External-Memory Pattern Databases Using Structured Duplicate Detection

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

Abstract

A pattern database is a lookup table that stores an exact evaluation function for a relaxed search problem, which provides an admissible heuristic for the original search problem. In general, the larger the pattern database, the more accurate the heuristic function. We consider how to build large pattern databases that are stored in external memory, such as disk, and how to use an external-memory pattern database efficiently in heuristic search. To limit the number of slow disk I/O operations needed to construct and query an external-memory pattern database, we adapt an approach to external-memory graph search called structured duplicate detection that localizes memory references by leveraging an abstraction of the state space. We present results that show this approach increases the scalability of heuristic search by allowing larger and more accurate pattern database heuristics.

Introduction

Pattern databases are lookup tables that store exact solutions to relaxed problems. Because they account for complex interactions among multiple sub-goals of a search problem, pattern databases can provide very accurate admissible heuristics that significantly improve the scalability of heuristic search. Their effectiveness has been established in solving a variety of search problems, including sliding-tile puzzles (Culberson & Schaeffer 1998; Korf & Felner 2002); Rubik's Cube (Korf 1997); 4-Peg Towers of Hanoi (Felner *et al.* 2004); heuristic-search planning (Edelkamp 2001); and guided model checking (Qian & Nymeyer 2004). Pattern database heuristics are also closely related to large table-based heuristics used in multiple sequence alignment (McNaughton *et al.* 2002; Zhou & Hansen 2004b).

In general, the larger the pattern database, the more accurate the heuristic. As a result, the bottleneck in building and using large pattern databases is their memory requirements. There has been some recent work on improving the memory efficiency of pattern databases. Felner *et al.* (2004) describe an approach to compressing pattern databases by merging adjacent entries with similar values, in order to allow larger pattern databases to fit in memory in their compressed form. Holte et al. (2004) show how to improve the accuracy of a pattern database without increasing its size, by using multiple smaller pattern databases. Zhou and Hansen (2004b) show how to use the start and goal states of a problem instance to restrict the region of the state space for which a pattern database needs to be computed. But in all of these techniques, the size of the pattern database is still limited by the size of internal memory. In this paper, we consider how to build larger and more accurate pattern databases using external memory, and how to make efficient use of these external-memory pattern databases in heuristic search.

There is increasing interest in using external memory, such as disk storage, to improve the scalability of heuristic search. Recent work uses external memory to store already explored nodes in order to detect duplicates and prevent node re-generation in graph search. Korf (2004) and Edelkamp (2004) use a technique called *delayed duplicate detection* that is also used by theoretical computer scientists to analyze the complexity of external-memory graph search (Munagala & Ranade 1999; Mehlhorn & Meyer 2002). In keeping with its use in worst-case complexity analysis, delayed duplicate detection makes no assumptions about the structure of the search graph (except that it is undirected). Zhou and Hansen (2004c) introduce a technique called *structured duplicate detection* that assumes the search graph has local structure that can be revealed in an abstract state space. For graphs with sufficient local structure, structured duplicate detection has advantages over delayed duplicate detection. It *never* generates duplicates, unlike delayed duplicate detection, and thus has lower overhead and reduced complexity. We adopt structured duplicate detection as an approach to creating and using external-memory pattern databases, in part for these advantages, and in part, because the kind of state abstraction used in structured duplicate detection is the same kind of state abstraction used in pattern databases. This makes the two techniques a good fit for each other, and ensures that whenever it is possible to use one, it is possible to use the other.

Background

We begin with a review of pattern database heuristics and the technique of structured duplicate detection. In the rest of the paper, we show how these two techniques can be combined.

