Excercise 3
Implementing a deliberative Agent

Group №: 90 Kyle Gerard, Yann Bolliger

October 18, 2018

1 Model Description

1.1 Intermediate States

Mathematically seen our state contains all positions of all the objects on the map. That means the current position of the agent, 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/tasks 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 and take the plan with the minimal cost at the end.

In order to make this memory consuming search fast, there are some important keypoints. 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 entire two properties for each search node.

Another small 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 for 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 linked list in order to prevent costly array copying. This makes our BFS reasonably fast for up to XXXX tasks.

2.2 A*

As in BFS, we closely followed to algorithm given and made use of an admissible heuristic. This lets us do simple cycle detection because we know that we find a state optimally in the first place. Again, we wrote a specialised `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` metod of `Node`. Therefore it automatically gives us the search node with the lowest $f()$ value in $\mathcal{O}(\log n)$ time. Which is an immense advantage compared to the approach where the entire list of search nodes is sorted in $\mathcal{O}(n \log n)$ time!

2.3 Heuristic Function

The heuristic in our A*-Algorithm must be admissible. It is defined as:

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

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

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

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.

–¿ monotonic, optimal?

3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

3.1.2 Observations

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

3.2.2 Observations