16-782 HW1 report

Qishun Yu

October 2021

1 Algorithm

In this algorithm, Dijkstra and weighted A* are used to generate valid paths. Both Dijkstra's algorithm and Weighted A* search are used once to find the path from the robot current pose to the goal pose. A sequence of path is then generated and executed at each step.

1.1 Dijkstra's Algorithm

Dijkstra is first used to calculate the 2D heuristic from the current position to targets in target trajectory, and the h value of each new Node is calculated as:

```
double newNode->hVal = currentNode->hVal +
(int)map[GETMAPINDEX(newX, newY, x_size, y_size)];
```

The reason for using Dijkstra's algorithm is the weighted A* wants an accurate heuristic function for faster calculation. The calculated h values at all valid locations are stored in an unordered_map and can later be used by Weighted A* fastly.

1.2 Weighted A*

Since map1 and map2 are very large, we want to expand less locations for faster calculations. I selected weighted A* to expands states in the order of $f = g + w * h$ values.

```
double gVal_new = currentNode->gVal +
(int)map[GETMAPINDEX(newX, newY, x_size, y_size)];
double hVal_new = h_cost[index_new2d]->hVal;
double fVal_new = gVal_new + w * hVal_new;
```

On the other hand, weighted A* is performed in 3D state space to consider the time frame. The state space of the search is $\{x, y, t\}$, therefore at each time step, the search space is $\{x \pm 1, 0, y \pm 1, 0, t+1\}$: a total of 9 successor Nodes are searched for each Node in the OPEN list. Therefore, the determining condition of path found is when a expanded Node and the goal Node are at the same location $\{x, y\}$ at the same time t

```
bool findPath(int x, int y, int t,
        int x_size, int y_size,
        int t_size, double* target_traj,
        int target_steps){
    int currentidx = calc_3dindex(x, y, t,
                    x_size, y_size, t_size);
    int goalidx = calc_3dindex(target_traj[t],
                    target_traj[t + target_steps],
                    t,
                    x_size, y_size, t_size);
    return (goalidx == currentidx);
}
```

1.3 Generate Path

Within the A* function, each Node's parentNode is updated and stored as a variable. Therefore, generating the path is a trivial task. Following is the core code of creating a stack of x, y positions of the path:

```
while (nextNode->parent != startNode) {
nextNode = nextNode->parent;
path.push(make_pair(nextNode->pose[0], nextNode->pose[1]));
}
```

2 Data Structure

2.1 Node

A new class called Node is created, Node has variables of the $fVal$, $gVal$, $hVal$ to store the values of f, g, and h. In order to trace the path from the goal location to the current robot location, a Node pointer $*parent$ is used to point to the previous Node.

```
class Node {
public:
    double fVal = (double) DBL_MAX;
    double gVal = (double) DBL_MAX;
    double hVal = (double) DBL_MAX;
    int tVal = -1;
    Node *parent = nullptr;
    vector<int> pose = {-1, -1};

    Node(){}

    Node(double f, double g, double h,
    int t, vector<int> p, Node* pa){
```

```
        fVal = f;
        gVal = g;
        hVal = h;
        tVal = t;
        pose = p;
        parent = pa;
    }
};
```

2.2 unordered_map

In order to find the Node and the h value at each location in the map, two unordered_maps are used. For 2D h value map, Key value is calculated using:

$$key = x * y\_size + y \tag{1}$$

For 3D Node map, Key value is calculated using:

$$key = (x * t\_size * y\_size + y * t\_size + t); \tag{2}$$

and the element value to insert is the Node pointer $Node*$.

```
unordered_map<long long int, Node*> NodeMap;
```

2.3 priority_queue

A priority queue is used for the OPEN list for faster retrieval of a Node with the smallest f value.

```
class Compare_Node {
public:
    bool operator()(Node *a, Node *b) {
        return(a->fVal > b->fVal);
    }
};

priority_queue < Node*, vector< Node*>, Compare_Node > openList;
```

