COMP30024 Artificial Intelligence

Project 1 Report

1. Search Problem Formulation

State: player and obstacle pieces' location on board

Action: a player can move, jump or exit one player piece per turn defined in the specification

Goal Test: no player's piece on board

Path Cost: 1 cost per action

2. Search Algorithms

Terminology:

b: branching factor for the search tree

d: length for the solution path in the search tree

δ: relative error in heuristic = |h*(s) - h(s)|

h(): heuristic function

n: number of hexagon on board not occupied by block

A* search

This is the search algorithm used in our program. It's a simple but efficient search algorithm. Not only it is complete and optimal, but also optimally efficient, meaning among similar algorithms(ones that expand paths using heuristic as a guide), which use the same heuristic, A* expands the least (or as least as others) number of nodes (states in this project).

Time Complexity:

best case ∈ O(d) if we disregard the complexity of the heuristic calculation

average case∈ $O(b^{\delta d})$ (from lecture)

worst case ∈ $O(b^{\delta d})$ (because it is uniform cost search now)

Space Complexity∈ $O(b^{\delta d})$ (because "keep all nodes in memory")

Completeness:

A* search is guaranteed to find a solution if one exists. As it is guaranteed in the specification thatthere is at least one solution, A* search is complete in the project.

Optimality:

Yes, as long as $h(s) \leq h^*(s)$ $\forall s \in$ *state space*

(A search is optimal if the heuristic is admissible(required in tree search) and consistent(required in graph search))

3. Heuristic Function

$$k * \sum_{piece_i \in player} min\_cost[piece_i],$$

$$where \begin{cases} k = 0, when\ state\ has\ an\ exit\ action \\ k = 1, otherwise \end{cases}$$

Our h() is processed before the A*.We use all obstacles location and “unoccupied piece” means a block is not at that piece’s hexagon. Firstly, initialize player’s unoccupied goal pieces with 1 cost. Then put all these pieces into priority queue and perform Dijkstra algorithm with a loop popping a piece until the priority queue is empty and update in each loop with rule:*min(popped piece’scost so far + 1, neighbor piece’s current cost to exit) or neighbor piece is unvisited and directly assignpiece’s current cost to exit+1* ∀neighbor pieces. If the neighbor piece’s value has updated, put this piece to the priority queue. As a result, we have a dictionary {unoccupied piece: cost to exit}named min_cost. Then, we use this preprocessed dictionary in the heuristic function.

We have k because we want our a* to process state with exit action firstly. Because an exit action means there is no better action for this piece to take. And a value equals 0 is always less than a state can perform an exit action with minimum possible cost equals to 1.

Time Complexity:∈O(n $\log_2$n) compared to a*’s average time complexity, we can discard it. And it requies a small amount of time (less than 100ms) in practical.

Admissibility:

As discussed above, when k = 0, it is admissible. As in our Dijkstra algorithm, we classify a piece can move with 1 cost which equals to real cost and piece can jump with a relaxed rule: can jump without any piece. Thus,from the current state to result state which requires two moves is classified with one jump which means less than true cost and requires one jump case still equals one cost.Above all,our heuristic is admissible.

4. Run time and Space Impact

A sample from the population is drawn to plot the graphs. For each combination of player number and block number, a sample of size 400 is drawn in random. Runtime is logged before plotting to mitigate the effect of large scaling.

branching factor

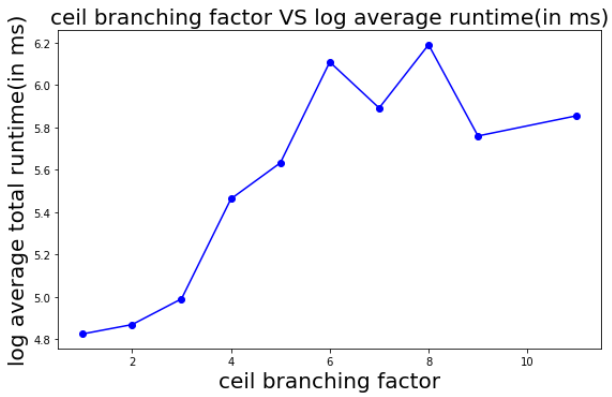

Figure 1

A bigger branching factor means more successors to examine when choosing the next move, which, by intuition, requires more time. Also, from the formula for time complexity above, a bigger b gives a greater time complexity.

