520,445	230,451,662	282,931,334	9,141	891,585	221,072,558	2,914,075,502	9,071
freecell-4	11,447,191	114,224,688	208,743,830	14,717	2,218,545	122,033,806	4,206,478,527	22,960
elevator-15	1,540,657	126,194,100	430,804,933	16,487	61,900	127,685,640	12,775,795,015	60,229
gripper-10	13,736,629	328,271,632	2,007,116,254	35,052	1,069,901	340,780,440	13,366,646,793	36,377
logistics-9	5,159,767	540,438,586	1,138,753,911	41,028	742,988	544,285,237	9,856,519,138	49,004
driverlog-13	49,533,873	2,147,482,093	2,766,380,501	108,051	4,299,806	2,147,483,535	38,879,000,039	122,877
satellite-7	12,839,146	571,912,557	838,488,709	160,687	429,971	584,308,516	28,532,162,097	129,608
trucks-10	-	-	-	-	7,085,621	231,515,502	6,282,870,888	70,963
depots-10	-	-	-	-	41,278,228	1,373,427,385	26,548,426,038	90,644

Table 1: Comparison of structured duplicate detection (SDD) with and without using edge partitioning on STRIPS planning problems. Columns show peak number of nodes stored in RAM (RAM), peak number of nodes stored on disk (Disk), number of full node expansions (Exp), number of incremental node expansions (Increment Exp), and running time in CPU seconds (Secs). A ‘-’ symbol indicates that the algorithm cannot solve the problem within 2 GB of RAM.

## 5 Computational results

We implemented SDD with edge partitioning in a domain-independent STRIPS planner that uses as its underlying search algorithm breadth-first heuristic search [Zhou and Hansen, 2006a]. The reason for using breadth-first heuristic search is that it uses internal memory very efficiently. Building SDD with edge partitioning on top of it improves the overall efficiency of search by limiting the need to access disk.

Our search algorithm uses regression planning to find optimal sequential plans. As an admissible heuristic, it uses the max-pair heuristic [Haslum and Geffner, 2000]. We tested our external-memory STRIPS planner in ten different domains from the biennial planning competition, including two domains (*trucks* and *storage*) from the most recent competition. Experiments were performed on an AMD Operton 2.4 GHz processor with 4 GB of RAM and 1 MB of L2 cache.

Table 1 compares the performance of SDD with and without edge partitioning. These problems are among the largest in each of the ten planning domains that SDD with edge partitioning can solve without either (a) using more than 2 GB of RAM or (b) taking more than 2 CPU days of running time. Two problems (*trucks-10* and *depots-10*) can only be solved within these limits using edge partitioning. For these two domains, the table also includes the largest instances that can be solved without edge partitioning. Both versions of SDD use the same state-space projection function.

Table 1 shows a couple of interesting things. First, it shows that edge partitioning can reduce the internal-memory requirements of SDD by an average factor of 11 times for these planning problems. In doing so, it only increases the peak number of nodes stored on disk by about 7.5%. Second, it shows that the overhead that results from using an incremental approach to expanding nodes is rather inexpensive. Although on average there are 16.8 times as many incremental expansions when edge partitioning is used as there are full expansions when it is not, this only increases running time by 53% on average. Note that the extra time taken by edge partitioning includes time for operator grouping.

Indirectly, the table shows roughly how much internal

memory is saved by using SDD with edge partitioning instead of A\*. The number of full node expansions in SDD gives an estimate of how many nodes A\* would need to store in order to solve the problem, since A\* has to store every node it expands, and breadth-first heuristic search (with an optimal upper bound) expands roughly the same number of nodes as A\*, disregarding ties. Based on the number of full node expansions shown in Table 1, A\* would need, on average, at least 1,340 times more internal memory to solve these problems than breadth-first heuristic search with SDD and edge partitioning. Because this estimate ignores the memory needed by A\* to store the Open list, which is usually larger than the Closed list, it is actually a considerable *underestimate*.

As the results show, SDD without edge partitioning is already very effective in solving these STRIPS planning problems, which indicates that these search problems contain a great deal of the kind of local structure that SDD exploits. This means that these problems actually present a serious challenge for edge partitioning, which must identify *additional* structure that can be exploited to reduce internal memory requirements even further. For search problems for which SDD without edge partitioning is less effective, SDD with edge partitioning is likely to reduce internal memory requirements by a much larger ratio.

Since edge partitioning is effective even when the abstract graph used by SDD does not capture any local structure, one might wonder whether such local structure is useful anymore. Although it is no longer needed to reduce internal memory requirements, it is still useful in reducing time complexity. First of all, if a problem can be solved by SDD without edge partitioning, the time overhead of incremental node expansion can be avoided. If edge partitioning is used, then the more local structure (i.e., the fewer successor nodes of an abstract node in the abstract graph), the fewer incremental expansions are needed before a node is fully expanded, and the overhead of incremental node expansion is reduced. The results in Table 1 show that edge partitioning reduces the amount of internal memory needed in exchange for an increase in average running time, although the actual increase is still fairly modest.

