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 new and recent advances in heuristic search, and apply them to the four-peg Towers of Hanoi problem. These include frontier search, disk-based search, parallel processing, multiple, compressed, disjoint, and additive pattern database heuristics, and breadth-first heuristic search. New ideas include pattern database heuristics based on multiple goal states, a method to reduce coordination among multiple parallel threads, and a method for reducing the number of heuristic calculations. 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 "presumed optimal solutions" to this problem from 24 to 30 discs. Verification of the 31-disc problem is in progress.

1 Towers of Hanoi Problems

The standard Towers of Hanoi problem consists of three pegs, and n different sized discs, initially stacked in decreasing order of size on one peg. The task is to transfer all the discs from the initial peg to a goal peg, by only moving one disc at any time, and never placing a larger disc on top of a smaller disc. To move the largest disc from the initial peg to the goal peg, none of the smaller discs can be on either peg, but must all be on the remaining auxiliary peg.[1] This generates a subproblem of the same form as the original problem, allowing a simple recursive solution. It is easily proven that the shortest solution to this problem requires $2^n - 1$ moves.

The problem becomes more interesting if we add another 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. The difficulty is that moving the largest disc from the initial to the goal peg requires that the remaining discs be distributed over the two auxiliary pegs, but we don't know a priori how to distribute them in an optimal solution. In 1941, a recursive strategy was proposed that constructs a valid solution [Frame, 1941; Stewart, 1941], and optimality proofs of this "presumed optimal solution" were offered, but they contained an unproven assumption [Dunkel, 1941], and the conjecture remains unproven. Absent a proof, the only way to verify optimality of this solution for a given number of discs is by a systematic search for shorter solutions. 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 the discs on any peg must be in sorted order. Thus, each new disc multiplies the size of the problem space by four.

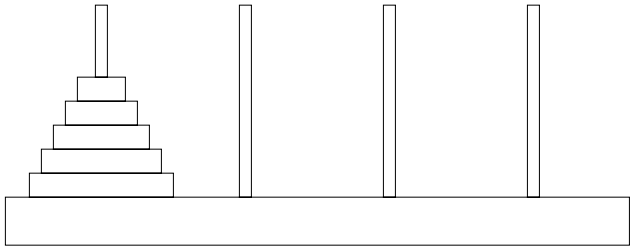

Figure 1: Four-Peg Towers of Hanoi Problem

2 Prior Work on Four-Peg Towers of Hanoi

The simplest search is a brute-force search, with no heuristics. Complete breadth-first searches, generating all states from a given initial state, have previously been done for up to 20 discs [Korf, 2004].

There are two different symmetries 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 equivalent problems. Thus, given any state, we sort the three non-initial pegs in order of their largest disc, reducing the number of states by almost a factor of six [Bode & Hinz, 1999]. The reduction is slightly less than six because if two non-initial pegs are empty, permuting them has no effect.

The other is a symmetry of the solution path [Bode & Hinz, 1999]. Assume that we find a path to a *middle state*, where all discs but the largest are distributed over the two auxiliary pegs. We can then move the largest disc from the initial to the goal peg. If we then execute the moves made to reach the middle state in reverse order, we will return all but the largest disc to the initial peg. If, however, while executing these moves, we interchange the initial and goal pegs, we will

[1] An auxiliary peg is any peg other than the initial or goal peg.

move all but the largest disc to the goal peg, completing the solution. Thus, once we reach a middle state, we can generate a complete solution, and a shortest path to a middle state generates a shortest solution path. In practice, this cuts the search depth in half, and we refer to this as a *half-depth search*.

[Korf, 2004] used both these symmetries in a brute-force half-depth search to verify the presumed optimal solution lengths for up to 24 discs. Additional techniques he used, such as frontier search, storing nodes on magnetic disk[2], and parallel processing, are discussed below.

