THE DELIVERY MAN LAB ASSIGNMENT

Group 6

Choi, Inho

Oh, Jaehee

Osipov, George

Takhchi, Medhi

Part 1 A* Algorithm – Theoretical Overview

A theoretical overview of the A* algorithm, including an explanation of optimality conditions.

A* Theoretical Overview

A* AND OPTIMALITY

A* algorithm:

A* algorithm is the most well-known way of best-first search. This means that the algorithm finds the way that minimizes total cost by first considering the routes that appear to be optimal. The f(n), estimated cost for node n, is calculated like this :

$$f(n) = g(n) + h(n)$$

g(n) : cost of the path to reach the node from starting node

h(n) : the estimated cost of the cheapest path from the node to the goal

In other words, f(n) is the estimated cost of the cheapest solution containing node n. It is reasonable that we examine the node with lowest value of f(n) to find the cheapest solution. Actually, under certain conditions for the heuristic function h(n), this strategy always leads to an optimal solution. Now, let's talk about these conditions for heuristic function, h(n).

<u>Conditions for optimality. (Admissibility and Consistency)</u>

Firstly, h(n) should be an admissible heuristic. It means that the heuristic function never overestimates the cost from specific node to the goal. If h(n) is admissible, the immediate estimation [f(n) = g(n) + h(n)] also never overestimates the true cost of a solution through n because g(n) is the actual cost to reach n by current path.

Secondly, h(n) should be consistent(this property is sometimes referred to as monotonicity). It is a stronger condition and is only required for applications of A* to graph search. For every node *n* and every successor *n'* of n acquired by any action *a*, *h(n)* is not greater than the step cost from *n* to *n'* via action *a* plus the estimated cost of reaching goal from *n'*.

$$h(n) \leq c(n, a, n') + h(n')$$

Every consistent heuristic is also admissible. Therefore, consistency is a stronger condition than admissibility. And most of admissible heuristics are also admissible.

Optimatlity of A*:

As we mentioned above, the consistency condition is applied on graph search only. So, the tree-search version of A* is optimal if h(n) is admissible, while the graph-search version of A* is optimal if h(n) is consistent. From now, we'll prove the statement : if h(n) is consistent, then graph-search version of A* algorithm is optimal.

Firstly, we can establish this : if h(n) is consistent, then the values of f(n) along any path are nondecreasing. We can prove it using definition of consistency. Suppose n' is a successor of n, then g(n') = g(n) + c(n, a, n') for some action a. And we can get this inequality :

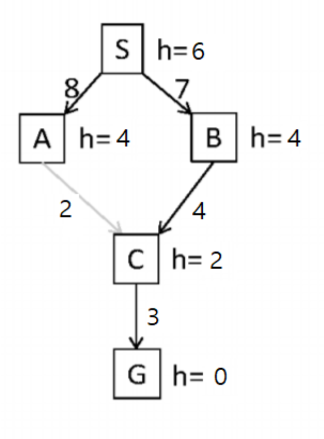
$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

Secondly, we have to prove that whenever A* selects a node for expansion, the optimal path from start to that node has been found. If not, there would have been another node *n'* on a path to *n* in the frontier and it would be selected first, because *f* is nondecreasing along any path.
By those two preceding consideration, we can claim that the sequence of nodes expanded by A* algorithm on graph-search is in nondecreasing order. Hence, the goal node picked first as expansion node must contain an optimal solution.

The A* algorithm is optimally efficient for any given consistent heuristic. There is no other optimal algorithm that is guaranteed to expand fewer nodes than A* because any algorithm that does not expand all nodes with f(n) less than the cost of optimal solution path take the risk of missing the optimal solution.

[Example]

- The heuristic used in this example is the number of steps from the goal, multiplied by the smallest edge cost.
- This heuristic is optimistic and monotonic.
- This example is a case of Graph Search.



Frontier	**Visited**
S(6)	
A(12), B(11)	S(6)
A(12), C(13)	S(6), B(11)
C(12)	S(6), B(11), A(12)
// Node C in Frontier is replaced by smaller estimated value.	
G(13)	S(6), B(11), A(12), C(12)

The cost of optimal solution is 13.

[References : *Russell, Stuart Jonathan; Norvig, Peter* Artificial intelligence : a modern approach 3. ed.: Boston: Pearson Education, cop. 2010]

THE DELIVERY MAN LAB ASSIGNMENT

Part 2 Our A* Implementation

Detailed explanation of implemented functions and A* algorithm using graph search.

Methods used in our code

OUR A* IMPLEMENTATION

General Overview:

A* algorithm performs best-first search expanding nodes according to the estimated cost of the cheapest path to the goal through current node n. We use priority queue as the underlying data structure for our frontier and use **priorFn** to calculate priority: f(n) = g(n) + h(n).

