Domain-Independent Structured Duplicate Detection

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

The scalability of graph-search algorithms can be greatly extended by using external memory, such as disk, to store generated nodes. We consider structured duplicate detection, an approach to external-memory graph search that limits the number of slow disk I/O operations needed to access search nodes stored on disk by using an abstract representation of the graph to localize memory references. For graphs with sufficient locality, structured duplicate detection outperforms other approaches to external-memory graph search. We develop an automatic method for creating an abstract representation that reveals the local structure of a graph. We then integrate this approach into a domain-independent STRIPS planner and show that it dramatically improves scalability for a wide range of planning problems. The success of this approach strongly suggests that similar local structure can be found in many other graph-search problems.

Introduction

The scalability of graph-search algorithms such as breadth-first search, Dijkstra's algorithm, and A*, is limited by the memory needed to store generated nodes in the Open and Closed lists, primarily for use in detecting duplicate nodes that represent the same state. Although depth-first search of a graph uses much less memory, its inability to detect duplicates leads to an exponential increase in time complexity that makes it ineffective for many graph-search problems. Recent work shows that the scalability of breadth-first and best-first search algorithms can be significantly improved without sacrificing duplicate detection by storing only the frontier nodes, using special techniques to prevent regeneration of closed nodes, and recovering the solution path by a divide-and-conquer technique (Korf *et al.* 2005; Zhou & Hansen 2006). But even with this approach, the amount of memory needed to store the search frontier eventually exceeds available internal memory. This has led to growing interest in external-memory graph-search algorithms that use disk to store the nodes that are needed for duplicate detection.

Because duplicate detection potentially requires comparing each newly-generated node to all stored nodes, it can lead to crippling disk I/O if the nodes stored on disk are accessed randomly. Two different approaches to performing duplicate detection efficiently in external-memory graph search have been proposed. *Delayed duplicate detection* (DDD) expands a set of nodes (e.g., the nodes on the frontier) without checking for duplicates, stores the generated nodes (including duplicates) in one or more disk files, and eventually remove duplicates by either sorting or hashing (Korf 2004; Edelkamp, Jabbar, & Schrödl 2004; Korf & Schultze 2005). The overhead for generating duplicates and removing them later is avoided in an approach called *structured duplicate detection* (SDD). It 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 expansion, and no duplicates are ever generated (Zhou & Hansen 2004). Although SDD is more efficient than DDD, it requires the search graph to have appropriate local structure.

In this paper, we develop an automatic approach for uncovering the local structure in a graph that can be leveraged by SDD. We integrate this approach into a domain-independent STRIPS planner, and show that, for a wide range of planning domains, it improves the scalability of the graph-search algorithm that solves the planning problems, often dramatically. We analyze the reasons for the success of this approach, and argue that similar local structure can be found in many other graph-search problems.

Memory-efficient graph search

Before reviewing approaches to external-memory graph search, we briefly review approaches to graph search that use internal memory as efficiently as possible. Since methods for duplicate detection in external-memory graph search are built on top of an underlying graph-search algorithm, the more efficiently the underlying search algorithm uses internal memory, the less it needs to access disk, and the more efficient the overall search.

Frontier search, introduced by Korf et al. (2005), is a memory-efficient approach to graph search that only stores nodes that are on the search frontier, uses special techniques to prevent regeneration of closed nodes, and recovers the solution path by a divide-and-conquer technique. It is a general approach to reducing memory requirements in graph search that can be used with A*, Dijkstra's algorithm, breadth-first search, and other search algorithms.

When frontier search or one of its variants is adopted, we previously showed that breadth-first branch-and-bound search can be more memory-efficient than A* in solving search problems with unit edge costs (Zhou & Hansen 2006). The reason for this is that a breadth-first frontier is typically smaller than a best-first frontier. We introduced the phrase *breadth-first heuristic search* to refer to this memory-efficient approach to breadth-first branch-and-bound search and a breadth-first iterative-deepening A* algorithm that is based on it. We adopt breadth-first heuristic search as the underlying search algorithm for the external-memory STRIPS planner developed in this paper, although we note that SDD and the automatic approach to abstraction that we develop can be used with other search strategies too.

External-memory graph search

In this section, we review structured duplicate detection and compare it to other approaches to duplicate detection in external-memory graph search.