Pattern databases

A pattern database is an admissible heuristic that is computed by projecting a search problem into an abstract state space, solving the search problem for all of the abstract states, and storing the optimal evaluation function in a lookup table (Culberson & Schaeffer 1998). Typically, the 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. Each projected (or abstract) state is called a *pattern*; the variables used in the projection are called the *pattern variables*; and the projected (or abstract) state space is called the *pattern space*. The table in which a pattern database is stored has one entry for each pattern, and the value stored in each entry provides an admissible heuristic for any state in the original state space that maps to this pattern. The size of a pattern database is the number of patterns, and, naturally, different projection functions result in pattern databases of different size. As a rule, the larger the pattern database (i.e., the finer-grained the abstraction), the more accurate the heuristic, and there is evidence that the speed of search is inversely related to the size of the pattern database (Hernádvölgyi & Holte 2000). Although a single pattern database can be useful, more informed heuristics can be obtained by combining several pattern databases based on different projections. For example, pattern databases are disjoint if the set of pattern variables considered in each projected problem is disjoint from the others *and* each operator only affects one of the projected problems, which ensures that the sum of the pattern costs is admissible (Korf & Felner 2002).

Structured duplicate detection

Structured duplicate detection (Zhou & Hansen 2004c) is an approach to external-memory graph search that uses a state-space projection function that is similar to the projection function used to create a pattern database, but uses it to leverage the local structure of a search graph in checking stored nodes for duplicates. As in a pattern database, the projection function is created by ignoring some state variables. We refer to the state variables used in the projection function of structured duplicate detection as *duplicate-detection variables*, to distinguish them from pattern variables.

The state-space projection function creates an abstract state-space graph in which an abstract node y' is a successor of an abstract node y if and only if there exist two nodes x' and x in the original state-space graph, such that (a) x' is a successor of x, and (b) x' and x map to y' and y, respectively, under the projection function. Figure 1 shows an example of a very simple abstract state-space graph for the Eight Puzzle. An abstract state-space graph reveals local structure in a search problem if the number of successors of any abstract node is small relative to the total number of abstract nodes.

In structured duplicate detection, stored nodes in the original search graph are divided into "$n$blocks" with each $n$block 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 $n$blocks) that map to the successors of the abstract node y that is the image of node x under the projection function. 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. As a result, an external-memory graph search algorithm can use RAM to store $n$blocks within the current duplicate-detection scope, and use disk to store other $n$blocks when RAM is full.

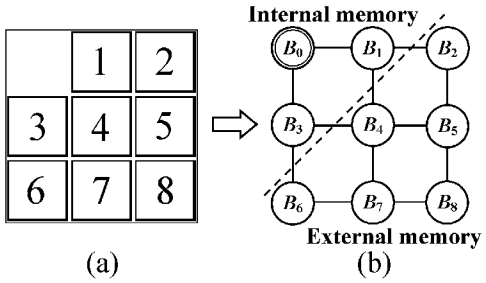

Figure 1: Panel (a) shows the goal state of the Eight Puzzle. Panel (b) shows an abstract state-space graph that is created by the simple projection function that considers the position of the "blank" only. The duplicate-detection scope of the goal state includes nodes that map to abstract nodes B_1 or B_3, and only these nodes need to be stored in internal memory when expanding the goal state.

Structured duplicate detection is designed to be used with a search algorithm that expands a set of nodes at a time, where the order in which the nodes in this set are expanded can be adjusted to minimize disk I/O. This includes A* (for search problems with many ties), breadth-first search, breadth-first branch-and-bound search, and related algorithms (Zhou & Hansen 2004a). The order of node expansions is chosen to ensure locality of memory references. Several rules can help achieve this. Expanding nodes in the same $n$block together, i.e., consecutively, results in reference locality because all nodes in the same $n$block have the same duplicate-detection scope. Expanding $n$blocks with similar duplicate-detection scopes consecutively also tends to result in reference locality. When RAM is full, $n$blocks outside the current duplicate-detection scope are flushed to disk. Writing the *least-recently used* $n$blocks to disk is one way to select which $n$blocks to write to disk. When expanding nodes in a different $n$block, any $n$blocks in its duplicate-detection scope that are stored on disk are swapped into RAM.

Unlike delayed duplicate detection, structured duplicate detection has a minimum internal-memory requirement, which is the size of the largest duplicate-detection scope. However, this is rarely an issue in practice because it can be controlled by changing the granularity of the state-space projection function. A finer-grained state-space projection function usually results in smaller duplicate-detection scopes, and thus, lower internal-memory requirements.

