Ecole Polytechnique Fédérale de Lausanne - EPFL

CS-430 Intelligent Agents

Mission 3

Deliberative Agent

Vladimir Somers
Armand Bosquillon de Jenlis

Group 27

Professor : Boi Faltings

Academic Year 2015 - 2016

1 Introduction

During this project, we had to build a deliberative agent for the Pickup and Delivery Problem. The most important parts of this project, which are explained in details in this report, were:

1. How to represent the state of the agent.
2. How to compute a plan (optimal or not) that allows the agent to deliver all the tasks that have not yet been delivered.

2 Design Choices

2.1 State representation: s

A state is composed of three variables:

- **City:** This is the city where the agent is located.
- **Carried:** This is the set of tasks that the vehicle carry *before* doing anything in **City**.
- **Delivered:** This is the set containing the tasks that have been delivered *before* arriving in **City**.

We denote the **final state** of the search by **s***.

$State = \{City, Carried, Delivered\}$

2.2 Node of the search tree n

A **Node** is an object composed of one State, a pointer to the **Parent Node** of the search tree and a **Cost**. The Cost is the parent's Cost plus the distance between the city of the parent's Node and the city of the current Node. We define the **final node** of the search as being the node containing **s***. We denote it by **n***.

$Node = \{State, Parent, Cost\}$

2.3 Transition

In our representation, a **possible transition** is composed of the three following kind of actions:

1. **Delivery** of the carried tasks that can be delivered in the current city.
2. **Pickup** of a set of tasks (that we name *PickedUp*) from the set of tasks available in the current city, such that the sum of the weights of all the tasks in $Carried \cup PickedUp$ will not exceed the capacity of the vehicle.
3. **Move** to a neighbor city.

We will now formalize the transition representation and the composition of the sets in a given state. Let's first define two successive states, $State_{old}$ and $State_{new}$, with :

- $State_{old} = \{City_{old}, Carried_{old}, Delivered_{old}\}$
- $State_{new} = \{City_{new}, Carried_{new}, Delivered_{new}\}$

We consider that the vehicle goes from $City_{old}$ to $City_{new}$.

Now, we are going to define more formally the different sets that we use, it will help us to see how to:

1. Compute $State_{new}$ when we have a transition from $State_{old}$. This is useful for the method **getSuccessor**.
2. Compute the transition that occurred between $State_{old}$ and $State_{new}$, given only these two states. This will be useful for the method **computePlan**.

We also define two important sets :

- *PickedUp* is the set of elements that the vehicle picks up in $City_{old}$.
- *DeliveredInCityOld* is the set of tasks delivered in $City_{old}$.

The mathematical definition of the different sets is:

$$DeliveredInCityOld = \{task \mid task \in Carried_{old} \ and \ task.deliveryCity = City_{old}\}$$

$$\begin{aligned} Delivered_{new} &= Delivered_{old} \cup DeliveredInCityOld \\ Carried_{new} &= (Carried_{old} \backslash DeliveredInCityOld) \cup PickedUp \end{aligned}$$

$$\begin{aligned} DeliveredInCityOld &= Delivered_{new} \backslash Delivered_{old} \\ PickedUp &= Carried_{new} \backslash Carried_{old} \end{aligned}$$

The **second and third** equations are the one that we use in the method **getSurccessors**. The **fourth and the fifth** equations are the one that we use in the method **computePlan**.

3 Important methods

3.1 BFS

The Breadth First Search algorithm is used to search a final state in the state space. The final state found with BFS has the minimal number of actions, but not necessarily the minimal cost. We used a *LinkedList* to store the expended nodes, because the *insertion* and *deletion* operations are in $O(1)$ whereas these operations are $O(n)$ in the worst case with an *ArrayList*. We used a *HashSet* to store the visited nodes (in order to detect cycles), because this structure gives a *contains* operation with a $O(1)$ time complexity.

3.2 A*

The A* algorithm is used to search a final state in the state space. The final state found with A* is guaranteed to have the minimal cost, given that the heuristic used is *admissible*. We used a *PriorityQueue* to store the expended nodes. This structure allows to automatically order the nodes according to their cost when they are added. We chose it because it provides a $O(log(n))$ time complexity for the enqueing and dequeing methods. We used a *HashMap* to store the visited nodes, because this structure gives *contains* and *get* operations with a $O(1)$ time complexity. Each entry of the *HashMap* is a key-value pair with the State as key and the corresponding cost as value. We use this *HashMap* in order to avoid expending a State which has already been expended with a lower Cost.

3.3 getSuccessors

This function returns an *ArrayList* of all possible nodes containing the states that can be reached from the state of the **currentNode**, which is the node received as argument. These states are given by all the **possible transitions** from the state of **currentNode**. This function has two nested loop so it was important to choose a state representation such that it was possible to do as few computation as possible in the most inner loop.

