A Comprehensive Study on Pathfinding Algorithm for Static 2D Square Grid

Xuemeng Wei, University of Alberta, xuemeng2@ualberta.ca

Lu Donglai, Sichuan University, 1157113687@qq.com

Abstract—**Path finding is essential and widely used in many fields, like video games, tour guide or escape routes, and artificial transportation. Based on those fundamental algorithms like the DFS and A*, we explore two optimized algorithms, which include Jump Point Search(JPS) , SubGoal Graphs, and an additive heuristic . In our paper, other than showing how to implement the basic work of these algorithms, we also design some regular grid maps, which vary in terms of obstacle numbers, map size, obstacle shape, aiming to explore which algorithm fits better in which situation. This paper is constructive for the reader who wants to know how single-agent path finding problems figure out by using programming skills and the detection of the otherness among those algorithms in the realistic applied range.**

I. Introduction

In many games, path finding strategies direct the player to get to the destination efficiently without taking a detour and wasting time. Highly qualified pathfing algorithms prevent the player from being defeated by the adversaries and escape when necessary. Pathfinding also plays a vital role in being a tour guide. In 2011, Zhiguang Xu and Michael Van Doren showed how A* algorithms work, by using MVG system as visualized scene, customizing optimal path for the tourist depending on which destination they want to choose, and shows how algorithms deal with the dynamics of an emergency on path finding. Although we will not discuss dynamic obstacles in this paper, this indeed shows the importance of path finding. As is known to us, Depth-First-Search(DFS) and A-star(A*) are traditional path finding algorithm. DFS is an efficient search algorithm. It floods the map to check if there is a path towards the goal at a high speed, but causes loss of accuracy. A* is a direct search method for solving the shortest path in a static network map, and is a common heuristic for many other pathfinding algorithms.

Integrating the traditional algorithms, Jump-Point-Search(JPS) and Subgoal-Graph-Algorithm are innovative algorithms for optimal pathfinding eight-neighbor grids. Jump-Point-Search uses recursive functions to help us find the jump points and reduce the operation of the frontier. Subgoal-Graph algorithm combines improved DFS and A* to best fit the both simple and complex pathfinding problems.

In this paper, we will discuss the pros and cons of above algorithms and correspondingly design some maps to test them.

II. Algorithms

A. Introduction of algorithms

1) Depth First Search: Depth-First-Search (DFS) is an algorithm for searching a graph or tree data structure. The algorithm starts at the root node of a tree and goes as far as it can down a given branch or path, then backtracks until it finds an unexplored path, and then explores it.

The logic of DFS is simple and the algorithm structure is as follows:

a) Create an empty frontier(stack list) and put the root node into it. Mark the root node's parent node as None in a dictionary.

b) Pop out the last location from frontier. If it is the target, set found target flag as True and turn to d)

c) Push the accessible neighbors who haven't been visited into the end of frontier. Mark the parent node of them using dictionary. Repeat b) and c) until the frontier is empty.

d) If found target flag is True, trace back the root node by using the dictionary and form a path.

2) A:* A* is a very common and efficient pathfinding technique. Focus on the function:

$$f(n) = g(n) + h(n)$$

The g value is how far we can move from the start point to n. And the h is the heuristic value that computes how far between the current node and the goal node. Various methods can be chosen to calculate the heuristic value. For example Manhattan distance or Euclidean distance. The step of processing an A* search is:

a) Have an openlist that collects all the cells with its value in a tuple. Have a dictionary which we could take down the current and the parent. And we also need a cost so far dictionary to record the g value.

b) Second, pop the item with the smallest . Actually, this uses a priority queue to select the minimum cost. And mark it as the current cell.

c) Keep track of the neighbors of current and update the g value which gives the cost value of each cell since we move further.

d) For each neighbor we keep track in c) , if we do not look at the neighbor before,or the updated new cost,g, is smaller than what we record, then finish the f value and push the tuple (f, neighbor) into the frontier. Continue step b), c), d) if the openlist is empty or the current we pop is the goal

