Recent Progress in Heuristic Search: A Case Study of the Four-Peg Towers of Hanoi Problem

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Ariel Felner
Department of Information Systems Engineering
Ben-Gurion University of the Negev
P.O. Box 653, Beer-Sheva, 84105 Israel
felner@bgu.ac.il

Abstract

We integrate a number of recent advances in heuristic search, and apply them to the four-peg Towers of Hanoi problem. These include frontier search, disk-based search, multiple compressed disjoint additive pattern database heuristics, and breadth-first heuristic search. The main new idea we introduce here is the use of pattern database heuristics to search for any of a number of explicit goal states, with no overhead compared to a heuristic for a single goal state. We perform the first complete breadth-first searches of the 21 and 22-disc four-peg Towers of Hanoi problems, and extend the verification of a "presumed optimal solution" to this problem from 24 to 30 discs, a problem that is 4096 times larger.

Four-peg Towers of Hanoi Problem

The three-peg Towers of Hanoi problem is well known in computer science. It consists of three pegs, and a number of different sized discs, initially stacked in decreasing order of size on one of the pegs. The task is to transfer all the discs from their initial peg to a goal peg, subject to the rules that only the top disc on a peg can move at any time, and a larger disc can never be placed on top of a smaller disc. It is easily proven that the shortest solution to this problem requires $2^n - 1$ moves, where n is the number of discs.

The problem becomes much more interesting if we add an additional peg (see Figure 1). While the four-peg Towers of Hanoi problem is 117 years old (Hinz 1997), the optimal solution length is not known in general. In 1941, a recursive strategy was proposed that always constructs a valid solution (Frame 1941; Stewart 1941), and proofs that this "presumed optimal solution" was indeed optimal were offered, but there was an error in the proofs (Dunkel 1941), and the conjecture remains unproved. Without a proof, the only way to verify that the presumed optimal solution is indeed optimal for a given number of discs is through systematic search. Previously this had been done for up to 24 discs (Korf 2004), and we extend it here to 30 discs. The size of the problem space is 4^n, where n is the number of discs, since each disc can be on any of four pegs, and all the discs on any peg must be in sorted order. Thus, each additional disc multiplies the size of the problem space by a factor of four.

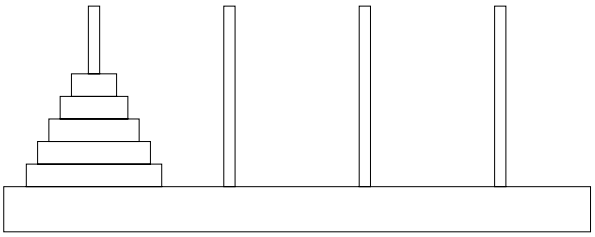

Figure 1: Four-Peg Towers of Hanoi Problem

In either problem, to move the largest disc from the initial to the goal peg, none of the smaller discs can be on either the initial or goal pegs. For the three-peg problem, this means that all but the largest disc must be on the remaining auxiliary peg, generating a subproblem of the same form as the original problem. This allows a recursive solution that is easily proven to be optimal. For the four-peg problem, however, moving the largest disc from one peg to another requires that all the remaining discs be distributed over the two remaining pegs. The difficulty is that we don't know how these discs should be distributed in an optimal solution.

Prior Work on Four-Peg Towers of Hanoi

Currently, systematic search is required to verify optimal solutions for given numbers of discs. The simplest approach is to perform a brute-force search from the initial state to the goal state. Complete breadth-first searches, which generate all states from a given initial state, have previously been done for up to 20 discs (Korf 2004).

There are two different sources of symmetry that can be used to speed up this search. The first is that transferring the discs from an initial peg to any of the other three pegs are all equivalent problems. Thus, given any state, we can permute the three non-initial pegs to derive an equivalent state, reducing the number of states that must be examined by almost a factor of six (Bode & Hinz 1999). The reduction is not quite a factor of six because if two of the non-initial pegs are empty, permuting them has no effect.

