Dynamic State-Space 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

The scalability of optimal sequential planning can be improved by using external-memory graph search. State-of-the-art external-memory graph search algorithms rely on a state-space projection function, or hash function, that partitions the stored nodes of the state-space search graph into groups of nodes that are stored as separate files on disk. Search performance depends on properties of the partition; whether the number of unique nodes in a file always fits in RAM, the number of files into which the nodes of the state-space graph are partitioned, and how well the partition captures local structure in the graph. Previous work relies on a static partition of the state space, but it can be difficult for a static partition to simultaneously satisfy all of these criteria. We introduce a method for dynamic partitioning and show that it leads to improved search performance in solving STRIPS planning problems.

Introduction

Recently-developed algorithms for external-memory graph search, including structured duplicate detection (Zhou & Hansen 2004; 2006b; 2007) and hash-based delayed duplicate detection (Korf & Schultze 2005; Korf 2008), rely on a hash function, or equivalently, a state-space projection function, that partitions the nodes of the state-space search graph into *buckets* of nodes that are stored as separate files on disk. For both structured duplicate detection and hash-based delayed duplicate detection, the state-space projection function must satisfy similar criteria. First, the set of unique nodes in each bucket must fit in RAM. In addition, for best performance, nodes should be relatively evenly distributed among buckets and buckets should be relatively full. Finally, search efficiency depends on how well the state-space projection function captures local structure in the graph, which takes the following form; for any bucket of nodes, successor nodes are found in only a small number of other buckets.

Finding a projection function that satisfies all of these criteria to achieve the best search performance presents a challenge. Korf relies on handcrafted projection functions (which he calls hash functions) that are tailored to specific search domains with well-understood structure. Zhou and Hansen (2006b) describe how to automatically generate an appropriate projection function by heuristic-guided greedy search through the space of possibilities. Both approaches have relied on static projection functions that do not change during the progress of the search, but this has drawbacks. It is relatively easy to design a static partition that captures local graph structure (assuming local structure is present), but difficult to predict in advance the number of nodes that will map to each bucket of such a partition; in practice, the distribution of nodes to buckets can be very uneven. It is also easy to use a randomized hash function to create a static partition that evenly distributes nodes among buckets, but a randomized hash function does not capture any local structure, since it allows nodes in one bucket to have successor nodes in any other random bucket. In short, it can be difficult to design a projection function that both captures local structure and evenly distributes nodes among buckets.

In this paper, we describe an approach to improving the performance of structured duplicate detection – and, by implication, hash-based delayed duplicate detection – by dynamically adjusting the projection function in the course of the search. This allows the search algorithm to monitor the distribution of nodes in buckets at runtime, and modify the projection function to improve search performance. In case the set of nodes in a bucket does not fit in RAM, the algorithm changes the partition so that the search can continue. Dynamic adjustment of the state-space projection function also can even out the distribution of nodes to buckets, while still capturing local structure in the state-space search graph. We show that the overhead for dynamically re-partitioning the state space is modest, and, in practice, it is more than compensated for by an improvement in the space and time complexity of the search.

Motivation and background

To motivate an external-memory approach to heuristic-search planning, we conducted experiments running state-of-the-art sequential optimal planners including the Fast-Downward planner (Helmert 2006) and the breadth-first heuristic search planner (BFHSP) (Zhou & Hansen 2006a), to see how fast these planners can expand and generate search nodes. To simplify our experiments, we ran these planners in brute-force search mode with $h = 0$ (the blind heuristic, although technically it may return a value of 0 for goal states and 1 for all other states), since results from the most recent Planning Competition have indicated that uninformed breadth-first search is quite competitive, in terms of the number of solved problems, against all the other sequential optimal planners equipped with state-of-the-art admissible heuristics (Helmert, Do, & Refanidis 2008).

Table 1 shows the brute-force search speed of these two planners. Our results indicate that it only takes about 4 $\sim$ 8 minutes for FastDownward to use up 4 GB of RAM using only one of the eight Intel Xeon 3.0GHz cores that our machine has. The speed of BFHSP is even faster, generating states about 3 times faster than FastDownward on average, also using a single core. In other words, both planners can run out of RAM in just a few minutes without resorting to parallelization. Of course, once planners begin to take advantage of multi-core processors, they may run out of RAM even sooner.

To scale up heuristic-search planners, we need to develop I/O-efficient search algorithms that can leverage the vast storage capacity of magnetic as well as solid-state drives that are increasingly popular. For background on disk-based search, we review and compare hash-based delayed duplicate detection and structured duplicate detection.

Both techniques use a state-space projection function to partition the stored nodes of a state-space search graph into *buckets*, where each bucket of nodes is stored in a separate file. The 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; thus, an abstract state corresponds to a bucket. A projection function can be defined by selecting a subset of state variables; states that have the same values for this subset of variables map to the same bucket. A projection function captures local structure in a graph if for any bucket of nodes, successor nodes are found in only a small number of other buckets, called its *neighbors*. In other words, it captures local structure when the largest number of neighbors of any bucket is small relative to the total number of buckets.

