Exploring solutions for the 4-pegged Tower of Hanoi problem

Dylan Farrer

December 2021

1 Introduction

The towers of hanoi problem describes a game that is traditionally set up with three wooden pegs. The first peg has n number of disks stacked on it, the widest disk at the base and the thinnest at the top. The other pegs are empty, and the goal is to move the disks so they replicate the starting tower configuration on the far peg. Only the top disk of any tower can be removed and placed onto another peg in a move. If the target peg is not empty, the top disk on that peg's tower must be larger than the disk that is being moved.

The 3 peg version of the game has an optimal solution of $2^n - 1$, but the optimal solution for the 4 peg variant is an 'open question' [1].

In this project, I implement two search algorithms to use on the 4 peg variant of the towers of hanoi problem, breadth-first search and A* search. I propose three heuristics for the A* search and compare these search methods against each other for increasing disc count n. I also implement a pattern database heuristic, and discuss it's applicability to the problem.

2 Implementation Discussion

2.1 Problem Representation

To decide upon the problem representation I would use, I considered the interactions and visibility needed in the system. The disks could only moved from the top of any peg to the top of any other peg, which made me consider using stacks. I then realised that to successfully judge a state, I would need to easily view all disks - stacks do not support structure iteration without popping off items. I decided to use a list for each peg instead. These lists are then stored in another list, which would represent a state.

The only information needed to represent each disk is it's width, so these were stored as integers.

To compare two states, each list is compared against each other, if each list in state A and state B have equal contents, state A and state B are the same.

A move is the operation of moving the last integer in any list and putting it on a different list. If the target list is not empty, the last integer must be greater than the integer that is being moved.

2.2 Chosen search algorithms

I have chosen A* search and breadth-first search as the algorithms to calculate solutions for the four-peg towers of hanoi problem.

Breadth first search is a classical search algorithm that will find the shallowest goal in a problem. It utilises a First-In-First-Out queue as it's frontier. If the possible next states are modelled as a tree, breadth-first search will visit every state at a given depth before traversing any further downward. The four-peg towers of hanoi problem has a finite branching factor and all moves are unweighted, which makes the breadth-first search algorithm complete and optimal in this case.

A* search is an informed search algorithm. It avoids traversing down potential routes that are deemed expensive. States are evaluated by adding two values together - The cost to reach the current state and the estimated cost to the goal state from the current state. The latter is evaluated with a heuristic. The sum of these two costs make up the estimated total cost of the path to the goal state via n, the current state. Like breadth-first search, A* search also has a frontier, but it is ordered by ascending total cost of the state. States are pulled from the front of this queue - the cheapest calculated state. A* is both optimal and complete.

2.3 A* heuristics

Heuristics are used in A* search to estimate the path length between a given state and the goal state. The given state is reasoned about in context to the goal state, to give it a 'score'. This score represents how favourable the state is as a potential next state to traverse. The Manhattan distance and Euclidean distance are two common ways to give a state a score; If the Euclidean distance between a current state and the goal state is equal to 5, then the cost of the traversal to the goal state must be at least 5, as the shortest possible route between two points is a straight line.

As the state space representation I had chosen was a matrix, my initial thoughts were to use Manhattan distance as a heuristic. I realised the heuristic would not fit directly onto my representation, as the possible moves after any given state were not multi-directional in a matrix. I decided to use Manhattan distance as an influence, and implement my own heuristics.

The heuristics I used:

- The number of misplaced disks
- The largest tower of disks (doubled, then minus one) (ignoring partial or complete goal state towers)

- The misplaced disk with the largest sum of (width + size of tower on top of it)

Using the number of misplaced disks as an A* heuristic seemed simple. If there are 4 disks that are not in their correct location in the goal state, at least 4 moves need to be made to get to the goal state. Because of this, it is clear that this heuristic is admissible. This heuristic is also consistent; If the calculated cost of a state is 4, the largest misplaced disk has a width of 4, if that disk is moved to it's goal position in the next move, the second largest disk - 3 - must be misplaced, so the score of the next state will be 3. The only other option is that the largest disk is not moved to the goal state, so the score stays the same.

The largest tower of disks is an extension to the first heuristic. If the largest tower that is 'misplaced' (a stack of the three largest disks at the goal peg would not count, for example) is 5 disks high, at least 5 moves need to be made to reach the goal state. I further reasoned that for any misplaced stack of height n, $n-1$ disks need to be moved off of it, then at least n moves need to follow that operation to reach the goal state. A simple example can help here:

Consider the case where the state is the initial state, the tower height is 3 and the number of available pegs is 4. The optimal set of moves to reach the goal state - the same tower on the far peg - is to first move the smallest disc onto either peg 2 or 3 (assuming the pegs are ordered $1, 2, 3, 4$, ranged from initial to goal peg), then move the second smallest disc to the other peg (the one that the smallest disc is not on), move the largest disc to the goal state, then the second smallest, then the smallest. The second heuristic is designed with this optimal solution in mind.