The other symmetry is the fact that moving all discs from the initial peg to the goal peg is equivalent to moving all discs from the goal peg to the initial peg. In practice, this

cuts the effective search depth in half, as follows (Bode & Hinz 1999). Assume that we find a path to a state where all discs but the largest are distributed over the two auxiliary pegs. Call such a state a *middle* state. We can then move the largest disc from the initial peg to the goal peg. If we then execute all the moves made to reach the middle state in reverse order, we will return all but the largest disc back to the initial peg. If, however, while executing these moves we interchange the initial peg with the goal peg, we will move all but the largest disc to the goal peg, completing the solution. Thus, once we reach a middle state, we can easily generate a complete solution, and once we find a shortest path to a middle state, we have found a shortest solution path. We refer to such a search as a *half-depth search*.

(Korf 2004) used both these symmetries in a brute-force half-depth search, verifying the presumed optimal solution length for up to 24 discs. Two additional techniques used in that work, frontier search (Korf *et al.* 2005), and storing the nodes on magnetic disk[1], are discussed below.

Verifying the optimal solution length for more discs requires a heuristic rather than a brute-force search. Previously, the largest four-peg Towers of Hanoi problem that had been solved optimally with heuristic search was 19 discs (Zhou & Hansen 2006). This is much smaller than the 24-disc problem solved with brute-force half-depth search, because the heuristic search was a full-depth search, computing the heuristic function to the single goal state.

The main new idea in this paper is to show how to compute a heuristic that estimates the length of a shortest path to any of a large number of different states. This allows us to combine heuristic search with half-depth searches, by estimating the length of a shortest path to any middle state. The naive approach of computing a heuristic estimate of the distance to each middle state, and taking the minimum of these values for each state of the search, is obviously impractical when the number of middle states is large. For the n-disc, four-peg Towers of Hanoi problem, there are 2^{n-1} possible middle states, one for each way to distribute the $n-1$ smallest discs over the two auxiliary pegs.

Pattern Database Heuristics

Pattern databases are heuristic functions based on lookup tables stored in memory (Culberson & Schaeffer 1998), and are the most effective heuristics known for many problems, including Rubik's cube (Korf 1997) and the sliding-tile puzzles (Korf & Felner 2002). A pattern database for the Towers of Hanoi problem contains an entry for each possible configuration of a subset of the discs, the *pattern discs*. The value of each entry is the number of moves required to move the pattern discs to the goal peg, ignoring any other discs in the problem (Felner, Korf, & Hanan 2004). Given a problem instance, for each state of the search, we use the configuration of the pattern discs to compute an index into the pattern database. The corresponding stored value is then used as a lower-bound heuristic on the number of moves needed to move all the discs to the goal peg.

[1]Throughout this paper we use "disc" to refer to a Tower of Hanoi disc, and "disk" to refer to a magnetic storage disk.

The size of a pattern database is limited by the memory available, and is a function of the number of pattern discs. For any configuration of n discs, we can compute a unique index in the range 0 to $4^n - 1$ by encoding the peg that each disc is on with two bits, and using the resulting binary number of $2n$ bits as the index. Thus, we only need to store the heuristic values and not the corresponding states in the pattern database. If the values are less than 256, each value can be stored in a byte of memory. Thus, a pattern database of 15 discs requires $4^{15} = 2^{30}$ bytes, or a gigabyte of memory.

To construct such a pattern database, we perform a breadth-first search starting with all 15 discs on the goal peg. As each new configuration of discs is first encountered, we store the depth at which it is encountered in the database. This search continues until all configurations have been generated. A pattern database only needs to be constructed once, written to disk, and then reloaded to solve different problem instances. Furthermore, note that any set of 15 different-sized discs will generate exactly the same pattern database, regardless of their actual sizes.

Pattern databases have been used in full-depth heuristic searches of the four-peg Towers of Hanoi problem with up to 19 discs (Felner, Korf, & Hanan 2004; Felner *et al.* 2004; Zhou & Hansen 2006). The main new idea in this paper is to use pattern databases with multiple goal states, such as the middle states in the Towers of Hanoi, for example.

Multiple-Goal Pattern Databases

Pattern databases are easily adapted to handle multiple goal states. Rather than seeding the breadth-first search queue with a single goal state, we start with all the goal states in the queue at depth zero. Thus, the depth of any state in such a breadth-first search is the length of a shortest path to any of the goal states. While this may appear obvious in hindsight, to our knowledge it has not been used nor published before.