3.4 isFinal

The goal of the agent is to deliver all the tasks that have not yet been delivered. This method *isFinal* allows us to know if a state is a **goal** state. When isFinal returns true, we can stop the search. Let **tasks** be the set of tasks not yet delivered. A **final state s*** is one for which:

1. $\forall \ task \in carried, \ task.deliveryCity = city$
2. $carried \cup delivered = tasks$

3.5 computePlan

This method computes the plan thanks to **n***. By using the pointer to the parent, it goes through the nodes starting from n* and reorder them by pushing each node on a stack. Then it goes through state1, state2, state3,..., state* and compute the plan thanks to this sequence.

3.6 Heuristic

In order to describe the heuristic, we are first going to give few definitions. In the following explanation, we talk about distances for ease of comprehension but what we really manipulate are costs.
Let **d1** be the following distance:

For each task in **carried**, compute the minimal distance to travel in order to deliver it. Then select the maximum of these distances and assign its value to **d1**. We denote by **t1** the task associated with this distance.

For each task that has not yet been taken, we define

1. **DpickUp**, the minimal distance between **currentCity** and task.pickupCity
2. **Ddeliver**, the minimal distance between task.pickupCity and task.deliveryCity.

Let **d2** be the following distance:
For each task in the map that has never been picked up, compute the distance **DpickUp** + **Ddeliver**, then select the maximum of these distances and assign its value to **d2**. We denote by **t2** the task associated with this distance.
For any state our **heuristic** returns **max(d1, d2)**.

We finally have:

- $d1 = max(dist(currentCity, deliveryCity_t)), \forall\ t \in carriedTasks$
- $d2 = max(dist(currentCity, pickUpCity_t) + dist(pickUpCity_t, deliveryCity_t)), \forall\ t \in toBePickedUpTasks$
- H(n) = max(d1,d2)

We chose this heuristic because it was the most efficient between the heuristic that we built. Among the heuristic we implemented, it is the one that seems to have the most accurate estimation of the minimal distance to go from **currentState** to $s*$.

3.6.1 Consistency

Let **n** denote a node of the search and **c(n,n')** be the cost to go from **n** to one of its successors: **n'**.
To prove that this heuristic is consistent, we have to prove that $h(n) \leq c(n, n') + h(n')$. The value of $c(n, n')$ is the cost to travel from **city** to **city'**. In the worst case, when we move from **city** to **city'**, we move on the direction of the shortest distance of **t1** or **t2** which implies $h(n') \geq h(n) - c(n, n')$ and thus $h(n) \leq c(n, n') + h(n')$.

3.6.2 Admissibility

Given that the heuristic is consistent, it is also admissible, which means that the solution found by A* is optimal! You can find a more intuitive proof of the admissibility of our heuristic in the annexes.

4 Simulation Results

4.1 Performances of A* and BFS

We can see that A* is always faster than BFS and, of course, finds a better solution. In some cases, BFS finds a solution which turns out to be optimal. Actually, the time is proportional to the number of node visited, which is, for these examples, always smaller with A*. You can find the tables of the other maps in the Annexes.

Switzerland	BFS			$A^*search$			naivePlan
Number of tasks	Explored nodes	Time (s)	Cost	Explored nodes	Time (s)	Cost	Cost
3	385	0.03	990	37	0.005	890	1590
6	12320	0.2	1380	2138	0.099	1380	3840
9	518077	4.584	1720	87100	1.736	1720	4610
12	12281156	138.397	1860	1871474	58.12	1820	5990

You can find the plotted results for 9 tasks on Switzerland with A* and BFS on Figures 1 and 2 in the Annexes. You can also find a comparison for different heuristics in the Annexes.

4.2 Results for three agents

You can see the results for one, two and three A* agents on Figures 3, 4, 5. We can see that the final average reward per kilometers is 275 for one agent, 180 for two agents and 150 for three agents. The increase of the number of agents is correlated with the drop of the average reward per kilometers, because the agents take each other tasks which causes useless movements and plan recomputations.

5 Conclusion

What was important in order to implement the deliberative agent was, first to chose a good state representation and to define correctly what is a transition, what it implies in the different sets, as explained in the subsection 2.3. It helped us to implement an agent which is correct, implement methods that are easy to reason about and an efficient search. In order to have an efficient search it was also important to optimize the code, use good data structures and a good heuristic for A*.

6 Annexes

6.1 Admissibility of the heuristic

Let **D** be the minimum total distance that the agent has to travel in order to deliver all the tasks that have not yet been delivered. If our heuristic returns **d1**, it is admissible because the vehicle has to travel at least **d1** to deliver the task **t1**. So our heuristic does not overestimate **D** if there was only **t1**. Adding other not yet delivered tasks does only increase **D**.
Same reasoning if our heuristic returns **d2**.
So in both cases the heuristic never overestimates **D**, it is thus admissible.

6.2 Performances of A* for different heuristics

Here we used six different heuristics. The three first heuristics gives a search slower than BFS. The heuristic H5 is the one described above. Heuristic H0 simply returns 0 in any case. The four other heuristics are simpler than H5. The results in the table are given for 9 tasks. The maximum number of tasks we can handle in less than one minute on the Switzerland map is 12.

