Final project report

Introduction:

This project focuses on designing AI algorithm for a multi-agent system. This algorithm will be able to plan a course for each agent in the system as well as avoid any collision among them, and each step taken by the agent should be generated within 1 second. Since I do the project individually, only two agent is involved.

Algorithm for path searching:

As it is a typical path planning problem, I choose A* algorithm. It is packed into the function routing() and set it to be triggered when the route is generated from the first time. It is consisted by three functions: expand(), expanding the node passed to the function and return its all possible successor; cost(), computing the overall efficiency of the actions has been taken; h(), the heuristic function to estimate the cost to target position. The overall cost is computed by cost()+h(), according to which nodes are prioritized in ascendant order. I borrowed the file util.py in the assignment 2 to use the PriorityQueue structure to help me with that.

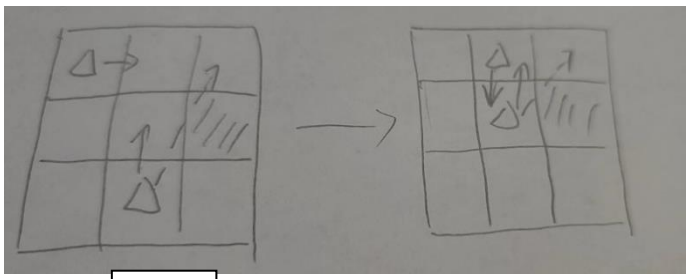
Collision avoidance:

As there are more than one agent, there is always a risk for them to bump into each other. However, when searching a route, one agent will treat the other agent as steady, and every movement of one agent will affect the route choose of the other. The basic idea to solve the problem is, when the agents enter unsafe condition, they will replan their route before taking every action until they are safe again.

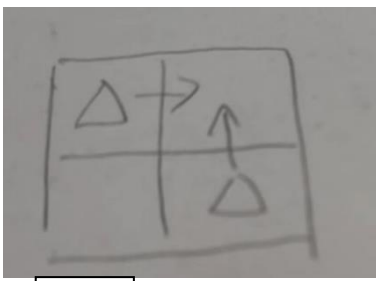
1. Threshold selection

   To avoid collision, an early detection is needed. As replan is still costly despite it is done in the middle way, we don`t want replanning to happen frequently. However, if the two agents come too close to each other, there may be not enough space so one agent has to maintain idle until the other pass by it. The most ideal case is, the two agents can keep moving and no collision happens.

   In my implementation, the avoidance function will be triggered when the distance between the two agents is no more than a threshold. In order to figure out the exact value of it, I analyze situations is possible for a collision:



P1



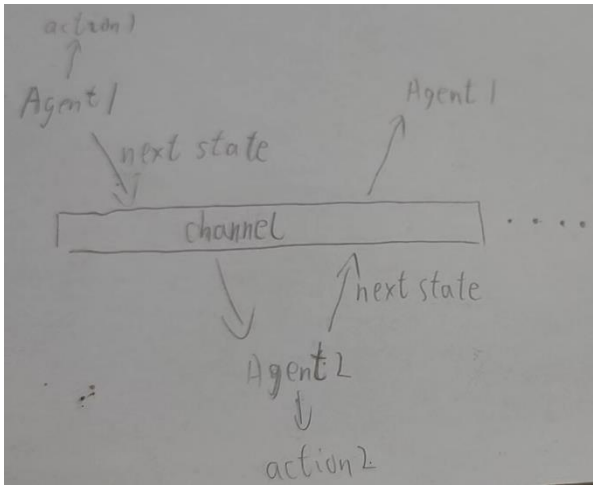
P2

In p1, collision happens when two agents attempt to move across each other. However, as the dash arrow indicated, the way of one agent is completely blocked by the other, which means they need to dodge the other at one stage ahead. The distance is calculated by $\Delta x + \Delta y$, which is 3 and set to be the threshold. Another condition is

shown in p2 when two agents attempt to enter the same block, which is included by the threshold=3.

