Implementing a Deliberative Agent

Smail Ait-Bouhsain, Maxime Gautier

October 23rd, 2018

1 Model Description

1.1 Intermediate States

We chose to use a chained list to represent states so we could generate all the states then reconstruct our plan from the goal state. States are therefore represented using several values :

PreviousState This allows us to reconstruct our plan through a chained list of states.

CurrentCity This variable stores the current location of the state, which is necessary for its definition

TaskStatus This is a string type variable where each character represents the status of a task. All tasks are represented in one string. The letter 'r' means the task has not yet been picked up, the letter 'p' means it has been picked up and is being carried and the letter 'd' means the task is delivered.

Task This is a task type variable which lets us know which task is being dealt with when creating a new state. It is linked to the action variable since it tells us if the task is to be pick up or delivered

Cost This variable represent the current cost of the state, which is the sum of the cost of all previous states plus the current one.

Weight This represents the total weight that the agent is carrying. If picking up a task will make the weight exceed the agent's capacity, it will not pick it up.

Action This Boolean type variable describes what is to be done with the task that creates a new state.

1.2 Goal State

The goal state is the state where all tasks are delivered and it is found through the string TaskStatus, when all characters are 'd' for delivered.

1.3 Actions

There are only two possible actions : either move to a city to pick up a task or move to a city to deliver a task. This explains our variables in the state representation.

2 Implementation

2.1 BFS

Our BFS algorithm was too slow so we added a condition so that it only checks states that have not already been treated or that have a lower cost than the same state treated before. It is the same condition as the one implemented by the C table in the A* algorithm and allows for much faster computing time without changing the BFS algorithm.

2.2 A*

In this algorithm we use the same condition as mentioned in the BFS implementation, for which we had to override the equals and hashcode in our State class so that it would compare according to the conditions we wanted.

Furthermore, we used a priority queue with a state comparator that we created to automatically sort each new addition and therfore optimize our computing time.

2.3 Heuristic Function

Our estimate for the minimal cost from the current state to the goal state is defined as follows. For each carried tasks, we compare the distance from the current city to the deliver city and for each remaining tasks we compare the sum of the distance to the pickup city and the path length of the task. We then keep the biggest value since the longest path will be, in extreme cases, the minimal distance to cover to deliver all the tasks. This is only true in the cases where all tasks can be picked up and delivered on the same path but it is a possibility. We then multiply the distance by the cost per km and add the cost of the current state to complete our heuristic function. This function is therefore optimal since it can never be higher than the potential shortest cost whilst minimizing computing time.

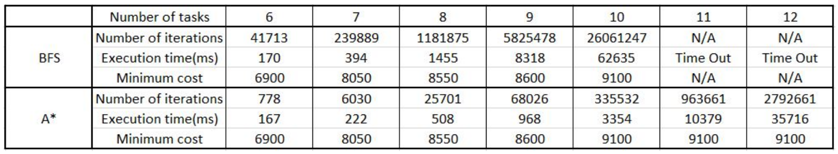
3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

We ran this experiment on the Switzerland topology with a seed 23456 for a number a tasks varying from 6 to 12 with one agent every time.

3.1.2 Observations



	Number of tasks	6	7	8	9	10	11	12
BFS	Number of iterations	41713	239889	1181875	5825478	26061247	N/A	N/A
	Execution time(ms)	170	394	1455	8318	62635	Time Out	Time Out
	Minimum cost	6900	8050	8550	8600	9100	N/A	N/A
A*	Number of iterations	778	6030	25701	68026	335532	963661	2792661
	Execution time(ms)	167	222	508	968	3354	10379	35716
	Minimum cost	6900	8050	8550	8600	9100	9100	9100

Figure 1: Figure 1 - Results for Comparison Experiment

As we can see the A* algorithm can run more tasks than the BFS without timing out but both always give the optimal plan that minimizes the cost. A* uses much less iterations than BFS due to its optimization, however each iteration takes more time which is logical since it must reorganize the queue in each iteration. This can be observed by comparing the number of iterations and computation time for 6 tasks in BFS and 9 tasks in A* : the number of iterations is close however A* takes about ten times more time.

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

We used the Switzerland topology with 7 tasks and the seed 23456. We compared the reward per Km of one, two then three agents in those settings to see how an agent's plan changes when there are more than one agent. All three agents run under the Astar algorithm for faster computation. We can see the initial configuration in Figure 2.

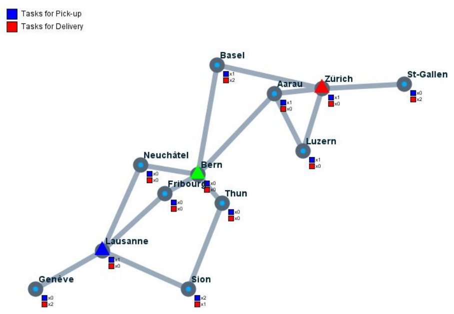
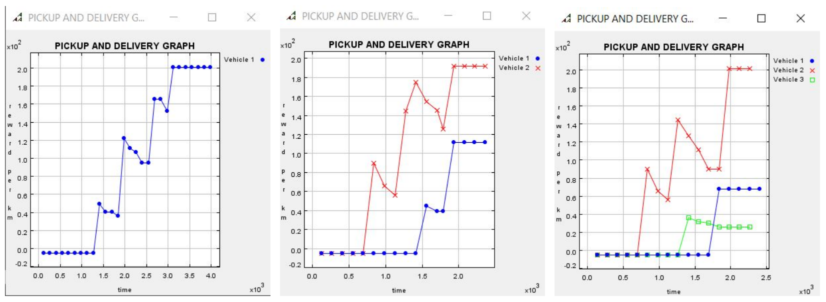


Figure 2: Figure 2 - Initialisation for Multi-agent Experiment

3.2.2 Observations



(a) One agent (b) Two agents (c) Three agents

Figure 3: Figure 3 - Rewards for one, two and three agents

When alone, the agent computes the optimal plan and runs it without any problems. When other agents are added, however, since they are not aware of each other's presence, all agents compute their optimal plan and run it until they come upon a city where the task has already been picked up by another agent. They then recompute a plan according to the available tasks at the moment, but since the other agents keep running, their plans rapidly become obsolete. It then becomes a race to pick up the most tasks the fastest, which is determined by the agent's initial location and the distribution of tasks. In this experience, the red agent spawns in Zurich in which there is a task and most tasks are distributed around him, which explains why it always ends up with the highest reward as we can see in Figure 3.