An Implementation of 5-Peg Towers of Hanoi with PDB Heuristics

Reza Ghanbari

Introduction

The Tower of Hanoi puzzle has been a subject of study in computer science and mathematics for many years. The problem involves moving a stack of disks of different sizes from one peg to another while following specific rules. One of the challenges of solving the Tower of Hanoi problem is the large search space that needs to be explored to find a solution. In the case of the classic Tower of Hanoi problem with three pegs and twelve disks, there are over 530,000 possible arrangements of the disks that need to be explored to find a solution.

To reduce the search space required to solve the problem, researchers have developed several techniques, including the use of pattern databases (PDBs). A pattern database is a pre-computed look-up table that stores information about the possible moves from a given state. By using pattern databases, it is possible to significantly reduce the number of states that need to be explored to find a solution to the problem. This makes the problem more tractable and allows for the use of more sophisticated algorithms.

In this project, we implemented the 5-peg Tower of Hanoi problem with 16 disks and explored the use of pattern databases to reduce the search space required to solve the problem. We implemented A* algorithm for searching through the state-space using the information stored in pattern databases as our heuristic. In the implementation, the State class is used to represent the state and to generate all possible states for the problem and ensure that the generated states were legal and non-symmetric. We then use different splits of state and look up the pattern databases to find the best heuristics value for each state.

Problem Definition

The problem of 5-peg Towers of Hanoi with 16 disks can be defined as follows: Given 16 disks of different sizes and five pegs arranged in a line, initially all the disks are placed on the first peg in decreasing order of size from bottom to top. The objective of the problem is to move all the disks from the first peg to the last peg while following the standard rules of Towers of Hanoi puzzle. The rules include that only one disk can be moved at a time and the larger disk cannot be placed on top of a smaller one. Moreover, in this problem, a disk can be moved from any peg to any other peg (not just adjacent pegs) as long as it obeys the rules. The problem is considered solved when all the disks are moved to the last peg in the same order as initially arranged on the first peg. The task is to find the shortest sequence of moves that solves the problem.

The inputs of the problem of 5-peg Towers of Hanoi with 16 disks are the initial state of the puzzle, which consists of the 16 disks arranged in a stack on one of the five pegs, and the goal state, which is to have all the disks stacked on one of the other pegs in the correct order from largest to smallest. The outputs of the problem are the sequence of moves required to transform the initial state to the goal state, and the length of this sequence, which is the minimum number of moves required to solve the puzzle optimally.

In our implementation, the configuration of the problem is given by a set of constant values stored in the Constants.h file, which include the number of pegs, the number of disks, and the size of each of two pattern databases used to solve the problem. In the output, it reports the progress by printing the number of expanded and generated nodes in each cost, the solution length, the path from the start state to the goal, and the execution time of both the algorithm and the PDB creation/loading time.

Methodology

In this section, we will explain the design decisions we have made to implement the problem, and the search algorithm. First, state representation and the pattern database creation are discussed, and then heuristic calculation is explained. We represent the state of the puzzle by an array, in which each item represents a disk, and the value of that item shows the peg number on which the is located. The disks are represented in increasing order, meaning that the bigger the disk, the bigger the index. An example of this is shown in Figure 1, which shows the array representing the state of a three-peg Towers of Hanoi puzzle with five s. Using this representation, we can easily convey all the information we need in a single array.

Additionally, it can be easily used for ranking. To generate the ranking of each state, we concatenate the binary representations of the values of each element of the array. The reason behind this ranking function is that it is faster to

generate such ranking since it only includes basic bit operations; moreover, considering each disk as a 5-bit number and converting that into a decimal as a ranking may not improve our implementation since we are using the symmetry avoidance (which is explained below) that removes the ranking of some of the states and avoids storing ranks in an array, which is the efficient implementation of that approach. Therefore, using bits is more useful when we are removing all symmetric states.

