CS5910 Final Project Report

Tower of Hanoi with 4 pegs

Chenyan Li
Matthew Lo

February 12, 2020

1 Basic Project

1.1 Rationale behind the problem representation chosen

The representation for a state is a n - sized tuple $(x_1, x_2, ...x_n)$, where $x_i \in \{0, 1, 2, 3\}, i \in \{1, ..., n\}$. The value of n indicates the number of discs used in the context, where $n \in \mathbb{N}$, the value of i represents the disc with size i, and the value of x_i represents the $Peg = \{0, 1, 2, 3\}$ it is on. In this context, we are given 4 pegs. Peg 0 indicates the source peg, peg 3 indicates the destination peg, and the rest are the auxiliary pegs. For example, the tuple $\big(0, \quad 1, \quad 1, \quad 0\big)$ represents the state that: $n = 4$, disc 1 and 4 are on peg 0, while disc 2 and 3 are on peg 1, and the rest pegs are empty. Although the ordering of discs on each peg is not shown on the state tuple, assume the discs are in ascending order from the top, and the actions will be making sure that no larger disc is on top of a smaller disc. Therefore, the initial state is represented by the tuple $\big(0, \quad 0, \quad 0, \quad 0\big)$, where all discs are on the source peg (rightmost), while the goal state is represented by the tuple $\big(3, \quad 3, \quad 3, \quad 3\big)$, where all discs are on the destination peg (leftmost).

An action is defined by the tuple $\big(i, \quad j\big)$, moving the toppest discs on peg i to peg j. The set of actions available initially is defined by:

$$Actions = \{0, 1, 2, 3\}^2 = \{\big(i, \quad j\big) \mid i, j \in \{0, 1, 2, 3\}\}$$

However, a disc on a peg cannot be moved to the same peg,

$$Actions = \{\big(i, \quad j\big) \mid i, j \in \{0, 1, 2, 3\}, i \neq j\}$$

In order the filter out actions which are unavailable to perform in a particular state, 2 rules are implemented. Consider an action $\big(i, \quad j\big)$:

1. If both peg i and peg j are not empty, and if the toppest disc on the peg i is larger than the one on peg j, then such action cannot be performed.
2. If peg i is empty, then any actions from peg i cannot be performed.

The impossible moves will be removed, and the remaining are the set of actions that can be performed in that particular state. For example, the possible actions set for the initial state is:

$$Actions = \{\big(0, \quad 1\big) \ \big(0, \quad 2\big) \ \big(0, \quad 3\big)\}$$

1.2 Search algorithms

2 search algorithm is chosen, A* graph search and the depth-first graph search.

A* graph search, as an informed search algorithm is an example of the best-first graph search, starting from the initial node, expanding nodes by the heuristic

value and the cost of reaching the corresponding node from the initial node. At each step, store the nodes in a *PriorityQueue* as the frontier and sorted in decreasing order of disability f (n) = g (n) + h (n). For each node, if the node is the goal node, then stop the iteration and return the solution. Otherwise, add the node into the explored node set, remove it from the frontier and add its child nodes to the frontier. The steps continue until the goal node is reached.

Depth–first graph search tends to explore same nodes as far as possible before proceed to the next node.

1.3 Heuristic functions

Three heuristic functions are created.

In general, a heuristic function is used to inform the search about the direction to the goal from the node it is on, by providing the estimated cost of path though node N to the goal. The first heuristic function is defined as the number of disc which are not on the destination peg. This most fundamental heuristic function provided the minimum number of actions to be performed to move all remaining disc from the initial peg and auxiliary pegs to the destination peg on the corresponding state that the node N represent, and will not exceed true number cost from node N. At goal state, the number of moves to be taken is zero so the heuristic value at the goal node is zero. Also, the heuristic value at every node is larger or equal to zero. Hence this heuristic function is admissible and consistent and giving out optimistic solution.