3) Subgoal Graph: Subgoal graph algorithm accelerates the pathfinding efficiency by building up subgoal graphs. The main idea of this algorithm is to identify a sequence of cells such that an agent always moves into a simple subgoal. Here introduce some definitions:

definition 1: *An unlocked cell s is a subgoal iff there are two perpendicular cardinal directions c_1 and c_2 such that s+c_1 +c_2 is blocked and s+c_1 and s+c_2 are not blocked.*

definition 2: *Two cells are* ***h-reachable*** *iff there is a path between them whose length is equal to the octile distance between them. Two h-reachable cells are* ***safe-h-reachable*** *iff there is no obstacle in the rectangle area made up of these two cells. Two h-reachable cells are direct-h-reachable iff none of the shortest paths between them contains a subgoal.*

Algorithm structure:

a) Construct a subgoal graph. Find all the subgoals of the map. Build up an undirected graph $G_S = (V_S, E_S)$ where V_S is the set of subgoals and E_S is the set of edges connecting direct-h-reachable subgoals.

b) Try a direct path. Test if the start cell and goal cell are safe-h-directable. If they are, return the direct path which contains moves in at most two directions from the start to the target and end the algorithm.

c) Find abstract path, which means find all the subgoals that compose the path between start cell and goal cell. Add the start cell and the goal cell into the subgoal graph: For each, find all the direct-h-reachable subgoals and connect them. Use A* algorithm to find the abstract path in the subgoal graph between the start cell and the goal cell. Restore the original subgoal graph.

d) If the abstract path exists, find a reachable path based on it. Use an improved DFS algorithm to form the concrete path between adjoining nodes in the abstract path. Return the path.

Three things are noticeable: 1) Step a) can be preprocessed since the algorithm is designed for the static maps. 2) Improved DFS mentioned above is target-oriented as two nodes chosen are h-reachable, which means there must be a path in the rectangle area composed by them. 3) In this algorithm, the agent can move diagonally only if both associated cardinal directions are also unblocked.

4) Jump-Point-Search: JPS search algorithm is a very efficient algorithm compared to A* in time efficiency and space efficiency in many cases. Although checking cell in each direction take some times, it adds fewer node to the frontier compared to A*. So, this also means we don't need to sort or pop off the node as often as when we use other algorithms. In this algorithm, recursive technique is the key point that makes sure we explore the route deeply. We introduce this algorithm by first understand the two main definitions:

definition 1: *A node n in* ***neighbors(x)*** *is a* ***forced neighbor*** *if n is a non-natural neighbor of x and the length link from parent(x) to x to n is smaller than the length links from parent(x) to n without passing x*

definition 2: ***Pruning rule is*** *to prune a cell from a nine cell grid followed by the two rules: First, if it is a* ***straight move****, then for n in neighbors(x), if the path length from parent(x) to n without passing x as the middle cell is shorter or equal to the path from parent(x) to x to n, then prune n. Second, if it is a* ***diagonal move****, then prune n if length from parent(x) to n without passing x as the middle cell is shorter than the path from parent(x) to x to n*

This is the general steps that we use in jump point search, very similar to the A*:

a. Use the list to hold the successors in the main function. We might call it frontier. We will always update this frontier when we end up finding all the successors of the current cell.

b.since for every node we traverse, we have a value for them. The value is the value of the parent plus the distance between current and parent. We pop the jump point which has the lowest heuristic value in the frontier, treat it as 'current' and start to find its successors until we find the node that we pop is the goal.

The term that appear in above's step is ' successor' and 'jump point'. To define successor, we use a list. This list will contain all the jump points of the current node. And we will return this to the main function when we confirm we find all the jumps point of that current node. In this process, we don't need to find the jump points in all 8 directions, but to use pruning rules to decide which specified direction we need to find jump points from.

And to define the jump point to return to the successor list, there are three type of decisions we need to make. First, the node we traverse is the goal. Second, in all the neighbors of the cell,lets say n, if there exist a neighbor which is the forced neighbor, then the cell n is a jump point. Third, when we now in a diagonal move, we use recursive to find the jump point in the two vertical directions he decompose into. So, when determine the jump point, we closely use the direction that from parent to current. But whatever which direction we deal with, we always use recursive, except when we meet an obstacle or get out from the grid.