## 6 Application to delayed duplicate detection

So far, we have described how to use edge partitioning in SDD, where it improves scalability by reducing internal-memory requirements. In particular, it reduces the proportion of generated nodes that need to be stored in internal memory at any one time in order to ensure detection of all duplicates. As we now show, edge partitioning can also be used in a form of delayed duplicate detection (DDD) that uses local structure to reduce the delay between generation of nodes and eventual detection and removal of duplicates. This has the advantage of reducing the disk storage requirements of DDD. We begin with a review of DDD and then describe how edge partitioning can enhance its performance.

### 6.1 Delayed duplicate detection

DDD alternates between two phases; successor generation and duplicate elimination. Depending on how duplicates are eliminated, there are two forms of DDD, as follows.

#### Sorting-based DDD

The first algorithms for external-memory graph search used sorting-based DDD [Stern and Dill, 1998; Munagala and Ranade, 1999; Korf, 2004; Edelkamp *et al.*, 2004]. Sorting-based DDD takes a file of nodes on the search frontier, (e.g., the nodes in the frontier layer of a breadth-first search graph), generates their successors and writes them to another file without checking for duplicates, sorts the file of generated nodes by the state representation so that all duplicate nodes are adjacent to each other, and scans the file to remove duplicates. The I/O complexity of this approach is dominated by the I/O complexity of external sorting, and experiments confirm that external sorting is its most expensive step.

#### Hash-based DDD

Hash-based DDD [Korf and Schultze, 2005] is a more efficient form of DDD. To avoid the I/O complexity of external sorting in DDD, it uses two orthogonal hash functions. During node expansion, successor nodes are written to different files based on the value of the first hash function, which means all duplicates are mapped to the same file. Once a file of successor nodes has been fully generated, duplicates can be removed from it. To avoid the overhead of external sorting in removing duplicates, a second hash function maps all duplicates to the same location of a hash table, which accomplishes by hashing what otherwise would require sorting. Since the hash table corresponding to the second hash function must fit in internal memory, hash-based DDD has a minimum internal-memory requirement that corresponds to the largest set of *unique* nodes in any file. Thus, this approach requires some care in designing the first hash function to make sure it does not map too many unique nodes to a single file.

Although hash-based DDD in its original form does not depend on, or leverage, the structure of a graph, an important improvement, called “interleaving expansion and merging” [Korf and Schultze, 2005], does. It works as follows. The nodes on the search frontier are stored in multiple files, called “parent files,” depending on the first hash function, and the successor nodes that are generated when the nodes in the parent files are expanded are also stored in multiple files,

called “child files.” Instead of waiting until all parent files at a given depth are expanded before merging any child files at the next depth to remove duplicates, a child file is merged as soon as all parent files that could possibly add successor nodes to it have been expanded. In other words, interleaving expansion and merging makes it possible to remove duplicates early. Because DDD generates duplicates and stores them on disk before eventually removing them, the technique of “interleaving expansion and merging” reduces the amount of extra disk storage needed by DDD. In fact, in the best case, it can reduce the amount of extra disk storage need by DDD by a factor of  $b$ , where  $b$  is the average branching factor, although the actual reduction may be less.

To allow “interleaving expansion and merging,” the first hash function must be designed in such a way that it captures local structure in the search graph. In particular, a child file must only contain successor nodes generated from a small number of parent files. A generic hash function cannot be used for this since it will typically hash nodes uniformly across *all* files. Instead, a problem-specific hash function that captures local structure must be designed. In the following, we explain how this enhancement of hash-based DDD exploits and, thus, depends on the local structure of a graph, and how it can further exploit edge partitioning.

### 6.2 Edge partitioning in DDD

To see how edge partitioning can be used to improve the performance of hash-based DDD, we first consider how the kind of local structure exploited by “interleaving expansion and merging” is related to the kind of local structure exploited by SDD. As previously pointed out, hash-based DDD requires a problem-specific hash function, and the “interleaving expansion and merging” technique is only effective when this hash function maps nodes to files in such a way that the nodes in one file (the child file) are generated from nodes in only a small number of other files (its parent files). In fact, these relationships can be represented by an abstract state-space graph in which abstract nodes correspond to files, and an abstract edge is present when nodes in one file have successor nodes in the other file. This should make it clear that the first hash function used by hash-based DDD is actually a state-space projection function, and, for “interleaving expansion and merging” to be effective, this hash function should capture the same kind of local structure that is exploited by SDD. The following concept will help make this more precise.

**Definition 2** *The predecessor-expansion scope of a child file for an abstract node  $y'$  under a state-space projection function  $\Pi$  corresponds to the union of nodes in the parent files for abstract nodes  $y \in \text{predecessors}(y')$ , that is,*

$$\bigcup_{y \in \text{predecessors}(y')} \Pi^{-1}(y),$$

where  $\text{predecessors}(y')$  is the set of predecessors of  $y'$  in the abstract graph, and  $\Pi^{-1}(y)$  is the set of nodes in the parent file for an abstract node  $y$ .

An important property of the predecessor-expansion scope is that it is guaranteed to contain all stored predecessors of nodes in a child file, which leads to the following theorem.

