AI 3603 Artificial Intelligence: Principles and Techniques

By: Luoyun Wang (521030910067)

HW#: 1

October 2, 2023

I. BASIC A* ALGORITHM

A. Description

A* is an informed search algorithm based on uniform-cost search algorithm and greedy algorithm. It is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost. In each loop, the way it determines the next step is to minimize the sum of the cost of the path and the cost required to extend the path all the way to the goal, which is often given by a heuristic function. We express this as a formula:

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

$g(x)$ is the exact cost from starting node to current node and $h(x)$ is an estimation from current node to goal node. The node with the smallest $f(x)$ is going to be chosen as the next node.

Specifically, in this lab, we need to implement the basic A* algorithm to find a path for a robot from starting position to target position. The robot can move in 4 directions and there are obstacles in the global map. For convenience, the robot is regarded as a point and the global map is discretized into a grid map.

B. Formulation

The focus of A* algorithm is to calculate $f(x)$ and to construct the loop. Thus, we can take the following approaches:

- Store searched points and $g(x)$ of them respectively.
- Choose an appropriate heuristic function $h(x)$ such as Manhatten distance.
- For each point being searched, get their neighbors and add them to a waiting list for search. Meanwhile, set those neighbors' parent the being searched point.
- If one neighbor of a point has been searched or has been in the waiting list, we compare the $g(x)$ of the two same points with different parents and keep the smaller-cost one.

With this, the prototype of A* algorithm is revealed: get $g(x)$ and $h(x)$ of each point in the waiting list, calculate $f(x)$, and choose one with the smallest $f(x)$ to search in the next loop.

C. Implementation

We use Python code to implement A* algorithm. There are some main points about the implementation:

- Use lists to store searched points and use multidimensional lists to store $g(x)$ and parents of points.
- Judge the neighbors of a point: being an obstacle or not, being searched or not and being in the waiting list or not. It is an important step before adding this neighbor into the waiting list.

Those are main points in algorithm level. As for points in syntax level, there is no need to give unnecessary details. The complete and specific Python code of A* algorithm is given in the **Task_1.py**. And the following image shows the path searched by A* algorithm using the code just mentioned.