B. Analysis of algorithms

1) DFS: This algorithm use stack(last-in-first-out) to implement, it can find the possible path from the start to the goal, but not guarantee the path is the shortest path. Examples will be show in the Measurement section.

2) A:* Compared to DFS, this algorithm work more efficiently and can find more optimal path. We use priority queue instead of a stack to find the path. For each current cell, by measuring the f value, we always need to one of the neighbors into the priority queue and pop. So the intermediate procedure is a bit complex compared to other advanced algorithms that we will discuss later in some cases. But when the start enable to go to the goal without blocked with a huge impenetrable wall, the path will be found at once.

3) Subgoal Graph: This algorithm conbines DFS and A* to flexibly offer suitable path searching method to get optimal result. Therefore, the integrated performance of it will be good.

In the preprocessing phase, it creates subgoal network graph, which later helps quickly build up connections between start cell and goal cell, and thus forms the path. Hence, subgoal plays a vital role.

Subgoal is the corner point of the obstacle. The more discrete obstacles, the more subgoals. So we deem that distribution of the obstacles in a map will have an influence on the efficiency of the algorithm.

4) Jump-Point-Search: Similar to A*, JPS use priority queue to find the path. But doing less steps when popping the cell from the frontier, since we have jump points. So, only successors will be added into the frontier, and this avoid the intermediate behavior on the frontier.

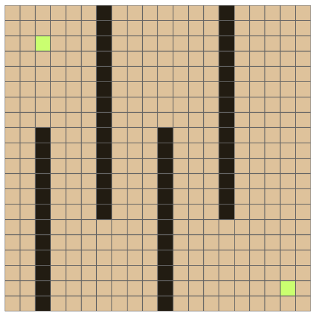
But unlike A*, the traversed cell do not go symmetric along the path like some general case A* preformed. Like for a diagonal move, JPS will search will traverse in the direction of the component vector along the diagonal direction.

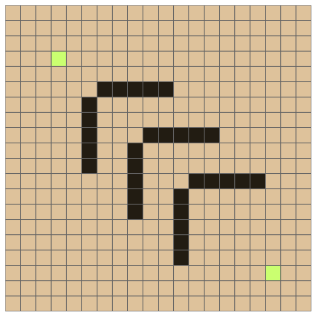
III. Measurement

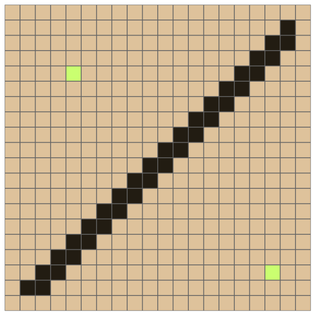
A. Maps we design

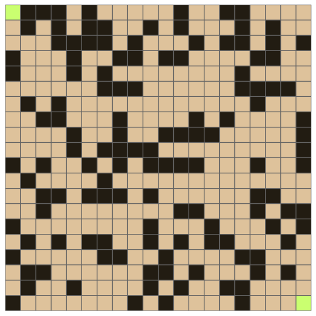
According to the analysis of 4 algorithms, we have designed 8 maps correspondingly, including 5 small size(20×20) maps and 3 big size (50×50) maps.

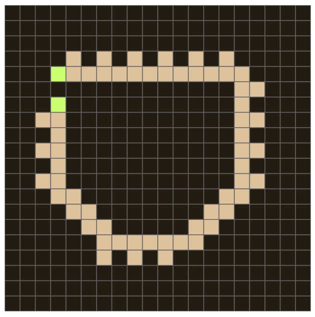
According to the analysis of the four algorithms and the exclusive rules and criterion, we design night maps and some of which could unambiguously present the speciality of the algorithms. Let's first talk about the regular shape graph, which is graph 1,2,3,5,7. For map1, the long bar obstacle lies regularly, we want to test how each algorithm preform on the map which has the best path with some wave curve. For map2, since the JPS can go through two cells even if two corner meet together, but some of the algorithms cannot, we want to use this map to show the speciality of JPS. For map3 and 6, very large empty space, we want to compare with the graph that has compact obstacles. For map7, there are so many forced neighbors for JPS and so many subgoals for subgoal graphs. So,for birth algorithms, this map is a challenge. For irregular map4 and map8,we want to see how compact obstacle influent the pathfinding algorithm performance.