2. Communication:

   In the program execution, the two agents are indeed taking actions by round. This makes it possible for the agent to inform their next location to each other. In order to do that I define a class variable “channel”. After one agent choose its action, it will compute its next state and add it to channel. In the round of the next agent, it will check where is already occupied from the channel, then take valid action and put its next state to the channel. The two agents provide feedback to each other so that they will not run into each other. The agent taking action first will be the one with more step count. The idea is, for the agent with less step, one step idle will not affect the overall step count for completing the task



Optimization:

The biggest problem of this project is the time limit. Because A* search requires the route be generated before taking any actions, the searching time is restricted to only 1 second. Thus, it requires a lot of work on optimization

1. History:

   The list history is used to store all visited node and avoid repeated visit. By estimation, it speeds up almost 200% after the history is added

2. Optimal cost function:

   When there is more than one node in the same priority, the A* search will do breadth first search. Apparently not all these nodes are needed to be expanded, so it is reasonable to avoid too many nodes cluster in the same priority. Instead of counting steps, I use the dot product of direction vector (from current position to goal) and action, normalized it with the length of direction vector. The great the product is, the faster the agent is approaching to goal. Since each action takes at least one step, the cost become $2 - product$

```
def cost(self, action, direction):
    return 2-(action_dict[action][0]*direction[0]+action_dict[action][1]*direction[1])/math.sqrt(direction[0]*direction[0]+direction[1]*direction[1])
```

3. Large map cost:

   On the large map, the scale of obstacle is much larger than the agent. If the agent

run into a wall, it takes several steps for it to move along to wall before it gives up. So I add an extract cost to the agent: for each side is blocked by an obstacle, the cost increase by 10. For most of the cases, the agent will start seeking an alternative route much faster

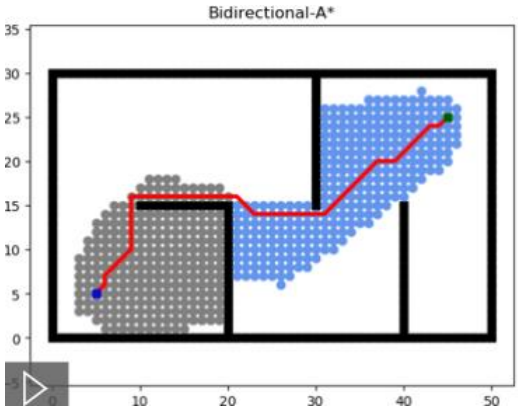
```
if(self.env.env_name=='large'):
    nodes.push(state,state[2]+self.h(state[0][self.name], goal)+10*(4-len(successor)))  # optimization
else:
    nodes.push(state,state[2]+self.h(state[0][self.name], goal))
self.history.append(state[0])
```

4. Avoid nil and moving back and forth:

   If the agent does not reach its goal, it should keep moving. If the agent moves upward just now, then attempting an immediate downward movement will make absolutely no sense, same with moving leftward and rightward. These two cases are straightly skipped to save time

5. Bi-directional A*:

   After applying all the technique above, it still takes about 3 second to search from the furthest point to goal. After searching for a lot of information online, I found bi-directional A*. The idea is, the goal will also search to the starting point. When the two search areas meet, we trace it back to recover half of route from each of them and combine them into a complete route. Although it sacrifices a little bit accuracy, it makes the program extremely fast!



Conclusion:

After a week of hard work, the agent eventually completed all tasks under the restriction. To be honest, all the algorithms taught in this course are not that abstract and I would still think implementing them are relatively easy. However, what makes a good AI engineer stand out from bad ones is the efficiency of program. For those programs I wrote in the past, as long as they can solve the problem, they are fine. But for AI program, the algorithm already guarantees a solution, so the speed matters now. Sometimes when we only require a good enough result, we may do trade-off and reach a balance between accuracy and speed. Clearly, AI is not only about design algorithms, but also make the algorithm achieve a desirable performance

As I`m a fan of real time strategy games, I know that A* is widely used in them, where hundreds of units, or agents, share the same map. It turns out that handling with only two agents already took me a lot of effort. I can image if the agent count grows to 3 as a team project has, or to hundreds, it will take much more optimization than what I have done and even new concepts. There is still a long way for me to go, to learn.