CS-430 : Problem Definition for the Reactive Agent Assignment

Titouan Renard, Christophe Marciot

October 18, 2020

1 Problem definition

We are trying to implement an agent that functions according to the following basic procedure.

Algorithm 1: Basic Agent Logic

```
while goal not reached do
    if current plan not applicable anymore then
        Compute optimal plan
    end
    else
        Execute next action in the plan
    end
end
```

In this case optimality is defined as *minimum cost function*. And the cost function is computed for any road taken as such :

$$c(\text{road}) = l_{road} \cdot c_{kil}$$

Where the value l_{road} is the length of a given road and the value c_{kil} is the cost per kilometer for a given agent.

Computing an optimal plan can be though of as finding a plan p_{opt} such that:

$$p_{opt} = argmin_p\{cost(p) \mid \text{ p reaches a goal}\}$$

Where the cost of a plan is simply given by :

$$cost(p) = \sum_{r \in \text{ roads in the plan}} cost(r)$$

The *goal* of our agent is to deliver all tasks on the map, all of these tasks as well as the length of every road and the cost per kilometer are known prior to planning.

Our agent has the ability to perform multiple tasks at the same time but has a maximum weight that it can carry, it has to verify :

$$\sum \text{ carried tasks } \leq \text{ agent capacity}$$

We will use *BFS* and *ASTAR* to find an optimal solution.

2 States, Transitions and Goals

Since we are working in a deliberative agent paradigm, we will not generate every possible state prior to planning, instead we will define states as a set of variables that are defined by our algorithm as it searches the solution space.

2.1 States

The states the agent can be in are defined by the following variables, for a given state **state**:

1. **state.position** : the city a given agent is in
2. **state.ctasks** : the list of all tasks carried by the agent in a given state
3. **state.cweight** : the total weight carried by the agent in a given state
4. **state.free_tasks** : the complete list of all undelivered, not yet picked-up tasks at a given state

2.2 Transitions

Our agent can perform three basic actions

1. It can drive from one city to another
2. It can decide to pickup or not to pickup a task when it is in a given city
3. It can decide to deliver a task when it is in the city it should be delivered to

We therefore construct our state representation around those three fundamental transitions :

1. **drive(start, destination)** : the movement action between two *adjacent* cities.

   Cost function defined as : $cost_{drive}(road_{start,destination}) = l_{road} \cdot c_{kil}$

   Possible $\iff$ the agent is in the city *start* and the city *destination* is adjacent to *start*.

   Consequences : agent is now in the city *destination*

2. **pickup(task)** : the action of adding a task to be carried by the agent

   Cost function defined as : $cost_{pickup} = 0$

   Possible $\iff$ the agent is in the city where *task* is and if

   $$state.cweight + \text{ w}_{task} \leq \text{ capacity}$$

   Consequences : agent appends *task* to *state.ctasks* and adds the weight w_{task} of the new task to *state.cweight*, *task* is removed from *state.free_tasks*

3. **deliver(task)** : the action of delivering a task carried by the agent to it's destination

   Cost function defined as : $cost_{deliver} = 0$

   Possible $\iff$ the agent is in the city where *task* should be delivered

   Consequences : agent removes *task* from *state.ctasks* and substracts the weight w_{task} from the task to *state.cweight*

2.3 Goal

The *goal* state is defined as any state where :

$$\text{state.free\_tasks} = \emptyset$$

3 Implementation of the re-planning algorithm

Problem: if every agent has it's own *state.free_ tasks* data-structure, it will not be able to cope with the fact that some other agent may take a task and remove it from the actual list of free tasks.
Solution: check at each state wether or not the free tasks have been modified, if it is the case, replan accordingly. This is essentially what the *Basic agent logic* algorithm suggests but this is a somewhat more detailed implementation:

Algorithm 2: Replanning Implementation

```
Initialization : p_opt ← compute_optimal_plan(global.free_ tasks)
while goal not reached do
    if state.free_task ≠ global.free_tasks then
        p_opt ← compute_optimal_plan(global.free_ tasks)
    end
    execute next action in p_opt
end
```

4 Tree Search algorithms

4.1 Breadth First Search

From the course :

Algorithm 3: Breadth-first(InitState)

```
Q ← (InitState)
repeat
    n ← first(Q), Q ← rest(Q)
    if n is a goal state then
        return Backtrack(n)
    end
    S ← succ(n)
    Q ← append(Q,S)
until Q = ∅;
return FAIL
```

Since in our case we look for a *path* in the state graph, not just the goal *state*, and in order to efficiently find a path (and to keep a smaller memory footprint of the algorithm) we give each state node a *parent* attribute, this enables us to implement the following algorithm.

Algorithm 4: Backtrack(goal)

```
let plan be a stack
n ← goal
repeat
    push n into plan n ← n.parent()
until n ≠ start state;
return plan
```

NOTE: we would need a nice implementation of a dynamic queue in order for BFS to work well.

4.2 ASTAR

From the course :

Algorithm 5: ASTAR(InitialNode)

```
Q ← (InitialNode) repeat
    n → first(Q), Q ← rest(Q)
    if n is a goal state then
        return n
    end
    if n ∉ C has a lower cost than its copy in C then
        C ← append(n,C)
        S ← succ(n)
        S ← sort(S,f)
        Q ← merge(Q,S,f) (Q is ordered by f(n) = g(n) + h(n))
    end
until Q is empty;
return FAIL
```

4.2.1 ASTAR Heuristics Definition

To be defined

4.3 Proving Optimality of Solution

To be done