Intelligent Agents - Deliberative Agent

Ana Manasovska - Valentin Rutz

October 29, 2015

1 Introduction

In this assignment, we make ready to tackle the Pickup and Delivery Problem yet again but with some differences with the previous assignment.
In this version of the PDP, once an agent enters the world, it is fixed. Meaning that in contrast to the reactive agent problem, there are no random events involved. The packages are generated before the agent enters the world (and none are generated afterwards), and at the same time, the agent knows what packages it is supposed to pickup and deliver. This implies that we can plan ahead our optimal route using different algorithms like BFS or A*.

2 Modeling the world

In order to model using programming the world described in the introduction, we decided to introduce several Java classes:

- Path: The Path class is the list of actions done to go from the start state (which will be made explicit in the second point) to the current state along with the cost of such a path. So the definition of Path is: Path(List¡Action¿ actions, double cost)
- State: The abstract State class is a model for concrete states (see below for concrete state description). It contains the current city, the set of remaining tasks of the agent, the set of tasks carried by the agent and the remaining capacity (it could have been computed using the carried tasks set but it is simpler that way and less costly). It has methods to check if it is a final State, check if it can pickup a task, check if it can deliver a task and the usual hashCode and equals methods to use it in a hash-based data structure.
  A State is a final state if and only if we have delivered all carried tasks and if we have picked up and delivered all remaining tasks.
- BFSState: The concrete state for BFS search. It does not contain any more attributes than its abstract parent but it contains behaviour information. It provides three methods for transitions:
  1. pickup(Task t, Path p) which returns the next State after picking up Task t. It adds t in carriedTasks, removes it from remainingTasks and lowers the capacity of the truck by the weight of t and it adds the new action in the Path p so that we can know how to get to that State.
  2. deliver(Task t, Path p) which returns the next State after delivering Task t. It removes t from carried tasks and adds back the weight of t in the capacity of the truck and adds the the delivering action into the Path p.

   3. move(City neighbor, Path p) which returns the next State after moving to neighbor. We simply add the action to the Path p that we move to that neighbor and we update the current city to the neighbor.

- AStarState: The concrete State for A* search. It contains the heuristic (h-value explained in section 3) and the current best F value for that state, along with the attributes inherited from State and transition information such as:

   1. pickup(Task t, Path p) which returns the next State after moving to the pickup city of Task t and picking up t. It essentially does the same as pickup in BFSState but it will also set the h-value of the newly created state accordingly to the heuristic.
   2. deliver(Task t, Path p) which returns the next State after moving to the delivery city of Task t and delivering t. It essentially does the same as deliver in BFSState but it will also set the h-value of the newly created state accordingly to the heuristic.

3 A* or the guy nobody invited to the party

Our heuristic for A* in state S is set to the cost of the most expensive task in that state.
In order to arrive in a final state from S, we have to deliver all of the tasks we currently have in S. The actual cost of delivering of the most expensive task will be at least equal to our heuristic function (if we choose the direct path for delivering the task) or larger (if we choose an indirect path in order to pickup/deliver other tasks).
This way the heuristic is always smaller than or equal to the actual cost from state S to a final state. That means we don't overestimate the cost and that our heuristic function is admissible which is important in order to find the shortest path to be guaranteed.
In the case of only one task left the heuristic is equal to the actual cost. If we have multiple tasks and their paths in general don't overlap then the actual cost is much higher than our heuristic. In this case, we would still get the optimal solution, but we could say the heuristic is not good enough to get the solution faster. The lower the heuristic is, the more nodes A* expands, making it slower. If the heuristic is zero, A* becomes Dijkstra and visits every state.

4 Performance comparison between BFS and A*

In our case A* works up until ten tasks (for less than a minute), and BFS works for eleven. A possible reason for that is that our heuristic is too low and A* explores a lot of states. In addition, A* works with data structure Priority Queue that has operations of O(logn) and BFS takes the advantage of a simpler queue with operations of O(1).
For example, for ten tasks (on the map of Switzerland), A* finds better solution, a cheaper one, than BFS. The total travelled distance of A* is 1820km, and for BFS is 1850km. When we have multiple agents, the sum of distances that all of the agents have travelled is larger than the total distance travelled by one single agent.
For multiple agents, the sum of distances of the both agents in case of A* is smaller than the one in the case of BFS. Regarding the comparison of the respected algoritms, in the case of multiple agents A* performs better. The The agents that use A* travel less kilometers together (1630km and 1110 km), while the sum of distances travel by the two agents that use BFS is larger (1650km and 1660km).

5 Conclusion

As expected, the results obtained from A* are better. We always get the optimal solution using an admissible heuristic function.

Figure 1: Our reaction to the performance results between BFS and A*