For example, to construct a pattern database containing the number of moves needed to move 15 discs to one of two auxiliary pegs, we generate all 2^{15} states in which the 15 discs are distributed among the two auxiliary pegs, and assign each a value of zero in the database. These states are used to initialize the breadth-first search queue, and the search runs until all possible configurations of the 15 discs have been generated, recording their distances in the pattern database. Thus, for any state, the value in the pattern database is the minimum number of moves needed to move all the discs to one of the two auxiliary pegs.

While multiple-goal pattern databases is the main new idea in this paper, optimally solving the 30-disc problem required integrating a significant number of recent research results in this area. We briefly discuss each of these in turn.

Frontier Search

The resource that limits both the breadth-first search to generate pattern databases, and the heuristic search to verify the presumed optimal solution lengths, is storage for the search nodes. To reduce the number of nodes that must be stored, we use frontier search (Korf *et al.* 2005). For simplicity, we describe it in the context of breadth-first search. Normally,

a breadth-first search stores all the nodes generated in either a Closed list of expanded nodes, or an Open list of nodes generated but not yet expanded. A complete search of the 22-disc, four-peg Towers of Hanoi problem, would generate 4^{22} or almost 17.6 trillion nodes. Frontier search only saves the Open list, and not the Closed list, reducing the space required to the maximum number of nodes at any depth. In addition, it saves with each node the operators used to generate it, to avoid regenerating the parent of a node as one of its children. For a complete search of the 22-disc problem, this is only 278 billion nodes, a reduction of over a factor of 63. Additional work is required to generate the actual solution path, but this adds only a small overhead to the algorithm. In our case there was no overhead, since we don't need the actual solution path, but simply the depth of a given node, both to generate the pattern database, and to verify the presumed optimal solution lengths.

Storing Nodes on Disk

Even with frontier search, however, 278 billion nodes is too many to store in memory. The solution is to store the nodes on magnetic disk (Korf 2003; 2004; Korf & Shultze 2005). The fundamental challenge is to design algorithms that only access the nodes on disk in a sequential fashion, since random access of data stored on disk is prohibitively expensive.

The main reason that nodes are stored is to detect duplicate nodes, which are nodes representing the same state arrived at via different paths. Nodes are normally stored in a hash table, which is randomly accessed each time a new node is generated to check for duplicate copies. When storing nodes on disk, however, newly generated nodes are not checked immediately, but written out to files, and then duplicate nodes are merged later in a single sequential pass through each file. When this *delayed duplicate detection* is combined with frontier search in a problem such as the Towers of Hanoi, two levels of a breadth-first search must be stored, rather than just one (Korf 2004).

This algorithm proceeds in alternating phases of node expansion and merging to eliminate duplicates. During expansion, files of parent nodes are expanded, and their children are written to other files. All the nodes in a given file are at the same depth, and have their largest discs in the same position. The position of the largest discs is encoded in the file name, and hence the individual nodes in the file specify only the positions of the smallest discs. In our implementation, the positions of the smallest 14 discs are stored in the actual records. This number is chosen because 14 discs require 28 bits, and 4 more bits are used for used operator bits, allowing a single 32-bit word to represent a state.

Parallel Processing

Any algorithm that makes extensive use of file I/O should be parallelized so that the CPU(s) can be occupied while a process blocks waiting for I/O. In addition, dual processor systems and multiple-core CPUs are becoming increasingly common, offering further performance gains from parallelism. Our algorithm is multi-threaded, with different threads expanding or merging different files in parallel.

Since all nodes in a given file have their largest discs in the same position, any duplicate nodes are confined to the same file, and duplicate merging of a file can be done independently of any other files. When expanding nodes in a file, however, their children will belong in different files if the large discs are moved. Furthermore, the expansion of two different files can send children to the same file. Previous implementations of this algorithm (Korf 2003; 2004; Korf & Shultze 2005) have coordinated multiple threads so that files that can generate children that belong in the same file are not expanded at the same time. In our current implementation, we remove this restriction so that two different expansion threads may write children to the same file. In order to determine the size of the resulting file, we perform a system call. This both simplifies the code and also speeds it up, since there is less coordination among multiple threads.