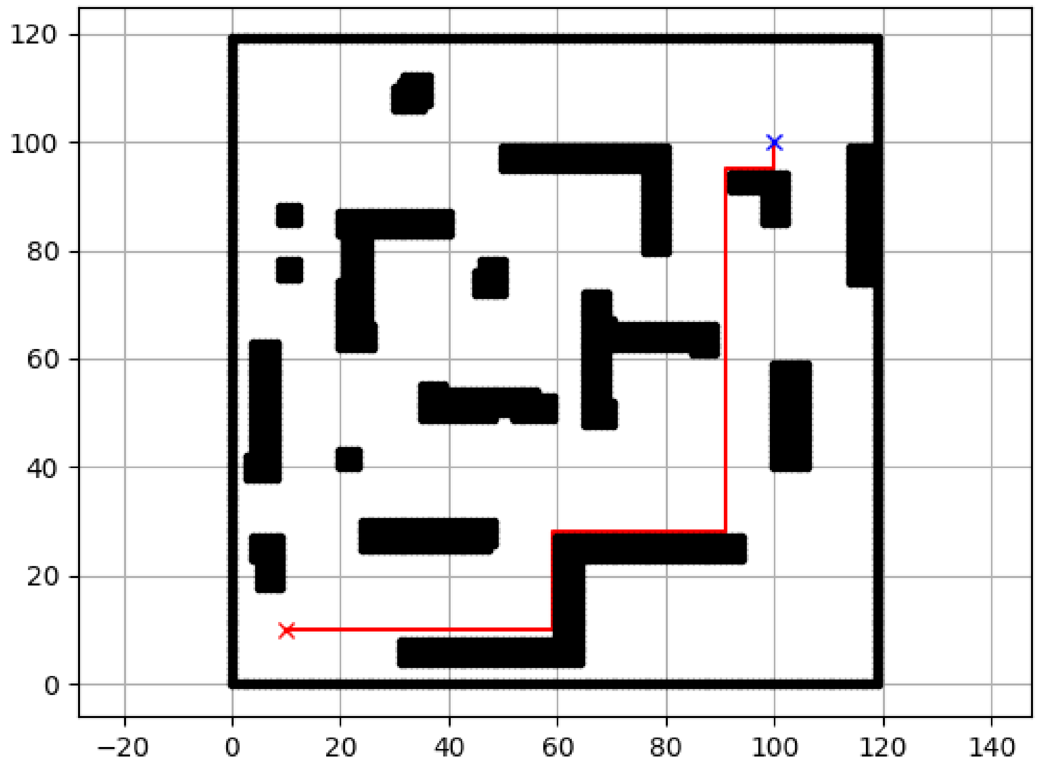

FIG. 1: Path searched by basic A* algorithm

II. IMPROVED A* ALGORITHM

A. Description

As we can see, the path searched by basic A* algorithm is not satisfying. The path is too close to obstacles so it is not a good path for robots. To be more realistic, there are 3 additional conditions:

1. The path should keep some distance away from obstacles.
2. Robots can move in 8 directions. That is to say, diagonal movement is possible.
3. Steering costs. We should reduce steering of robots.

Considering these conditions, A* algorithm needs to be improved.

B. Formulation

For each conditions above, there are corresponding methods to solve them respectively:

1. Add a function that checks whether the current point is too close to obstacles or not with a distance parameter.

2. Improve the neighbor-search function. Now we consider the neighbor of a point not only in the same line but also in the diagonal position.
3. Add a function that calculates steering costs. Cost of one turn is adjustable and it depends on how much we don't want to turn.

Notice that the moving cost of diagonal movement is different from that of same-line movement. Therefore we'd better change the moving cost to adjust the new moving pattern.

C. Implementation

Improvement of neighbor-search function is just adding more points into the neighbor list and I only give the implementation of the other two functions:

```
# Judge whether too close to obstacles
def close_to_obstacle(map, pos):
    if map[pos[0]][pos[1]] == 1:
        return True
    # Set distance away from obstacle as 2
    for i in neighbors(pos, 2):
        if map[i[0]][i[1]] == 1:
            return True
    return False

# Calculate steering cost
def steer_cost(last, now, next):
    if last[0] == -1 and last[1] == -1:
        return 0
    elif (now[0] - last[0] == next[0] - now[0]) and (now[1] - last[1] == next[1] - now[1]):
        return 0
    else:
        # Set steering cost as 1
        return 1
```

The complete and specific Python code of improved A* algorithm is given in the **Task_2.py**. And the following image shows the new path searched by improved A* algorithm using the code just mentioned (See next page).

D. Comparison with basic A* algorithm

Let's compare improved A* algorithm with basic A* algorithm in the following aspects:

- **Computational time**: Using the timer in the Python library, we can measure the running time of the codes of two A* algorithms. The result shows that the running time of improved A* algorithm is more than 10 times faster than that of basic A* algorithm. Possible reasons can be the reduction in searching points due to loop-breaking of close-to-obstacle points and diagonal search, which makes it get the goal more quickly.
- **Safety**: It is obvious that improved A* algorithm is much safer than basic A* algorithm. Because the improving search will deliberately avoid obstacles.
- **Optimality**: Improved A* algorithm may be not so optimal as basic A* algorithm. Considering it has to avoid obstacles, there are less choices to reach the goal. And $h(x)$ is implemented as Manhatten distance so it may be larger than the real distance. These two aspects decrease optimality.