Given a state-space projection function, we can construct an abstract state-space graph as follows: for any two abstract nodes, there is a arc from one abstract node to the other if and only if there is a node in the first with a successor node in the second, under an action corresponding to the arc. A projection function captures local structure in a graph if the largest out-degree of any abstract node is small relative to the total number of abstract nodes. This means that for any bucket of nodes, successor nodes are found in only a small number of other buckets.

Hash-based delayed duplicate detection

Korf introduced hash-based delayed duplicate detection (DDD) to avoid the overhead of sorting-based DDD, which relies on external sorting of files to detect and remove duplicate nodes. Hash-based DDD uses two hash functions, where the first corresponds to what we call a state-space projection function. As new states are generated they are placed into separate files based on this first hash function. This guarantees that all duplicate nodes end up in the same file. To remove duplicates from a file, the file is read into a hash table that fits in RAM; associated with this hash table is the second hash function. Duplicate nodes that map to the same slot of this hash table are "merged." Finally, the contents of this hash table are written back to disk as a file of unique nodes.

	FastDownward			BFHSP		
Problem	Exp	Gen	Sec	Exp	Gen	Sec
logistics-9	23M	28M	400	30M	360M	111
depots-13	12M	193M	307	20M	329M	200
gripper-8	23M	106M	200	50M	229M	88
driverlog-14	18M	225M	205	11M	144M	107
blocks-10	22M	98M	237	36M	149M	76
satellite-5	10M	340M	449	7M	236M	89
freecell-5	11M	78M	264	37M	259M	1,007

Table 1: Brute-force search speed of sequential optimal STRIPS planners. Columns show the number of nodes expanded (Exp) and generated (Gen) in millions, and the CPU seconds (Sec) for FastDownward and a breadth-first heuristic search planner (BFHSP) that uses structured duplicate detection.

Consider a breadth-first search algorithm that expands one level of the search space at a time. The files at the current level of the search space contain no duplicate nodes, and are called *expansion files*. As the nodes in each of these files are expanded, their successor nodes are written to files at the next depth of the search space, with the value of the first hash function determining which file they are written to. Since these files contain duplicate nodes that will be removed in the merge phase of the algorithm, they are called *merge files*. Because expanding all files at the current level before merging any files at the next level could require a large amount of extra disk space to store all of the duplicate nodes, Korf (2005) proposes to *interleave expansion and merging*. This is possible only if the projection function captures local structure in the state-space search graph; that is, it depends on the nodes in each merge file being generated from only a small number of expansion files. As soon as all the expansion files that have successors in a particular merge file have been expanded, duplicates can be removed from the merge file (by copying the file into a hash table in RAM) even if all expansion files in the current level of the search space have not yet been expanded. To save disk space, merging duplicates in a file is given priority over expanding another file.

Structured duplicate detection

Zhou and Hansen (2004) describe an alternative strategy for external-memory graph search called structured duplicate detection (SDD). To facilitate comparison of SDD to hash-based DDD, we describe a form of SDD that uses a technique called *edge partitioning* (Zhou & Hansen 2007). This will help explain why the state-space projection function used by both approaches must satisfy similar criteria.

Like hash-based DDD, SDD uses two "hash functions." It partitions the nodes in each level of the search space into separate files based on the first hash function, which is the projection function. For each file, it expands the nodes contained in the file and writes the successor nodes to files in the next level of the search space. The key difference between

SDD and hash-based DDD is that SDD does not delay duplicate detection; all duplicate nodes are detected and eliminated as soon as they are generated, without writing any duplicate node to disk. In its original form, SDD accomplishes this by copying the *duplicate-detection scope* of the currently-expanding file into a hash table in RAM associated with the second hash function. The duplicate-detection scope consists of all nodes in any of the neighbor files of the expanding file. Since this requires the nodes in multiple files to fit in RAM at once, this form of SDD has a larger internal-memory requirement than hash-based DDD, if they both use the same projection function. (Hash-based DDD never requires more than one file of unique nodes to fit in RAM at once.)

Edge partitioning reduces the internal-memory requirement of SDD in the following way. If the duplicate-detection scope of an expansion file does not fit in RAM, then one or more of the neighbor files of the current expansion file are not copied into RAM. Instead, when expanding the nodes in the expansion file, the successor nodes that map to one of these neighbor files are simply not generated. After every node in the expansion file is expanded, the ignored neighbor file(s) are copied into RAM, replacing the neighbor files that were previously in RAM, and the nodes in the same parent file are expanded again. This time, the successor nodes that mapped to one of the neighbor files that is no longer in RAM are not generated, and only the successor nodes that map to the neighbor file(s) in RAM are generated and saved, as long as they are not duplicates. Thus, by *incrementally* expanding the nodes in a file, the internal-memory requirements of SDD can be reduced to the point where no more than one (neighbor) file needs to be stored in RAM at once, the same as for hash-based DDD. It is a classic time-space tradeoff; the amount of RAM that is required is reduced in exchange for the increased time complexity of node re-expansions.

