CSC384

Term Project: Search and Path Finding

Brandon Oliver g4oliver

Jack Irish j5irish

Kevan Hollbach g4hollba

Team Member Roles

Brandon Oliver: Research, Code base manager, Implementation of Search optimization.

Jack Irish: Experimental Assessment, Debugging.

Kevan Hollbach: Optimality adjustments and analysis, README author.

Project Type

Search with Heuristic Design and Pruning. A* search and additional research and methods to optimize.

Problem

The problem we are addressing is path finding. Within path finding there are many opportunities to challenge and optimize search algorithms. In this project we explored A* Search and A* Search with pruning (based on Jump Point Search). We have gathered a range of map data to test challenge each algorithms benefits and flaws and analyzed them. The goal of this project was to reduce our number of expanded nodes while reducing the total runtime of the search.

Methods

We first started by creating our A* search algorithm and setting up our State object. Then we simply used f = g to produce our uniformed cost search algorithm. Finally we used a technique called “Jump Point Search”. Below is an outline of the type of structures and functions we used:

Map Files: We used n x m .map files which we would read as input and use to precompute a 2-d array of characters called “char_map” and a 2-d array of states called “state_map”. Within each map is a char “s” and “d” for start and destination, which is used to predefine our initial state and goal state.

States: For our states we used many variables to help create our results and some were required by our implementation of JPS (Jump Point Search). Some are more specific to our needs like:

map_rep: character representaion on the map file

time_expanded: When the state was expanded so that we can update our path.

loc: location on the map as a tuple: (x, y)

accessed_diagonally: Used for JPS. Represents whether or not this node was a successor of a state that was diagonally adjacent.

JPS_pruned: Used for JPS. True if this state has been pruned, false otherwise. During JPS algorithm, any state that is pruned will be skipped.

Successors:

For our successors we simply checked expanded to all adjacent states, given that they were in range of the 2-d array (ex: state_map[-1][0] is out of range). The successors for JPS were handled differently and will be explained in the JPS section. Expansion was uniformed cost of 1 or root(2) if diagonal movements.

Cycle Checking:

We simply used a dictionary for our Closed list where the key was our state's hash id and its value was the cost of the path to get to that state. We used this value so that we could reevaluate a state if a cheaper path was found.

Heuristics Used: We used Manhattan distance and Euclidean distance as our heuristics for this project. These two were common amongst most path finding algorithms and therefore seemed and have been the most effective. They are also already admissible and monotonic heuristics allowing for completion and optimality.

JPS:

Description: We used a pruning technique known as Jump Point Search, written by Daniel Harabor and Alban Grastien, which works by "jumping" to states that would be optimally reachable. The idea of "jumping" is similar to how we usually expand our nodes in pathfinding but instead of stopping at the state directly adjacent to our current state, we continue down that adjacent path under certain conditions. These pruning conditions are:

p = curr state

y = alternative state

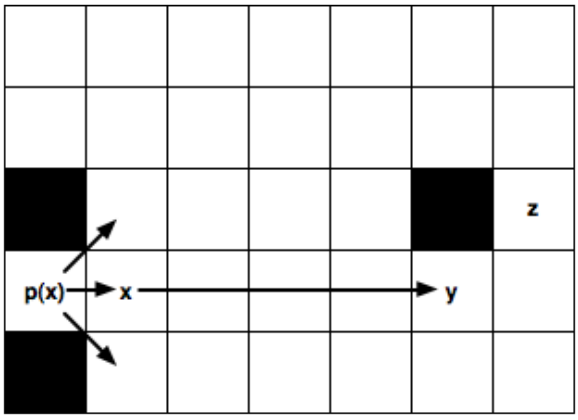
x = successor of p

n = some destination state

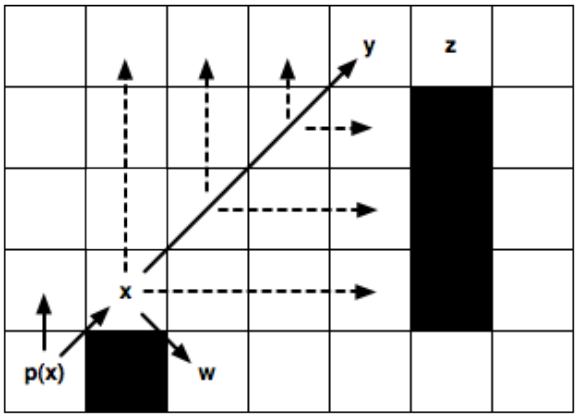
1. there exists a path π’ = <p, y, n> or simply π’ = <p, n> that is strictly shorter than the path π = <p, x, n>;

2. there exists a path π’ = <p, y, n> with the same length as π = <p, x, n> but π’ has a diagonal move earlier than π.