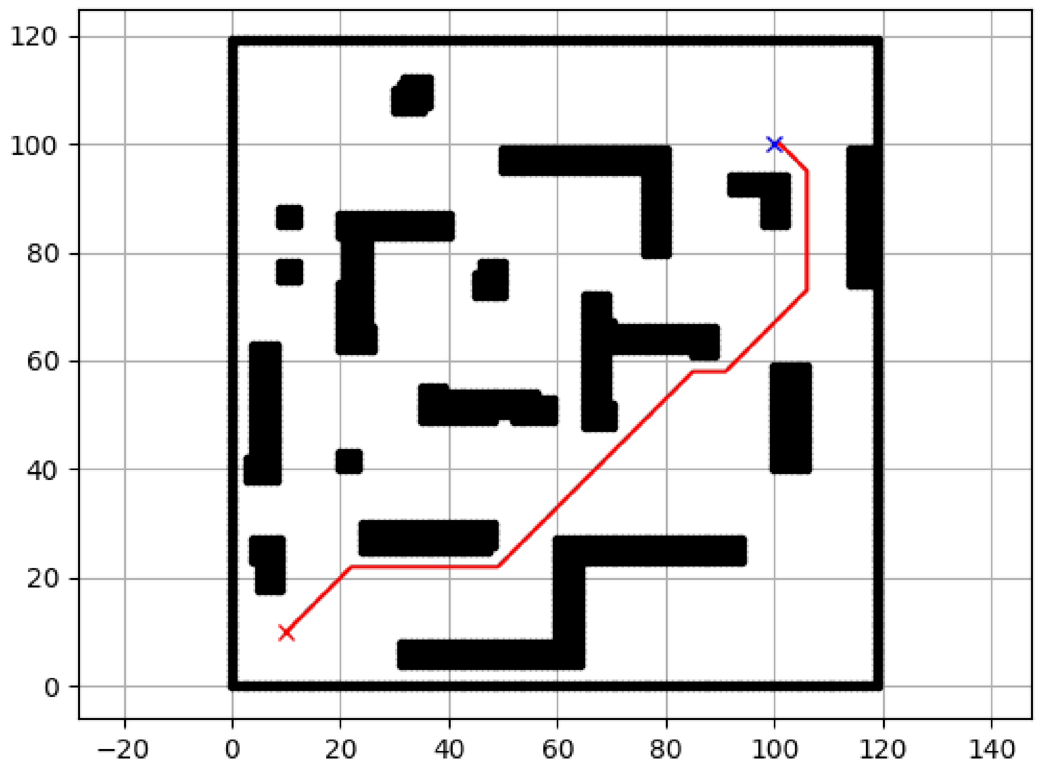

FIG. 2: Path searched by improved A* algorithm

III. SMOOTHNESS-IMPROVING ALGORITHM FOR PATH

A. Description

Although improved A* algorithm gives a good path in the last section, it is not ideal for real robots. Real movement tracks should be smoother, which is usually required to guarantee the comfort and energy efficiency. So we need to improve the smoothness of the path.

To reach the goal, let's choose *Bézier curve* to make the path smoother. A set of discrete control points defines a smooth, continuous curve by means of a formula. That is the using method of Bézier curve. For different numbers of control points, curve-generating formulas are different. For $n+1$ control points (n degrees), There is a general definition of it:

$$B(t) = \sum_{i=0}^{n} \tbinom{n}{i} (1-t)^{n-i} t^i P_i \qquad 0 \le t \le 1$$

where $\tbinom{n}{i}$ are the binomial coefficients and all P_i are different $n+1$ control points.

B. Formulation

Steering points are the most smoothness-needed points, so we just set them and their neighbors on both sides as control points and change the original broken line to Bézier curve.

- For relatively independent steering points, we set 3 control points and use quadratic Bézier curves:

$$B(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \qquad 0 \leq t \leq 1$$

- For two steering points that close to each other, we set 4 control points and use cubic Bézier curves:

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)t^2 P_2 + t^3 P_3 \qquad 0 \leq t \leq 1$$

Notice that the distances between chosen neighbors and steering points can be set individually for each steering points. So we can set different distance values for points in different places.