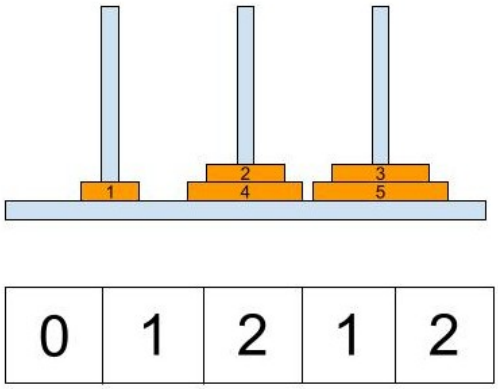

Figure 1: An example of representation of the problem's state as an array

To generate the pattern database, we use the Breadth-First Search (BFS) algorithm to find the distance of the states in the abstract state space from the goal state. As we are generating two pattern databases, considering 12-disks and 4-disks abstractions of the original problem, we would end up generating hundreds of millions of states. The number of states for 12-disks would be equal to 5^12, and 5^4 is the number of states in the state space of 4-disks. Therefore, we would need to find the heuristic value for more than 244 million states, which is a very time-consuming operation. To cope with this problem, we avoid generating symmetric states. In this problem, two states are symmetric if we can generate one from another one only by changing the order of pegs - all pegs except the goal peg can be reordered, and the goal peg is the one in which all the disks are located in the goal state. An example of such symmetry is shown in Figure 2.

Regarding the fact that we have four pegs without considering the goal peg, each state would have $4!$ symmetric states. So, by removing symmetric states, we would achieve a pattern database that contains 24 times fewer states than the original one.

We avoid generating symmetric states by mapping all the states to one of their symmetries in which the pegs are sorted according to the number of disks on them. This way, all the different symmetric states would be mapped to only one of their representations, so we can store that one and use it to avoid generating other symmetries.

We also generate pattern databases for the center state goal (Korf and Felner 2007), which means the goal in this case are the states in which all of the disks except the last one are located in neither in the initial nor the goal peg, and the last disk could be in any of these pegs.

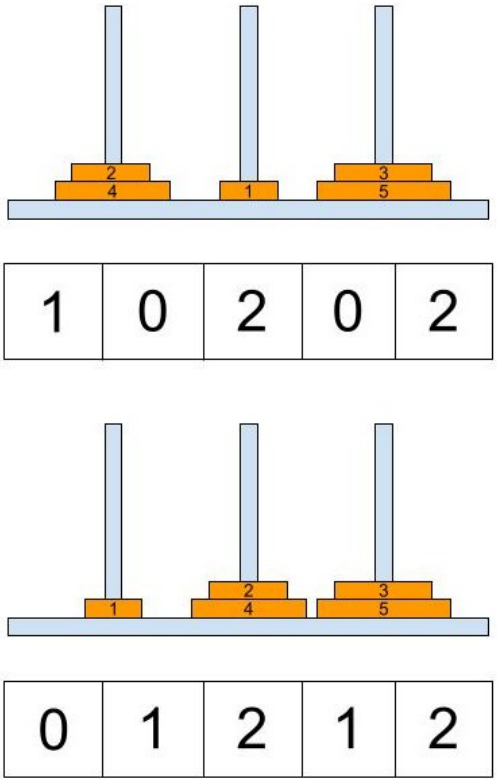

Figure 2: An example of two symmetric states in 3-peg TOH, the goal peg is the most right one

To generate this PDB, we should generate all the goal states. With that in mind, we first generate all those combinations by running a DFS to generate the combinations and then add them one by one to our BFS queue. Here, we also avoid symmetric states to generate fewer goals. In this problem symmetric states could be found by exchanging the position of the middle pegs, so for each state we would have 6 symmetric states. Considering our problem configuration, without symmetry avoidance, we should generate $3^11 \times 5$ states as goal states for 12 disk abstraction (since we do not use 2 of 5 pegs, the base is 3, and the power is 11 since we should not put the last disk in the middle pegs, in fact it is similar to generating a PDB with 11 disks, also the 5 is the number of pegs that the last disk could be located), and $3^3 \times 5$ goal states for 4 disk abstraction. By removing symmetric states, we generate almost 6 times fewer states, which would be 147624 + 24 states as the goal in total. This makes the search time for generating the PDB much shorter; this heuristic would result in optimal solutions since the last disk could be in any peg; Consequently, in the original search where only one of the two PDBs takes the last disk, the other PDB that does not have the last disk would not over-estimate the cost since its last selected disk could also be in one of the middle pegs with the heuristic value of 0. Therefore, the

heuristic value of the state that moves the last disk to its final position would be zero using these two PDBs.

After knowing how we built two pattern databases, we can explain how we query the PDB to get the heuristic values. Using the two pattern databases - one with 12 disks and another with four disks abstraction - we can calculate the heuristic for each state by adding the values of the corresponding ranks of that state in these two pattern databases, as long as the disk sets we are choosing to rank are disjoint. From now on, a valid disk separation will be referred to as a split, as we are dividing the state array into two sets. Another important point is that the only thing that matters for the ranking function is the order of the disks, and to query a PDB for k disks, we can use any k disks for the ranking as long as they form a sequence with correct order, and ignore all other disks; for example to query 4-disk PDB, we can use the first four disks for the ranking, or the disks 1, 3, 5, and 7, and ignore all the disks that are between them; they both would result in a valid heuristic value. Therefore, we can use any random splits that preserves the ordering, and give them to the ranking function to get a heuristic value. By taking the maximum of these heuristics, we would reach a final heuristic which is admissible. So, one important matter would be what splits of disks we are using to query the pattern databases. One way to tackle this issue is to randomly select a large enough number of splits and calculate the heuristic for them for a constant number of states, and then, take the splits that generate the best heuristic values.