Sorting-based delayed duplicate detection

The first algorithms for external-memory graph search used delayed duplicate detection (Stern & Dill 1998; Munagala & Ranade 1999; Korf 2004; Edelkamp, Jabbar, & Schrödl 2004). In its original and simplest form, delayed duplicate detection 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 bottleneck.

Structured duplicate detection

Sorting a disk file in order to remove duplicates is only necessary if duplicate detection is delayed. Structured duplicate detection (Zhou & Hansen 2004) detects duplicates as soon as they are generated by leveraging the local structure of a search graph. As long as SDD is applicable, it has been proved to have better I/O complexity than DDD. Moreover, the more duplicates are generated by DDD in the course of searching in a graph, the greater the relative advantage of SDD.

To identify the local structure in a graph that is needed for SDD, a state-space projection function is used to create an abstract state space in which each abstract state corresponds to a set of states in the original state space. Typically, the method of projection corresponds to a partial specification of the state. For example, if the state is defined by an assignment of values to state variables, an abstract state corresponds to an assignment of values to a subset of the state variables. In the abstract state-space graph created by the projection function, 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 projection function. The abstract state-space graph captures local structure in the problem if the maximum number of successors of any abstract node is small relative to the total number of abstract nodes. We refer to this ratio as the *locality* of the graph. The *duplicate-detection scope* of a node in the original search graph is defined as all stored nodes that map to the successors of the abstract node that is the image of the node under the projection function. The importance of this concept is that the search algorithm only needs to check for duplicates in the duplicate-detection scope of the node being expanded. Given sufficient locality, this is a small fraction of all stored nodes. An external-memory graph search algorithm uses RAM to store nodes within the current duplicate-detection scope, and can use disk to store nodes that fall outside the duplicate detection scope, when RAM is full.

Structured duplicate detection is designed to be used with a search algorithm that expands a set of nodes at a time, like breadth-first search, where the order in which nodes in the set are expanded can be adjusted to minimize disk I/O. Stored nodes are partitioned into "buckets," where each bucket corresponds to a set of nodes in the original search graph that map to the same abstract node. Because nodes in the same bucket have the same duplicate-detection scope, expanding them at the same time means that no disk I/O needs to be performed while they are being expanded. When a new bucket of nodes is expanded, nodes stored on disk are swapped into RAM if they are part of the duplicate-detection scope of the new bucket, and buckets outside the current duplicate-detection scope can be flushed to disk when RAM is full. The general approach to minimizing disk I/O is to order node expansions so that changes of duplicate-detection scope occur as infrequently as possible, and, when they occur, they involve change of as few buckets as possible. That is, expanding buckets with overlapping duplicate-detection scopes consecutively also tends to minimize disk I/O.

Hash-based delayed duplicate detection

Hash-based delayed duplicate detection (Korf & Schultze 2005) is a more efficient form of delayed duplicate detection. To avoid the time complexity of 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, and all duplicates are mapped to the same file. Once a file of successor nodes has been generated, duplicates can be removed. To avoid the time complexity of sorting to remove duplicates, a second hash function is used that maps all duplicates to the same location of a hash table. Since the hash table corresponding to the second hash function must fit in internal memory, this approach requires some care in designing the hash functions (which are problem-specific) to achieve efficiency. As a further enhancement, the amount of disk space needed for hash-based DDD can be reduced by interleaving expansion and duplicate removal. But interestingly, this is only possible if the successor nodes written to a file are generated from a small subset of the files being expanded; that is, its possibility depends on the same kind of local structure leveraged by SDD.

Korf and Schultze (2005) give some reasons for preferring hash-based DDD to SDD. Their observation that "hash-based DDD only reads and writes each node at most twice" applies to duplicate nodes of the same states, and since hash-based DDD allows many duplicates to be generated, it does not actually bound the number of reads and writes per state. Their observation that hash-based DDD does not require the same local structure as SDD is an important difference (even though we noticed that some of the techniques used by hash-based DDD to improve efficiency over sorting-based DDD exploit similar local structure). They also point out that SDD has a minimum memory requirement that corresponds to the largest duplicate-detection scope. However, the minimum memory requirement of SDD can usually be controlled by changing the granularity of the state-space projection function. Given appropriate local structure, a finer-grained state-space projection function creates smaller duplicate-detection scopes, and thus, lower internal-memory requirements. In the end, whether a search graph has appropriate local structure for SDD is the crucial question, and we address this in the rest of the paper.