Handling Large Numbers of Files

Previous implementations of this algorithm (Korf 2003; 2004; Korf & Shultze 2005) have maintained in memory a data structure containing information about each file, such as its size. This is not practical with very large numbers of files, however. For example, to verify the optimal solution to the 30-disc problem, we need to store states with 29 discs, as will be explained later. The positions of the 14 smallest discs are stored with each node, and the positions of the remaining 15 discs are encoded in the file name. This requires keeping track of over $2 \times 4^{15}/6 \approx 358$ million different files, since we have separate expansion and merge files. Thus, storing even the sizes of these files in memory is too expensive.

Instead, we store only two bits for each possible file, one for the expansion file at the current depth, and the other for the merge file at the next depth. If the corresponding file exists, the bit is set to one, and zero otherwise. If the file exists, then we perform a system call to determine its size. Performing a system call on every possible file just to tell if it exists is prohibitively expensive.

Since we only store one bit per file, we don't keep track of which expansion files have already been expanded, and thus we can't merge a file as soon as it is complete. Instead, we do all the file expansions at a given depth, followed by all the file merges. This further simplifies and speeds up the code, but at the cost of some additional disk space, since files are not merged as soon as possible. As the results below will show, however, time is a more constraining resource than disk space at this point.

Compressed Pattern Databases

As a pattern database includes more components of a problem, the heuristic values become more accurate, since the database reflects the interactions between the included components. In the case of the Towers of Hanoi, the more discs that are included, the more accurate the heuristic. The limitation is the memory available to store the pattern database. For example, a full pattern database based on the 22-disc four-peg Towers of Hanoi problem would require 4^{22} or about 18 trillion entries. We can "compress" such a pattern database into a smaller database that will fit in memory,

however, with only a modest lost of accuracy, as follows.

We generate the entire problem space of the 22-disc problem, in about 11 days. As each state is generated, we use the configuration of a subset of the discs, say the 15 largest, as an index into a pattern database in memory, and store in that database the shallowest depth at which that configuration of 15 discs occurs. This is called a compressed pattern database (Felner *et al.* 2004), and occupies only a gigabyte of memory. Each entry corresponds to a particular configuration of the 15 largest discs, but contains the minimum number of moves required to move all 22 discs to their goal peg, where the minimization is over all possible configurations of the remaining six smallest discs. Note that the values in this pattern database are significantly larger than the corresponding entries in a simple 15-disc pattern database, because the latter values ignore any smaller discs.

To use this compressed pattern database in a search of a problem with at least 22 discs, for each state we look up the position of the 15 largest discs, and use the corresponding table value as the heuristic for that state.

Disjoint Additive Pattern Databases

Assume we have a Towers of Hanoi problem with 29 discs, and two different simple pattern databases, one for 15 discs and one for 14 discs. Note that any move only moves a single disc. Thus, given any configuration of the 29 discs, we can divide them into two disjoint groups of 15 discs and 14 discs, look up the configuration of each group in the corresponding pattern database, and sum the resulting values to get an admissible heuristic value for the 29-disc problem. Furthermore, any disjoint partition of the discs will work. This is called a disjoint additive pattern database (Korf & Felner 2002; Felner, Korf, & Hanan 2004).

Combining compressed and additive pattern databases is straightforward. Again assume a problem with 29 discs, and a 22-disc pattern database compressed to the size of a 15-disc database. The compressed pattern database contains the number of moves needed to solve the 22 discs. We can construct a separate pattern database for the seven remaining discs, requiring only 4^7 or 32 kilobytes of memory. Then given a state, we can use the 15 largest discs to represent the 22 largest discs, look up the corresponding entry in the compressed pattern database, then look up the configuration of the seven smallest discs in the seven-disc pattern database, and finally add the two heuristic values together, to get an admissible heuristic for all 29 discs.