- **H0** : returns 0.
- **H1** : Maximal distance between the distances to travel to each not already picked up tasks (= current city to pickup city).
- **H2** : Maximal distance between the distances to travel for each carried tasks (= current city to delivery city).
- **H3** : Maximal distance between the distances to travel to deliver each not already picked up tasks (= pickup city to delivery city).
- **H4** : Maximal distance between the distances to travel for each not already picked up tasks (= current city to pickup city + pickup city to delivery city).
- **H5** : Maximal distance between the distances to travel for each carried tasks (= current city to delivery city) and the distances to travel for each not already picked up tasks (= current city to pickup city + pickup city to delivery city).

Switzerland	$A^* search$		
Heuristic	Explored nodes	Time (s)	Cost
H0	535713	6.995	1720
H1	299060	5.960	1720
H2	246002	4.967	1720
H3	216658	4.265	1720
H4	92712	2.782	1720
H5	87100	1.736	1720

6.3 Supplementary tables

England	BFS			$A^* search$			naivePlan
Number of tasks	Explored nodes	Time (s)	Cost	Explored nodes	Time (s)	Cost	Cost
3	237	0.019	591.4	9	0.002	591.4	1006.5
6	9632	0.188	1471.4	1344	0.071	1471.4	3222.8
9	184104	2.359	1701.4	19226	0.433	1701.4	4601.8
12	6720144	74.69	1924.4	778920	21.6	1913.8	5807.2

6.4 Simulation results

France	BFS			*A*search*			naivePlan
Number of tasks	Explored nodes	Time (s)	Cost	Explored nodes	Time (s)	Cost	Cost
3	345	0.043	2988	37	0.007	2540	3776
6	9664	0.179	4729	716	0.037	4281	8679
9	222571	2.793	4880	18134	0.54	4791	13021
12	2164536	22.549	4880	46865	1.272	4791	17420

Netherlands	BFS			*A*search*			naivePlan
Number of tasks	Explored nodes	Time (s)	Cost	Explored nodes	Time (s)	Cost	Cost
3	605	0.032	668.6	56	0.005	633.2	779.7
6	23715	0.357	808.5	1068	0.043	804.2	1425.9
9	492497	5.043	879.1	7299	0.195	838.4	2178.4
12	10042004	114.877	879.1	80301	1.924	859.3	2986.2

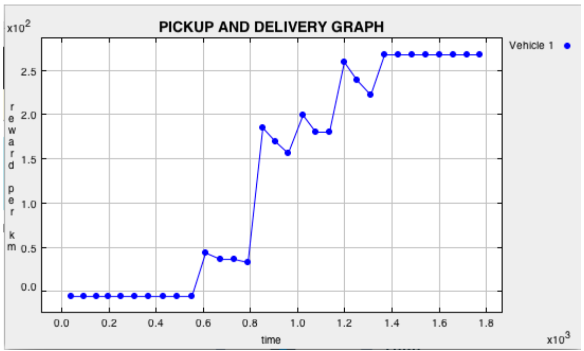

Figure 1: BFS, 9 tasks, Switzerland

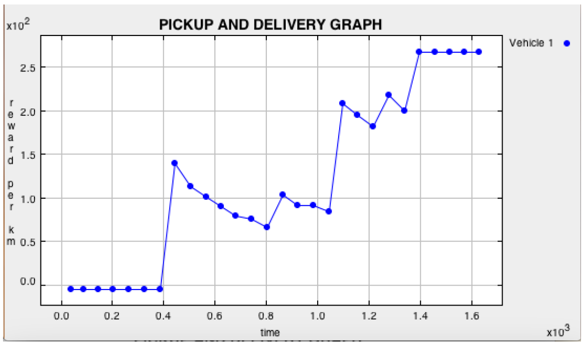

Figure 2: A*, 9 tasks, Switzerland

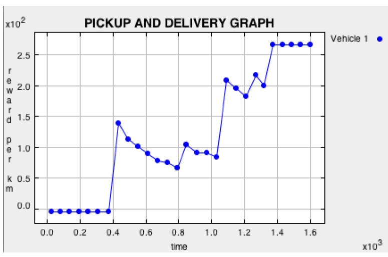

Figure 3: One A* agent

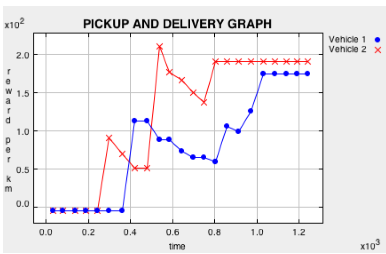

Figure 4: Two A* agents

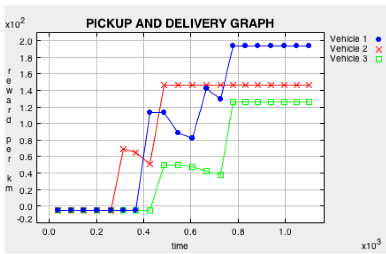

Figure 5: Three A* agents