An advantage of SDD worth mentioning is that it can be used to create external-memory pattern databases (Zhou & Hansen 2005). Because a heuristic estimate is needed as soon as a node is generated, the delayed approach of DDD cannot be used for this. But the most important advantage of SDD is that, when applicable, it is always more efficient than hash-based DDD. The reason for this is that even though hash-based DDD significantly reduces the overhead of removing duplicates compared to sorting-based DDD, it still incurs overhead for generating and removing duplicates, and this overhead is completely avoided by SDD.

Domain-independent abstraction

We have seen that SDD is preferable to DDD when it is applicable. To show that it is widely applicable, we introduce the main contribution of this paper: an algorithm that automatically creates an abstract representation of the state space that captures the local structure needed for SDD. We show how to do this for a search graph that is implicitly represented in a STRIPS planning language. But the approach is general enough to apply to other graph-search problems.

We begin by introducing some notation related to domain-independent STRIPS planning and state abstraction.

STRIPS planning and abstraction

A STRIPS planning problem is a tuple $\langle \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{A}$ is a set of atoms, $\mathcal{O}$ is a set of grounded operators, and $\mathcal{I} \subseteq \mathcal{A}$ and $\mathcal{G} \subseteq \mathcal{A}$ describe the initial and goal situations, respectively. Each operator $o \in \mathcal{O}$ has a precondition list $Prec(o) \subseteq \mathcal{A}$, an add list $Add(o) \subseteq \mathcal{A}$, and a delete list $Del(o) \subseteq \mathcal{A}$. The STRIPS problem defines a state space $\langle \mathcal{S}, s_0, \mathcal{S}_\mathcal{G}, \mathcal{T} \rangle$, where $\mathcal{S} \subseteq 2^\mathcal{A}$ is the set of states, s_0 is the start state, $\mathcal{S}_\mathcal{G}$ is the set of goal states, and $\mathcal{T}$ is a set of transitions that transform one state s into another state s'. In *progression planning*, the transition set is defined as $\mathcal{T} = \{(s, s') \in \mathcal{T} | \exists o \in \mathcal{O}, Prec(o) \subseteq s \ \wedge \ s' = s \setminus Del(o) \cup Add(o)\}$. In *regression planning*, which involves searching

backwards from the goal to the start state, the transition set is defined as $\mathcal{T} = \{(s, s') \in \mathcal{T} | \exists o \in \mathcal{O}, Add(o) \cap s \neq \emptyset \ \wedge \ Del(o) \cap s = \emptyset \ \wedge \ s' = s \setminus Add(o) \cup Prec(o)\}$. A *sequential plan* is a sequence of operators that transform the start state into one of the goal states. Since STRIPS operators have unit costs, an optimal sequential plan is also a shortest plan.

A state-space abstraction of a planning problem is a projection of the state space into a smaller abstract state space. The projection is created by selecting a subset of the atoms $\mathcal{P} \subseteq \mathcal{A}$. The projection function $\phi_\mathcal{P} : 2^\mathcal{A} \rightarrow 2^\mathcal{P}$ is defined so that the projected state is the intersection of s and $\mathcal{P}$, and the projected or abstract state space is defined as $\phi_\mathcal{P}(\mathcal{S}) = \{s_\mathcal{P} | s \in \mathcal{S} \ \wedge \ s_\mathcal{P} = s \cap \mathcal{P}\}$; for convenience, we write $\phi_\mathcal{P}(\mathcal{S})$ as $\mathcal{S}_\mathcal{P}$. The *projected operators* are similarly defined by intersecting the subset of atoms $\mathcal{P}$ with the precondition, add, and delete lists of each operator.

Locality-preserving abstraction

The number of possible abstractions is huge – exponential in the number of atoms. For SDD, we need to find an abstraction that has appropriate local structure. Recall that we defined the locality of an abstract state space graph as the maximum number of successors of any abstract node compared to the overall number of abstract nodes. The smaller this ratio, the more effective SDD can be in leveraging external memory.

To see why this is so, first assume that the abstract nodes evenly partition the stored nodes in the original graph. Although this is never more than approximately true, it is a reasonable assumption to make for our analysis. Next, recall that the *duplicate-detection scope* of any node $s \in \mathcal{S}$ in the original search graph is defined as all stored nodes that map to the successors of the abstract node that is the image of the node under the projection function $\phi_\mathcal{P}$. The largest duplicate-detection scope determines the minimal memory requirement of SDD, since it is the largest number of nodes that need to be stored in RAM at one time in order to perform duplicate detection. From the assumption that the abstract nodes evenly partition the stored nodes in the original graph, it follows that the locality of the abstract state-space graph determines the largest duplicate-detection scope.