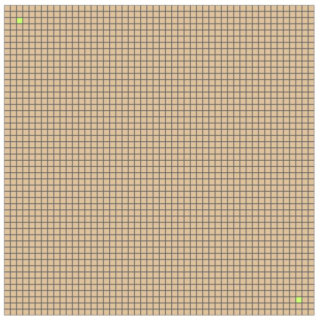

(a) map1

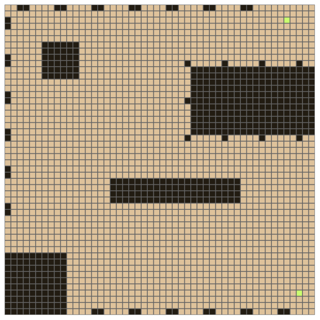

(b) map2

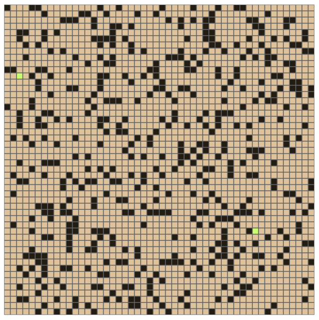

(c) map3


(d) map4


(e) map5


(f) map6


(g) map7


(h) map8

B. Results

We measure the algorithms in terms of speed, length of the path and memory occupation. For speed, we employ time library in Python to record running time of the algorithms. And it is worth noting that we divide the Subgoal Graph algorithm into preprocessing part and running part,which is benificial for further discussion. For the length of the path, we determine that the respective lengths of cardinal and diagonal moves are 1 and $\sqrt{2}$. For the memory occupation, we take memory profiler as measurement tool. Small maps take up almost no memory, so we simply record the data for big maps. Dynamic changes in memory are difficult to capture, so the data obtained from memory profiler may not be accurate as expected, but small changes does have value, we'll discuss it later.

For each map, we test 50 times, record the experimental data and obtain the average value. The experimental results are presented in tabular form as follows.

TABLE I
TIME EVALUATION(TIME:MS)

Algorithm	DFS	A*	Subgoal Graph	JPS
Map1	1.1018	17.032	(34.008)+1.003	251.055
Map2	2.141	1.001	(41.028)+2.017	531.135
Map3	1.993	26.006	(58.576)+2.985	378.816
Map4	0.997	4.116	(31.022)+0.988	563.877
Map5	0.998	2.998	(10.003)+0.976	39.010
Map6	14.008	3.014	(976.398)+2.002	1822.889
Map7	7.018	40.328	(573.129)+6.018	11252.545
Map8	5.001	9.001	(264.507)+7.001	6446.372

TABLE II
PATH EVALUATION(LENGTH:UNIT)

Algorithm	DFS	A*	Subgoal Graph	JPS
Map1	91.68	36.63	43.07	36.63
Map2	45.84	19.8	26.24	19.8
Map3	44.42	33.9	34.48	33.9
Map4	75.05	29.21	35.07	29.21
Map5	51.5	38.73	44	38.73
Map6	132.11	63.64	63.64	63.64
Map7	144.01	64.18	78.24	64.18
Map8	560.02	48.36	55.38	48.36

TABLE III
SPACE EVALUATION(MEMORY:MIB)

Algorithm	DFS	A*	Subgoal Graph	JPS
Map6	0.1133	0.0078	(0.0703)+0	-
Map7	0.0039	0	(0.5742)+0	-
Map8	0	0	(0.9297)+0.0039	-

IV. DISCUSSION

A. Individual algorithm performance

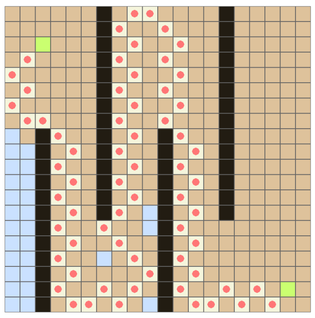
a) For DFS, in terms of time efficiency, no obvious differences shown in small maps. And for the large map, since it will take the agent to move down almost all the cell under the start node to get to the goal, which lies on the bottom right of the whole page, the whole time cost is larger than any of the map it measures. Hence, the ranking order of the neighbors of the agent matters for DFS, which directly determines the moving direction.