Multiple Pattern Databases

In order to sum the values from two different pattern databases, the corresponding sets of elements must be disjoint, and every move must move only a single element. However, given any two admissible heuristic functions, their maximum is usually a more accurate admissible heuristic. It has been observed that the maximum of two or three different pattern database heuristics is often more accurate than a single pattern database heuristic of the same total size, and more than compensates for the overhead of the multiple lookups (Holte *et al.* 2004).

Continuing our example with 29 discs, we can construct two different admissible heuristics as follows: One divides the 29 discs into the 22 largest and seven smallest discs, adding their pattern database values, and the other divides them into the seven largest and 22 smallest discs, again adding their database values. Finally, we take the maximum of these two values as the overall heuristic value. We use the same 22-disc database and 7-disc database for both heuristics, since the absolute sizes of the discs doesn't matter.

Breadth-First Heuristic Search

Frontier search with delayed duplicate detection can be combined with a best-first heuristic search such as A* (Hart, Nilsson, & Raphael 1968), but the frontier of a best-first search includes nodes at different depths, and is usually larger than a frontier of nodes all at the same depth. A better solution to this problem is an algorithm called breadth-first heuristic search (Zhou & Hansen 2006). Breadth-first heuristic search is given a cost threshold, and searches for solutions whose cost does not exceed that threshold. The search is a breadth-first search, except that as each node is generated, the cost of the path to the node, $g(n)$, is added to the heuristic estimate of the cost of reaching a goal from that node, or $h(n)$. If the cost of the node $f(n) = g(n) + h(n)$ exceeds the associated cost threshold, the node is deleted. Breadth-first heuristic search is a form of frontier search, and only stores a few levels of the search at a time. Thus, once a solution is found, additional work is required to construct the solution path. If the optimal solution cost is not known in advance, a series of iterations can be performed with successively increasing cost thresholds, until the optimal solution is found. This algorithm is called breadth-first iterative-deepening-A* (Zhou & Hansen 2006).

Breadth-first heuristic search is ideally suited to our Towers of Hanoi problem for several reasons. One is that to verify the presumed optimal solution length, we don't need to construct the solution path, since these solutions are known. Of course, if we were to find a shorter solution, we would want to generate it to verify that it is correct. Secondly, since we know the presumed optimal solution length in advance, we can set the cost threshold to this value, eliminating the need for iterative deepening.

Minimizing Heuristic Calculations

For many search problems, the time to compute the heuristic evaluations is a significant fraction of the running time. This is particularly true with multiple heuristics, or for large pattern databases that reside in main memory, since they are randomly accessed, resulting in poor cache performance.

For pattern database heuristics, it is easy to determine a maximum possible heuristic value, from the maximum value in each database. For many search algorithms, such as breadth-first heuristic search or iterative-deepening-A* (IDA*) (Korf 1985), pruning occurs by comparison to a known cost threshold. Under these circumstances, we can speed up a heuristic search by only computing heuristics for nodes that could possibly be pruned. In particular, if t is the cost threshold, and m is the maximum possible heuris-

tic value, then we need not compute any heuristics for those nodes n for which $g(n) + m \leq t$. To the best of our knowledge, this technique has not appeared in the literature before.

Brute-Force Search of Towers of Hanoi

We first describe complete brute-force searches of the four-peg Towers of Hanoi problem. These are used to compute pattern databases, and are also of interest in this problem for another reason. For the three-peg problem, the minimum number of moves needed to transfer all n discs from one peg to another is 2^{n-1}. Furthermore, if we perform a breadth-first search starting with all discs on a single peg, once the search to depth 2^{n-1} is complete, all states in the problem space will have been generated at least once. This is known as the *radius* of the problem space from this initial state. In other words, the radius of the problem from the standard initial state is the same as the optimal solution length from the standard initial state to the standard goal state.

It was believed that this was true of the four-peg problem as well. However, (Korf 2004) showed that this is not true for the 15-disc and 20-disc four-peg problems. For the 15-disc problem, the optimal solution length to transfer all discs from one peg to another is 129 moves, but there exist 588 states that are 130 moves from the standard initial state. For the 20-disc problem, the optimal solution depth is 289 moves, but the radius of the problem space from the standard initial state is 294 moves. An obvious question is what happens for larger problems.