To measure the degree of the local structure captured by a state-space projection function $\phi_\mathcal{P}$, we define the *maximum duplicate-detection scope ratio*, δ, as follows,

$$\delta(\mathcal{P}) = \max_{s_\mathcal{P} \in \mathcal{S}_\mathcal{P}} \frac{|Successors(s_\mathcal{P})|}{|\mathcal{S}_\mathcal{P}|}.$$

Essentially, δ is the maximum fraction of abstract nodes in the abstract state-space graph that belong to a single duplicate-detection scope. The smaller the ratio, the smaller the percentage of nodes that need to be stored in RAM for duplicate detection, compared to the total number of stored nodes. Reducing this ratio allows SDD to leverage more of external memory to improve scalability. Thus, we propose searching for an abstraction that minimizes this ratio as a way of finding a good abstraction for SDD.

In our experience, however, this ratio can almost always be reduced by increasing the resolution of the abstraction, that is, by adding atoms to the projection function. But we

don't want to increase the resolution too much, for a couple of reasons. First, the size of the abstract graph can increase exponentially in the number of atoms used in the projection, and, naturally, we don't want the abstract graph to become so large that it doesn't fit comfortably in RAM. Second, and more importantly, increasing the size of the abstract graph decreases the average number of nodes from the original state-space graph that are assigned to the same abstract node. If the same number of nodes are divided into too many buckets, this can increase the frequency with which the duplicate-detection scope changes during node expansion, which has the potential to increase the amount of disk I/O. In effect, there is a tradeoff between reducing the size of the largest duplicate-detection scope, which reduces the minimum internal memory requirement, and increasing the frequency with which the duplicate-detection scope changes, which potentially increases the amount of disk I/O. The tradeoff between these two factors is complicated and problem-dependent, and we do not yet try to optimize it. Instead, we search for an abstraction that minimizes δ subject to a bound on the maximum size of the abstract graph. That is, we define the best abstraction for SDD as

$$\mathcal{P}^* = \arg\min_{\mathcal{P}}\{\delta(\mathcal{P})|M \geq |\mathcal{S}_{\mathcal{P}}|\}, \qquad (1)$$

where M is a parameter that bounds the size of the abstract graph.

In our experiments, we create the abstract graph before the search begins and don't change it. An alternative is to increase the granularity of the abstraction on the fly, as needed. The algorithm starts with a very coarse abstraction, if any. If it can solve the problem without running out of RAM, there is no need for a more fine-grained abstraction. If not, the algorithm refines its abstraction function to the next level of granularity, and resumes the search. The search continues in this way until the problem is solved or some upper bound M on the size of the abstract graph is exceeded.

Exploiting state constraints

As we have seen, adding atoms to the projection function increases locality in the abstract graph, but it also increases the size of the abstract graph, and we don't want to make the abstract graph too big. A useful way to add atoms to the projection function without increasing the size of the abstract graph too much is by exploiting state constraints. State constraints are invariants or laws that hold in every state that can be reached from the initial state. Although not specified in the description of a planning domain, they can be discovered using a domain analysis tool (Gerevini & Schubert 1998). With such constraints, the value of one atom limits the possible values of other atoms, and this can be exploited to limit the reachable state space.

Our approach to creating an abstraction of the state space exploits XOR constraints to limit the size of the abstract graph as a function of the number of atoms. XOR constraints are state constraints that specify that only one atom in a group of atoms can be true in any complete state. An example of an XOR-constraint in the *blocks* domain is (XOR (on x y) (clear y)), stating that any object y is either

clear or has some x on it, but not both. To discover these constraints, we use an algorithm described by Edelkamp and Helmert (1999), who refer to the process of discovering XOR constraints as *merging predicates*. We note that their technique can find n-ary (for $n > 2$) XOR constraints, which are common in recent planning benchmark domains.