Considering the admissibility of heuristics, we would expect that the ones that perform well in the first couple of states would continue to suggest useful heuristic values during the search, at least in the early stages.

Implementation

In this section we briefly explain the implementation details. The code is written in C++ 20, and Clang++ with LLVM15 is used as the compiler. The code is compiled by -O3 and -fopenmp flags.

One of the most critical classes in our implementation is the *State* class, which represents the state of the puzzle in our implementation. The state class contains three main arrays and an integer that determines the total number of disks. One of the arrays called *state* is the main array that represents the state; The other two are auxiliary arrays that are used during the process of symmetry avoidance. *numberOfDisksInPegs* keeps track of the total number of disks in each peg, and *topDiskInPegs* stores in index of the disks which are on top of each peg. This array is useful for legality check, using this the operation could be done in $O(1)$, as well as tie-breaking during the symmetry avoidance, since during the process of sorting the pegs, two pegs might have the same number of disks in them. Moreover, in generating the children, we can use this array to find the disk which we can move to other pegs; using this we can do the process of children generation in $O(p^2)$ instead of $O(n^2)$ (p and n stand for the number of pegs and the number of disks respectively).

One of the main functions of the state class is to generate all the possible states that can be reached from the current state. This is done by looping through the pegs and checking if we can move the top disk of the current peg to the target peg while following the puzzle's rules. Once we have found an eligible target peg, we call the *generateChildState* function. In this function, we create a new state by making a copy of the parent state's arrays, moving the disk to the target peg, and reorganizing the arrays according to this change. This process is done in the *move* function. In that function, we modify the arrays to reflect the current status of the state by changing the number of disks on the pegs involved in the movement of the disk and finding the new top disk for the peg where the disk is moved from.

In addition to these functions, there is another function called *getCompressedState*, which generates the binary representation of each item in the state array and concatenates them all together to create a number that represents the entire state. This way, we save up a considerable amount of memory by compacting a State object that has three arrays and a number of type uint8_t into a uint64_t number (for each State, it reduces the amount of space allocated to the state representation from 27 bytes to 8 bytes). This compact representation is stored in the closed list of A* algorithm.

The second main class in our implementation is the Heuristic class, which contains the functions that generate the pattern databases and also ranking/unranking functions. Moreover, it contains some functions that serve as interfaces that enable other entities to interact with the PDB.

One of the main functions in this class is *getRank* that generates the rank of a given state. To do so, it first finds the mapping from this state to its sorted symmetric by the help of *getMappingForSymmetry* function; This mapping is used in the *convertStateToInt* function to generate the same representation for symmetric arrays. This function concatinates the bit representations of each item of the *state* array, and returns the result as the rank of that state. This process is similar to *getCompressedState* but with a small yet important difference, which is the fact that this function compresses the sorted symmetry of a state instead of the state itself.

getMappingForSymmetry function, that is used in above process, is a sorting network that sorts all the pegs except the goal peg according to the number of disks on them; in cases where two pegs have the same number of disks, it uses *topDiskInPegs* array to break the tie. This tie-breaking has a noticeable impact on the final number of states, for example for the PDBs that do not use the midpoint optimization, the process would generate 25,108,270 PDB entries for 12-disk, while using this tie-breaking process decreases the number of entries to 10,172,527 (almost 60% fewer entries). For midpoint optimization, it applies the explained process only on the 3 pegs in the center.

Another important function in this class is createPDB, that generates the entries of the PDB by running BFS over all the states in the abstract state-space, starting from the goal state. In this implementation of BFS, the closed list would be the PDB that the search has generated, and the open list is a queue of numbers, that are the ranking of each state in the frontier.

There are some smaller functions in this class that are used during the process of ranking and accessing the PDB as well. *getRankFromArrays* is a function that does the exact

same things as *getRank* with a difference in its argument list. On the other hand, *getUnrankedState* function returns the corresponding state of rank. Also *getHeuristicValue* functions are used to read the corresponding value of a rank from the PDB. Finally, *readFromFile* and *saveToFile* functions are used to load and store PDB entries in a file.