Our search node has the following structure:

x,y - coordinates of the node
cost - actual cost of path from start to current node (g(n))
estimate - heuristic estimate of the cost of path from current node to goal (h(n))

We use a data structure - **minCostGrid** - to keep track of the minimal cost of reaching each node in the grid. It allows us to avoid revisiting nodes - every time a node from the frontier is chosen to be expanded, we check whether we have already explored a cheaper path to it and do not proceed if we have. As well as that, minCostGrid allows us to keep the frontier thin: before adding a node to the frontier we check whether we have already checked a cheaper way through it.

The heuristic we are using is Manhattan distance from a node to the goal. This heuristic is optimistic - on a rectangular grid we have at least as many actions left before reaching the goal as the Manhattan distance, and each action cost at least 1. Manhattan distance is also consistent - cost of any transition c(n, n') is at least 1 and any action changes Manhattan distance with exactly 1: $|h(n) - h(n')| = 1 \leq c(n,n') \rightarrow h(n) \leq h(n') + c(n,n')$

Detailed Explanation of Implemented Functions:

There are several functions used to implement A*:

Function "isPackagePresent" checks whether a package is present at position (x, y).

Function "isGoalState" works differently in case we are to pick a package or to deliver one. If a package is not loaded, we check whether we've reached the location of the next package (according to some order); if a package is loaded, we compare destination coordinates to current position. If it is the same, current state is a goal state.

Function "getSuccessor" returns successor state position. For example, if action is 6, go right means increasing *dx* by 1. If action is 2, go down means decreasing *dy* by 1.

Function "getLegalActions" returns a list of legal actions, and we need it because edges of the environment blocks car's movement. For example, if x coordinate is 1, going left is an illegal action. If y coordinate is less than height, going up is legal action. We do not regard stop here (this is explained in the write-up about our non-A* strategies).

Function "**heuristic**": we use Manhattan distance - standard heuristic for a square grid. We calculate *dx* and *dy* distance between current state and the next package or current state and the delivery position depending on what our current goal is.

We are using graph-search and not revisiting nodes, which requires monotonicity and optimism.

h is optimistic because Manhattan distance is equal to the minimal number of actions we would have to take to reach the goal from current state, and the cost of each action is at least 1.

Also, h is monotonic because c(n, n') is always greater or equal to 1 and any move is only reducing or increasing Manhattan distance by 1. So, h(n) is less than or equal to h(n') + c(n, n') for all nodes n and all successors n' of n.

Function "**getTransitionCost**" returns cost of performing action at current state. For example, if action is going down, transition cost is hroads[x, y-1].

Function "**performAction**" returns resulting state after performing given action in current state. It determines successor position and transition cost, heuristic of successor. Creates successor state and calculates cumulative cost equal to current state's cost + transition cost between current and successor state, and 1, for a new turn.

Now get back to our main function.

At first, we initialize package order. "minCostGrid" is a matrix which lets us keep track of minimal cost (so far) of getting to any node. At first, all costs are initialized to infinity except start state - cost of reaching it is 0.

First, we create a node for start state. x is car's x coordinate, y is car's y coordinate, estimate and cost are both 0.

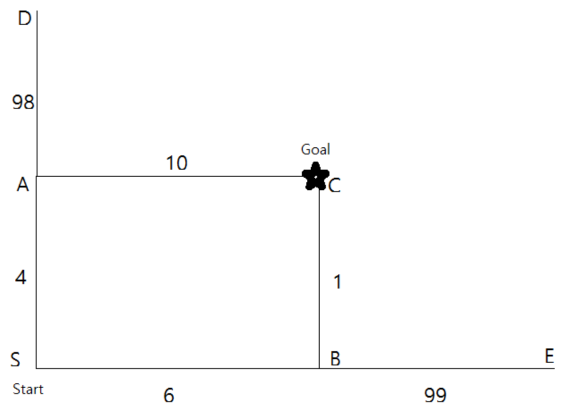
Function "**estimateCost**" returns the estimated cost of reaching the goal if we perform given action from the given state. First, we extract environment dimensions from vroads and hroads. We use priority queue to find least cost nodes and to maintain frontier. We perform action and push start node to frontier. Now, we continue while priority queue is not empty. Popping current state first. If minimal cost of getting into the node is greater or equal to current state's cost, we add successors of current state to frontier. Perform legal action with successor state. Add successor state to frontier only if it was reached with a better cost than before. If goal state is reached, return the cost it took to reach the goal.

Function "**priorFn**" is priority function for a search node. It is defined as cost of reaching this node + heuristically estimated cost of path to goal. In priority queue, it pops max value first, but in this algorithm the least one is best one, so we use minus here.