The next heuristic function is partially based on the first function, while we are trying to penalize smaller discs on the destination peg while the largest peg isn't on it. That means extra actions have to be taken to move the smaller discs away from the destination peg in order to spare space for the largest disc to move in. Therefore the heuristic function is defined as the number of discs which are not on the destination pegs + λ × the number discs on the destination peg while the largest disc isn't on it, where λ is the penalty parameter. In order to decide the value of λ, consider the following case: If the largest disc isn't on the destination peg (node N), then all smaller discs have move to any other pegs to give way (move to node P), then return to the destination peg to reach the goal state (node G), so at least 2 actions has to be performed for each smaller plate, hence the minimum value of λ here is 2. This minimum value also ensured that the heuristic function is admissible and consistent, also more efficient then the previous one as more condition is being considered.

For an extreme case, the third heuristic function to implement is for $\lambda = 1000$. It is clearly over-penalizing and overestimating the path cost hence this is not an admissible heuristic function.

Consider the following case: Let the node A be the state that $\big(0, \quad 3, \quad 3, \quad 0\big)$, and node B be $\big(0, \quad 1, \quad 3, \quad 0\big)$, path cost from node A to node B is $c(A, B) = 1$ because it takes one action only (moving disc 2 to peg 2). The heuristic value at node A is $h(A) = 2 + 2 \times 1000 = 2004$ and $h(B) = 3 + 1 \times 1000 = 1003$. Clearly, $h(A) = 2004 > h(B) + c(A, B) = 1003 + 1 = 1004$. This heuristic is not consistent. Hence some of the solutions produced by this heuristic function may be suboptimal.

1.4 Running results

[Table 1.1 the runtime (sec) of each algorithm vs. number of discs n (adjusted significant figure)]

n	A* Search H1	A* Search H2	A* Search H3	DFS
1	0.00011	8e-05	0.030977	0.0
2	0.00044	0.00026	0.00010	0.0
3	0.00200	0.00010	0.00301	0.00010
4	0.02967	0.01751	0.02596	0.01097
5	0.29117	0.17262	0.24834	0.20046
6	2.98054	1.80109	2.24902	2.46338
7	58.8580	44.7259	24.4293	21.9794
8	917.967 >600	776.758 >600	306.390 <600	814.867 >600

From table 1.1, we can see that the running time for 8 discs for all algorithms are more than 10 mins except for the A* search with heuristic function 3. However heuristic function 3 is inconsistent and the results are suboptimal (see Table 1.2). Therefore, the maximum number of discs n that can handle the problem optimally is 7 and the maximum number of discs n that can handle the problem suboptimally is 8, with a runtime limit of 10 minutes.

[Table 1.2 the solution length of each algorithm vs. number of discs n]

n	A* Search H1	A* Search H2	A* Search H3	DFS
1	1	1	1	1
2	3	3	3	5
3	5	5	5	26
4	9	9	9	86
5	13	13	15	223
6	17	17	27	1190
7	25	25	47	3846
8	33	33	83	17554

As above, the maximum number of discs that can handle the problem optimally is $k = 7$. At $n = 7$, the solution length for the optimal is 25, while the one for the suboptimal is 47.

2 Extension

2.1 Rationale behind the PDB heuristic

Let the m be the number of discs to construct the Pattern Database. The complete set of possible states will then be 4^m. For each possible state, perform the breadth - first graph search and compute the corresponding solution length. Convert the state tuples to indices for the database, otherwise it takes much more time to perform brutal search on finding the corresponding solution length. Combine the indices and the solution lengths into as data frame PDB. Then we will treat those solution lengths as the heuristic values. Here we consider they are the m largest discs in all cases, and we match last m entries of each node state tuple with length n, assign the corresponding heuristic value to it, then run the A* search on them.

2.2 Running results

The biggest m such that constructing the PDB takes less than 5 minutes is 5, it takes 51.0053 seconds.

[Table 2.1 the runtime (sec) of A* Search with PDB vs. number of discs n]

n	A* Search PDB
6	0.37450289726257324
7	24.09334683418274
8	592.1565272808075

As $m = 5$, we then start from $n = 6$, and this time the running time is slightly improved. Now we can handle 8 discs within 10 minutes. The reason for the slight improvement may due to the PDB heuristic values are providing us a more accurate information about the path cost. However the running time is still highly depends on the device and the search algorithm. For example, using *PriorityQueue* on A* search is much more slower than using the *Queue* on depth-first search.