We ran the first complete breadth-first searches of the 21- and 22-disc four-peg Towers of Hanoi problems, starting with all discs on one peg. The machine used for all our experiments was an IBM Intellistation A Pro, with dual two gigahertz AMD Opteron processors, and two gigabytes of memory, running CentOS Linux. We have three terabytes of disk storage available, consisting of four 500 gigabyte Firewire external drives, and a 400 and two 300 gigabyte internal Serial ATA drives.

Table 2 shows our results. The first column shows the number of discs, the second column the optimal solution length to transfer all discs from one peg to another, the third column the radius of the problem space from the standard initial state, the fourth column the width of the problem space, which is the maximum number of unique states at any depth from the standard initial state, and the last column the running time in days:hours:minutes:seconds, running six parallel threads. We conjecture that for all problems with 20 or more discs, the radius of the problem space from the standard initial state exceeds the optimal solution length.

D	Len.	Rad.	width	time
21	321	341	77,536,421,184	2:10:25:39
22	385	395	278,269,428,090	9:18:02:53

Table 1: Complete Search Results for Towers of Hanoi

Heuristic Search of Towers of Hanoi

We now consider heuristic searches of the four-peg Towers of Hanoi problem. As mentioned above, there exists a solution strategy that will move all discs from one peg to another, and a conjecture that this strategy generates a shortest solution, but the conjecture is unproven. Prior to this work, the conjecture had been verified for up to 24 discs (Korf 2004). Here we extend this verification to 30 discs. Note that the 30-disc problem space contains 4^{30} states, which is 4096 times larger than that of the 24-disc problem.

We first built a 22-disc pattern database, compressed to the size of a 15-disc database, or one gigabyte of memory. This was done with a complete breadth-first search of the 22-disc problem, seeded with all states in which all discs were distributed over two auxiliary pegs. We used the 6-fold symmetry described above, since all non-initial pegs are equivalent. The pattern database itself did not use this symmetry, but contained an entry for all 4^{15} possible configurations of the 15 largest discs, in order to make database lookups more efficient. It took 11 days to construct the pattern database, and required a maximum of 784 gigabytes of disk storage. The main reason that this is longer than the time for the complete search of the 22-disc problem described above is that the compressed pattern database occupied a gigabyte of memory, and thus we were only able to run four parallel threads instead of six. We also built small complete pattern databases for up to seven discs, in almost no time.

The half-depth heuristic searches were done using breadth-first heuristic search, starting with all discs on the initial peg. The goal of the heuristic searches was any middle state in which all but the largest disc are distributed over the two auxiliary pegs. For example, in the 30-disc problem, we are looking for a middle state where we have moved the smallest 29 discs off the initial peg to two auxiliary pegs. Thus, the largest disc is not represented at all. This search also used the six-fold symmetry described above.

If a middle state is found in k moves, then a complete solution to the problem with $2k + 1$ moves exists, utilizing the symmetry between the initial and goal states. Breadth-first heuristic search was run with the cost threshold set to $(p - 1)/2$, where p is the presumed optimal solution depth. In fact, it would be more efficient to set the cost threshold to one move less, checking only for the existence of shorter solutions. To increase the confidence in our result, however, we used the $(p - 1)/2$ cost threshold, and checked that an actual solution was found. Note that a search with a given cost threshold will find all solutions whose cost is less than or equal to the threshold.

For the 30-disc problem, the search was pure breadth first until depth 272, the shallowest depth where heuristic pruning was possible, based on the maximum possible heuristic value. Beyond this depth, for each state we added the depth to its heuristic value, and deleted the state if the sum exceeded the cost threshold.

For each state we took the maximum of two heuristics. For the 29-disc problem, for example, which only uses 28 discs, one value was the compressed pattern database value for the 22 largest discs, added to the complete pattern database value for the six smallest discs. The other was the compressed database value for the 22 smallest discs, plus the complete database value for the six largest discs. For the 28-disc problem, we used the 22-disc compressed database,

and looked up the remaining five discs in a complete pattern database, and similarly for the smaller problems.