C. Implementation

Just modify the final path and dispose each steering points. We get the new points by setting 1000 or more t and calculating $B(t)$ that we've got from original control points. Here is the function of calculating new points:

```python
def Bezier_func(index, path, case): # index: path indexes of control points
    res = []
    # Set iterations as 10000
    t = np.linspace(0, 1, 10000)
    for dt in t:
        if case == 2:
            # Quadratic
            x = path[index[0]][0]*(1-dt)**2 + path[index[1]][0]*2*dt*(1-dt) \
            + path[index[2]][0]*dt**2
            y = path[index[0]][1]*(1-dt)**2 + path[index[1]][1]*2*dt*(1-dt) \
            + path[index[2]][1]*dt**2
        elif case == 3:
            # Cubic
            x = path[index[0]][0]*(1-dt)**3 + path[index[1]][0]*3*dt*(1-dt)**2 \
            + path[index[2]][0]*3*(1-dt)*dt**2 + path[index[3]][0]*dt**3
            y = path[index[0]][1]*(1-dt)**3 + path[index[1]][1]*3*dt*(1-dt)**2 \
            + path[index[2]][1]*3*(1-dt)*dt**2 + path[index[3]][1]*dt**3
        res.append([x, y])
    return res
```

Searching for indexes of steering points is simple but checking whether two points are suitable for cubic Bézier curves or not is not easy. So a good method is to statically set every point and their neighbors. The specific code is a bit long so it is not going to be quoted here.

The complete and specific Python code of smoothness-improving algorithm is given in the **Task_3.py**. And the following image shows the new path searched by improved A* algorithm with smoothness-improving using the code just mentioned.

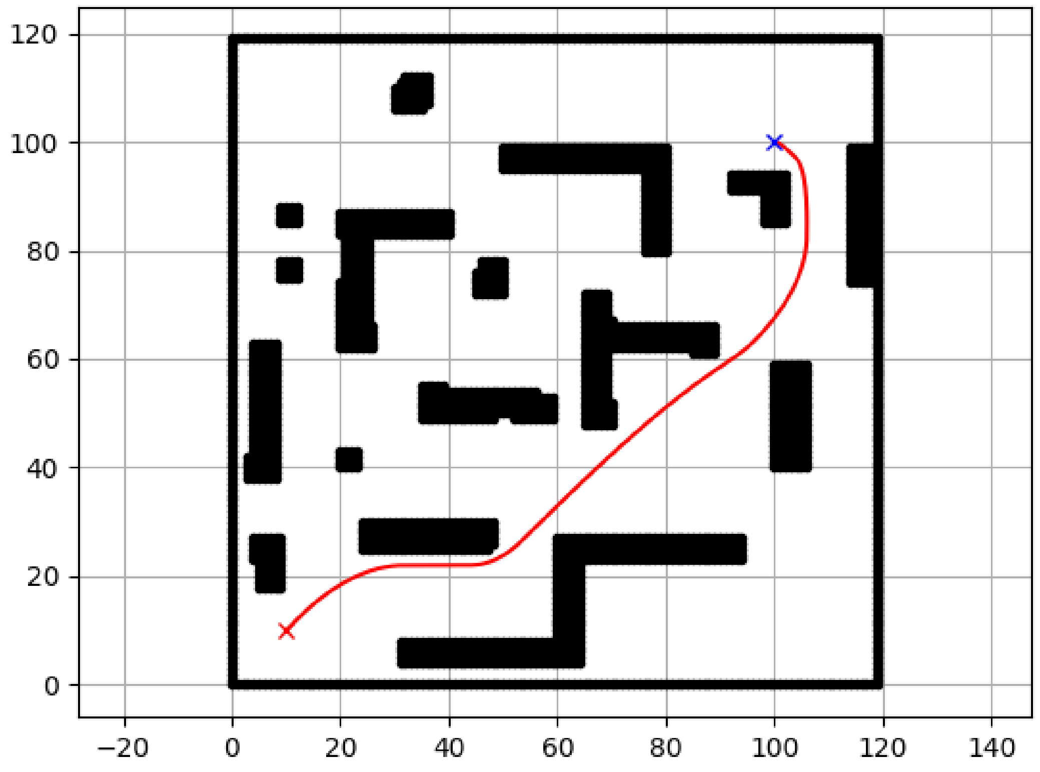

FIG. 3: Path searched by improved A* algorithm with smoothness-improving