If SDD with edge partitioning uses the same projection function as hash-based DDD, it is clear they have the same peak RAM requirement – the amount of RAM required to store all the unique nodes in the largest file. But the two approaches offer different tradeoffs. Hash-based DDD incurs extra time and space overhead for writing all duplicate nodes to disk and removing them in a later merge step. SDD with edge partitioning incurs the extra time overhead of having to read the same expansion file multiple times, if a duplicate-detection scope does not fit in RAM.

Comparison of approaches

As for disk storage, hash-based DDD always needs at least as much disk space as SDD, since both approaches store all unique nodes. In addition, hash-based DDD needs disk space to store duplicate nodes. How much additional disk storage it needs depends on how well the projection function captures local structure in the state-space search graph. If local structure is leveraged to allow interleaving of expansion and merging, it may need very little additional disk space. In the worst case, when merging of files must be postponed until all files in the current layer are expanded, it could require much more disk storage than SDD, by a factor equal to the ratio of duplicate nodes to unique nodes. Note that if the projection function does not capture any local structure, both the internal memory requirement and the disk storage requirement of SDD with edge partitioning remain the same; only its time complexity increases due to incremental node expansions and multiple reads of the same expansion file.

Comparing the time complexity of hash-based DDD and SDD is more challenging, but we can make some general remarks about their relative advantages and disadvantages. Both approaches perform extra work that is not performed by the other approach, and that is what we compare. For hash-based DDD, the extra work is writing all duplicate nodes to disk and then eliminating them in a later merge step that copies the nodes in each merge file back to RAM, eliminates duplicates, and writes an expansion file of unique nodes. For SDD with edge partitioning, the extra work consists of incremental node expansions and reading the same file from disk multiple times. The extra work performed by hash-based DDD is proportional to the ratio of duplicate nodes to unique nodes in the search space, which is problem-dependent. The extra work performed by SDD depends on how much local structure is captured by the projection function, since the number of times a file may need to be read from disk is bounded above by the number of its neighbor files.

Hash-based DDD may have an advantage in time complexity when there are few duplicates relative to unique nodes in the search space. (An example of such a problem would be the Rubik's Cube search problem used a test case by Korf (2008).) SDD may have an advantage when the ratio of duplicates to unique nodes is large, and the projection function captures local structure in the state-space search graph. If the projection function does not capture any local structure, there may be a blowup in the time complexity of SDD; but in this case, there could be a corresponding blowup in the disk space requirements of hash-based DDD. The best approach is likely to be problem-dependent. Our concern in this paper is not to establish which approach is better. The point of our comparison is to show that both approaches rely on a state-space projection function that must satisfy the same criteria. It follows that the method for dynamic state-space partitioning introduced in this paper can be effective for both approaches. Our experimental results will demonstrate the effectiveness of dynamic partitioning for SDD.

Criteria of a good projection function

As we have seen, the projection function used in external-memory graph search should capture local structure in the graph. For hash-based DDD, this allows interleaving of expansion and merging. For SDD with edge partitioning, it limits the time overhead of incremental expansions.

In addition, and more critically, the projection function should ensure that the set of unique nodes in each file fits in a hash table in RAM. One way to ensure this is to use a high-resolution projection function that partitions the nodes of the state-space search graph into so many files that each is guaranteed to fit in RAM. But partitioning the state space into too many files can degrade search performance. Besides decreasing the average size of a file, the typically uneven distribution of nodes among files means that many files could be empty or nearly empty. The search algorithm also needs to maintain a table in RAM that keeps track of all files, whether they are open or not, whether they have a write or read buffer, their status as an expansion or merge file (in hash-based DDD), a list of their neighbor files, etc., and the size of this table can grow exponentially with the resolution of the projection function. Refining the partition also tends to increase the number of neighbors of a file. In hash-based DDD, this means that more file buffers must be maintained to allow generated nodes to be written to their corresponding file. For SDD with edge partitioning, it could lead to more incremental expansions. In general, increasing the resolution of the projection function reduces the peak RAM consumption of the search algorithm in exchange for an increase in its running time, for all of these reasons.

Thus, it is generally not enough to ensure that the set of unique nodes in each file fits in RAM. It is also important to use as coarse a partition as possible, so as not to degrade search performance too much. Given an uneven distribution of nodes among files, however, a coarse partition increases the risk that the set of unique nodes in a particular file may not fit in RAM. This motivates the approach to dynamic state-space partitioning that is developed in this paper, which will allow us to manage this tradeoff more effectively. It will let us find the coarsest partition that still allows the largest file to fit in RAM.

Although the issues we address are important for the performance of hash-based DDD, we should clarify that they do not arise for test cases for which Korf (2008) reports experimental results. For Rubik's Cube, he reports results for (partial) breadth-first search that does not use a heuristic. For search problems involving sliding-tile puzzles and the four-peg Towers of Hanoi, Korf uses handcrafted hash functions that are perfect and invertible, allowing the use of direct-address tables in memory that just need to store a few bits of information for each entry, instead of the entire state description. This allows him to partition the nodes of the state-space search graph into sufficiently many files (still hundreds of millions of files) that the maximum number of unique nodes in each file is guaranteed to fit in the direct-address table in RAM. But in general, perfect *and* invertible hash functions are not possible. In particular, they are not feasible for either domain-independent planning or model checking.