This essentially means that as long as we can reach all the successors of x optimally through x or p, then we keep going down this path. However, if there is a wall/blocking successor state, then we have to consider whether a successor is no longer reachable. Therefore we stop, and return the farthest most state as the successor of p. This is illustrated in Figure 1, where the dotted lines are checks and solid lines indicate a successful jump, where y is the new successor of p. The time taken to add every state and extract it later is saved since we iterate through the states beforehand, eliminating the insertion and extraction cost from out Open/Frontier list.



a) Horizontal and vertical case.



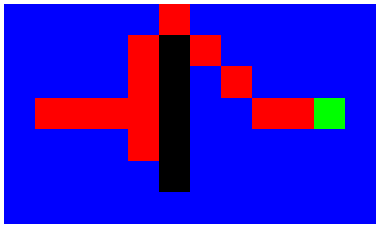
b) Diagonal Case.

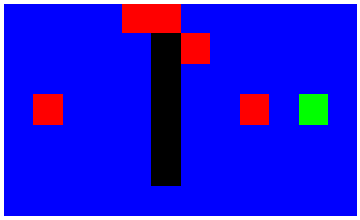
Figure 1

Evaluation and Results

Each test was run 10 times using python.time() and an average was taken for the runtime. Below we have split our results based on the different map files we tested with. We have two map images. On the left is A* and on the right is A* with JPS. Both are using euclidean distance as their heuristic.

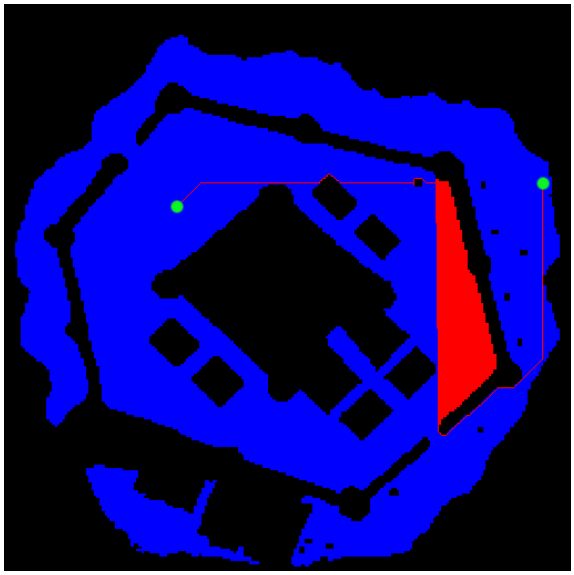
Red: nodes expanded, Blue: is open space, Black is a wall, Green: goal and start

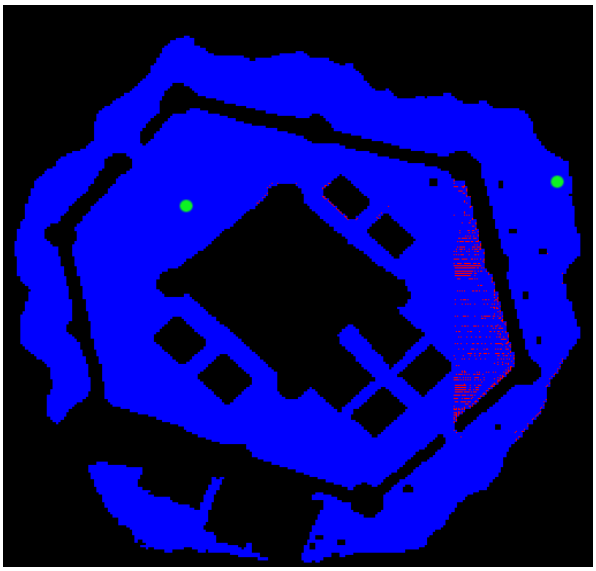




Map 1 (map_3.map):