depth of search tree

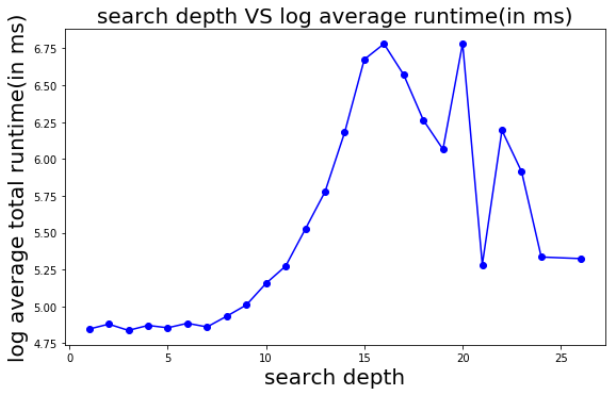

Figure 2

In cases with a large depth of search tree, the player takes a long way to finish the game, thus using more time. Although it is unstable when depth is big, compared to small depth, there is a large gap in time used

number of bocks

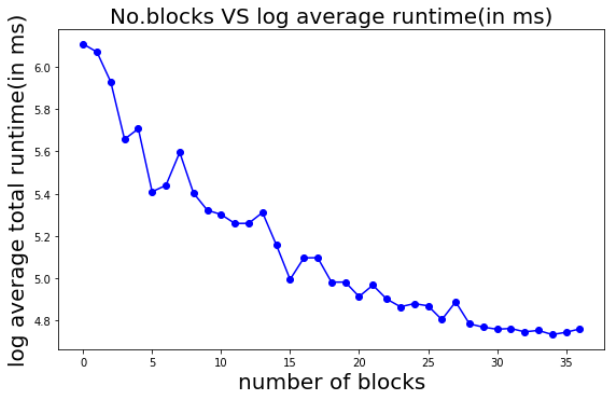

Figure 3

As the number of blocks increases, the size of search space decreases, resulting in faster execution time.

number of player pieces

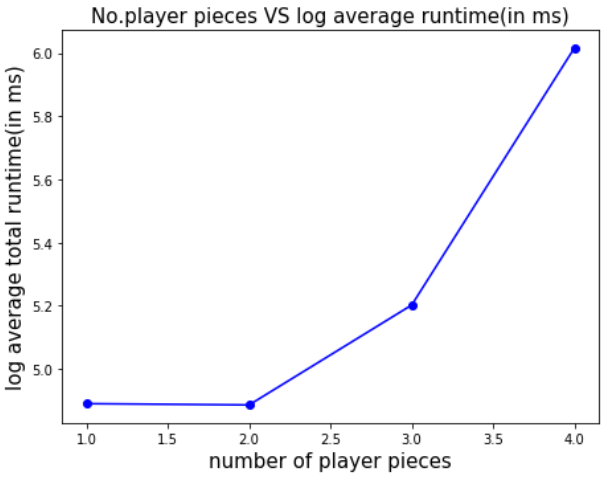

Figure 4

The size of the search space increases as the number of player pieces increases. Hence, it takes a longer time for the program to find the solution.

relative error

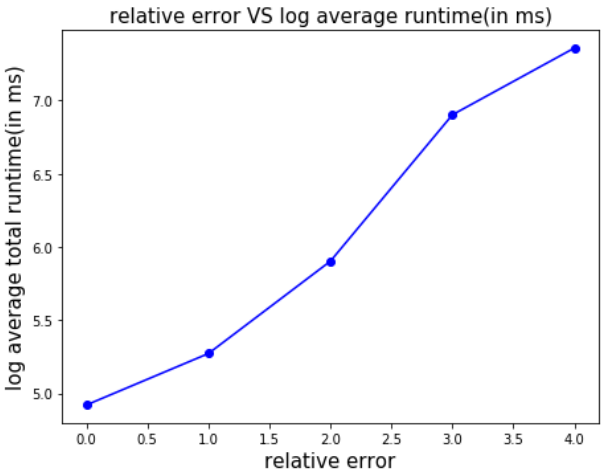

Figure 5

A larger relative error means greater deviation from true cost when using the heuristic for estimation. Therefore, the program needs more time to find the path.

5. Conclusion:

In conclusion, branching factor, search depth, number of pieces and relative error have a positive relation with runtime, whereas number of blocks has a negative relation with runtime.