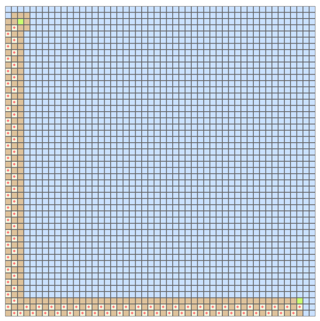
For path searching, DFS always gets the sinuous path although it is correct.

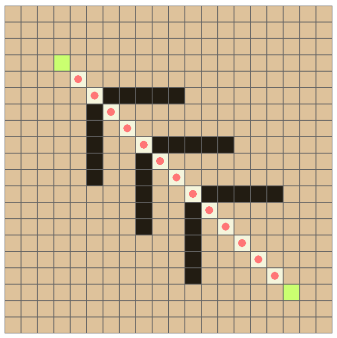
(Blue cells represent the cells the agent visited.)

b) For A*, since it can go through when obstacle meet together, map2 and map6 behaves quite similar, almost going from top left to bottom right. And when in map2,4,5,6, because of the large clear space and the direct destination point, A* runs very well in these maps. In terms of path choice, A* choose the right path consistently.

c) For Subgoal Graph Algorithm, the construction of subgoal graph brings a lot of benefits, because it is done in the


(i) DFS1


(j) DFS2


(k) A*

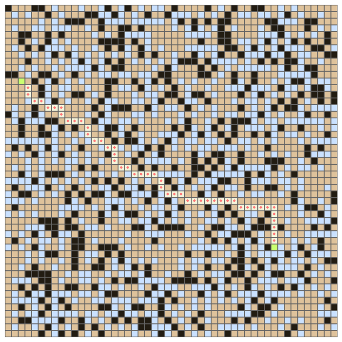
preprocessing phase. When a map is static, it is acceptable for us to spend some time getting some stuff done before the agent come in to move. From the result, we know that the preprocessing takes a mount of time while the the real running time is little.

For all kinds of maps, the Subgoal Graph Algorithm performances well since the existence of the subgoal graph makes the pathfinding problem more like matching. But for the drawbacks of it, two things are worth discussing.

First, the limitation of the moving directions. In this algorithm, there is only one rule: The agent can move diagonally only if both associated cardinal directions are also unblocked. Without such rule, the algorithm cannot work at all, because if the agent can move freely, then the subgoal(the turning point of an obstacle) will not be necessarily right on the path of the agent when it arrives the angle of an obstacle. Such limitation will definitely lengthen the searching path, which decrease the superiority of the algorithm to some degree.

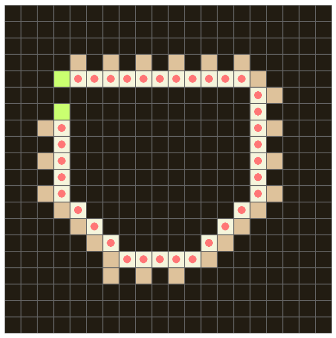
Second, the types of maps indeed have an impact on the efficiency of the algorithm. Because it requires plenty of time to construct subgoal graph and store some necessary data. If a map is full of discrete obstacle, namely, a map is full of subgoals, it needs much more time and more memory to build up the subgoal network graph. Besides, in path searching phase, start cell and goal cell will be added into the subgoal graph and make connections with some of the nodes. The more complex subgoal graph, the more running time of the algorithm. However, experiment result indicates that the complexity of subgoal graph mostly influences the preprocessing time which we don't care so much.

(The blue cells represents the subgoals.)



(l) subgoal graph

d) For Jump-Point-Search, because of our code works not as expected,although it find the right path, it seems that the time efficiency is so slow. So, in the discussion part, we just want to talk about its individual algorithm among 8 maps. Among small maps, JPS runs faster in map1 and map5. In map1,unlike the random scattered map like map4 and map7, successors can easy be found. So, path can be found quicker than others.Contrast with map1, in map7, too many In map5, the map almost filled with obstacle, JPS runs dramatically faster than other maps. For this one path map, because JPS doesn't need to explore the long path recursively, so it can immediately find the forced neighbor of the current cell, and put that cell into the successor immediately. And for path choosing, this algorithm find the absolute shortest path.



