ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Computer Science Master, Course: Intelligent Agents
Students: Ivan Gavrilović, Boris Perović

Assignment 3

Programming: Implementing a deliberative Agent

A brief solution description:

The solution was developed in IntelliJ IDEA 12 IDE, using JDK 7 update 40 and the provided LogistPlatform framework. Git was used as collaboration tool and a version control system. Provided template (*deliberative.zip*) was used as a base for the project and agent deliberative-random given in it was used for testing purposes in multiple agent scenarios (e.g. our implemented agent + 1 random agents).

For this assignment, the most important thing was to define very precisely all the theoretical aspects / model of the world. These include: suitable representation of the states in which an agent can find itself, transitions between states and final (goal) states.

State representation:

While analysing the problem in order to find suitable agent state representation we needed to be precise enough so that the all the relevant states can be well defined, while trying to carefully generalize the states in order to reduce their global number, speeding up the calculation of the plan. Therefore we define a state as the current city a vehicle is in, a set of tasks that it is carrying, and a set of tasks that have not yet been picked up and a plan that stores all the actions an agent took from the starting state up to the current state. This is implemented in the class `State`, which contains fields that represent mentioned properties: `City currentCity`, `TaskSet inProgress`, `TaskSet toPickUp` and `Plan plan`. Cost of a state if calculated as a total distance travelled (can be obtained from `plan` field, which is correct and saves us some multiplications, since cost/km for each vehicle is constant.

Transitions:

Transitions are supposed to lead an agent from one state to another. This is implemented in method `List<State> getChildren()` that generates states to which an agent can transit from the current state (children states) and will be explained here in details.

Transition from one state to next one should provide us with either delivering a task or picking up a new task. Transitions can be more specific (e.g. single move to a neighbour), but this would provide us with unnecessary increase of number of states, slowing down plan calculation. For each of the tasks T(I, J) (I being pickup and J delivery city) that we should pick up, we do the following. We move from current city (A) to the pickup city (I) and consider this to be the valid child state if we do not encounter some task to be picked up on our route from A to I. If we encounter other task to be picked up, then we can skip generating this state, as it will be generated later (one of the child states of the encountered state will be not picking up any of the tasks in that city and proceeding towards the original pickup city I). If we encounter a city in which we should deliver some of the tasks we carry (contained in the set `inProgress`), we always deliver them, since it makes no sense carrying them anymore.

Movement to other city is followed by adding a move action to the plan; delivery of the package is followed by addition of delivery action to the plan; at the end we add the pickup action for the task. It is important to notice that we perform a pickup of a package only if the maximum capacity of the vehicle will not be exceeded.

Also, for each of the tasks T(I, J) that we are carrying, and that have not been delivered in some of the movements to the valid pickup destinations, we generate the sequence of move actions from I to the city J. The same principle stands here as above. If we encounter the city on the route from current city A to city J in which we have some other task to deliver, we will not generate the child state for J delivery as it will be generated later. Movement across the route is followed by addition of move actions to the plan and delivery action when we reach J.

It is of great importance to note that while generating child states cycles may be created. We consider that a cycle exists, when an agent comes to the city it has already been to, without making any progress towards the goal, i.e. not making any pickup or delivery actions (goal/final states will be explained later). We must remove cycles since other than being useless to plan-making, they would lead to an extreme loss in performance. Cycle detection is implemented using the `State parent` field in the `State` class. Each child state that we create checks whether there is an equal state in the hierarchy of its parents. If some ancestor of the potential child state is equal to it, the child state is not generated as it would form a cycle.

Final (goal) states:

Goal of the pickup and delivery problem solution is to process (deliver) all of the tasks. Therefore we set the delivery of all tasks as a goal. This is represented with a final state that can have arbitrary city as the current city (more precisely one of the delivery cities) and an empty sets of tasks that are being carried and an empty set of tasks that need to be picked up.

Heuristics:

When using A* algorithm for search of the state-space, we use a heuristic. Heuristic is an estimate of the remaining cost of all of the pickup/delivery tasks. Combined with current cost of the road travelled, it gives us an estimate of total cost of the solution (considered final state). It is very important to note that if we want to preserve A* algorithm's optimality property (guarantee that it returns an optimal solution), we must not use a heuristic that overestimates the remaining cost. Reason for this can easily be shown in a counter-example: if we have a heuristic that overestimates, we could come to a solution in final state Y that costs y, but the actual path of the optimal solution (state X and potentially further) is not expanded since it for example has a cumulative cost $x+h(X) > y$ (heuristic overestimated).