**Theorem 2** *Merging duplicate nodes after expanding all nodes in the predecessor-expansion scope of a child file is guaranteed to eliminate all duplicates that could be generated for the child file.*

The concept of predecessor-expansion scope lets us identify the local graph structure needed by “interleaving expansion and merging” in a principled way, and relate it to the kind of local structure exploited by SDD. For undirected graphs, they are exactly the same, since the set of predecessors of an abstract node always coincides with the set of its successors. For directed graphs, they may or may not be the same.

This analysis also lets us specify a condition under which “interleaving expansion and merging” will not be effective, at least by itself. When the abstract graph is fully connected, the predecessor-expansion scope of any child file is the entire set of parent files. This means the earliest time a child file can be merged is when all nodes at the current depth have been expanded, which prevents the application of interleaving expansion and merging. We are now ready to show how edge partitioning allows “interleaving of expansion and merging” to be effective *even in this case*.

The idea is to force nodes within the predecessor-expansion scope of a child file to generate successors *only* for that child file alone, without generating successor nodes for other child files at the same time. This can be achieved as follows. Let  $y'$  be the abstract node that corresponds to the child file that is the target of merging. To merge duplicate nodes in this file as early as possible, edge partitioning only uses operators  $o \in \mathcal{O}_{y,y'}$  to generate the successors of nodes in the parent files for abstract nodes  $y \in \text{predecessors}(y')$ .

Once all nodes in the parent files have generated their successors for this child file, merging can take place as usual. The advantage of edge partitioning is that it saves external memory by not generating (possibly duplicate) successors for any other child files before merging is performed. After merging duplicates in a child file, edge partitioning picks another child file as the next merging target, until all the child files have been merged. It can be shown that by the time the last child file is merged, edge partitioning must have used all the operators to generate all the successor nodes for the current depth. By doing so in an incremental way, it ensures that the local structure needed by the “interleaving expansion and merging” technique is always present.

Although we do not present empirical results for edge partitioning in DDD, our analysis helps to clarify the relationship between SDD and hash-based DDD with interleaving of expansion and merging. Both exploit the same local structure of a graph, and thus the performance of both can be enhanced by using edge partitioning in a similar way.

## 7 Conclusion

We have introduced a technique, called edge partitioning, that improves the scalability of structured approaches to external-memory graph search by using a strategy of incremental node expansion to localize memory references in duplicate detection. Results show that it significantly reduces the internal memory requirements of structured duplicate detection (SDD). Moreover, it is guaranteed to be effective regardless

of the structure of the graph, and this guarantees that SDD can be applied to any search problem. Finally, we have shown that it can also be used to reduce the amount of disk storage needed by delayed duplicate detection.

There are a number of directions for future work. One possibility is to vary the degree of incremental expansion. For example, instead of using one operator group at a time, edge partitioning can use multiple (but not all) operator groups to generate successor nodes at a time. If enough internal memory is available, this can reduce the overall number of (incremental) expansions. With edge partitioning, we now have two options to reduce the internal-memory requirements of SDD. We can either increase the granularity of the state-space projection function, or we can use edge partitioning. Which option is better under what circumstances is an interesting question, and the answer is likely to help us understand how to best trade off internal-memory requirements with the number of disk I/O operations needed by SDD.

## References

- [Edelkamp *et al.*, 2004] S. Edelkamp, S. Jabbar, and S. Schrödl. External A\*. In *Proc. of the 27th German Conf. on Artificial Intelligence*, pages 226–240, 2004.
- [Haslum and Geffner, 2000] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Proc. of the 5th International Conference on AI Planning and Scheduling*, pages 140–149, 2000.
- [Korf and Schultze, 2005] R. Korf and P. Schultze. Large-scale parallel breadth-first search. In *Proc. of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 1380–1385, 2005.
- [Korf, 2004] R. Korf. Best-first frontier search with delayed duplicate detection. In *Proc. of the 19th National Conf. on Artificial Intelligence (AAAI-04)*, pages 650–657, 2004.
- [Munagala and Ranade, 1999] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. of the 10th Symposium on discrete algorithms*, pages 687–694, 1999.
- [Stern and Dill, 1998] U. Stern and D. Dill. Using magnetic disk instead of main memory in the mur(phi) verifier. In *Proc. of the 10th International Conference on Computer-Aided Verification*, pages 172–183, 1998.
- [Zhou and Hansen, 2004] R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *Proc. of the 19th National Conference on Artificial Intelligence (AAAI-04)*, pages 683–688, 2004.
- [Zhou and Hansen, 2005] R. Zhou and E. Hansen. External-memory pattern databases using structured duplicate detection. In *Proc. of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 1398–1405, 2005.
- [Zhou and Hansen, 2006a] R. Zhou and E. Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4-5):385–408, 2006.
- [Zhou and Hansen, 2006b] R. Zhou and E. Hansen. Domain-independent structured duplicate detection. In *Proc. of the 21st National Conference on Artificial Intelligence (AAAI-06)*, pages 1082–1087, 2006.