Verifying the optimal solution length for more discs requires a heuristic search. Previously, the largest four-peg Towers of Hanoi problem that had been solved optimally with heuristic search was 19 discs [Zhou & Hansen, 2006], compared to 24 discs for brute-force half-depth search. The reason is because the heuristic search was a full-depth search, computing the heuristic to the single goal state.

The most important new idea in this paper is how to compute a heuristic that estimates the length of a shortest path to any of a number of goal states. This allows us to implement a heuristic half-depth search, by estimating the length of a shortest path to a middle state. The naive approach of computing a heuristic to each middle state, and taking the minimum of these values for each state of the search, is impractical with a large number of middle states. For the n-disc, four-peg 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. This can be reduced to 2^{n-2}, by assigning the second-largest disk to one of the auxiliary pegs.

3 Pattern Database Heuristics

Pattern databases 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 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 from their given configuration to the goal peg, ignoring all other discs [Felner, Korf, & Hanan, 2004]. Given a problem instance, for each state in the search, we use the configuration of the pattern discs to compute an index into the database. The corresponding value is then used as a lower-bound estimate of the number of moves needed to move all the discs to the goal peg.

The size of a pattern database depends on the number of pattern discs. We can represent any configuration of n discs by a unique index in the range 0 to $4^n - 1$, by encoding the location of each disc with two bits, and using the resulting $2n$-bit binary number as the index. Thus, we only store the heuristic values and not the corresponding states in the database. If the values are less than 256, each value is stored in a single byte. 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 corresponding search depth in the database. This search continues until all configurations have been generated. A pattern database is only constructed once, and can be used to solve any problem instances with the same goal state. Note that any set of 15 different-sized discs generate exactly the same pattern database, regardless of their actual sizes.

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

4 Multiple-Goal Pattern Databases

Our main new contribution is pattern databases for multiple goal states. Rather than initializing the breadth-first search queue with one goal state, we seed it with all the goal states at depth zero. Then, the depth of any state in the breadth-first search is the length of a shortest path to any goal state.

For example, to construct a pattern database of the number of moves needed to transfer 15 discs to two auxiliary pegs, we initialize the search queue with 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. The breadth-first search runs until all configurations of 15 discs have been generated, recording their depths in the database. Thus, for any state, the database value is the minimum number of moves needed to move all the discs to the two auxiliary pegs.

While this may appear obvious in hindsight, it escaped the attention of at least three groups of researchers working on the Towers of Hanoi. [Felner, Korf, & Hanan, 2004] used heuristic search with additive pattern database heuristics to solve up to 17 discs. [Felner *et al.*, 2004] used compressed pattern databases to solve up to 18 discs. [Zhou & Hansen, 2006] solved up to 19 discs, using breadth-first heuristic search with pattern databases. All of these were full-depth heuristic searches to a single goal state, using pattern database heuristics. At the same time, all of these researchers knew that brute-force half-depth searches to multiple goal states could solve larger problems [Korf, 2004], but didn't see how to combine the two together.

Multiple-goal pattern databases also have real-world applications. Consider a car navigation system that must quickly plan a route to the nearest hospital. We first run Dijkstra's algorithm [Dijkstra, 1959] in a preprocessing step, initially seeded with all hospital locations, storing with each map location the distance to the closest hospital. For this search, the direction of any one-way streets is reversed. Then, when the user requests a route from his current location to the nearest hospital, A* will find an optimal path in linear time, since the heuristic is exact. As another application, suppose we want to label each address in a city with the identity of the closest fire station, to aid in dispatching fire trucks. We run Dijkstra's algorithm, initially seeded with the location of all fire stations, but modified so that each node also includes the identity of the closest fire station, in addition to its distance. In this case, we don't reverse one-way streets, because we are traveling from the fire station to the address.

In addition to multiple-goal pattern databases, verifying the presumed optimal solution of the 30-disc problem required integrating a number of additional recent and new research results in heuristic search, which we discuss below.