The strategy of split selecting is implemented in the *Selections* class. First, it randomly generates a constant number of unique splits that are large enough to cover a sufficient amount of splits space. In our case, we have $\binom{16}{4}$ = 1820 cases in total, and 1000 unique splits have been randomly selected; so the chance of selecting each possible split would be more than 50%. After that, it evaluates these splits in a constant number of states (100 in our example) at the very beginning of the search process and keeps the heuristic values generated by each split. Finally, it selects a constant number of the splits whose average heuristic values are higher than the others, and keeps them together with the heuristics that have already generated maximum values as the splits that could be used during the rest of the search process.

In our case, we keep 100 splits during the search, and we use parallelism to calculate the heuristic values of all of them to make use of the system resources and reduce the execution time. This approach would give our implementation enough flexibility to fit in different hardware configurations, and also scale well when there are more hardware resources available. To do so, the only thing that is needed is to change the constant values related to the Selection class that are stored in the Constants header file.

The process of randomly selecting the splits is defined in the constructor of the Selection class. Knowing that we should choose 4 disks out of 16, we generate a vector containing the numbers of all disks, shuffle that vector in each iteration (to avoid repetition), select the first four as the split and store them after sorting the numbers (since order of disks is important for the PDBs, we keep and use the sorted numbers). To store the splits, we consider each of 4-selected numbers as a hexadecimal value and store the equivalent integer of that value in a map; since the selected numbers are sorted and each disk appears at most once in a selection, we would never see two numbers that represent same set of disks.

The main function of this class is *getHCost*, which calculates the heuristic value for a given state array. First, we take the maximum of the two splits mentioned in the assignment description. Then, if we are still in the process of evaluating heuristics, we calculate the heuristics suggested by all the random splits that were generated in the constructor, and keep the results in a map. If we have already passed the process of finding heuristics, we just calculate the heuristic value of each of the selected heuristics in a parallel for loop. For that, the *getHCost* function calls *getHCostOfSelection* for each specific split. This function generates the arrays needed for calculating the heuristic, and uses the interface given by Heuristic class to get the result.

Selecting the best heuristics is done in *getRandomSelections* function. In this function, whenever we reach the limit of evaluating the heuristics, we first add all the heuristics we have already used their values into the final set. Then, we take the heuristics that generated the higher values on average for the remaining spots.

There are also two functions, *generatePairFromNumber* and *convertNumberToSelection*, that are used to convert a split into an integer number and vice versa.

The main body of A* is implemented in the solver class. In our implementation, the close list is a map that maps every state to its parent, and the open list is stored in a priority queue. We store the compressed representation of the state as well as its g-cost and h-cost in the open list. The reason for keeping h-cost is the computation overhead of recalculating the h-cost value, and we keep g-cost to break the tie in favor of the nodes with higher g-cost whenever there is more than one state with the same f-cost. This tie-breaking changes if we set our implementation to search for the midpoint, since in that case the goal is near to the start, breaking ties in favor of higher g-cost would result in generating states that are in a deeper parts of the search tree than the goal.

In the main function, we instantiate all the needed classes and create the PDBs, and run the algorithms. We also calculate the execution time of the algorithm and PDB creation. There are some flags in main, like *SAVE_PDB* or *READ_PDB*, that are used to set whether we want to read our PDB entries from a file or write those entries into a file. All of these flags could be configured in the Constants header file.

Experiments and Results

In this section, the results of the experiments are discussed. The hardware specification of the system used for the experiment is an x86-64 architecture with 20 CPUs, 14 cores per socket, and 2 physical threads per core. It is a 12th generation Intel Core i9-12900H processor with a maximum frequency of 5000 MHz, 24 MB L3 cache, and approximately 16 GB memory.

Four experiments have been conducted using the implementation. Two of them examine the impact of tie-breaking by g-cost when using midpoint optimization. Another one is for the case where we find the optimal solution without using midpoint optimization; in this experiment, we use a PDB that is generated regarding the goal state as the one with all the disks on the goal peg. The last experiment is for the case in which we find sub-optimal solutions without using midpoint optimization; in this experiment, we also use the same PDB as the previous one. Each of these experiments have been conducted 3 times and the average of these executions has been reported.

The first experiment is evaluating the impact of tie-breaking with the g-cost when we use midpoint optimization. In Figure 3, we show the difference in the number of generated nodes when we break the tie in favor of high g-cost versus low g-cost. As can be observed, as the cost increases, the difference between these two approaches would become more noticeable. This may have happened due to the fact that the goal is quite near to the start state. As a result, by breaking ties in favor of the higher g-costs, we direct the search towards deeper parts of the search tree and would generate more nodes.