We define an XOR group P as a set of atoms $\{p_1, p_2, \cdots, p_n\} \subseteq \mathcal{A}$ such that (XOR p_1 p_2 $\cdots$ p_n) holds, i.e., one and only one atom must be true. An example of an XOR group in the *logistics* domain is the set of atoms that denote the location of a vehicle. There are usually many XOR groups in a domain. For example, each truck or airplane in *logistics* can be associated with an XOR group that encodes the location of the vehicle. Note also that the XOR groups do not have to be disjoint from one another. By selecting XOR groups of atoms for the projection function, we include the state constraints in the abstract state space and limit the size of the abstract graph. To see how this works, consider an abstraction created by selecting ten atoms. If there are no state constraints, the abstract graph has $2^{10} = 1024$ nodes. But if the ten atoms consist of two XOR groups of five atoms each, the abstract graph has only $5^2 = 25$ nodes.

Greedy abstraction algorithm

We use a greedy algorithm to try to find a projection function that minimizes Equation (1). Let $P_1, P_2, \cdots, P_m$ be the XOR groups in a domain. Instead of attempting to minimize $\delta(\mathcal{P})$ for all possible combinations of all XOR groups, a greedy algorithm adds one XOR group to $\mathcal{P}$ at a time. The algorithm starts with $\mathcal{P} = \emptyset$. Then it tries every single XOR group $P_i \in \{P_1, P_2, \cdots, P_m\}$ by computing the corresponding $\delta(P_i)$, finds the best XOR group P_i^* that minimizes $\delta(P_i)$ for all $i \in \{1, 2, \cdots, m\}$, and adds P_i^* to $\mathcal{P}$. Suppose the best XOR group added is P_1. The greedy algorithm then picks the best remaining XOR group that minimizes $\delta(P_1 \cup P_i)$ for all $i \neq 1$, and adds it to $\mathcal{P}$. The process repeats until either (a) the size of the abstract graph exceeds the upper bound (M) on the size of abstract graphs, or (b) there is no XOR group left. Typically, we don't run out of XOR groups before exceeding the size bound. If we did, we could continue to add single atoms to try to improve the abstraction.

Example

We use an example in the *logistics* domain to illustrate the algorithm. The goal in *logistics* is to deliver a number of packages to their destinations by using trucks to move them within a city, or airplanes to move them between cities. Consider a simple example in which there are two packages {pkg1, pkg2}, two locations {loc1, loc2}, two airports {airport1, airport2}, two trucks {truck1, truck2}, and one airplane {plane1}. A domain constraint analysis discovers several XOR groups, and uses these XOR groups to construct several candidate abstract state-space graphs, two of which are shown in Figure 1. An oval represents an abstract state and the label inside an oval shows the atom(s) in that abstract state. An arrow represents a projected operator that transforms one abstract state into another. Figure 1(a) shows an abstract state-space graph created by using a projection function based on the 7 possible locations of pkg1. Note
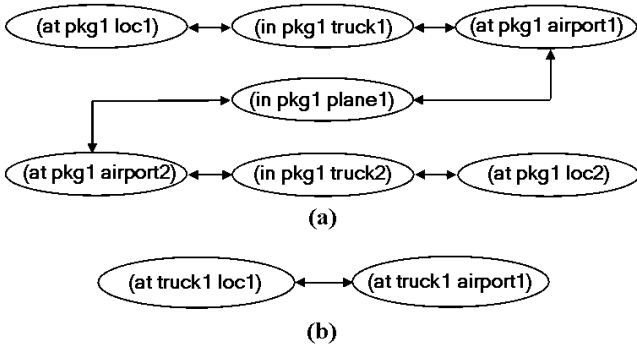
Figure 1: Abstract state-space graphs (with self loops omitted) for *logistics*. Panel (a) shows an abstract state-space graph based on the location of pkg1. Panel (b) shows another abstract state-space graph based on the location of truck1.

that all the atoms shown in Figure 1(a) belong to a single XOR group. Figure 1(b) shows another abstract state-space graph based on the location of truck1. For each candidate abstract graph, the algorithm computes its corresponding δ. For the abstract graph in Figure 1(a), $\delta = \frac{3}{7}$, since the maximum number of successors of any abstract node is 3 (note that self loops are omitted) and the graph has a total of 7 nodes. For the abstract graph in Figure 1(b), $\delta = \frac{2}{2} = 1$, which means it has no locality at all. Thus, the abstract graph shown in Figure 1(a) is preferred, since it has a smaller δ value.