5 Frontier Search

The limiting resource for both breadth-first search to generate pattern databases, and heuristic search to verify optimal solutions, is storage for search nodes. Linear-space depth-first searches are completely impractical for this problem, because they generate too many duplicate nodes representing the same state. To reduce the number of stored nodes, we use frontier search [Korf *et al.*, 2005]. For simplicity, we describe breadth-first frontier search. A standard breadth-first search stores all nodes generated in either a Closed list of expanded nodes, or an Open list of nodes generated but not yet expanded. Frontier search saves only the Open list, and not the Closed list, reducing the nodes stored from the total number of nodes in the problem, to the maximum number of nodes at any depth, or the *width* of the problem. For example, the total number of states of of the 22-disc, four-peg Towers of Hanoi problem is 4^{22} or 17.6 trillion, while the width of the problem is only 278 billion states, a savings of a factor of 63. It also saves with each node the operators used to generate it, to avoid regenerating the parent of a node as one of its children. Additional work is required to generate the solution path. In our case, however, we don't need the solution path, but simply the depth of a given node, both to generate pattern databases, and to verify presumed optimal solutions.

6 Storing Nodes on Disk

Unfortunately, 278 billion nodes is too many to store in memory. One solution is to store nodes on magnetic disk [Korf, 2003; 2004; Korf & Shultze, 2005]. The challenge is to design algorithms that rely on sequential access, since random access of a byte on disk takes about five milliseconds.

The main reason to store nodes is to detect duplicates, which are nodes representing the same state, but reached via different paths. This is normally done with a hash table, which is randomly accessed whenever a node is generated. On disk, however, duplicates nodes are not checked for immediately, but merged later in a sequential pass through the disk. When this *delayed duplicate detection* is combined with frontier search in a problem such as the Towers of Hanoi, two levels of the search tree must be stored at once [Korf, 2004].

The algorithm expands one level of the search space at a time. There are two types of files: expansion files that contain no duplicate nodes, and merge files that contain duplicates. All nodes in a file are at the same depth. When expansion files are expanded, their children are written to merge files at the next depth. For fault tolerance, and ease of interrupting and resuming the program, we save all expansion files at the current depth, until completing the iteration to the next depth.

All nodes in the same file have their largest discs in the same positions. The nodes within a file specify the positions of the 14 smallest discs. The 28 bits needed to specify 14 discs, plus four used-operator bits, equals 32 bits or a single word. For example, the 30-disc problem requires storing the positions of 29 discs. 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, generating up to $4^{15}/6 \approx 179$ million file names.

Previous implementations of this algorithm [Korf, 2003; 2004; Korf & Shultze, 2005] have stored in memory an array entry for each possible file name, but this doesn't scale to very large name spaces. Instead, we maintain in memory a hash table with the names of the files that exist at any point in time. For each file name, we store whether a corresponding expansion file exists at the current and/or next depth, whether the former has been expanded, whether a merge file exists, and a list of the expansion file names that could contribute children to this merge file. To conserve disk space, we merge merge files as soon as all such expansion files have been expanded.

7 Parallel Processing

Any I/O intensive algorithm must be parallelized to utilize the CPU(s) while a process blocks waiting for I/O. Furthermore, increasingly common multi-processor systems, and multiple-core CPUs, offer further opportunities for parallelism. Our algorithm is multi-threaded, with different threads expanding or merging different files in parallel. Merging eligible files takes priority over file expansions.

Since all nodes in a given file have their largest discs in the same positions, any duplicate nodes are confined to the same file, and duplicate merging of different files can be done independently. When expanding nodes, however, any children generated by moving the large disks will be written to different merge files, and the expansion of two different files can send children to the same merge file. Previous implementations of this algorithm [Korf, 2003; 2004; Korf & Shultze, 2005] have coordinated multiple threads so that files that can write children to the same merge file are not expanded at the same time. In our implementation, we remove this restriction. Two different threads may independently write children to the same merge file, as long as the files are opened in append mode. This both simplifies the code and speeds it up, since it requires almost no coordination among multiple threads.