External-memory pattern databases

We now begin to discuss how to use structured duplicate detection to limit disk I/O in creating and using an external-memory pattern database. Our approach assumes the pattern database is used inside a search algorithm that also uses structured duplicate detection to limit the internal-memory requirements of graph search. The initial idea is simple: just as structured duplicate detection can localize memory references to nodes stored for duplicate detection, it can be used to localize memory references to patterns in a pattern database. However, it must localize memory references in pattern-database lookups in a way that exploits the *shared* local structure in both the abstract state space used for duplicate detection *and* the pattern space. Since the two are abstractions of the same state space, they usually have shared local structure.

Pattern-space projection function

Our approach to external-memory pattern databases extends the idea of structured duplicate detection from the state space to the pattern space. It relies on a *pattern-space projection function* that partitions the pattern space in much the same way that the state-space projection function used in structured duplicate detection partitions the state space. A pattern-space projection function is a many-to-one mapping from a pattern space P to an *abstract pattern space* $\tilde{P}$, in which each *abstract pattern* corresponds to a set of patterns in the original pattern space. If a pattern p is mapped to an abstract pattern $\tilde{p}$, then $\tilde{p}$ is called the *image* of p, and p is called the *pre-image* of $\tilde{p}$. A pattern-space projection function can be defined by ignoring some pattern variables in the encoding of a relaxed problem. We call the set of pattern variables used in the pattern-space projection function the *abstract pattern variables*.

However, the above definition of the pattern-space projection function is incomplete because it does not consider how to integrate an external-memory pattern database with structured duplicate detection. To integrate memory references to nodes stored for duplicate detection with memory references to patterns in a pattern database, we add the further restriction that abstract pattern variables must also be duplicate-detection variables. In other words, the abstract pattern variables must be selected from the intersection of the set of pattern variables and the set of duplicate-detection variables. (This entails that one must choose the set of duplicate-detection variables carefully so that their intersection with the set of pattern variables is non-empty.) The reason for this additional restriction is that structured duplicate detection partitions nodes stored in the Open and Closed lists into $n$blocks such that nodes in the same $n$block share the same value for the duplicate-detection variables. With this restriction on the abstract pattern variables, we can guarantee that nodes in the same $n$block map to the same abstract pattern, and that storing only the pre-images of a few of these abstract patterns in RAM is sufficient to answer all pattern-database queries when generating the successors of these nodes. We discuss the details of this idea next. Since a pattern-space projection function partitions a pattern database into groups of patterns, with each group corresponding to one abstract pattern, we introduce the term "$p$block" to refer to a set (or "block") of patterns in the pattern space that are pre-images of an abstract pattern.

The relationship between the pattern-space projection function and the state-space projection function used in structured duplicate detection, as well as the relationships between various abstractions of the state space, are illustrated in Figure 2. We adopt the following notation. Let S denote the original state space, let $\tilde{S}$ denote the abstract state space used in structured duplicate detection, and let $\Pi_{\tilde{S}}$ denote the state-space projection function. Let P denote the pattern space that is formed in creating a pattern database, let $\tilde{P}$ denote the abstract pattern space, and let $\Pi_{\tilde{P}}$ denote the pattern-space projection function. Using this notation and with these distinctions in mind, we can define the central concept of pattern-lookup scope, which determines which $p$blocks must be stored in RAM and which can be stored on disk only, at any point in the search process.

Pattern-lookup scope

Let x denote the node that is being expanded, let abstract node $y = \Pi_{\tilde{S}}(x)$ be the image of node x under the state-space projection function used for structured duplicate detection, and let $successors(y)$ be the set of successor abstract nodes of y in the corresponding abstract state-space graph.

Definition 1 *The* pattern-lookup scope *of node x with respect to an abstract pattern space $\tilde{P}$ is the union of sets of patterns that are pre-images of an abstract pattern $\tilde{p}$ such that an abstract node $y' \in successors(y)$ maps to the abstract pattern $\tilde{p}$ under the pattern-space projection function $\Pi_{\tilde{P}}$, or equivalently,*

$$\bigcup_{y' \in successors(y)} \Pi_{\tilde{P}}^{-1}(\tilde{p})$$

where $\tilde{p} = \Pi_{\tilde{P}}(y')$ and $\Pi_{\tilde{P}}^{-1}(\cdot)$ is a function that takes as input an abstract pattern in the abstract pattern space $\tilde{P}$ and returns the set of patterns that are pre-images of the abstract pattern.