There is no quicker way to reach the goal state than the method described in the above example, even if we increase the peg number. As the second heuristic is modelled off of this optimal case, it is admissible.

This heuristic is not consistent. Consider the case where the state has four pegs, the second peg has the tower 1 and the third peg has the tower $4, 3, 2$. The first and fourth peg are the initial and goal pegs respectively. The estimated score for this state is height of the largest tower - 3 - doubled and then decremented by one - $(3 * 2) - 1 = 5$. Any next action that reduces the size of this tower will decrease the estimated score by 2, but any state that increases the tower size will increase the score by 2. This violates consistency; The estimated score of any state must decrease or equal the estimated score of the previous state plus the transition cost.

Rather than building upon the first two heuristics, I decided to consider how else a state could be analysed. I considered using the width of the largest misplaced disk as a heuristic, although after some though I realised this heuristic is equal to the first one; For any state, the largest misplaced disk width is always equal to the number of misplaced disks.

For the third and final heuristic, I combined the reasoning behind my second heuristic and the thought experiment on using max width. This heuristic scores a state by finding the largest misplaced disk, and adding the width of that disk to the size of the tower on top of that disk. This can be viewed as a variant to the second heuristic; In some cases, they will produce the same score - if the largest misplaced disk is at the base of the largest misplaced tower. Another scenario will be useful to describe the benefit of this heuristic.

Consider the state that has 4 pegs, the second peg has the tower $5, 4, 3, 2$ on it, the third peg has the tower $7, 6, 1$. The first and fourth peg are the initial state and goal state respectively. The second heuristic will notice the largest misplaced tower and score the state 7, the third heuristic will see the largest disc and add that (7) to the height of the tower on top of the largest disc (2), scoring 9. The actual cost is greater than both estimates, but the third heuristic is closer.

For the largest misplaced disc in a state and the size of the tower on top of that disc, the minimum set of moves to reach the goal state would be to remove the tower ($n - 1$ moves, if the total tower size is n including the bottom disc), and then move the largest disc to the base of the goal state, then build up the remaining discs. The third heuristic is based upon this process. Because there is no quicker path than the one just mentioned, this heuristic is also admissible.

This heuristic not consistent. Consider the state that has four pegs, the first and fourth are the goal states respectively, the second peg has the tower $4, 3, 2, 1$ and the third peg has the tower 5. The estimated score for this state would be the width of the largest disk - 5 - plus the size of the tower on top of it - 0 - giving 5. A possible - and recommended - action is to move the largest disk to the goal state. To score this new state, the heuristic will now see 4 as the largest disk, it will then add the height of the tower - 3 - on top of this disk to output a score of 7. This score difference violates consistency.

Heuristic	Admissible	Consistent
Misplaced disk no.	Yes	Yes
(Largest disk tower * 2) - 1	Yes	No
Largest misplaced disk + tower height on top	Yes	No

Table 1: Heuristic Properties

3 Experimental findings

3.1 Search algorithm and heuristic comparison

Table 2 shows the compute time of the different search methods against an increasing tower size n. Notice that the heuristics only make relatively significant improvements on compute time from $n = 3$. The heuristics 'Largest misplaced disk + tower height on top' and 'Misplaced disk no.' provide the lowest compute times and provide very similar time-to-compute values, but 'Misplaced disk no.' proves to be the most efficient heuristic.

Disk no.	Breadth-First-Search	Misplaced disk no.	(Largest disk tower * 2) - 1	Largest misplaced disk + tower height on top
1	0.00010	0.00015	0.00008	0.00007
2	0.00041	0.00027	0.00034	0.00031
3	0.00430	0.00091	0.00394	0.00090
4	0.04439	0.00652	0.01726	0.00602
5	0.48622	0.12336	0.31319	0.15314
6	7.87020	0.66759	2.60725	0.95565
7	122.04973	32.83669	70.69461	36.57899

Table 2: Comparison of Heuristics: Time taken (seconds) to find solution against increasing disk count

3.2 Heuristic capability and optimality

Using these heuristics, the largest tower size n of an initial state that a solution can be computed for in under 10 minutes is 7. As the A* search and breadth-first search algorithms are both optimal, all values that are computed are the optimal solution. Within the 10 minute time limit, the system I have implemented can compute the optimal solution for disc sizes $[1 - 7]$.

