Excercise 3
Implementing a deliberative Agent

Group №: 90 Kyle Gerard, Yann Bolliger

October 22, 2018

1 Model Description

1.1 Intermediate States

Mathematically seen our state contains the position of the agent and all the not yet delivered tasks. That means all the tasks the agent has already picked up and all the tasks to be picked up yet. This could be written as the following state space:

$$\mathcal{S} = \mathcal{C} \times \{A_{carried} : A_{carried} \subseteq \mathcal{T}\} \times \{A_{pending} : A_{pending} \subseteq \mathcal{T}, A_{pending} \cap A_{carried} = \emptyset\}$$

where $\mathcal{C}$ is the set of cities and $\mathcal{T}$ is the overall set of tasks.

1.2 Goal State

A goal state is reached when all packets are delivered. Therefore it is defined as:

$$\mathcal{S}_{goal} = \{(C, A_{carried}, A_{pending}) \in \mathcal{S} : A_{carried} = \emptyset \wedge A_{pending} = \emptyset\} \subset \mathcal{S}$$

1.3 Actions

At each state there are up to three possible actions: move, pickup, deliver. They all have some conditions on the current state:

- The agent can move only to neighbors of the current city encoded in the state. This changes the city for the next state, the rest stays the same.
- The agent can deliver a task if $\exists T \in A_{carried}$ which has to be delivered at the current city of the agent. Then this task is removed from $A_{carried}$ for the next state.
- The agent can pickup a task only if $\exists T \in A_{pending}$ which has to be picked up from the current city of the agent and if $\sum_{T \in A_{carried}} \text{weight}(T) <$ maximal capacity of the vehicle. In that case T is removed from $A_{pending}$ and added to $A_{carried}$ for the next state.

2 Implementation

2.1 BFS

We closely followed the BFS algorithm. However, instead of stopping the search at the first goal node, we traverse the entire tree, collect all goal nodes and take the plan with the minimal among them.

There are some important keypoints to speed in this. We defined a specialised node class that acts as nodes in the search tree. These nodes don't only store the formal `State` of the agent but also the plan

up to this state and the corresponding cost. This saves us from recomputing the two properties for each search node at each iteration. Another point is cylce detection. This is where we prevent reconsidering states we have already seen unless we find them with a smaller cost.

The most important speedup comes however from choosing the correct data structures. For $\mathcal{O}(1)$ lookups of states in the cycle detection and other lookups we always use a `HashMap` together with the hash function of the class `State`. For the search agenda queue we rather use a linked list in order to prevent costly array-copying. This makes our BFS reasonably fast for up to 11 tasks.

2.2 A*

As in BFS, we closely followed to algorithm given and made use of an admissible heuristic. This lets us do simpler cycle detection because we know that we find a state optimally in the first place. Again, we wrote a `Node` class that keeps track of the state, the plan, the cost and the heuristic in the search tree.

We also reused `HashMap` for cycle detection but here, we leveraged even more of Java's rich collection API. In fact, the search agenda queue is a `PriorityQueue`. This is a binary heap implementation and works with the `compareTo` method of `Node`. Therefore it automatically gives us the search node with the lowest $f()$ value in $\mathcal{O}(1)$ time (insertion is $\mathcal{O}(\log n)$). This is an immense advantage compared to sorting the entire list of search nodes in $\mathcal{O}(n \log n)$ time!

2.3 Heuristic Function

The heuristic in our A*-Algorithm must be admissible. It is defined for a state $S = (C, A_{carried}, A_{pending})$:

$$h(S) = \max(h(A_{carried}), h(A_{pending})) \tag{1}$$

$$h(A_{carried}) = \max_{T \in A_{carried}} \text{distance}(C, \text{destination}(T)) \tag{2}$$

$$h(A_{pending}) = \max_{T \in A_{pending}} \text{distance}(C, \text{origin}(T)) + \text{distance}(\text{origin}(T), \text{destination}(T)) \tag{3}$$

This is admissible because if the agent has at least one pending task, it has *at least* to go to the task's origin, pick it up and deliver it. In that case the heuristic exactly calculates the cost. If the agent has more tasks, the heuristic will underestimate the cost. The same is true if the agent only has one carried task. Therefore the heuristic is admissible.