The following theorem follows from the definition.

Theorem 1 *The pattern-lookup scope of a node contains all the patterns that can be queried when generating the successors of the node.*

The concept of pattern-lookup scope provides the foundation for external-memory pattern databases because it allows a search algorithm to use internal memory to store patterns within the pattern-lookup scope of a set of expanding nodes, and use external memory to store any or all of the other patterns, when internal memory is full. That is, only the $p$blocks in the pattern-lookup scope need to be stored in RAM.

The concept of pattern-lookup scope is analogous to the concept of duplicate-detection scope in structured duplicate detection, and there is a close correspondence between the two. For each duplicate-detection scope, there is a single pattern-lookup scope, and the pattern-lookup scope can only

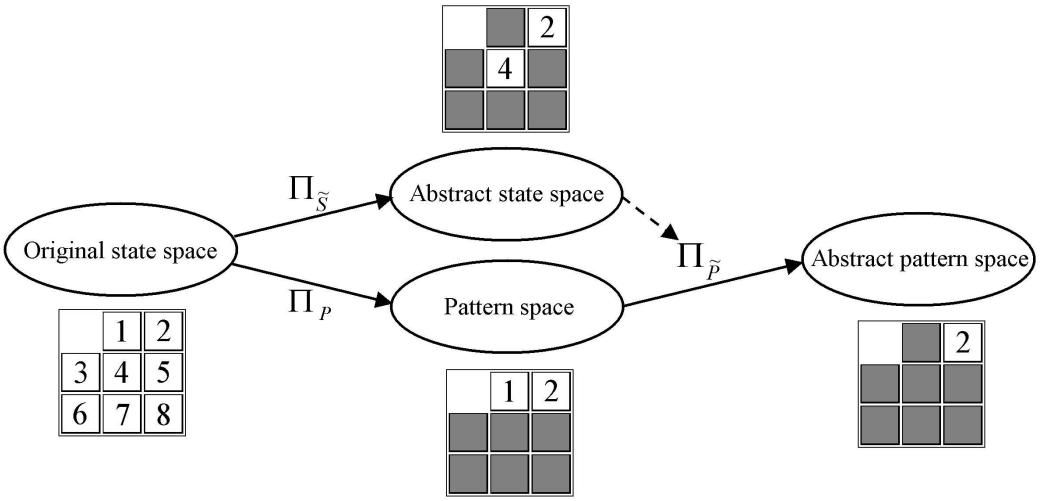

Figure 2: The diagram shows the four different spaces involved in external-memory pattern databases: the original state space S, the abstract state space $\tilde{S}$ used for structured duplicate detection, the pattern space P, and the abstract pattern space $\tilde{P}$. Solid arrows represent projection functions that transform one space into another. The dashed arrow represents the influence of the abstract state space on the choice of pattern-space projection function. In each space, a sample state of the Eight Puzzle is shown in which shaded tiles represent ignored state variables.

change when the duplicate-detection scope changes. But the pattern-lookup scope does not necessarily change when the duplicate-detection scope changes, since the same pattern-lookup scope can correspond to more than one duplicate-detection scope. Note that the largest pattern-lookup scope establishes a minimum internal-memory requirement for the external-memory pattern database.

Creating external-memory pattern databases

Pattern databases are typically created by a complete breadth-first traversal of the pattern space, in the backward direction from the goal pattern. Because we are interested in creating very large pattern databases that cannot fit in RAM, and since a breadth-first search algorithm that creates such a large pattern database usually cannot store all the nodes it visits in RAM, we create an external-memory pattern database by using structured duplicate detection inside a memory-efficient implementation of breadth-first search, such as breadth-first frontier search (Korf 2004).