Edelkamp and Sulewski (2008) make this point about model checking. An application of hash-based DDD to model checking is described by Evangelista (2008). Instead, the hash table must store the complete state description with each entry, in order to resolve collisions. In this situation, it is not possible to ensure that all the nodes in a bucket at any point in the search will fit in RAM. Since it is usually unrealistic for the projection function to partition the nodes of the search graph into so many files that it is possible to guarantee that the set of all possible nodes that map to a bucket can fit in RAM at once, an open-address hash table is used instead that allows collisions. The hope is that the actual nodes in a bucket at any point during the search will fit in RAM. Since the number of nodes that will be generated and stored in any file is not known until run time, it motivates a dynamic approach to state-space partitioning.

Pathological state-space projection functions

Dynamic partitioning is useful when it is difficult to predict the distribution of nodes in buckets, and the primary reason it is difficult to predict the distribution is because it is difficult to predict reachability. From different start states, different sets of states may be reachable. In heuristic search, the choice of heuristic also affects which states are reachable. In frontier search, which saves memory by only storing nodes on the frontier of the search, the distribution of nodes among buckets varies with the depth of the search, because reachability varies with search depth.

To illustrate the effect of reachability on the distribution of nodes among buckets in heuristic search, we use the 15-Puzzle shown in Figure 1 as an example. Suppose the state-space projection function hashes a node to a bucket based on the position of tiles $3, 7, 11, 12, 13, 14$, and 15 (shown as gray tiles). Since there are $16!/9! = 57,657,600$ different combinations for the positions of these 7 gray tiles, each bucket should get the fraction $57,657,600^{-1} = 1.73 \times 10^{-8}$ of the total number of nodes generated, if the distribution of nodes is perfectly balanced among buckets. However, for the start state shown in Figure 1, all the gray tiles are already at their goal positions. Thus an optimal solution does not need to move these gray tiles far away from their current positions. In fact, the white tiles in Figure 1 form a solvable instance of (a variant of) the 8-Puzzle with an optimal solution length of 20, which happens to be also the optimal solution length of the entire 15-Puzzle instance. Thus, there is at least one optimal solution that does not require moving a single gray tile.

Since all states that share the same positions of these gray tiles are mapped to the same bucket, the bucket with all the gray tiles located at their goal positions would get the majority of nodes. This is because the heuristic biases the search to move the white but not the gray tiles. Unfortunately, this means that almost every node generated is mapped to the same bucket. We refer to projection functions that create highly imbalanced buckets as *pathological state-space projection functions*. In this example, an external-memory search algorithm that uses either SDD or hash-based DDD would not save any RAM, and could potentially use *more* RAM, because the partition could have orders of magnitude more buckets than the number of search nodes expanded in solving the problem. This example also shows that increasing the resolution of the state-space projection function is not guaranteed to work in heuristic search. If the projection function used is pathological, then creating more buckets does not necessarily reduce the size of the largest bucket, which determines the peak RAM requirements, in both SDD and hash-based DDD.

Note that the same state-space projection function would have worked fine if it were used inside a brute-force breadth-first search algorithm, because, in the absence of any search bias, the search is just as likely to move the gray tiles as it is to move the white tiles, resulting in a more balanced distribution of nodes among buckets. However, in the exper-

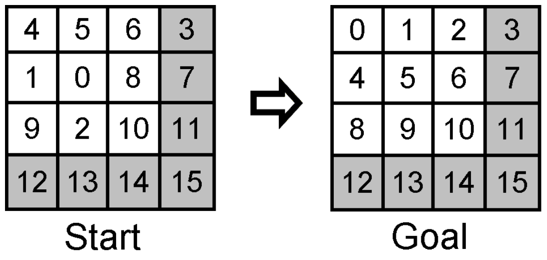

Figure 1: Example of a pathological state-space projection function for an instance of the 15-Puzzle. Only the positions of the gray tiles are considered in the projection function.

imental results section, we will see that even in brute-force breadth-first search, unreachability of states in the Sokoban domain affects the performance of a static partition.

Dynamic state-space partitioning

Zhou and Hansen [2006b] describe an automatic state-space partitioning algorithm for a domain-independent STRIPS planner that uses external-memory graph search with SDD. The projection function that partitions the state space is defined by selecting a subset of state variables. Beginning with the null set of variables, the algorithm performs greedy search in the space of projection functions by selecting at each step a multi-valued variable (or a related group of Boolean variables) that maximizes the locality of the partition, where locality is defined as the largest number of neighbors of any bucket divided by the total number of buckets. This measure of locality is a good objective function for the greedy search under the assumption that the projection function evenly partitions the stored nodes of the graph. But as shown in our previous example (and our experiments will show) this assumption is an over-idealization; in practice, the distribution of nodes among buckets can be very uneven.