3 Results

Map ID	Target Caught	Time Taken	Moves Made	Path Cost
1	YES	2892	2850	2892
2	YES	4314	3679	2360939
3	YES	395	349	395
4	YES	395	371	395
5	YES	173	160	2084
6	YES	126	79	3120

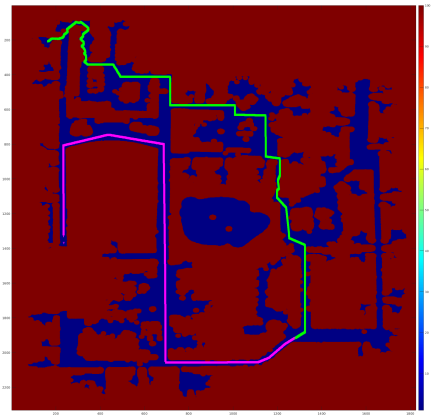

Figure 1: map1 result

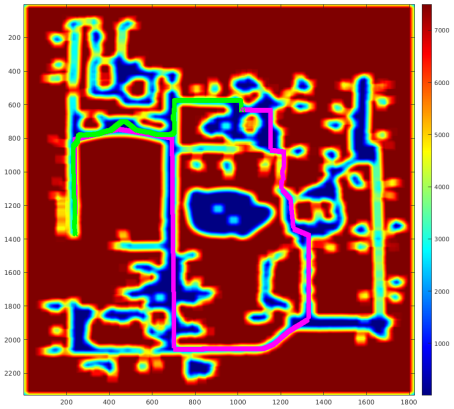

Figure 2: map2 result

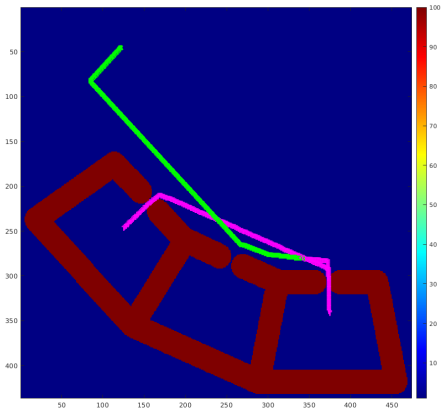

Figure 3: map3 result

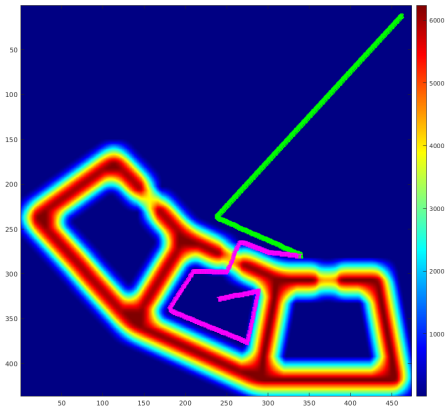

Figure 4: map4 result

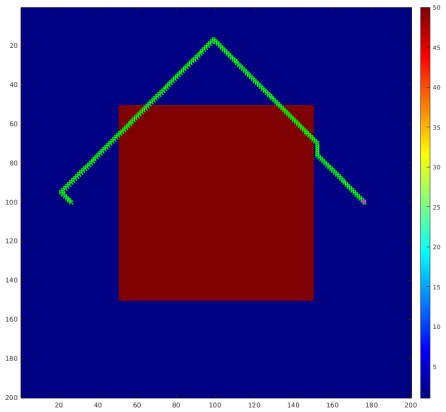

Figure 5: map5 result

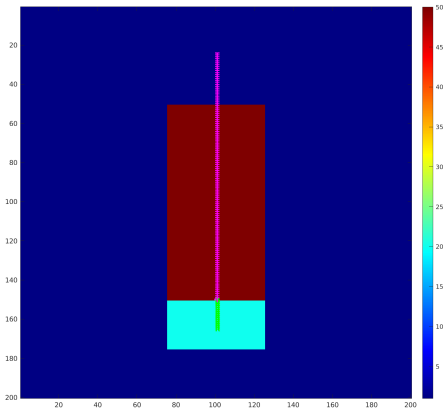

Figure 6: map6 result