Completion of the breadth-first search is not necessarily the last step in creating a pattern database. For efficient access to a pattern database, it is important to arrange patterns in a systematic way such that a unique index into the pattern database can be easily computed for each pattern. We refer to this pattern-arrangement process as *compilation* of a pattern database. Compiling an external-memory pattern database is challenging because one cannot take the ordinary approach of storing in RAM an array that maps each node to its unique index, since such an array would be as big as the entire pattern database. But since one can easily fit an entire $p$block in RAM, this suggests another way to compile an external-memory pattern database. When the breadth-first search algorithm expands a pattern, it finds the abstract pattern to which it maps, and then writes the pattern's abstract encoding and the cost of the shortest path to it from the goal pattern to a disk file associated with the abstract pattern. Because each abstract pattern corresponds to a file stored on disk, the algorithm needs to store in RAM an array of file descriptors, one for each abstract pattern. (Note that this array is the size of the abstract pattern space, which is usually exponentially smaller than the pattern space.) Once a complete breadth-first search is finished, each of these files is processed by a pattern-database compilation algorithm that sequentially reads in one pattern at a time, computes its unique index in the $p$block based on its pattern-space encoding, writes its cost to the corresponding position in the $p$block stored in RAM, and, if there are no more patterns left in the file, writes the entire $p$block to a pattern-database file stored on disk. This completes the compilation process for one $p$block, and the construction of the entire external-memory pattern database is completed when every $p$block has been compiled. Note that in a compiled $p$block, only the cost of a pattern is stored, because its encoding can be computed based on its position in the $p$block.

Compressing external-memory pattern databases

Like regular pattern databases, external-memory pattern databases can also be compressed, but in a different way. Recall that compressing pattern databases involves finding cliques in the pattern space such that the cost of patterns in the same clique differs by at most one (or a small constant). Unlike regular pattern databases, in which cliques often occur at adjacent positions (Felner *et al.* 2004), cliques in external-memory pattern databases usually do not occur at adjacent positions in the same $p$block, because each $p$block is an abstraction (and thus, a distortion) of the original pattern space. In fact, patterns that map to the same position in *different* $p$blocks tend to form cliques, because they may be

closer in the original pattern space.

We introduce an algorithm for compressing an external-memory pattern database. First, the algorithm finds C abstract patterns that form a clique in the abstract pattern space, where C is a *compression ratio* that usually depends on the domain as well as the pattern-space projection function. Then, it reads the cost of the first pattern from each of the C $p$blocks that correspond to these abstract patterns and writes their minimum cost to a file that serves as the compressed $p$block. Next, the algorithm reads the cost of the second pattern from each of these C $p$blocks, writes their minimum cost to the compressed $p$block, and so on, until there is no pattern left in each input $p$block. It is interesting to note that after compression, the size of a $p$block does not change, but the number of $p$blocks is reduced by a factor of C.

Using external-memory pattern databases

We next describe how to use an external-memory pattern database in a heuristic search algorithm. Because our approach relies on structured duplicate detection, it must be used as part of a search algorithm that expands a set of nodes at a time, where the order in which the nodes in this set are expanded can be determined entirely by structured duplicate detection. Except for depth-first search, this assumption holds quite broadly in many search algorithms, including breadth-first search, A* (for search problems with many ties), and related algorithms.

Our approach allows the user to specify the maximum number of $p$blocks that can be stored in RAM. When the search algorithm expands nodes in a different $n$block, it must check if the $p$blocks that form the pattern-lookup scope of the nodes in the $n$block are stored in RAM, and, if not, read them from disk. If the search algorithm already stores the maximum number of $p$blocks in RAM, it must remove from internal memory one or more $p$blocks that do not belong to the current pattern-lookup scope. Unlike structured duplicate detection, which needs to write $n$blocks to disk when RAM is full, our approach does not need to write any $p$block to disk at all, because each and every $p$block is stored on disk before the search begins.

When the part of internal memory reserved for $p$blocks is full, the search algorithm must decide which $p$blocks to remove from internal memory. We adopt the least-recently used strategy (Sleator & Tarjan 1985), because it is easy to implement and performs reasonably well. In our implementation, each $p$block has a time stamp that keeps track of the most recent access to it. That way, the algorithm can remove the $p$block that has not been accessed for the longest time.