We can improve on this static approach by introducing a dynamic partitioning algorithm that monitors the distribution of nodes among buckets in the course of the search and modifies the projection function to adapt to the distribution. The dynamic partitioning algorithm searches for a projection function that both captures local structure *and* keeps the size of the largest bucket of nodes as small as possible – in particular, small enough to fit in RAM. The algorithm we describe is simple and could be improved in obvious ways, but it is sufficient to show the effectiveness of the approach.

Like the static partitioning algorithm, the dynamic partitioning algorithm is greedy and adds a new state variable to the projection function each iteration. In the initial iteration, no state variables have been selected and the partition consists of a single bucket for all nodes. For each bucket in the partition, it keeps a vector of counters, one for each state variable that has not yet been selected. It scans all generated nodes and computes values for the counters, as follows. As it scans each node, it maps it to one of the buckets of the partition created in the previous iteration. Then, for each state variable that has not yet been selected and the corresponding bucket in a refined partition, it determines whether the node maps to this potential bucket. If so, it increments the corresponding counter. At the end of the iteration, the algorithm selects the state variable which results in the greatest reduction in the size of the largest bucket (and also captures locality in the state-space graph) and adds it to the projection function. This refines the partition. Then the process repeats, with new counters. The algorithm terminates when either the size of the largest bucket is below a threshold or the maximum number of buckets is reached. It checks whether the partition found by the dynamic algorithm is significantly better than the partition used to organize the current set of files. If so, it creates a new set of files based on the new partition and copies the nodes on disk to the new files. To save disk space, a file that corresponds to an old bucket is deleted immediately after all of its nodes are moved to their new buckets. Thus, dynamically changing the partition barely increases the peak disk space requirements of the search algorithm; its effect is to reduce the peak RAM requirements.

Vars	Values	Nodes
$\{X\}$	$\{X=1\}$	$\{a,b\}$
	$\{X=2\}$	$\{c,d\}$
	$\{X=3\}$	$\{e,f\}$
$\{Y\}$	$\{Y=4\}$	$\{a,c,e\}$
	$\{Y=5\}$	$\{b,d,f\}$
$\{Z\}$	$\{Z=6\}$	$\{a,b,c,d\}$
	$\{Z=7\}$	$\{e,f\}$

Table 2: First iteration of dynamic partitioning.

Vars	Values	Nodes
$\{X,Y\}$	$\{X=1,Y=4\}$	$\{a\}$
	$\{X=1,Y=5\}$	$\{b\}$
	$\{X=2,Y=4\}$	$\{c\}$
	$\{X=2,Y=5\}$	$\{d\}$
	$\{X=3,Y=4\}$	$\{e\}$
	$\{X=3,Y=5\}$	$\{f\}$
$\{X,Z\}$	$\{X=1,Z=6\}$	$\{a,b\}$
	$\{X=1,Z=7\}$	$\emptyset$
	$\{X=2,Z=6\}$	$\{c,d\}$
	$\{X=2,Z=7\}$	$\emptyset$
	$\{X=3,Z=6\}$	$\emptyset$
	$\{X=3,Z=7\}$	$\{e,f\}$

Table 3: Second iteration of dynamic partitioning.

Example *An example illustrates how the greedy dynamic partitioning algorithm works. Suppose a search problem has three state variables: $X \in \{1,2,3\}$, $Y \in \{4,5\}$, and $Z \in \{6,7\}$. The algorithm has generated and stored 6 states (encoded in $\langle X,Y,Z\rangle$ format): $a=\langle 1,4,6\rangle$, $b=\langle 1,5,6\rangle$, $c=\langle 2,4,6\rangle$, $d=\langle 2,5,6\rangle$, $e=\langle 3,4,7\rangle$, and $f=\langle 3,5,7\rangle$. Tables 2 and 3 show the first two iterations of the algorithm. In both tables, the "Vars" column shows the set of state variables being considered for the projection function, the "Values" column shows the assignment of values to the state variables, and the "Nodes" column shows the set of stored nodes whose state encoding matches the correspond-*

ing "Values" column. In the first iteration, only three singleton state-variable sets, $\{X\}$, $\{Y\}$, and $\{Z\}$, are considered. The largest bucket size, as a result of using a single state variable, is 2 for X, 3 for Y, and 4 for Z. Thus, at the end of the first iteration, the state variable X is chosen for the projection function. Since there are only two variables Y and Z left, the second iteration only has two candidates – the variable sets $\{X, Y\}$ and $\{X, Z\}$ – for the refined projection function. Clearly, the variable set $\{X, Y\}$ should be chosen, because it reduces the largest bucket size to one, achieving a perfect balance across all buckets.

