Exercise 3
Implementing a deliberative Agent

Group №76: Simon Honigmann, Arthur Gassner

October 22, 2018

1 Model Description

1.1 Intermediate States

For this exercise, the state representation could be simplified compared to the previous because task information is available to the agents. State was represented by the current city of a given agent, a list of all available tasks, and a list of all tasks currently being carried by the agent. A `State` class was created to conveniently encode this information.

In addition to the information included in the vehicle's state, we added some information to the nodes to reduce the complexity of planning algorithms. This information included: the total distance travelled from the root node, the node's parent, the total weight of tasks being carried (which can be computed directly from the list of carried tasks), a list of actions, *actions required*, that the agent needs to take to transition from the parent node, and the tree level on which the node exists. A `Node` class was created to encapsulate the state and supplemental information.

1.2 Goal State

The goal state for the agent is to have no remaining tasks to pick-up or deliver. This is represented in the tree as reaching a node which has no children nodes. Nodes satisfying this goal state are not unique. The optimality of a node is determined by the magnitude of the distance travelled to reach the node.

1.3 Actions

At any given state, an agent is given up to N+M possible transitions, where N represents that number tasks available for pick-up and M represents the number of tasks currently being carried which can be dropped-off. Each of these possible actions will cause the agent to transition to the city of the respective pick-up or drop-off task.

In order to reduce the amount of nodes, a given node can transition into three different kinds of new nodes :

1. a node where the vehicle has picked up a task in a City A and delivered a non-zero amount of tasks in that same City A;
2. a node where the vehicle has picked up a task in a City A and delivered nto task in the City A;
3. a node where the vehicle has picked up no task and delivered a non-zero amount of tasks to a City A.

On top of these improvements, if the vehicle *has to* pass over a city where it can deliver a task, it does automatically.

2 Implementation

Prior to implementing either search algorithm, a `Tree` class was created, which established the functions and data structures required to create a tree of `Node` states. The `Tree` can be initialized differently, to either generate and store the whole tree immediately or to generate the tree incrementally, depending on the search algorithm being implemented. The `Tree` class also has functions to remove nodes, return all nodes at a given level, check the goal condition for a node, and to generate children for a node.

2.1 BFS

The Breadth First Search (BFS) algorithm determines the absolute optimal pick-up and delivery plan for an agent by searching through all possible paths. To do this, the tree is incrementally generated. Then, starting at the tree's root node and working down level-by-level, each node is evaluated. As nodes are added to the working list of nodes, they are ordered based on their distance from the root. Nodes are evaluated against the goal condition, then removed from the working list. A list of actions required to get to the optimal node is generated by travelling up the tree until the root is found, and storing each node's *actions required* in an ArrayList. Finally, a plan is generated using the agent's starting city and the generated list of actions. This plan is returned and is implemented by the agent.

2.2 A*

The A* algorithm is used along a heuristic that estimates the distance between the current node and a goal node.

It goes deeper into the tree by trying first the nodes who, according to the heuristic, are closer to a goal node.

Once a goal node is found, the implementation of A* has been made so that it computes the actions necessary to go from the root node to this goal node, and creates a Plan object out of this list of actions.

When it comes to the implementation of A*, we made the `abstract class AstarPlan`, with the method `abstract double h(Node node)` representing the heuristic. For each A* algorithm (using each a different heuristics), we make a class that inherits from the class AstarPlan and then implement the heuristic.

2.3 Heuristic Function

We tried two heuristic functions, represented by the classes `AstarPlanWithZeroHeuristic` and `AstarPlanWithMinDistanceHeuristic`.

The class `AstarPlanWithZeroHeuristic` has a heuristic always returning zero, clearly underestimating the distance from the current node to a goal node. In that sense, the heuristic is *admissible*. The admissibility of the heuristics leads A* to always finding the optimal solution. Therefore, this heuristic is *optimal*. It is however important to note that this heuristic is very inefficient, with a large execution time (10.5 seconds for 6 tasks in Switzerland).

The class `AstarPlanWithMinDistanceHeuristic` is the second A* algorithm implemented. Its heuristic considers the minimum amount of distance it had to travel to get to its first task at the beginning. Then, it considers that to deliver each task the vehicle is currently carrying, it has to travel that same distance. It also considers that to pick-up and deliver each task it has not picked up yet, it has to travel twice that same distance. The heuristic then returns the "expected distance to travel" varying with the

Seed	Algorithm	Distance Travelled for 6 Tasks (km)	Computation Time for 6 Tasks (s)
23456	BFS	1380	8.7
	1380	10.7	
	1460	0.7	
	BFS	1510	10.7
A* Zero Heuristic	1510	63	1380
	1590	0.5	
	BFS	910	15.5
	190	16	
A* Minimum Distance	190	0.5	1460
	BFS	1266.7	11.6
	1026.7	29.9	
	1080.0	0.6	

number of carried tasks and the number of tasks left to pick up. This heuristic is not *admissible* since it does not underestimate the distance to a goal node. Therefore, it leads A* to not always finding the optimal solution (This heuristic is hereby not *optimal*). It is however important to note that this heuristic finds a solution much faster than the previous admissible heuristic (0.5 second for 6 tasks in Switzerland).

3 Results

Each search algorithms was thoroughly tested to ensure its robustness and optimality. Each search function was run on all 4 provided topologies. Additionally, different starting seeds were attempted to ensure they performed consistently across different initial conditions. Generated plans were compared between the BFS and the ZeroHeuristic to ensure they were identical, as both should return the same unique optimal solution. The following experiments present performance and efficiency metrics for each of the algorithms with several different seed values.

3.1 Experiment 1: BFS and A* Comparison

Our BFS algorithm was compared against both A* search algorithms using the Swiss topology and different random seeds.

As is apparent in the above table, BFS is the slowest algorithm

3.1.1 Setting

3.1.2 Observations

c A* can handle more tasks than BFS

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

3.2.2 Observations