Of course, a δ of $\frac{3}{7}$ may still be too large. The algorithm can decrease δ by increasing the granularity of the abstract graph. It does so by adding another XOR group to $\mathcal{P}$. Suppose it adds a new XOR group that is based on the location of pkg2. Since pkg2 also has 7 possible locations, the number of combinations for the locations of these two packages is $7 \times 7 = 49$, the size of the new abstract graph. But the maximum number of successors of any abstract node only increases from 3 to 5. Thus, the new δ is $\frac{5}{49}$. In the more general case where there are n packages, δ can be made as small as $\frac{1+2n}{7^n}$, a number close to zero even for small n's.

Implementation

An important implementation detail in external-memory graph search is how to represent pointers that are used to keep track of ancestor nodes along a least-cost path. One possibility is to use conventional pointers for referencing nodes stored in RAM and a separate type of pointer for referencing nodes stored on disk. A drawback of this approach is that every time a node is swapped between RAM and disk, its successors along a least-cost path (including those stored on disk) must change the type of their ancestor pointer, and this can introduce substantial time and space overhead .

In our implementation of external-memory breadth-first heuristic search, a pointer to a node at a given depth contains two pieces of information; (a) the abstract node to which the node maps, and (b) the order in which the node appears in the bucket of nodes of the same depth that map to the same abstract node. It can be shown that these two pieces of information about a node do not change after it is generated by a

breadth-first search algorithm. Thus, in our implementation, the same pointer is always valid, whether the node it points to is stored in RAM or on disk.

Results

We implemented our abstraction-finding algorithm in a domain-independent STRIPS planner that uses breadth-first heuristic search (BFHS) with SDD to find optimal sequential plans. We used regression planning, and, as an admissible heuristic, we used the max-pair heuristic (Haslum & Geffner 2000). We tested the planner in eight different domains from the biennial planning competition. Experiments were performed on an AMD Operton 2.4 GHz processor with 4 GB of RAM and 1 MB of L2 cache.

The first eight rows of the Table 1 compare the performance of BFHS with and without SDD. The eight problems are the largest that BFHS can solve without external memory in each of the eight planning domains. The slight difference in the total number of node expansions for the two algorithms is due to differences in tie breaking.

The results shows several interesting things. First, they shows how much less RAM is needed when using SDD. Second, the ratio of the peak number of nodes stored on disk to the peak number of nodes stored in RAM shows how much improvement in scalability is possible. Although the ratio varies from domain to domain, there is improvement in every domain (and more can be achieved by increasing the resolution of the abstraction). The extra time taken by the external-memory algorithm includes the time taken by the abstraction-finding algorithm. The greedy algorithm took no more than 25 seconds for any of the eight problems, and less than a second for some. (Its running time depends on the number of XOR groups in each domain as well as the size of the abstract state-space graph.) The rest of the extra time is for disk I/O, and this is relatively modest. The last column shows the maximum duplicate-detection scope ratio, where the numerator is the maximum number of successors of any node in the abstract graph and the denominator is the total number of abstract nodes. The next to last column shows the number of atoms, $|\mathcal{P}|$, used in the projection function that creates the abstract graph. Comparing $2^{|\mathcal{P}|}$ to the number of nodes in the abstract graph shows how effective the state constraints are in limiting the size of the abstract graph.

The last four rows of Table 1 show the performance of the planner in solving some particularly difficult problems that cannot be solved without external memory. As far as we know, this is the first time that optimal solutions have been found for these problems. It is interesting to note that the ratio of the peak number of nodes stored on disk to the peak number of nodes stored in RAM is greater for these more difficult problems. This suggests that the effectiveness of SDD tends to increase with the size of the problem, an appealing property for a technique that is designed for solving large problems.

Conclusion

For search graphs with sufficient local structure, SDD outperforms other approaches to external-memory graph