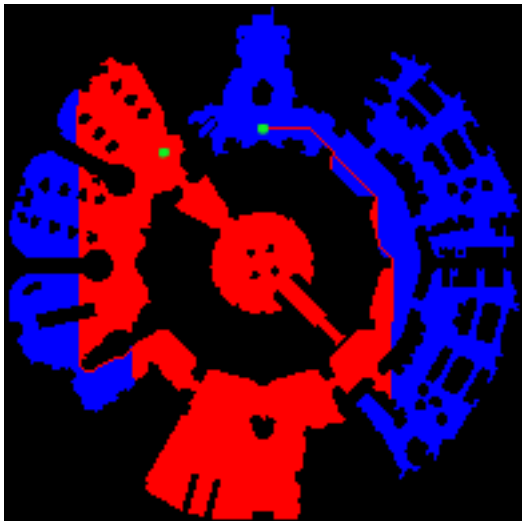
Algorithm w heuristic	Runtime(seconds)	Nodes expanded	Nodes in path
A* w manhattan	0.0	28	14
A* w euclidean	0.0	12	12
A* + JPS w manhattan	0.0	9	5
A* + JPS w euclidean	0.0	5	6

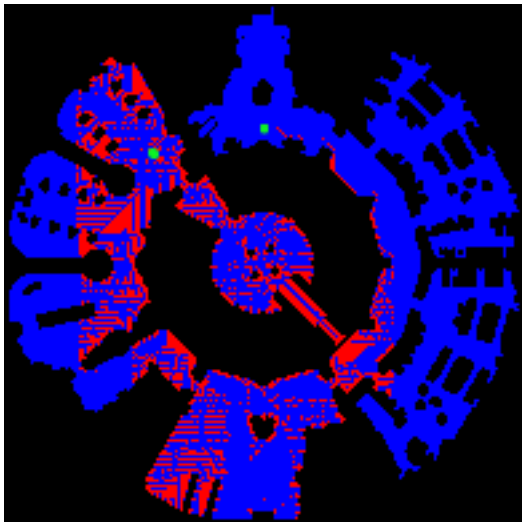




Map 2 (map_large_medium.map): 512 x 512

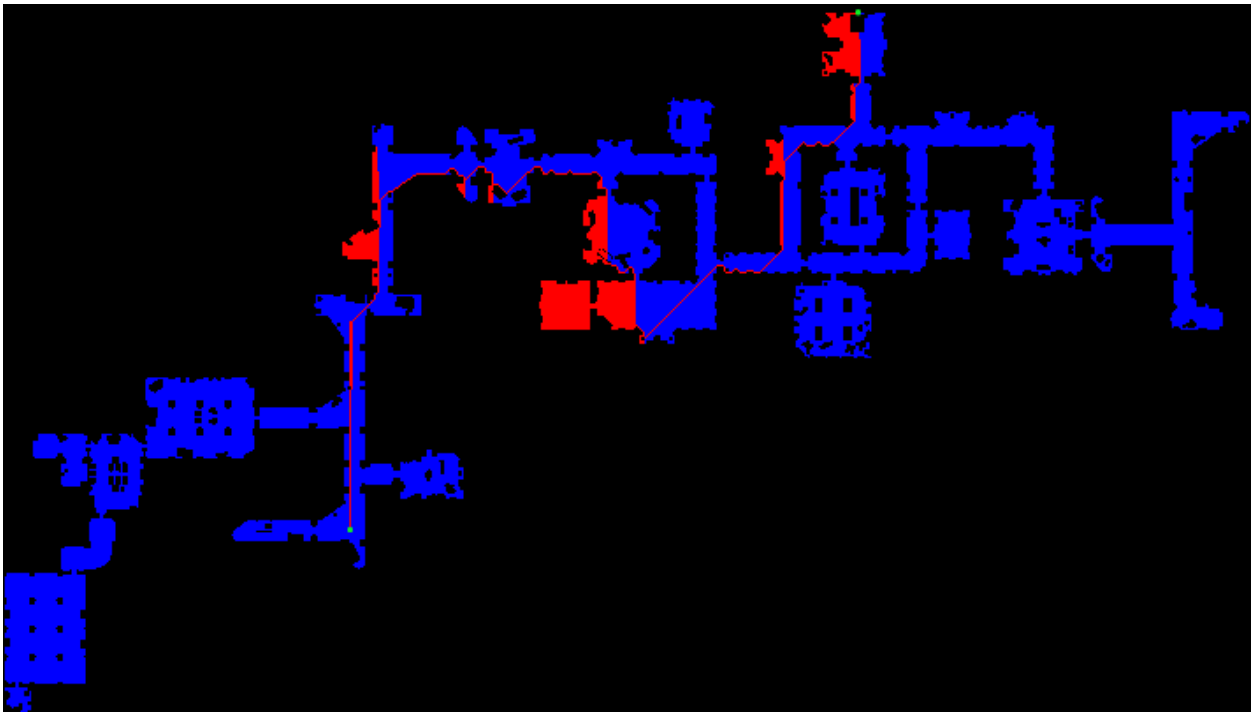
Algorithm w heuristic	Runtime(seconds)	Nodes expanded	Nodes in path
A* w manhattan	12.03	40166	677
A* w euclidean	2.46	7504	748
A* + JPS w manhattan	6.7	1598	75
A* + JPS w euclidean	3.21	895	98