8 Compressed Pattern Databases

A pattern database heuristic that includes more components of a problem is more accurate, since the database reflects 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 to store the database. For example, a full pattern database based on the 22-disc four-peg problem would require 4^{22} or about 17.6 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 22-disc problem space in about 11 days, using disk storage. As each state is generated, we use the configuration of say the 15 largest discs as an index into a pattern database, and store in that entry the shallowest depth at which that configuration of 15 discs occurs. That entry is 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 seven smallest discs. 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. This is called a compressed pattern database [Felner *et al.*, 2004], and occupies only a gigabyte of memory in this case.

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.

9 Disjoint Additive Pattern Databases

Consider a Towers of Hanoi problem with 29 discs, a simple 22-disc pattern database, and an seven-disc pattern database. A move only moves a single disc. Thus, given any configuration of the 29 discs, we can divide them into any two disjoint groups of 22 and seven discs each. We look up the configuration of the 22 discs in the 22-disc database, look up the configuration of the seven discs in the eight-disc database, and sum the resulting values to get an admissible heuristic for the 29-disc problem. This is called a disjoint additive pattern database [Korf & Felner, 2002; Felner, Korf, & Hanan, 2004]. There is nothing special about this 22-7 split, but maximizing the size of the largest database gives the most accurate heuristic values.

We can add values from a compressed database the same way. Again assume a problem with 29 discs, a 22-disc pattern database compressed to the size of a 15-disc database, and a simple seven-disc database. Given a state, we use the 15 largest discs to represent the 22 largest discs, and look up the corresponding entry in the compressed pattern database. We also look up the configuration of the seven smallest discs in the eight-disc database, and add these two heuristic values together, to get an admissible heuristic for all 29 discs.

10 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 also admissible. Furthermore, the maximum of several different pattern database heuristics is often more accurate than the value from a single pattern database of the same total size, and more than compensates for the overhead of the additional 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 can use the same two databases for both heuristics.

11 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 a best-first search frontier includes nodes at different depths, and is usually larger than a breadth-first search frontier, with all nodes at the same depth. A better solution to this problem is breadth-first heuristic search (BFHS) [Zhou & Hansen, 2006]. (BFHS) is given a cost threshold, and searches for solutions whose cost does not exceed that threshold. The cost of a node n is $f(n) = g(n) + h(n)$, where $g(n)$ is the cost from the initial state to node n, and $h(n)$ is the heuristic estimate of the cost of reaching the goal from node n. BFHS is a breadth-first search, but any node whose total cost exceeds the cost threshold is deleted. BFHS is also 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, breadth-first iterative-deepening-A* [Zhou & Hansen, 2006] performs a series of iterations with successively increasing cost thresholds, until an optimal solution is found.

BFHS is ideally suited to the Towers of Hanoi for several reasons. One is that to verify optimality of the presumed optimal solutions, we don't need to construct the solution paths, but simply show that no shorter solutions exist. If we were to find a shorter solution, however, we would need to generate it to verify its correctness. Secondly, since we know the presumed optimal solution length in advance, we could set the cost threshold to one less than this value, and perform a single iteration, eliminating iterative deepening.

12 Minimizing Heuristic Calculations

For many search problems, the time to compute heuristic evaluations is a significant fraction of the running time. This is particularly true of large pattern databases, since they are randomly accessed, resulting in poor cache performance.