Problem	Len	No external memory			External memory					
		RAM	Exp	Secs	RAM	Disk	Exp	Secs	$|\mathcal{P}|$	δ
logistics-6	25	151,960	338,916	2	1,725	165,418	339,112	89	28	9/4096
satellite-6	20	2,364,696	3,483,817	106	51,584	2,315,307	3,484,031	359	33	31/1728
freecell-3	18	3,662,111	5,900,780	221	1,069,901	2,764,364	5,900,828	302	22	81/1764
elevator-12	40	4,068,538	12,808,717	256	172,797	3,898,692	12,829,226	349	33	25/1600
depots-7	21	10,766,726	16,794,797	270	2,662,253	9,524,314	16,801,412	342	42	4/2150
blocks-16	52	7,031,949	18,075,779	290	3,194,703	5,527,227	18,075,779	387	54	43/4906
driverlog-11	19	15,159,114	18,606,835	274	742,988	15,002,445	18,606,485	507	49	15/3848
gripper-8	53	15,709,123	81,516,471	606	619,157	15,165,293	81,516,471	1,192	29	37/4212
freecell-4	26	-	-	-	11,447,191	114,224,688	208,743,830	15,056	22	81/1764
elevator-15	46	-	-	-	1,540,657	126,194,100	430,804,933	17,087	42	32/7936
logistics-9	36	-	-	-	5,159,767	540,438,586	1,138,753,911	41,028	40	13/14641
driverlog-13	26	-	-	-	49,533,873	2,147,482,093	2,766,380,501	145,296	92	25/21814

Table 1: Comparison of breadth-first heuristic search with and without using domain-independent structured duplicate detection on STRIPS planning problems. Columns show solution length (Len), peak number of nodes stored in RAM (RAM), peak number of nodes stored on disk (Disk), number of node expansions (Exp), running time in CPU seconds (Secs), size of projection atom set ($|\mathcal{P}|$), and maximum duplicate-detection scope ratio (δ). A '-' symbol indicates that the algorithm cannot solve the problem without storing more than 64 million nodes in RAM.

search. In this paper, we have addressed two important questions about the applicability of SDD: how common is this kind of local structure in AI graph-search problems, and is it possible to identify it in an automatic and domain-independent way.

Previous work shows that the local structure needed for SDD is present in the sliding-tile puzzle, four-peg Towers of Hanoi, and multiple sequence alignment problems (Zhou & Hansen 2004; 2005). The empirical results presented in this paper show that it can be found in a wide range of STRIPS planning problems. Although it is no guarantee, this encourages us to believe that similar structure can also be found in many other graph-search problems.

In previous work, the abstractions of the state space that capture this local structure were hand-crafted. An advantage of automating this process is that locality-preserving abstractions may not be obvious to a human designer, especially in domains with hundreds of variables and complex state constraints. In addition, automating this process makes it possible to search systematically for the best abstraction, which even a knowledgable human designer may not find.

Many further improvement of this approach are possible. In the near future, we will explore ways to improve the algorithm for identifying locality-preserving abstractions and test it on additional problems. We will also consider whether SDD can leverage other forms of abstraction. So far, we have used state abstraction to partition the nodes of a graph in order to create an abstract state-space graph that reveals local structure. For problems where state abstraction (node partitioning) does not reveal enough local structure, operator abstraction (edge partitioning) may provide another way of creating an abstract state-space graph that reveals useful local structure. Given that most AI search graphs are generated from a relatively small set of rules, we believe they are likely to contain local structure of one form or another that can be leveraged for SDD. Finally, we believe the local structure used for SDD can eventually be leveraged in parallel graph search as well as external-memory graph search.

References

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proceedings of the 5th European Conference on Planning (ECP-99)*, 135–147.

Edelkamp, S.; Jabbar, S.; and Schrödl, S. 2004. External A*. In *Proceedings of the 27th German Conference on Artificial Intelligence*, 226–240.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proc. of the 15th Nat. Conf. on Artificial Intell. (AAAI-98)*, 905–912.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. of the 5th International Conference on AI Planning and Scheduling*, 140–149.

Korf, R., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *Proc. of the 20th National Conference on Artificial Intelligence (AAAI-05)*, 1380–1385.

Korf, R.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

Korf, R. 2004. Best-first frontier search with delayed duplicate detection. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, 650–657.

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.

Stern, U., and Dill, D. 1998. Using magnetic disk instead of main memory in the mur(phi) verifier. In *Proc. of the 10th Int. Conf. on Computer-Aided Verification*, 172–183.

Zhou, R., and Hansen, E. 2004. Structured duplicate detection in external-memory graph search. In *Proc. of the 19th Nat. Conf. on Artificial Intelligence (AAAI-04)*, 683–688.

Zhou, R., and Hansen, E. 2005. External-memory pattern databases using structured duplicate detection. In *Proc. of the 20th National Conf. on Art. Int. (AAAI-05)*, 1398–1405.

Zhou, R., and Hansen, E. 2006. Breadth-first heuristic search. *Artificial Intelligence* 170(4–5):385–408.