Map 3 (map_large_hard.map): 194 x 194

Algorithm w heuristic	Runtime(seconds)	Nodes expanded	Nodes in path
A* w manhattan	2.63	8961	542
A* w euclidean	2.07	7496	472
A* + JPS w manhattan	1.08	2975	72
A* + JPS w euclidean	1.06	2667	144



Map 4 (map_large_hard_3.map): 402 x 711

Algorithm w heuristic	Runtime(seconds)	Nodes expanded	Nodes in path
A* w manhattan	6.13	147078	793
A* w euclidean	1.2	3588	891
A* + JPS w manhattan	1.9	3530	109
A* + JPS w euclidean	0.76	813	170

Based on our results, we see that with JPS pruning vs standard A* we cut our runtime by almost half. Another major improvement was using Euclidean over Manhattan. Although Euclidean takes slightly longer that Manhattan to compute, the runtime and node expansion benefits are worth it, with as high as an 80.5% runtime reduction on map_large_hard_3.map and an 82% reduction on expansion on map_large_medium.map.

Therefore we can conclude from all 4 cases, A* + JPS with Euclidean distance heuristic is the most optimal in terms of runtime and node expansion.

With JPS we still would have the same branching factor as our standard A* algorithm but what we are able to reduce is the depth of our search, therefore if d' is our depth expanded through JPS we can conclude through our research and data that:

$(b^{d'}) <= (b^d)$

Therefore we use this to conclude that our JPS is optimal.

Limitations/Obstacles

We had implemented an AVL tree for our Open/Frontier to reduce our O(n) extraction to a O(logn) extraction, but we had some implementation issues that caused our search to fail. We also wanted to tackle the addition of other agents into our map with their own goals and start states using a Windowed Heuristic Cooperative A* search, but we did not have enough time to get it to work correctly. We were only able to create a basic implementation of Cooperative A* search with some minor issues. This attempt can be found in coop_search.py.

Some of the obstacles we faced were, correctly understanding the JPS algorithm. Since it is based on a paper which offers no pseudocode, we had test many small map cases to make sure that our algorithm was pruning correctly. At one point we had reduced the nodes expanded by 90% (much higher than current tests) but the run time was much longer than A* search, which disproved the optimality of our solution. Another similar issue was reducing both the runtime and node expansion significantly but losing completeness on some of our maps. We had pruned too many nodes, which happened to be useless to our smaller map cases, but were necessary to our larger map cases. Another obstacle we face was with Cooperative A*. We wanted to implement a hierarchy to handle more complicated multi agent search situations but we ended up running out of time to implement the details outlined by David Silver's paper “Cooperative Pathfinding”.

Conclusion

Overall we are happy that we were able to implement a successful pruning algorithm to optimize A* search within path finding. However, if we had more time we would have liked to explore path finding more through non-uniform cost search, cooperative multi-agent path finding (knowledge of each other agent). We have also realized how much research has been put into path finding, especially in optimizing A* and multi agent search. If we had more time, we would have really liked to finish WHCA* (Windowed Heirarchical Cooperative A*) or at least produce an optimal multi agent path finder. Although A* search is a good base to start with search problems, there are clearly a lot more techniques to push the boundaries of better space and time complexity. With the addition of pruning and better heuristic design we were able to make basic A* look like BFS compared to our JPS implementation, yet this was only the beginning.

References

A* references:

class notes:
http://www.cdf.toronto.edu/~csc384h/winter/Lectures/csc384w16-Lecture02-HeuristicSearch.pdf

JPS references:

http://www.benicourt.com/blender/wp-content/uploads/2015/03/5396-22335-1-PB.pdf

http://zerowidth.com/2013/05/05/jump-point-search-explained.html

Cooperative Pathfinding:

http://www.aaai.org/Papers/AIIDE/2005/AIIDE05-020.pdf

Maps:

http://www.movingai.com/benchmarks/