Exercise 3: Implementing a deliberative Agent

Group №76: Simon Honigmann, Arthur Gassner

October 22, 2018

1 Model Description

1.1 Intermediate States

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), the list of actions, 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 can have up to N+M possible children, where N represents the 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. Two unique node types were identified : a node where the vehicle has picks up a task in a city, and a node where the vehicle goes directly to a city to drop off (a) task(s).To further reduce the number of nodes, the following conditions are checked :

1. If there are packages to deliver on the target path, all delivery actions are all added to the node.
2. If adding a task exceeds the vehicle's capacity, the node is invalid and is not added to the tree.

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 former option is not used in our final optimized versions of BFS and A*. 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 optimal pick-up and delivery plan for an agent by searching through all feasible paths. To do this, the tree is incrementally generated, starting with the root. Then, working down level-by-level, each node is evaluated. The algorithm will first check the distance to root parameter for a node and compare it to the current best distance (initialized as `double.MAX_VALUE` such that any path reaching the goal state can trump the default value). If the node's distance to root is lower than the current best, the algorithm checks if the node satisfies the goal condition of having no children and updates the best distance and nodes accordingly. If the node has children, a flag is set to continue the search of remaining nodes on the next tree level. On each level the flag is reset. The algorithm continues until no children remain, then returns the best node. 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*. Finally, a plan is generated from these actions.

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` (A* ZH) and `AstarPlanWithMinDistanceHeuristic` (A* MD). 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 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

3.1.1 Setting

Our BFS algorithm was compared against both A* search algorithms using for single agents operating in the Swiss topology and different random seeds, listed below. All agents start in Lausanne. All relevant parameters are specified in `deliberativeE1.xml`.

3.1.2 Observations

The Zero Heuristic algorithm ends up being the slowest due to the poor choice of heuristic, and potentially the increased optimization efforts for BFS. Both BFS and A* with the Zero Heuristic return the same, optimal path.

A* with the Minimum Distance heuristic performed excellently. While not always optimal, its plans were within 4% of the distance travelled of the optimal solution with a considerable 20x decrease in computation speed compared to BFS. A* with the Minimum Distance heuristic also trumped the other algorithms in its capacity to handle tasks. In under 1 minute, BFS could handle 6 tasks reliably, A* with Zero heuristic could handle 10 tasks, and A* with Minimum Distance heuristic could handle about 53 tasks.

Table 1: Comparison of BFS, A* ZH, and A* MD Algorithm Performance for the Delivery of 6 Tasks, Starting from Lausanne

Seed	Algorithm	Distance Travelled (km)	Computation Time (s)
23456	BFS	1380	8.7
	A* ZH	1380	10.7
	A* MD	1460	0.7
23457	BFS	1510	10.7
	A* ZH	1510	63
	A* MD	1590	0.5
23458	BFS	910	15.5
	A* ZH	910	16
	A* MD	910	0.5

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

This experiment is run using configuration as specified in the `deliberativeE2.xml` file. It is run three times : with 1 vehicle, 2 vehicles and 3 vehicles, for both BFS and A* MD.

3.2.2 Observations

As we can see in Tables 2, despite doubling (respectively tripling) the amount of vehicles in the country, the time it takes for the tasks to all be delivered decreases only slightly. This decrease has a huge cost, since the two (respectively three) cars' total distance traveled increases much faster. This is due to the car not cooperating. If they were able to coordinate themselves, the total distance traveled would decreased as well as the time to deliver all tasks.

Table 2: Table comparing the results of one, two and three BFS and A* (Minimum Distance Heuristic) agents running in parallel in Switzerland with 6 tasks

Algorithm (# Agents)	Total Delivery Time (s)	Total Distance Travelled (km)	Algorithm (# Agents)	Total Delivery Time (s)	Total Distance Travelled (km)
BFS (1)	27.1	1380	A* MD (1)	20	1460
BFS (2)	24.1	2410	A* MD (2)	16	2410
BFS (3)	23.4	3120	A* MD (3)	15.4	2830