COMP3211 Project Report

Authors: YIN Zhuohao ZHOU Siyuan

Email: zyinad@connect.ust.hk szhoubd@connect.ust.hk

Project roles:

- Siyuan implemented the path-finding task using Dijkstra algorithm.
- Zhuohao tested and improved the system.
- The policy design part was done together.

1. Methodology

1.1 Path-finding by Dijkstra Algorithm

We used Dijkstra Algorithm to generate 9 **action maps** corresponding to each goal and each map and saved them as lists in `agent.py`. The implementation of the algorithm can be found in `dijkstra_test.py`.

For example, here is an **action map** for the blue agent corresponding to a small map. 5: 'nil', 0:'up',1:'right', 2:'down', 3:'left'. It shows what the agent should do in each position without considering other agents.

$$\begin{aligned}
&[[5,5,5,5,5,5,5,5],\\
&[5,1,1,2,3,3,3,5],\\
&[5,2,5,2,3,3,3,5],\\
&[5,1,1,5,5,0,0,5],\\
&[5,0,5,0,3,3,5,5],\\
&[5,1,1,0,5,0,3,5],\\
&[5,1,1,0,3,0,0,5],\\
&[5,5,5,5,5,5,5,5]]
\end{aligned}$$

A visualization is given below:

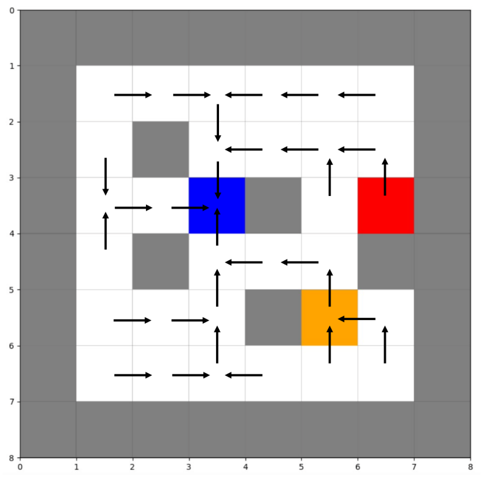

Figure 1: An illustration of how the action map works.

1.2 Rule-based System to Avoid Collision

1.2.1 Types of Conflicts

Here we consider different scenarios of impending collision in the context of two agents, say p1 and p2, with the following notations.

$p1.loc$: the current location of agent p1.

$p1.next$: the location of agent p1 in the next time step by taking the action produced by the Dijkstra algorithm.

$p2.loc$: the current location of agent p2.

$p2.next$: the location of agent p2 in the next time step by taking the action produced by the Dijkstra algorithm.

- **Vertex conflict.** A *vertex conflict* between agent p1 and p2 occurs when `p1.next == p2.next`. In other words, p1 and p2 plan to go to the same location at the same step.
- **Following conflict.** A *following conflict* occurs when `p1.next == p2.loc` (p1 follows p2) or `p1.next == p2.loc` (p2 follows p1).
- **Swapping conflict.** A *swapping conflict* occurs when `(p1.next == p2.loc) and (p2.next == p1.loc)`. Intuitively, the two agents try to swap their locations at the same step.

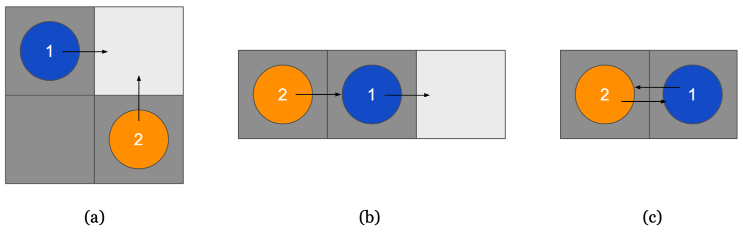

Figure 2: An illustration of the 3 types of conflicts. From left to right: a vertex conflict, a following conflict, and a swapping conflict.

After observing the patterns, we found that vertex conflicts and swapping conflicts are quite similar and we can identify them by evaluating `(p1.next == p2.loc) or (p2.next = p1.loc)`. From now on, we refer to them both by **meeting conflicts**.

1.2.2 Policies for Conflicts

For *following conflicts*, a helpful observation is that the agent being followed is free to make its move. Thus, whenever we identify a *following conflict*, we instruct the agent being followed to make it move according to its action map, while the other agent stays put. In the illustration above, the action of agent 1 will be ‘right’ while the action of agent 2 is ‘nil’.

In *meeting conflicts*, neither of the involved agents can safely make its planned next move. One of them must be the ‘nice guy’ to move out of the way and let the other one through. By this token, we manually set a priority for the three agents.

Color	Orange	Blue	Red
Priority	0	1	2

The smaller the number, the more ‘considerate’ the agent is, that is, when in conflict, it will always let the other agent through first by making a **random available action**.

In terms of implementation, we defined the following functions for conflict-checking:

- `noCollisionNextMove(a1_loc,a2_loc,a3_loc,a1_action,a2_action,a3_action)` returns true if a1_action does not induce any type of conflict or potential collision with the other two agents a1 and a2, and false otherwise.
- `twoAgentCollision(a1_loc,a2_loc,a1_action,a2_action)` detects the meeting conflict between agents a1 and a2 and returns true if one is detected.
- `freeToMove(a1_loc,a2_loc,a3_loc,a1_action)` checks if a1_action steps into a location currently occupied by a2 or a3.

To summarize, we first check if there exists any type of conflict if every agent follows the optimal action computed by Dijkstra. If no conflicts, then just do it. Otherwise, there must be either a following conflict or a meeting conflict. For *following conflicts*, we let the agent that is being followed move according to its action map, while the action of the one that is following it is ‘nil’. For *meeting conflicts*, we let the high-priority agent to choose a random available action to clear the way, while the low-priority one stays still.

2. Testing and Improvement

With the above conflict-handling policy, we tried running it with different initial positions of the 3 agents, and we discovered some cases where they ended up in a dead lock.

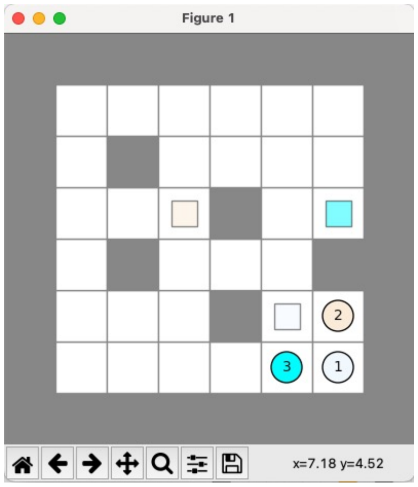

Figure 3: A scenario where our first implementation achieved low score or 0 score.

The adjustment we made lies in the random available action that is chosen for high-priority agents. Originally, the available actions included ‘nil’, which means the agent has a chance to stay still in *meeting conflicts* even if it has other choices. Consider a scenario of type (c) conflict, a *swapping conflict*, if the high-priority agent chooses ‘nil’ for multiple times (note that the low-priority one always chooses ‘nil’ in *meeting conflicts*), the conflict is still yet to be resolved, and the those time steps will simply be a waste. As an improvement, we excluded ‘nil’ in the first place when choosing a random available action. In this way, there will no longer be cases where two agent in a *swapping conflict* remains still and confront each other for several rounds.

3. Performance

We altered the number of test iterations up to 100 and obtained the following mean scores.

Map size	small	medium	large
Mean score	69.1333	72.9667	72.3634

Though the results are of high randomness, we believe our system has done a good job finding paths and avoiding collisions.