Homework 2: Search-based Path Finding

Huan Chen

1. Matlab Part

Test the matlab codes with a 2D map: 250×250 map, obstacles ratio is 0.3, target location (250, 250). Calculation result is showed in Table 1.

Dijkstra is the slowest, as showed in the Figure 1, it visits all the nodes on the map. A* with Manhattan distance as heuristic function is the fastest while fails to find the shortest path.

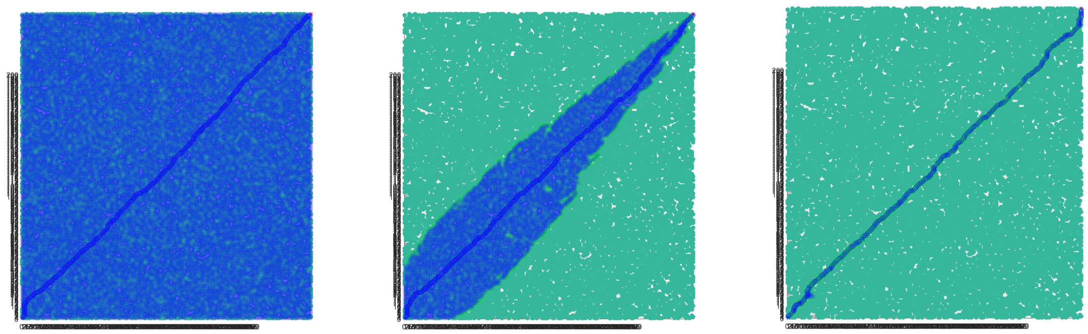

Figure 1: Path planning on 250×250 map, heuristic function used (from left to right): 0 (Dijkstra), Euclidean distance, Manhattan distance

2. ROS Part

2.1 The influence of heuristic function

Table 2 shows the path planning result with three heuristic function, targeting three points: A, B, C. We can conclude the following conclusions:

Table 1: A* path planning with three heuristic functions

	0 (Dijkstra)	Euclidean	Manhattan
Path length	370.30	370.30	378.40
Time	78.43	25.93	0.62

Table 2: Path planning with three heuristic functions, 3 target points

Target Point 1-A			
	Euclidean	Manhattan	Diagonal
Time (ms)	0.4732	0.0968	0.1322
Path cost (m)	5.5799	5.6971	5.5799
Visited nodes	496	26	99

Target Point 1-B			
	Euclidean	Manhattan	Diagonal
Time (ms)	1.2010	0.1205	0.3946
Path cost (m)	5.7999	5.7999	5.7999
Visited nodes	983	24	369

Target Point 1-C			
	Euclidean	Manhattan	Diagonal
Time (ms)	0.6728	0.1267	0.1695
Path cost (m)	5.4876	5.6340	5.4876
Visited nodes	288	21	85

1. Calculation speed: Manhattan $>$ Diagonal $>$ Euclidean, Manhattan is the fastest.
2. Number of visited nodes: Euclidean $>$ Diagonal $>$ Manhattan
3. Path cost (length): Manhattan $>$ Euclidean $=$ Diagonal, Manhattan cannot promise a shortest path.

2.2 The influence of tie breaker

Use Euclidean heuristic function, compare the influence of the tie breaker, the result is showed in Table 3. Tie breaker can reduce the visited nodes therefore shorten the calculation time, while keep the shortest length at the mean time.

2.3 A* and jumping point search

In lots of situations in this map, JPS can find a path faster than A*, as the **case 1** and **case 2** showed in Table 4. Because there are many obstacles on the map, JPS has no chance to expand too much. But the improvement is not so obvious in these two cases.

Sometimes, JPS has less visited nodes, but still takes longer time to find a path, as the **case 3** in Table 4. Because JPS reduces the number of nodes in the open list, but increases the number of status query.

Also, sometimes JPS is less efficient than A*. Like the **case 4** showed in Table 4. As you can find in Figure 3, the target point locates high space in 3D map, there are much less obstacles around, JPS could have more nodes to expand and have less chance to find jump points.

Table 3: Path planning with and without tie breaker

Target Point 2-A		
	No Tie Breaker	With Tie Breaker
Time (ms)	1.1697	0.5940
Path cost (m)	6.4683	6.4683
Visited nodes	496	26

Target Point 2-B		
	No Tie Breaker	With Tie Breaker
Time (ms)	1.3600	0.8106
Path cost (m)	6.0734	6.0734
Visited nodes	972	472

Target Point 2-C		
	No Tie Breaker	With Tie Breaker
Time (ms)	0.8412	0.2806
Path cost (m)	6.0971	6.0971
Visited nodes	514	220

Table 4: JPS and A* comparisons

JPS is faster	Case 1		Case 2	
	A*	JPS	A*	JPS
Time (ms)	0.4520	0.2721	0.4760	0.4520
Path cost (m)	4.3321	4.3321	4.8634	4.8634
Visited nodes	268	96	355	159

A* is faster	Case 3		Case 4	
	A*	JPS	A*	JPS
Time (ms)	0.6286	0.9159	0.3929	1.2010
Path cost (m)	5.7655	5.7655	6.7554	6.7554
Visited nodes	407	343	212	250