External-memory disjoint pattern databases

To create multiple pattern databases, one needs to define multiple pattern-space projection functions, one for each pattern database. For multiple pattern databases, the pattern-lookup scope of a node is defined as the union of the node's pattern-lookup scopes for all pattern databases. It is straightforward to show that Theorem 1 still holds in this case.

Disjoint pattern databases are a special case of multiple pattern databases in which the set of pattern variables for each pattern database is disjoint from the pattern variables of all other pattern databases. In disjoint pattern databases, one must choose the set of duplicate-detection variables carefully so that it overlaps with the set of pattern variables for *each* external-memory pattern database. The reason for this is that the abstract pattern variables used in a pattern-space projection function must be duplicate-detection variables as well. If the two sets of variables do not overlap, then the pattern-space projection function trivially maps all patterns to a single abstract pattern, the $p$block for this abstract pattern is the entire pattern database, and there is no way to use external memory to store part of the pattern database.

As a design guideline, the larger the pattern database, the more its pattern variables should overlap with duplicate-detection variables. This creates a finer-grained pattern-space projection function that reduces the size of each $p$block by increasing the number of abstract patterns.

#d	Sol	Int Mem	Ext Mem	Exp	Secs
18	225	529K	0	20,890,457	20
19	257	12,011K	0	379,977,147	505
20	289	20,000K	52,005K	2,786,693,382	8,163
21	321	20,000K	366,297K	13,926,234,207	57,559

Table 1: Results for the 4-peg Towers of Hanoi problem. Columns show the number of disks (#d), solution length (Sol), peak number of nodes stored in RAM in thousands (Int Mem), peak number of nodes stored on disk in thousands (Ext Mem), number of node expansions (Exp), and running time in CPU seconds (Secs).

Computational results

We tested our approach to external-memory pattern databases in three domains, and summarize our results below. While the results show the effectiveness of our approach, they do not fully illustrate its potential, as it is possible to compute *much* larger pattern databases than these using this approach. All experiments were performed on a Pentium IV 3.2 GHz processer with 1 GB of RAM, 512 KB of L2 cache, and a 7200RPM Seagate disk with 400 GB of storage.

4-peg Towers of Hanoi

For the 4-peg Towers of Hanoi problem (TOH4), unlike the well-known 3-peg problem, the only way to find a provably optimal solution is by systematic search. Previously, the largest pattern database built for this problem contains 16 disks (Felner *et al.* 2004), because the algorithm that creates the pattern database only needs a bit array of size $4^{16} = 4$ gigabits (or 512 megabytes) of RAM to keep track of all patterns visited in the breadth-first search. It is not possible to create a 17-disk pattern database using the same method, because it would require a bit array of size $4^{17} = 16$ gigabits (or 2 gigabytes) to search the 17-disk pattern space, and our machine only has 1 gigabyte of RAM.

We used breadth-first frontier search with structured duplicate detection to create a 17-disk external-memory pattern database, which contains 17,179,869,184 patterns and has a size of 16 gigabytes. To the best of our knowledge,

this is the largest pattern database ever built. (Note that our approach would allow us to build much larger pattern databases than this *without* using more RAM.) In creating this pattern database, the search algorithm used less than 0.7 gigabytes of RAM and approximately 120 gigabytes of disk storage. We then compressed the 17-disk pattern database using the external-memory compression algorithm described previously. The compression ratio C used is 64, which reduces the size of the entire 17-disk pattern database to 256 megabytes. The maximum loss of accuracy due to compression is 5 (the number of steps for solving the 3-disk problem), because the positions of the three smallest disks are ignored when querying each compressed $p$block.

Using the 17-disk compressed pattern database and breadth-first heuristic search (Zhou & Hansen 2004a), we can solve larger instances of TOH4 than the previous state-of-the-art. Table 1 shows the results. Not only were we able to solve both the 18-disk problem (the largest previously-solvable problem) and the 19-disk problem in RAM, we also solved the 20- and 21-disk problems in external memory, using structured duplicate detection.

Note that for the standard start state in which all disks are placed on a single peg, one only needs to search half as deep due to symmetry. Korf's (2004) breadth-first frontier search (with delayed duplicate detection) exploits this symmetry to solve TOH4 with up to 24 disks. But this symmetry only applies to fixed start and goal states. For any other start state, one has to use a complete search to find a shortest path to the goal state, as our algorithm does.