In A* algorithm, set initial action as stop and cost as infinity. Estimate cost of getting into the goal and set as current cost. If current cost is less than best cost so far, update cost and action. Also, car's next move is determined.

Example of storing the nodes that we have visited.

Value in brackets correspond to the value of f(n) = g(n) + h(n) at this node.
Heuristic is equal to the number of steps to goal.



Frontier: <u>S(0+2)</u>, store [S(0)]

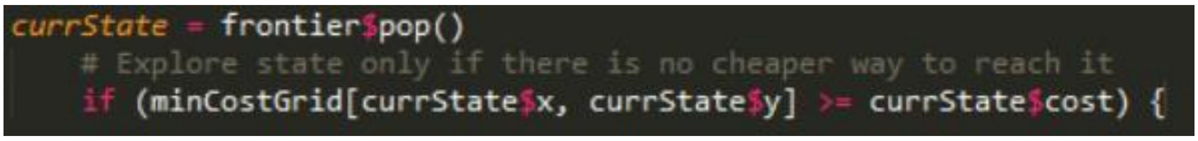
Frontier: <u>SA(4+1)</u>, SB(6+1), store [S(0), A(4)]

Frontier: SAC(14+0), <u>SB(6+1)</u>, SAD(102+2), store [S(0), A(4), B(6)]
We avoided adding SBS to the frontier as its cost was 12 and we can already reach S with cost 0.

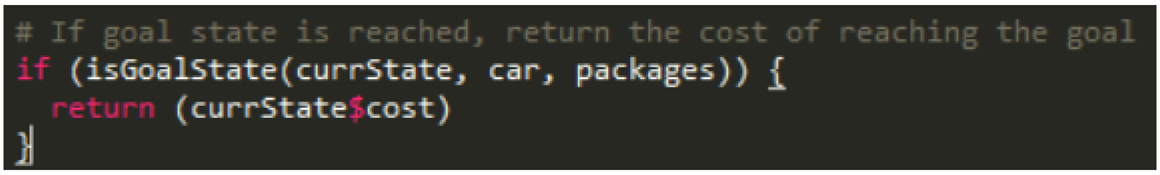
Frontier: SAC(14+0), <u>SBC(7+0)</u>, SAD(102+2), SBE(105+2), store [S(0), A(4), B(6), C(7)], goal reached.

A* Using Graph Search:

The biggest difference between Graph search and Tree search is whether it stores the nodes that we have visited. In Graph search, we store visited nodes and frontier nodes both. And then, when we expand a specific node, we add the neighbors which are not already in the visited set or frontier set to the frontier. Therefore, the Graph search doesn't revisit nodes. It is effective to reduce time complexity because it visits all the nodes only once at most and there is no meaningless repetition. On the other hand, the Tree search doesn't store visited nodes. It only stores frontier nodes. If we use DFS algorithm, the space complexity of it is O(b*d) when b is breadth of each node and d is depth of goal. So, it offers enormous memory advantages over Graph search. However, it revisits nodes and is not guaranteed to find the goal node if the search space contains directed loops.
In our A* implementation, we used a graph search. The reason for our choice is that graph search has advantage in time complexity because it avoids repeated states. Usually, the graph search keeps all visited nodes in memory and check whether the node was visited or not to do so. However, we used slightly different way. We remembered minimal total estimated cost(denoted f(n)) in grid format(minCostGrid). At each step, we pop a node from frontier(Priority queue prioritised by f(n)) and check whether the popped state's cost is less or equal than 'minCostGrid' value of that state. If it's not, the popped state would be a revisited state because revisited state should have a higher estimated cost than the before cost in the environment of this project.

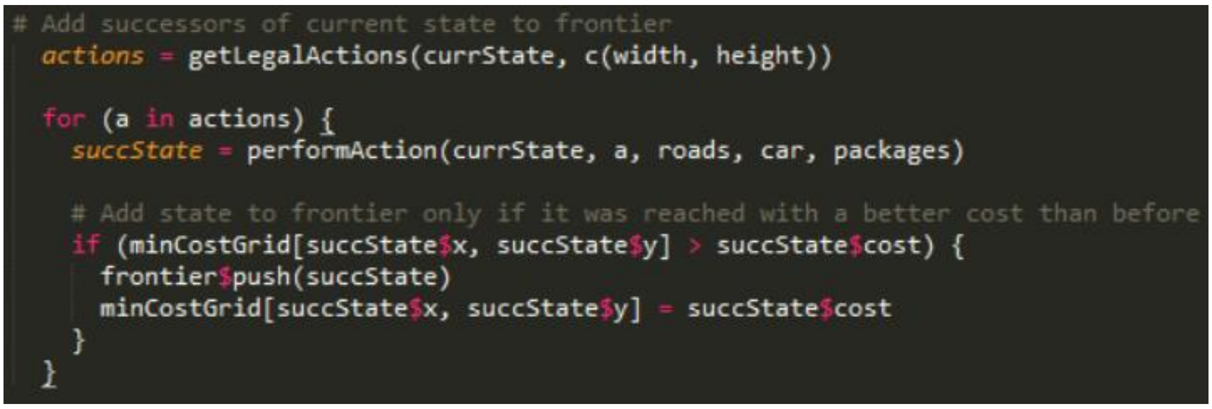

```
currState = frontier$pop()
    # Explore state only if there is no cheaper way to reach it
    if (minCostGrid[currState$x, currState$y] >= currState$cost) {
```