In our initial description of the algorithm, we assumed that all nodes are stored on disk when the dynamic partitioning algorithm is invoked. In fact, some nodes could be stored only in RAM and we need to handle the case where changing the partition requires moving nodes from RAM to disk, or vice versa. The following four cases need to be handled correctly: (1) moving a RAM node to a RAM bucket, (2) moving a disk node to a disk bucket, (3) moving a RAM node to a disk bucket, and (4) moving a disk node to a RAM bucket. Furthermore, one or more RAM buckets may need to be flushed to disk, if internal memory is exhausted in the middle of moving nodes to their new buckets. Thus, in the fourth case above, the algorithm needs to make sure there is space in RAM to hold a node read from disk; if not, all the nodes in the new bucket need to be written to disk. The procedure for handing the second case is then invoked, because the new bucket is no longer in RAM. To save RAM, once a bucket is flushed to disk, it is never read back into RAM until the search resumes.

Excessive overhead for dynamic partitioning is avoided in a couple of ways. First, since repartitioning the state space involves the time-consuming process of moving nodes on disk and creating new files, it is done only when the dynamic partitioning algorithm finds a significantly better partition. (In our implementation, the reduction in the largest bucket size must be greater than 10%.) Second, dynamic partitioning is invoked only if there is a significant imbalance in bucket sizes *and* the largest bucket consumes a substantial fraction of available RAM. (In our implementation, the ratio of largest bucket size to average bucket size must be greater than three *and* the largest bucket size must be greater than half of available RAM; these choices could be tuned to improve performance.) How often dynamic partitioning is invoked is problem-dependent. If a good partition is found early in the search, it may not need to be changed. This can be viewed as choosing a partition based on a sampling of the search space.

In each iteration, the algorithm sequentially scans the buckets (for good I/O performance) and uses a set of counters to keep track of the number of stored nodes whose state variable being considered (for inclusion in the abstraction) has a particular value. In progression (i.e., forward state-space search) planning, the number of counters should equal the number of possible legal values for the state variable being considered; whereas in regression (i.e., backward state-space search) planning there is a need for one more counter, since the value of a state variable may not be defined in regression search. Equivalently, one can compute the value of this additional counter by subtracting the sum of all "legal-value" counters from the total number of nodes stored, which is easy to obtain in our system.

As the abstract search space is being refined, a vector of counters is stored at every abstract node such that the size of the largest bucket can be accurately computed without actually moving the nodes to their new buckets. Of course, once the refinement of the abstraction function is done, all the nodes need to be moved to the new buckets where they belong, but this is only performed in the end of the dynamic partitioning algorithm.

A simplification of our approach is that we adopt a uniform-resolution method of partitioning. This allows us to reduce the size of the largest bucket and, in practice, it also tends to even out the distribution of nodes among buckets. But a variable-resolution partitioning scheme, in which the inclusion of some state variables in the projection function depends on the values of other state variables, would allow much more fine-grained control of the distribution of nodes – and it would likely find a better partition that could further reduce the time complexity of the external-memory search.

Experimental results

We implemented dynamic state-space partitioning inside an external-memory STRIPS planner that uses as its underlying search algorithm *breadth-first heuristic search* (Zhou & Hansen 2006a). Experiments were run on a Xeon 3.0 GHz processor with 4 GB of RAM and 6 MB of L2 cache.No parallel processing was used in the experiments.

Table 4 shows the performance of our external-memory STRIPS planner on the 15-Puzzle. While domain-specific solvers can find optimal solutions to randomly generated instances of the 24-Puzzle, the 15-Puzzle remains a challenge for domain-independent planners. With an accurate disjoint pattern database heuristic, the best planner can solve only 93 of Korf's 100 15-Puzzle instances (Haslum *et al.* 2007). Here we show that with the equivalent of a basic pattern database heuristic (same as Manhattan distance), our planner can solve the entire set. For reference, the most difficult instance (#88) can be solved by our planner in less than an hour, storing about 350 thousand nodes in RAM and 300 million nodes on disk. In our implementation, each 15-Puzzle node takes 36 bytes to store. Thus, the peak RAM consumption for storing the Open and Closed lists is roughly 12.6 MB for solving instance #88. Without external-memory search, it would take roughly 10.8 gigabytes of RAM just to store the nodes, even though the underlying breadth-first heuristic search algorithm only stores 30% of the nodes expanded by A* (Zhou & Hansen 2006a).

For all instances in Table 4 and for both static and dynamic partitioning, the number of buckets in the partition is 3360. Dynamic partitioning uses a slight amount of extra disk space for moving nodes on disk when the partition is changed dynamically. Some of the extra time overhead for dynamic partitioning is due to the partitioning algorithm itself. Most is due to extra disk I/O as a result of using less

		Static partitioning				Dynamic partitioning			
#	Len	RAM	Disk	Increm Exp	Secs	RAM	Disk	Increm Exp	Secs
17	66	908,902	50,871,643	711,180,658	589	149,054	51,443,638	718,502,872	854
49	59	927,906	80,987,861	812,948,341	697	105,021	81,209,329	817,962,744	1,021
53	64	244,889	48,518,100	592,797,672	511	102,365	48,650,054	593,080,899	690
56	55	498,854	49,436,882	477,575,355	424	95,883	49,570,631	480,107,665	522
59	57	957,496	52,834,528	531,743,811	484	169,941	53,504,361	539,097,048	543
60	66	867,509	218,185,611	2,582,825,054	2,184	680,840	218,181,871	2,566,169,284	2,568
66	61	309,651	81,919,509	920,508,447	793	124,377	82,084,656	922,526,667	1,014
82	62	385,486	177,927,698	1,865,565,899	1,582	245,695	177,963,645	1,859,645,376	2,115
88	65	1,776,317	295,406,768	3,357,109,415	2,923	349,901	296,507,472	3,354,622,475	3,234
92	57	324,196	48,085,400	512,701,523	450	71,998	48,130,370	511,378,919	702