Fifteen Puzzle

To solve the Fifteen Puzzle, we use structured duplicate detection inside an algorithm called breadth-first iterative-deepening A*, or BFIDA* (Zhou & Hansen 2004a). The algorithm is the same as breadth-first branch-and-bound search, except that it uses an iterative-deepening upper bound (similar to Korf's IDA*) and divide-and-conquer solution reconstruction to reduce its memory requirements. To guide the search, we created a disjoint pattern database heuristic that is based on two external-memory pattern databases, one for a group of 7 tiles and one for a group of 8 tiles. The 7-tile group contains tiles 1, 4, 5, 8, 9, 12, and 13. The 8-tile group contains the rest of the tiles.

The set of duplicate-detection variables considers the position of tiles 3, 5, 10, 12, and 15, plus the "blank." Note that tiles 5 and 12 belong to the 7-tile group, and tiles 3, 10, 15 belong to the 8-tile group. This creates an abstract pattern space with $16 \cdot 15 = 240$ abstract patterns for the 7-tile group, and an abstract pattern space with $16 \cdot 15 \cdot 14 = 3360$ abstract patterns for the 8-tile group. Hence a $p$block for the 7-tile group contains $(16!/9!)/240 = 240240$ patterns, and a $p$block for the 8-tile group contains $(16!/8!)/3360 = 154440$ patterns. The algorithm stores a maximum number of 100 $p$blocks for each external-memory pattern database. Thus, the peak internal-memory requirement is $(240240+154440)\cdot 100 = 39,468,000$ bytes, which is 15 times *less* than the internal-memory requirement of disjoint pattern databases based on these two groups of tiles that are stored entirely in RAM.

Name	Cost	Int PDB	Ext PDB	Exp	Secs
1amk	33,960	6,918	487,851	12,893K	148
1tis	37,581	5,963	453,733	10,249K	165
actin	52,117	14,469	1,814,077	88,657K	1,309
1ton	32,707	30,383	2,078,950	218,533K	2,826
1gtr	58,010	48,846	4,725,039	360,697K	5,885
2cba	34,294	46,371	3,729,917	666,682K	16,292
S52	35,713	37,555	4,545,352	1,074,340K	19,877
1bco	23,703	62,513	4,440,847	2,699,068K	98,236

Table 2: Results for aligning groups of 5 protein sequences from reference set 1 of BAliBASE. Columns show name of instance, cost of optimal alignment, peak number of patterns stored in RAM (Int PDB), peak number of patterns stored on disk (Ext PDB), number of node expansions in thousands (Exp), and CPU seconds (Secs).

We ran BFIDA* on the 10 most difficult of Korf's 100 random instances of the Fifteen Puzzle (Korf 1985). On average, each instance took 22.6 seconds to solve using structured duplicate detection and our external-memory disjoint pattern databases. For comparison, BFIDA* using structured duplicate detection and the Manhattan-distance heuristic took an average of 1083 seconds to solve each instance, although the internal-memory requirements of the two algorithms are about the same. In other words, using external-memory disjoint pattern databases results in a 48-times speedup of BFIDA* *without* using more RAM.

Note that the average runtime of BFIDA* in our experiments also includes the time it takes to read $p$blocks stored on disk in solving *every* instance. On the other hand, the average runtime using regular pattern databases may increase if there are only a handful of instances to solve. This is because the time it takes to read the entire disjoint pattern databases from disk, if not amortized over a sufficiently large number of instances, may become a significant part of average runtime. For example, reading the two regular disjoint pattern databases from disk actually takes about 12 seconds on our machine, and if only a single instance were to be solved, the average runtime would be at least 12 seconds no matter what algorithm is used; whereas the average runtime of BFIDA* in our experiments remains unaffected.