(m) JPS

B. Comprehensive comparison

Based on the time efficiency of way-finding, each algorithm has its own advantage. If discuss the map with large blank area, and the move from the start do not obstructed by obstacles, like map2 and map 6, then A* is very optimal. This is because there is always a cell that move near to the goal. But for the same map6, DFS is not a good choice since both the distance of the row index and the column index have large span. If the start cell want to find the goal, almost all the cell will be traversed. JPS also has the similar situation when solving map6. As JPS searching for all the jump points, no jump points return until we reaching the goal and return the goal. Additionally, for map7, both Subgoal graphs and JPS have poor performances for finding subgoals or forced neighbors throughout the map.

The dynamic changes of memory also reflect on the algorithms. For map6, DFS takes up the most memory while the A* takes up the least memory, it shows the differences between aimless and goal-oriented algorithm. When there is no obstacle to force the agent to change the moving direction, DFS acts like flood filling rather than searching. On the contrary, A* move directly towards the goal without visiting many cells which are unnecessary. For map7 and map8, both of the maps have multiple discrete obstacle, in that case, there will be a great number of subgoals when constructing subgoal graph. Therefore, the memory occupied by Subgoal Graph Algorithm will significantly increase. Contract with A*, JPS always find jump points and them into openlist instead of dealing the openlist in each move. Consequently, less memory will be taken than A*.

V. Limitation

1) JPS algorithm should works fast, but from the TABLE I, some errors might been made when we find what is the next cell to go next, or we recursive meaningless times and traverse the same cell several times.
2) Due to insufficient technique capacity and the time limitation, we cannot implement the Two-level subgoal graph and the improved Jump Point Search. But we will explore them and try to implement later so as to find more valuable information.

VI. Conclusion

1) DPS runs at a fast speed but not in an accurate way. The efficiency of it depends heavily on the position of the start cell and the goal cell as well as the distribution of the obstacles. Hence it is an unstable pathfinding algorithm.
2) A* preform efficiently and always find the shortest path. It is a wise choice to use it when there exist a one direction path. If the start cell is cut off by a set of consecutive obstacles from reaching the goal, it may take more time when using A*.
3) Subgoal Graph is an algorithm of good comprehensive performance. Preprocessing for constructing subgoal graph costs a lot of time, but saves a great amount when running it. It takes long to deal with the map with multiple complicated and discrete obstacles. Correspondingly, it is efficient to manage open maps with consecutive obstacles.
4) Jump-point-search algorithm increasing the speed of detecting jump point by using pruning rules. It only search the jump points and put them in the openlist, and use some of the jump points to find the shortest path. It is efficient since it avoid of add meaningless cell. But

although there is few cell in the openlist, it there are vast tracts on the map, JPS will take some time explore deeply and weast time.

ACKNOWLEDGMENT

In this paper, we want to thank for the professor Bill Nace, who always be patient when we have questions or problems on our project. And when we have no ideas about where to choose topics, we are grateful to Bill for giving us inspirations and resolving our confusion after class. We would also want to thank to our tutor Ziyi Wang. When we get into trouble on finding the bug in our code, he always glad to think our questions carefully and give us suggestions, which are extremely constructive for our project.

REFERENCE

[1] . Abd Algfoor, Z., M.S. Sunar and H. Kolivand, A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. International Journal of Computer Games Technology, 2015. 2015: p. 1-11.

[2] D. D. Harabor and A. Grastien, ”Online graph pruning for pathfinding on grid maps,” in Proceedings of the 25th National Conference on Artificial Intelligence (AAAI ’11), San Francisco, Calif, USA, 2011

[3] T. Uras, S. Koenig, and C. Hernández, ”Subgoal graphs for optimal pathfinding in eight-neighbor grids,” in Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS ’13), Rome, Italy, June 2013.

[4] . Xu, Z. and M. Van Doren. A Museum Visitors Guide with the A pathfinding algorithm. 2011: IEEE