We chose two heuristics, both preserving the optimality property of A* algorithm.

First one is "furthest child" which returns the cost of travel to the furthest child as an estimate of the cost of remaining tasks. This is a rather coarse heuristic, but it gives us decent performance (9 tasks under 1 minute), which can be observed in results section.

Second one is an adaptation of a well-known heuristics for Travelling Salesman Problem – Minimum Spanning Tree (MST) heuristic. This heuristic uses adapted Prim's algorithm to calculate MST of remaining cities of interest (delivery and pickup cities) and returns it size as an estimate of the cost of remaining tasks. Proof of correctness of this heuristic is simple: minimum spanning tree

cannot be traversed without backtracking, therefore its size does not overestimate the actual remaining cost.

Results:

One agent:

The following table represents measurements obtained using different algorithms for different tasks set sizes:

Algorithm ↓	Tasks number →	5	6	7	8	9	10	11
A* hMST	Time [ms]	33	112	208	888	4117	23489	80013
	Reward [cost/km]	171	185	200	228	267	290	320
A* hFurthest	Time [ms]	86	180	685	10976	53391	-	-
	Reward [cost/km]	171	185	200	228	267	-	-
BFS	Time [ms]	103	221	530	2970	8395	timeout	-
	Reward [cost/km]	111	116	132	120	159	-	-
Naive	Time [ms]	0	0	0	0	0	0	0
	Reward [cost/km]	64	63	73	85	97	99	101

Table 1 - performance comparison

It is clearly observable that A* algorithm with MST heuristic gives us the best time performance amongst non-naive search algorithm implementations (naive search is constant in time, but very suboptimal). Although BFS returns suboptimal solution – the first final state it finds, A* algorithm with MST heuristic is still faster. This due to a good heuristic function, which makes relatively precise estimations early and leads the search in right direction. The maximum number of tasks for which we can build a plan in less than 1 minute is 10, but we are able to process 11 tasks under 1.5 minute.

Multi-agent:

One of the main drawbacks of deliberative agents can be observed in multi-agent systems. Each of the agents calculates the optimal plan on its own, which will often lead to conflicts (e.g. one agent comes to a city to pick up a task, but the task has already been picked up). In this case an agent may have travelled unnecessarily, which leads to a lesser reward and also the agent needs to recalculate the optimal plan, which takes additional time. Graphs of different multi-agent execution scenarios are shown below:

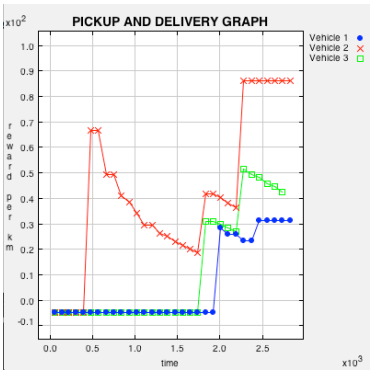

Figure 3 - 3 BFS agents

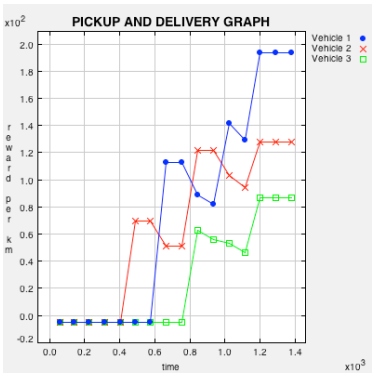

Figure 2 - 3 A, MST heuristic*

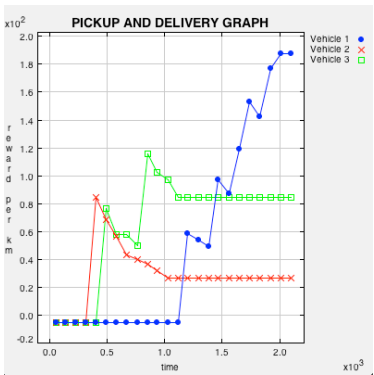

Figure 1 - 1 A, MST heuristic + 2 naive*