We implemented our parallel search algorithm using multiple threads. The number of threads is limited by the available memory, since each thread needs it own local storage. The pattern databases are shared, however, since they are read-only. In our experiments we ran four parallel threads on two processors. We used all seven disks, evenly dividing the files among the disks, to maximize I/0 parallelism.

The 30-disc problem took over 28 days to run, generated a total of over 7.1 trillion unique nodes, and required a maximum of 1.28 terabytes of disk storage. Unlike the others, this problem was run with only a 21-disc compressed pattern database, but is currently running with the full 22-disc compressed database.

The results are shown in Table 1, for all problems not previously verified. The first column gives the number of discs, the second column the length of the optimal solution, the third column the running time in days:hours:minutes:seconds, the fourth column the number of unique states generated, and the last column the maximum amount of storage needed in gigabytes. In all cases, the presumed optimal solution depth was verified.

D	Len.	time	unique states	storage
25	577	2:37	443,870,346	.1
26	641	14:17	2,948,283,951	.5
27	705	1:08:15	12,107,649,475	2.1
28	769	4:01:02	38,707,832,296	9.1
29	897	2:05:09:02	547,627,072,734	111
30	1025	28:16:23:34	7,106,487,348,969	1,284

Table 2: Heuristic Search Results for Towers of Hanoi

Conclusions

The main new idea of this paper is how to efficiently perform a heuristic search for a shortest path to any of a large number of goal states. The solution we propose is based on using a pattern database for the heuristic function. A pattern database heuristic for a large number of goal states can be constructed by initially seeding the breadth-first search queue with all the possible goal states at depth zero. We also integrate a large number of recent advances in heuristic search, including frontier search, disk-based search, breadth-first heuristic search, multiple pattern databases, disjoint pattern databases, and compressed pattern databases. Additional new improvements we introduce include simplifying and speeding up the parallel search algorithm by reducing coordination between different threads, handling very large numbers of files by storing only one bit per file, and a method to eliminate many heuristic calculations by knowing the maximum possible heuristic value. We demonstrate these techniques by verifying the presumed optimal solution depth for four-peg Towers of Hanoi problems with up to 30 discs, which is 6 more discs than the previous state of the art. We also performed the first complete breadth-first searches of the 21 and 22-disc problems, uncovering further examples where the radius of the problem space exceeds the optimal solution length. We conjecture that this will be true of all larger problems as well.

Acknowledgements

Thanks to Peter Schultze for keeping our disk array running. This research was supported by NSF under grant No. EIA-0113313 to Richard Korf.

References

Bode, J.-P., and Hinz, A. 1999. Results and open problems on the Tower of Hanoi. In *Proceedings of the Thirtieth Southeastern International Conference on Combinatorics, Graph Theory, and Computing*.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Dunkel, O. 1941. Editorial note concerning advanced problem 3918. *American Mathematical Monthly* 48:219.

Felner, A.; Meshulam, R.; Holte, R.; and Korf, R. 2004. Compressing pattern databases. In *AAAI-04*, 638–643.

Felner, A.; Korf, R.; and Hanan, S. 2004. Additive pattern database heuristics. *JAIR* 22:279–318.

Frame, J. 1941. Solution to advanced problem 3918. *American Mathematical Monthly* 48:216–217.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

Hinz, A. M. 1997. The Tower of Hanoi. In *Algebras and Combinatorics: Proceedings of ICAC'97*, 277–289. Hong Kong: Springer-Verlag.

Holte, R.; Newton, J.; Felner, A.; Meshulam, R.; and Furcy, D. 2004. Multiple pattern databases. In *ICAPS-2004*, 122–131.

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

Korf, R., and Shultze, P. 2005. Large-scale, parallel breadth-first search. In *AAAI-05*, 1380–1385.

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

Korf, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. 1997. Finding optimal solutions to Rubik's cube using pattern databases. In *AAAI-97*, 700–705.

Korf, R. 2003. Delayed duplicate detection: Extended abstract. In *IJCAI-03*, 1539–1541.

Korf, R. 2004. Best-first frontier search with delayed duplicate detection. In *AAAI-04*, 650–657.

Stewart, B. 1941. Solution to advanced problem 3918. *American Mathematical Monthly* 48:217–219.

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