Table 4: Comparison of edge partitioning with and without dynamic state-space partitioning on the 10 hardest of Korf's 100 15-Puzzle instances encoded as STRIPS planning problems. The number of buckets in the partition is the same for both static and dynamic partitioning. Columns show solution length (Len), peak number of nodes stored in RAM (RAM), peak number of nodes stored on disk (Disk), number of incremental node expansions (Exp), and running time in CPU seconds (Secs).

		Static partitioning					Dynamic partitioning			
Problem	Len	Disk	RAM	Increm Exp	Secs	Buckets	RAM	Increm Exp	Secs	Buckets
blocks-14	38	381,319	37,129	10,763,944	21	2,660	8,637	13,738,732	40	2,644
gripper-7	47	2,792,790	13,000	177,532,311	506	3,726	14,999	169,318,244	298	2,568
freecell-3	18	4,279,315	151,546	107,699,115	284	1,764	35,247	101,070,199	327	1,764
depots-7	21	12,877,783	410,815	184,606,201	300	4,240	32,000	255,291,983	584	4,240
driverlog-11	19	15,780,803	89,999	233,976,409	305	3,848	75,000	299,947,271	414	2,752
gripper-8	53	14,099,800	59,999	894,274,064	1,427	4,212	50,000	857,433,260	1,210	3,210
depots-13	25	1,110,708	81,003	27,711,837	25	625	14,653	25,411,325	38	700
driverlog-14	28	26,356,967	911,288	664,087,448	775	784	472,011	544,381,999	810	616
logistics-10	42	81,728,366	8,505,120	1,434,271,308	3,267	2,744	448,343	1,961,380,522	3,959	2,940

Table 5: Comparison of edge partitioning with and without dynamic state-space partitioning on STRIPS planning domains. Columns show solution length (Len), peak number of nodes stored on disk (Disk), peak number of nodes stored in RAM (RAM), number of incremental node expansions (Exp), running time in CPU seconds (Secs), and the peak number of buckets. For the first 6 problems, the planner used the max-pair heuristic; for the last 3, it used a more accurate pattern database heuristic.

RAM. The results in Table 4 show the effectiveness of dynamic state-space partitioning in reducing the peak RAM requirements of the algorithm.

Figure 2 provides additional insight into why the approach is effective. With dynamic partitioning, the distribution of stored nodes among files is more concentrated around the average file size than with static partitioning, which has both larger files and empty files.

Table 5 shows the performance of the external-memory planner on domains from the biennial Planning Competition. A couple interesting observations can be made. First, peak RAM consumption for problems such as depots-7 and logistics-10 is substantially reduced. Previously, the only way to reduce peak RAM consumption was to use a more fine-grained projection function that creates more buckets. Yet in some cases, the reductions we achieved in peak RAM consumption are a result of using a *coarser* partition with fewer buckets. This shows that the resolution of the projection function, while important, is not the only factor that determines the amount of RAM saved in external-memory search. Our results show that there can be a large difference in peak RAM consumption among projection functions that have the same resolution. This illustrates the benefit of dynamic partitioning based on monitoring the actual distribution of nodes among buckets. The only problem instance for which dynamic partitioning uses more RAM than static par-

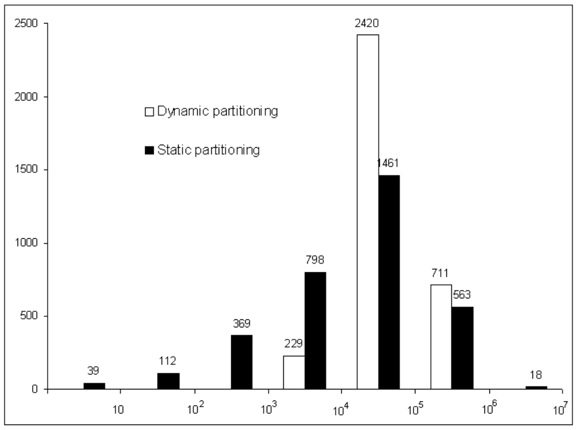

Figure 2: Distribution of nodes among buckets using static and dynamic partitioning for Korf's 15-puzzle problem instance #88. The x-axis is bucket size in number of nodes and the y-axis is count of buckets that have a size that falls in one of the following 7 ranges; $[0, 10]$, $(10, 10^2]$, $(10^2, 10^3]$, $(10^3, 10^4]$, $(10^4, 10^5]$, $(10^5, 10^6]$, $(10^6, 10^7]$.