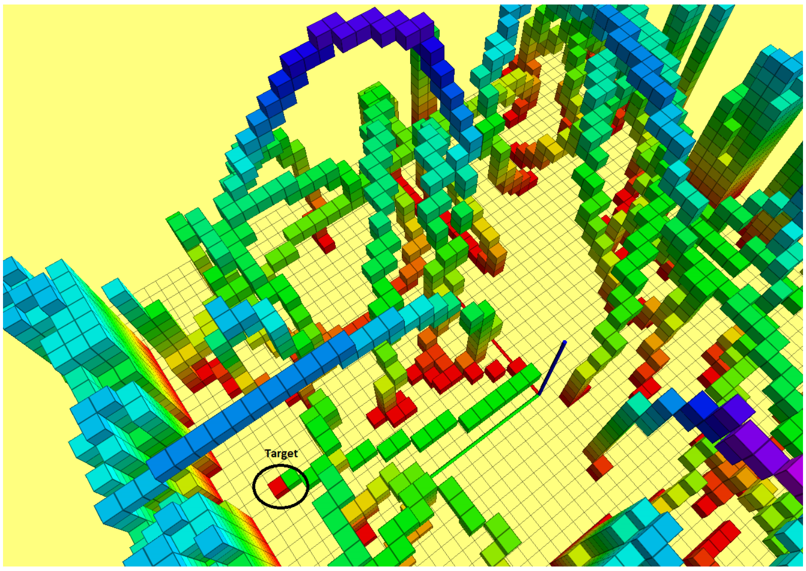

Figure 2: JPS and A* path search comparison case 1

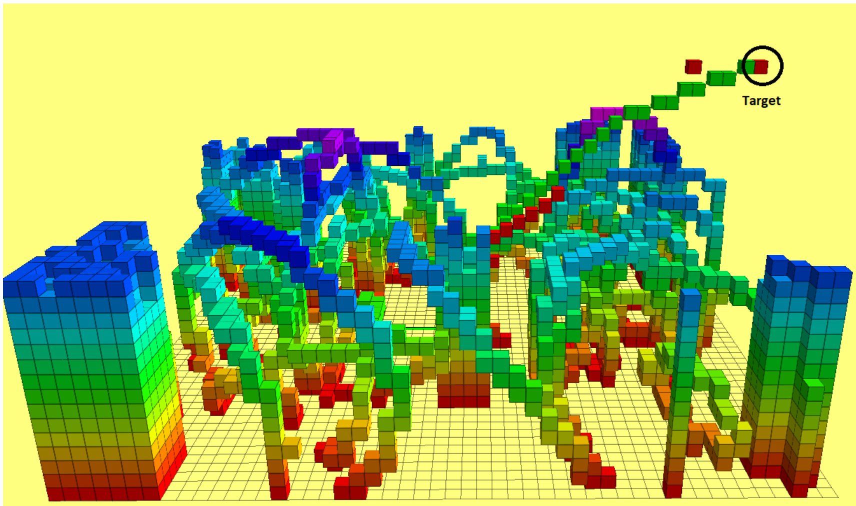

Figure 3: JPS and A* path search comparison case 4

3. Main Takeaways

This lecture is mainly about search-based path find algorithms.

1. Graph search basis: configuration space and work space.
2. Breadth first search (BFS) and depth first search (DFS).
3. Dijkstra and A*: pros/cons, heuristic function, tie break.

- Dijkstra, $h = 0$.
- Manhattan, $h = dx + dy$.
- Euclidean, $h = \sqrt{dx^2 + dy^2}$
- Diagonal, $h = dx + dy + (\sqrt{2} - 2) * min(dx, dy)$
- Tie break, $h = h \times (1 + p)$, $p < \frac{\text{minimum cost of one step}}{\text{expected maximum path cost}}$

4. Jump point search (JPS), node expansion method is different

Algorithm 1 A* Algorithm

Input: Map, startNode, targetNode
Output: Path

```
for node in the map do
    node.visited = false
    node.gScore = inf
    node.fScore = inf
    node.parent = NULL
end for
startNode.gScore = 0
startNode.fScore = 0
OPEN = startNode
while OPEN ≠ ∅ do
    currentNode = findLowestScore(OPEN)
    currentNode.visited = true
    eraseFromOpen(currentNode)
    if currentNode == targetNode then
        return buildPath(targetNode)
    end if
    neighborNodeSet = getSuccors(currentNode)
    for neighborNode in neighborNodeSet do
        edgeCost = calculateEdgeCost(currentNode, neighborNode)
        if neighborNode.visited == false then
            neighborNode.gScore = current.gScore + edgeCost
            neighborNode.fScore = neighborNode.gScore + calculateHeuristicScore(neighborNode,
            targetScore)
            neighborNode.parent = currentNode
            insertIntoOpen(neighborNode)
        else if neighborNode.gScore ≥ current.gScore + edgeCost then
            neighborNode.gScore = current.gScore + edgeCost
            neighborNode.fScore = neighborNode.gScore + calculateHeuristicScore(neighborNode,
            targetScore)
            neighborNode.parent = currentNode
        end if
    end for
end while
```