Multiple sequence alignment

Optimal alignment of multiple DNA or protein sequences is a challenging search problem for which A* has been shown to be very effective (McNaughton *et al.* 2002; Zhou & Hansen 2003; 2004b). When the standard sum-of-pairs cost function is used, admissible heuristics can be created by aligning subsets of the sequences. This creates large table-based heuristics that are essentially pattern databases, and can be summed in a manner similar to disjoint pattern databases. For example, in order to optimally align 5 sequences, we can create two triple-alignment pattern databases such that the sum of their values will be an admissible heuristic. However, the size of a triple-alignment pattern database is cubic in the length of a sequence, where the length (of a protein sequence) is typically in the few hundreds. To limit the amount of memory required to store

these large table-based heuristics, we used a technique described in (Zhou & Hansen 2004b) in combination with the external-memory pattern database technique presented in this paper. We used both in a memory-efficient version of A* that is very effective for multiple sequence alignment, called Sweep A* (Zhou & Hansen 2003), using structured duplicate detection.

We ran an external-memory version of Sweep A* using our external-memory pattern databases on real protein sequences from reference set 1 of BAliBASE, a widely-used benchmark (Thompson, Plewniak, & Poch 1999). All problems involve aligning 5 protein sequences. The cost function used is a Dayhoff substitution matrix with a linear gap penalty of 8. We built two triple-alignment pattern databases in solving each problem. The pattern-space projection function used ignores all but the longest sequence that is shared by both triples of sequences, thereby dividing each triple-alignment pattern database into L $p$blocks, where L is the length of the longest sequence.

Table 2 shows that our approach reduces the internal-memory requirements of the pattern databases by an average factor of 83 times. We also found that using external-memory pattern databases made Sweep A* run 16% *faster* than using traditional pattern databases of the same accuracy that are stored entirely in internal memory. The reason is that the amount of internal memory needed to store the $p$blocks is often small enough to fit (or almost fit) in the 512 kilobytes of L2 cache that our machine has, and pattern lookups are *much* faster in cache than in RAM. Of course, this is not guaranteed to happen. For example, it took a disproportionately long time to solve the most difficult problem (1bco) in Table 2, partly because the peak number of patterns stored in internal memory for this problem (62,513 patterns) does not quite fit in cache. With external triple-alignment pattern databases, we can solve multiple sequence alignment problems that could not be solved previously. For example, no one before has found provably optimal solutions for the largest three problems listed in Table 2.

Conclusion

Pattern databases have proved to be an effective way of automatically creating highly-accurate admissible heuristics. The accuracy of the heuristics typically increases with the size of the tables in which they are stored, and thus memory is a bottleneck in creating improved heuristics. We have introduced an efficient approach to creating and using external-memory pattern databases that uses structured duplicate detection to leverage shared local structure in both the pattern space and the abstract state space used for duplicate detection. This approach allowed us to create the largest pattern database ever constructed, and we showed that it significantly improves search performance in three domains. In the future, we plan to use this approach to create much larger external-memory pattern databases that could significantly improve the scalability of heuristic search.

Acknowledgements

We thank the anonymous reviewers for helpful comments. This work was supported in part by NSF grant IIS-9984952.

References

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(4):318–334.

Edelkamp, S. 2001. Planning with pattern databases. In *6th European Conference on Planning (ECP-01)*.

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

Felner, A.; Meshulam, R.; Holte, R.; and Korf, R. E. 2004. Compressing pattern databases. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, 638–643.

Hernádvölgyi, I., and Holte, R. 2000. Experiments with automatically created memory-based heuristics. In *Proc. of the Symposium on Abstraction, Reformulation, and Approximation (SARA-2000), Lecture Notes in Artificial Intelligence 1864*, 281–290.

Holte, R.; Newton, J.; Felner, A.; Meshulam, R.; and Furcy, D. 2004. Multiple pattern databases. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling*, 122–131.

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

Korf, R. 1997. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of 14th National Conference on Artificial Intelligence (AAAI-97)*, 700–705.

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.

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

McNaughton, M.; Lu, P.; Schaeffer, J.; and Szafron, D. 2002. Memory-efficient A* heuristics for multiple sequence alignment. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, 737–743.

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

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

Qian, K., and Nymeyer, A. 2004. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 497–511.

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

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

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

Zhou, R., and Hansen, E. 2004a. Breadth-first heuristic search. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling*, 92–100.

Zhou, R., and Hansen, E. 2004b. Space-efficient memory-based heuristics. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, 808–814.

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