#	Len	Disk	RAM	Increm Exp	Secs
106	205	189,096K	41,278K	46,595,322K	41,208
108	238	51,275K	28,665K	36,281,692K	24,009
137	177	51,183K	34,399K	15,178,194K	9,290
40	82	2,147,471K	13,824K	79,572,038K	102,516

Table 6: Performance of dynamic state-space partitioning in STRIPS planning. The first 3 are Microban instances (Haslum *et al.* 2007) solved with $h = 0$; the last is a 24-Puzzle instance (Korf & Felner 2002). (1K = 1000 nodes)

titioning is gripper-7. In this case, we intentionally forced the dynamic partitioning algorithm to use a coarser projection function in order to see if it could still find a good partition. Note that with fewer buckets for gripper-7, the external-memory search algorithm runs 70% faster in return for a 15% increase in peak RAM consumption.

Table 6 shows the performance of dynamic partitioning on four hard STRIPS problems. The first three are Microban ("introductory" Sokoban) instances (Haslum *et al.* 2007). To show the benefit of our approach when the heuristic is weak, we used brute-force breadth-first search. Surprisingly, dynamic partitioning with $h = 0$ not only solved #137, an instance that needs the most number of node expansions according to (Haslum *et al.* 2007), it also solved #108, a previously unsolved instance. Static partitioning works poorly due to the abundance of unreachable states in the search graph of Sokoban. Because it is generally intractable to determine if states are unreachable (Zilles & Holte 2009), static partitioning cannot take this into account, resulting in an uneven distribution of nodes. The last row of Table 6 corresponds to a 24-Puzzle instance #40 (Korf & Felner 2002). To our knowledge, this is the first time a graph-search algorithm with full duplicate detection was able to solve a non-trivial instance of the 24-Puzzle using the Manhattan distance heuristic. This instance may even be a challenge for a domain-specific graph-search algorithm to solve, since it requires expanding tens of billions of unique nodes and storing a frontier that consists of over two billion unique 24-Puzzle states – in fact, we are unaware of a domain-specific 24-Puzzle solver that can solve the same instance with full duplicate detection *and* the same heuristic. Finally, note that both the first and the last instance in Table 6 could not be solved using static partitioning, with available RAM.

The columns labeled "RAM" in Tables 4, 5 and 6, which show the peak RAM nodes for structured duplicate detection, also show the peak RAM nodes for hash-based delayed duplicate detection if it uses the same search algorithm and state-space projection function. Thus, as far as reducing peak RAM consumption is concerned, dynamic partitioning improves SDD and hash-based DDD *equally*.

Conclusion

For state-of-the-art external-memory graph search algorithms that partition the stored nodes of a state-space search graph into buckets that are stored as separate files on disk, we have introduced an approach to dynamic state-space partitioning in external-memory graph search that substantially reduces peak RAM consumption in exchange for a modest increase in running time. It can also reduce running time while leaving peak RAM consumption roughly the same. In some cases, it can even reduce both RAM consumption and running time. It achieves improved performance and a more favorable time-memory tradeoff than static partitioning because the partition is adapted to the actual distribution of stored nodes. Although the timing results of our experiments are for structured duplicate detection only, the approach can also be used to reduce the peak RAM requirements of hash-based delayed duplicate detection.

Although we have focused on external-memory search, we expect a similar approach to be effective for parallel graph search. For parallel search, dynamic partitioning limits the size of the largest sub-problem(s) created, effectively reducing the need for load balancing. As the number of cores packaged in the same processor increases, the issue of load balancing can become harder. The techniques described here suggests a new way to attack the load balancing problem in parallel graph search that does not increase the synchronization overhead among multiple cores.

References

Edelkamp, S., and Sulewski, D. 2008. Model checking via delayed duplicate detection on the GPU. Technical report, University of Dortmund.

Evangelista, S. 2008. Dynamic delayed duplicate detection for external memory model checking. In *Proc. of the 15th Int. SPIN workshop*, 77–94.

Haslum, P.; Helmert, M.; Bonet, B.; Botea, A.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. of the 22nd Conference on Artificial Intelligence (AAAI-07)*, 1007–1012.

Helmert, M.; Do, M.; and Refanidis, I. 2008. IPC 2008: Deterministic competition. `http://ipc.informatik.uni-freiburg.de/Results`.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Korf, R., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1–2):9–22.

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. 2008. Linear-time disk-based implicit graph search. *Journal of the ACM* 35(6).

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

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

Zhou, R., and Hansen, E. 2006b. Domain-independent structured duplicate detection. In *Proc. of the 21st National Conf. on Artificial Intelligence (AAAI-06)*, 1082–1087.

Zhou, R., and Hansen, E. 2007. Edge partitioning in external-memory graph search. In *Proc. of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2410–2416.

Zilles, S., and Holte, R. 2009. Downward path preserving state space abstractions (extended abstract). In *Proc. of the 8th Symposium on Abstraction, Reformulation, and Approximation (SARA-09)*, 194–197.