For pattern database heuristics, we can determine the maximum possible heuristic value from the maximum values in each database. For search algorithms such as BFHS or iterative-deepening-A* (IDA*) [Korf, 1985], pruning occurs by comparison to a given cost threshold. We can speed up these searches by only computing heuristics for nodes that can possibly be pruned. In particular, if t is the cost threshold, and m is the maximum possible heuristic value, then we don't compute heuristics for those nodes n for which $g(n)+m \leq t$, since they can't possibly be pruned. We don't need to compute the heuristic at the shallowest possible depth that pruning could occur either, since the cost of additional node expansions may be compensated for by the faster speed of a brute-force search. Since BFHS searches breadth-first, we don't even need to load the pattern databases until the depth at which we start computing heuristics, saving the memory for additional parallel threads.

13 Brute-Force Search Experiments

We first ran complete brute-force searches of the four-peg Towers of Hanoi, both to compute pattern databases, and for another reason as well. For the three-peg problem, a breadth-first search to depth 2^{n-1}, starting with all discs on one peg, will generate all states of the problem at least once. Thus, 2^{n-1} is the *radius* of the problem space from this initial state. For the three-peg problem, this radius is the same as the optimal solution length to move all discs from one peg to another.

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, 129 moves are needed to move all discs from one peg to another, but there exist 588 states that are 130 moves from the standard initial state. For the 20-disc problem, the optimal solution is 289 moves, but the radius of the problem from the standard initial state is 294 moves. What happens for larger problems was unknown.

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 we used is 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. Note that for both problems, the radius from the standard initial state exceeds the optimal solution depth, and we conjecture that this is true for all problems with 20 or more discs.

D	Optimal	Radius	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

14 Heuristic Search Experiments

We next ran heuristic searches of the four-peg Towers of Hanoi. As explained above, there exists a solution strategy that moves all discs from one peg to another, and a conjecture that it is optimal, but the conjecture is unproven. Prior to this work, the conjecture had been verified for up to 24 discs [Korf, 2004]. We extended this verification to 30 discs.

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 pegs. We used the 6-fold symmetry described above to generate the database, since all non-initial pegs are equivalent. The database itself did not use this symmetry, however, but contained an entry for all 4^{15} possible configurations of the 15 largest discs, to make database lookups more efficient. It took almost 11 days to construct the pattern database, and required a maximum of 392 gigabytes of disk storage. The reason that this is longer than the time for the complete search of the 22-disc problem described above is that we had to access the pattern database for each node expanded, and were only able to run five parallel threads instead of six. We also built simple pattern databases for up to seven discs, essentially instantaneously.

We then ran breadth-first heuristic search, starting with all discs on the initial peg, searching for a middle state in which all but the largest disc are distributed over the two auxiliary pegs. For example, in the 30-disc problem, a middle state is one where the 29 smallest discs have moved 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.

If a middle state is found in k moves, then a complete solution of $2k + 1$ 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. It would be more efficient to set the cost threshold to one move less, checking only for the existence of shorter solutions. To increase our confidence in our results, however, we used the $(p - 1)/2$ cost threshold, and checked that an actual solution was found. A search with a given cost threshold will find all solutions whose cost is less than or equal to that threshold.

For each problem, we took the maximum of two heuristics. For the 30-disc problem, which only uses 29 discs, one value was the compressed pattern database value for the 22 largest discs, plus the complete pattern database value for the seven smallest discs, and the other was the compressed database value for the 22 smallest discs, plus the complete database value for the seven largest discs. For the smaller problems, we also used the 22-disc compressed database, and looked up the remaining discs in a complete pattern database.

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 five parallel threads on two processors. We used all seven disks, evenly dividing the files among the disks, to maximize I/0 parallelism.

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 expanded, and the last column the maximum amount of storage needed in megabytes. In all cases, the presumed optimal solution depth was verified. The 30-disc problem took over 17 days to run, generated a total of over 4.3 trillion unique nodes, and required a maximum of 398 gigabytes of disk storage. Even with two processors, the limiting resource is CPU time.