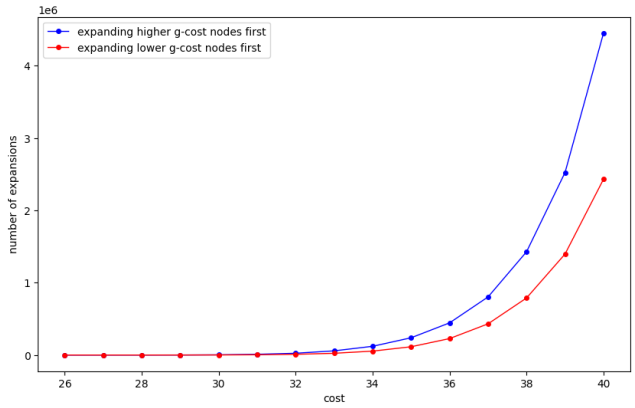

Figure 3: The number of nodes generated by expanding higher g-cost first vs. expanding lower g-cost first

Approach	Solution Length
Optimal midpoint with low g-cost	40.0
Optimal midpoint with high g-cost	40.0
Sub-optimal midpoint	41.0
Optimal without midpoint	79.0

Table 1: The solution length for each approach

Table 1 displays the solution length of each approach. It is evident from the table that using the optimal midpoint heuristics result in reaching the solution exactly in the middle of the path toward the goal (here the goal state is the state in which all the disks are in the goal peg). This implies that the use of heuristics significantly reduces the search space and leads to more efficient solutions. On the other hand, using sub-optimal heuristics leads to a non-optimal solution that is relatively close to the optimal solution.

Table 2 reports the number of node expansions for each approach, with the results indicating that the higher g-cost tie-breaking strategy resulted in more node expansions than the lower g-cost tie-breaking strategy, which was expected. Furthermore, the sub-optimal midpoint heuristic approach resulted in significantly higher node expansions than the first two approaches. This is due to the fact that the PDB used in the optimal approach guides the search towards a goal state where all disks are on the goal peg, resulting in the generation of more states before reaching the midpoint.

Interestingly, the number of node expansions for the optimal solution that does not use midpoint optimization was much higher than all the other approaches. This can be attributed to the fact that midpoint optimization stops the search right in the middle of the process and allows for the reconstruction of the remaining path using the path that has already been generated. It is also worth mentioning that the solution length for this approach was 79, whereas the solution length for the first two approaches was only 40. Thus, considering the exponential growth in the number of generated nodes, it is expected to see this huge gap between the last approach and the others.

The number of generated nodes and the execution time also followed the same trend. In both of these measurements, there is an observable gap between each of those four methods of solving the problem.

Approach	Number of Expanded Nodes
Optimal midpoint with low g-cost	2,219,532.67
Optimal midpoint with high g-cost	4,616,802.0
Sub-optimal midpoint	55,333,048.0
Optimal without midpoint	431,808,097.67

Table 2: The number of expanded nodes for each approach

Approach	Number of Generated Nodes
Optimal midpoint with low g-cost	5,486,544.33
Optimal midpoint with high g-cost	10,109,170.67
Sub-optimal midpoint	95,904,999.33
Optimal without midpoint	698,254,655.67

Table 3: The number of generated nodes for each approach

Approach	Execution Time (s)
Optimal midpoint with low g-cost	52.82
Optimal midpoint with high g-cost	170.09
Sub-optimal midpoint	1219.05
Optimal without midpoint	6772.44

Table 4: The execution time for each approach

In addition to these values, another important factor is PDB creation and the time it takes to generate proper PDBs. For this implementation we have developed two PDB creation approach, the first approach considers the goal as when all the disks are in the goal peg, and the second one is the case where all the disks except the last one are in the middle pegs. According to the methodology section, fewer states should be generated for the first approach, so the execution time should also be lower. The average execution time for the first approach is 37.939s while the average for the second one is 153.689s. It is expected since the second approach is generating almost four times more states than the first one.

In conclusion, the experiments conducted on the implementation have demonstrated the effectiveness of using heuristics in reducing the search space and finding more efficient solutions. The results have also highlighted the impact of tie-breaking and the use of midpoint optimization on the number of generated nodes and solution length. Moreover, the comparison of the two PDB creation approaches has shown that the goal state where all the disks are in the goal peg results in a lower execution time due to the generation of fewer states. Overall, these findings provide valuable insights for the development of more efficient and effective algorithms for solving the Tower of Hanoi problem.

References

Korf, R., and Felner, A. 2007. Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. 2324–2329.