In order to have an optimal search algorithm the heuristic also needs to be monotonic. This can be shown with the following argument. At start, the agent has no picked up tasks and $h()$ will be according to (3). This will not change until the agent picks up the task that maximises (3) because clearly $h(A_{carried}) \leq h(A_{pending})$. Therefore, at each move, the heuristic changes with *at most* the distance of that move. This means $h()$ satisfies $h(x) \leq d(x, y) + h(y)$, the condition for monotonicity. The same is true, once the agent has picked up all tasks. Hence, the heuristic given above is monotonic.

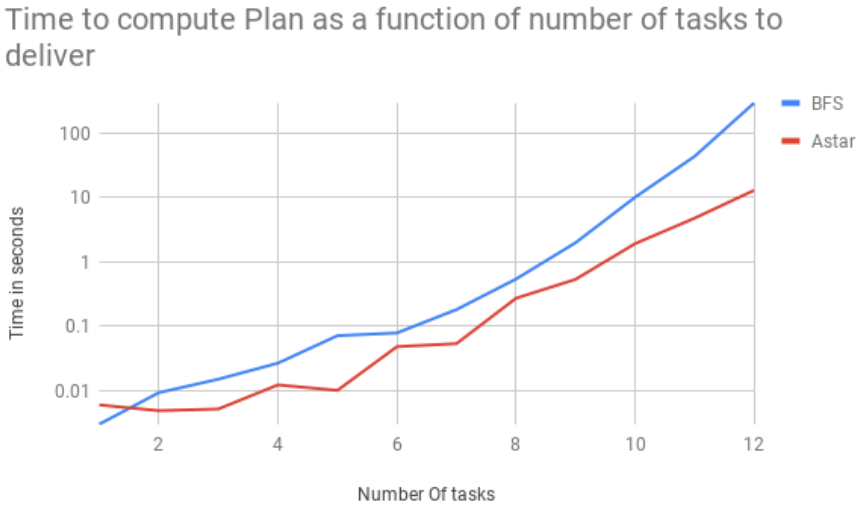
3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

The configuration file used was the one given in the handout called `deliberative.xml` with `rngSeed` left as is at 23456. Both the BFS and ASTAR algorithms were used. For each algorithm, we measured the time to compute the optimal plan as a function of the number of tasks to pickup and deliver.

3.1.2 Observations



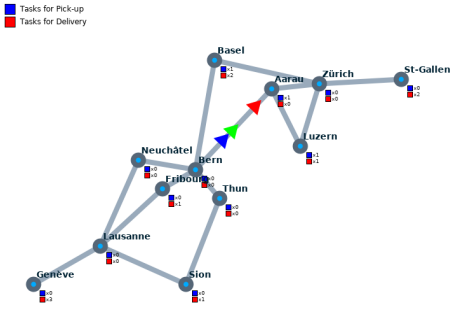
As seen in the graph above, both the ASTAR and BFS algorithms' time to compute the optimal plan grows exponentially in the number of tasks to deliver. We also remark that the ASTAR algorithm runs significantly faster than BFS. Using BFS we can build a plan with 11 tasks in 38 seconds. Using ASTAR we can build a plan with 13 tasks in 15 seconds. Therefore, even though both algorithms compute the optimal plan, ASTAR is more efficient than BFS. However as both algorithm's complexity is exponential, both algorithms are of limited use (much too slow) as the number of tasks keeps increasing.

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

Here we use the ASTAR algorithm for all agents because it is faster than BFS. Still using the given `deliberative.xml` configuration file, we examine the behavior of the agents depending on the number of agents in the environment.

3.2.2 Observations



With more than one agent in the environment we rapidly observe that the agents no longer act optimally. Indeed, their "optimal" plans computed by the ASTAR algorithm do not take into account the other agents' plans. This means that the agents will travel to a city to pickup a task even though another agent is closer to the city and thus will pickup the task first. In the figure above we see three agents going to the same pickup city even though there is only one task to be picked up. In this case the optimal plan would be to not try and pickup a task from a city when another agent will reach it earlier.