D	Opt.	time	unique states	space
25	577	2:37	443,870,346	43
26	641	14:17	2,948,283,951	263
27	705	1:08:15	12,107,649,475	1,079
28	769	4:01:02	38,707,832,296	4,698
29	897	2:05:09:02	547,627,072,734	56,298
30	1025	17:07:37:47	4,357,730,168,514	397,929

Table 2: Heuristic Search Results for Towers of Hanoi

A search of the 31-disc problem is in progress at press time.

It has been running for three months, and is expected to finish in a couple weeks. It required two terabytes of disk storage.

For comparison purposes, we also ran a brute-force half-depth search of the 25-disc problem using substantially the same program with no heuristics. This took 100 hours, compared with 2.6 minutes for the heuristic half-depth search.

15 Future Work: Perimeter Search

Multiple-goal pattern databases also provide a way to significantly improve the efficiency of a technique called *perimeter search* [Dillenburg & Nelson, 1994; Manzini, 1995]. The idea is to perform a small breadth-first search backward from the goal state, and store a perimeter of nodes surrounding the goal state at a fixed distance d. Then, in the forward search, instead of computing a heuristic to the single goal state, we compute a heuristic to each of the perimeter states, and take the minimum of these distances, plus d, as the heuristic estimate of the cost to the goal state. The drawback of this technique is that it requires for each state as many heuristic calculations as there are perimeter nodes. If we use a pattern database heuristic, however, and seed the breadth-first search used to construct the pattern database with all the perimeter nodes, then a single database lookup gives us the minimum heuristic value to any of the perimeter nodes. If we add d to this value, we get the improved accuracy of the perimeter-based heuristic, with no additional overhead. Integrating this technique with multiple and/or additive pattern databases is non-trivial, however, and the subject of future work.

16 Conclusions

We show how to efficiently perform a heuristic search for a shortest path to any of a large number of explicit goal states, using a pattern database heuristic. It is constructed by initially seeding the breadth-first search queue with all the goal states at depth zero. Multiple-goal pattern databases allow us to perform heuristic searches to any of a large number of "middle states" at half the solution depth in the Towers of Hanoi. This idea also has applications to real-world multiple-goal problems, such as, "take me to the nearest hospital". We also integrated a large number of recent advances in heuristic search, including frontier search, parallel disk-based search, breadth-first heuristic search, multiple pattern databases, disjoint additive pattern databases, and compressed pattern databases. Additional new improvements we introduce include simplifying and speeding up our parallel search algorithm by reducing coordination between different threads, and a method to eliminate many heuristic calculations, based on a maximum possible heuristic value. By combining all these techniques, we were able to verify the presumed optimal solution depth for the four-peg Towers of Hanoi problem with up to 30 discs, a problem 4096 times larger than the 24-disc previous state of the art. A search of the 31-disc problem is still in progress at press time. We also performed the first complete breadth-first searches of the 21 and 22-disc problems, showing that the radius of the problem space from the standard initial state exceeds the optimal solution length. We conjecture that this will be true of all larger problems as well.

17 Acknowledgements

Thanks to Peter Schultze for technical support, and to IBM for donating the Intellistation. This research was supported by NSF grant No. EIA-0113313 to Richard Korf, and by the Israel Science Foundation grant No. 728/06 to Ariel Felner.

References

[Bode & Hinz, 1999] 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 & Schaeffer, 1998] Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

[Dijkstra, 1959] Dijkstra, E. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

[Dillenburg & Nelson, 1994] Dillenburg, J., and Nelson, P. 1994. Perimeter search. *Artificial Intelligence* 65(1):165–178.

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

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

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

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

[Hart, Nilsson, & Raphael, 1968] 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, 1997] Hinz, A. M. 1997. The Tower of Hanoi. In *Algebras and Combinatorics: Proceedings of ICAC'97*, 277–289. Hong Kong: Springer-Verlag.

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

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

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

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

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

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

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

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

[Manzini, 1995] Manzini, G. 1995. BIDA*, an improved perimeter search algorithm. *Artificial Intelligence* 75(2):347–360.

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

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