4 Pattern Database implementation

4.1 Introduction and rationale

To extend this project further, I implemented a pattern database heuristic that I plugged into my A* algorithm. A pattern database heuristic pre-computes the 'true' score of subproblems in a problem space into a database. States that are encountered in A* can then be given the score of the subproblem that fits into that state.

The pattern database heuristic works as follows (in the context of the towers of hanoi problem). Given an initial state and tower size, a pattern database is calculated. This database includes every state possible from the initial state provided, with the true distance that state is from the goal state. This database is calculated by performing a reverse breadth-first search from the goal state to the initial state, noting the path length at each state. The database will match the number of discs in the initial state up to a maximum (7 in the case of my system, as it is the highest disc count where constructing the PDB takes under 5 minutes), if the problem tower size is larger, the database is created at the maximum size (7) and no larger. When the A* search algorithm needs to calculate the estimated path to the goal state for a state in the system, it is matched to an equivalent state in the database and given the associated score.

For a problem size within the maximum tower size of the pattern database heuristic, the returned estimated cost to the goal state value is the actual cost to the goal state, resulting in a very fast search (once the database has been created). However, as the database must be created before the traversal begins, the heuristic will take at least the same amount of time to complete as a breadth-first-search with a tower size of the problem specification or a tower size of 7, whichever is smaller.

4.2 Discussion

Table 3 shows the time taken to compute a solution for a given towers of hanoi problem using a pattern database heuristic that produces a table with a maximum initial tower size of 7. As can be seen from this table, the PDB heuristic does not provide better performance than a breadth-first search unless the problem complexity (tower size) is larger than the PDB tower size. For a disk number of 8, the total time to compute a solution using the PDB heuristic is 201.61375 seconds (build time + A* search time). A* search using the next best heuristic in my system 'Misplaced disk no.' takes 682.62262 seconds. The PDB heuristic improves the largest initial tower size I can handle within 10 minutes to 8 discs.

The A* run time using PDB dramatically increases from initial tower size 8 to 9. One reason for this is how the heuristic scores states that have a higher disc number then the database stores. It will remove the larger disks from the state until it matches a state stored in the database, and then output the score associated with that found state. For every state in a 7 disc problem set, there are 4 states in an 8 disc problem set that are 'equivalent' if the largest disk is removed. All of these states will be given the same score, irrespective of their differences. For each additional starting disc, the number of states that are treated as the same state will increase by powers of 4.

Disk no.	PDB build time (seconds)	Time taken to reach solution (seconds)
1	0.00015	0.00006
2	0.00097	0.00039
3	0.00771	0.00098
4	0.05135	0.00106
5	0.54142	0.00447
6	8.13493	0.00468
7	124.40000	0.03019
8	127.74754	73.86621
9	125.27344	1449.73630

Table 3: Compute time (seconds) of (max) 7-disc pattern database heuristic against increasing disk size

Table 4 shows the increased time to compute a solution for increasing initial tower sizes, given a pattern database that has a disc number of 3. For every initial tower size, the pattern database heuristic performs worse than the other

Disk no.	PDB build time (seconds)	Time taken to reach solution (seconds)
3	0.00852	0.00109
4	0.00684	0.02529
5	0.00810	0.48493
6	0.00509	6.74916
7	0.00607	140.47783

Table 4: Compute time (seconds) of 3-disc pattern database heuristic against increasing disk size

heuristics in my system. From Table 4 and 3, it is clear that to reap the benefits of the pattern database heuristic, the size of the database should be configured against the problem size; At the edge of my system's computational power (at the 10 minute mark) PDB performs the best. At every other point, it does not perform better than the other heuristics in the system.

5 Conclusion

In this project, I have explored various methods to compute a solution to the four-peg towers of hanoi problem. I have created a list-based representation of the problem so that I can implement a system that can reason about and compare different states. I created three heuristics to use in the A* search algorithm and discussed their admissibility and consistency. These heuristics were compared against each other and against a breadth-first search algorithm for time-to-compute for increasing initial tower size. I determined that the best heuristic in my system was 'Misplaced disc no.', which is both admissible and consistent.

In addition, I implemented a pattern database heuristic for the A* search algorithm and compared it's time-to-compute against other search methods in my system. I discovered that a pattern database heuristic can provide efficient time-to-compute, but does not always perform better than other search methods and is dependent on the relationship between the pre-computed database size and problem size.

My system can efficiently compute the optimal solution for a four-peg tower of hanoi problem up to an initial tower size of 8 (7 if only considering the non-pdb heuristics).

References

[1] Chu, I.-P., and Johnsonbaugh, R. The four-peg tower of hanoi puzzle. *ACM SIGCSE Bulletin 23*, 3 (1991), 2–4.