If the cost of popped state(currState) is less or equal than 'minCostGrid' value of that state, we pick it up as a node to expand. Now, there are two cases for this picked node: first - if it is a goal node; in this case, we return the cost of this state and compare this value to other action's values to find the best action in that state.



```
# If goal state is reached, return the cost of reaching the goal
if (isGoalState(currState, car, packages)) {
  return (currState$cost)
}
```

In the second case it is not a goal state. In this case, we expand this node by all legal actions. The legal actions mean that the car does not get off the boundary of board by those actions. Then, we get the neighbors of that node. In A* algorithm, neighbors are added to the frontier only if the value of estimated cost is less than the cheapest route to this node so far.



```
# Add successors of current state to frontier
  actions = getLegalActions(currState, c(width, height))

  for (a in actions) {
    succState = performAction(currState, a, roads, car, packages)

    # Add state to frontier only if it was reached with a better cost than before
    if (minCostGrid[succState$x, succState$y] > succState$cost) {
      frontier$push(succState)
      minCostGrid[succState$x, succState$y] = succState$cost
    }
  }
```

THE DELIVERY MAN LAB ASSIGNMENT

Part 3 Non - A* Strategies

Using A* algorithm which is widely used in pathfinding and graph traversal has shown some remarkable results, however its performance and accuracy could be improved by using some other strategies.

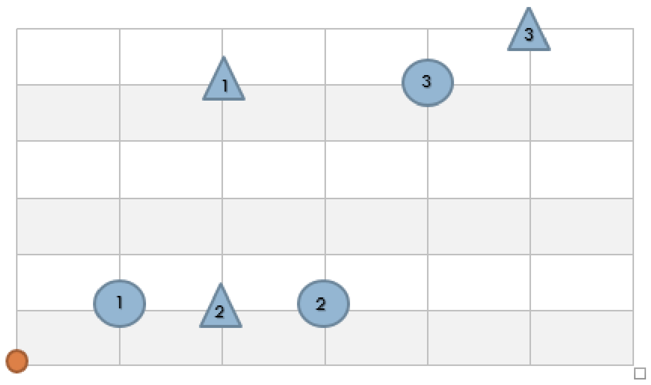
Methods used in our code

NON A* STRATEGIES

Ordering:

This method based on computing to choose first the best order of delivery can improve the performance of our code. It's similar to Dijkstra algorithm in a way that it doesn't use an informative heuristic function: h(v)=0 for each v.

Let's see an example to compare the efficiency of these two strategies:



Example uses a uniform cost grid. In this picture, circles denote pickup locations for the packages and triangles denote delivery destinations.

If we use a greedy ordering approach in A* (pick up closest package – deliver), we end up with the following ordering:

Start -> 1 -> 3 -> 2

2 + 5 + 2 + 2 + 7 + 1 = 19

But if we plan ahead and determine best order of delivery we had already a better solution which is:

Start -> 2 -> 1 -> 3

4 + 1 + 1 + 5 + 2 + 2 = 15

Thus, ordering method shows better results (for example, in the case above 15 < 19) and helped us to make less turns in the delivery game than greedy A* algorithm did.

No "stop" alternative:

In DeliveryMan "5" is used to stop. This is counted as a turn. We never use stop action because it is never a rational choice. More formally: let's compare expected cost of making an action to the cost of stopping for a turn and then performing this action. If the cost of the action is **w** which is greater than 1, expected cost of performing the action directly is **w**, the expected cost of stopping and making a move is: (0.9)***w** + (0.05)*(**w**-1) + (0.05)*(**w**+1) = **w** (according to transition probabilities in the environment). In case **w**=1 expected cost is even larger than 1.

Other methods

NON A* STRATEGIES

Djikstra algorithm

A* is basically an informed variation of Dijkstra. A* is considered a "best first search" because it greedily chooses which vertex to explore next, according to the value of f(v) [f(v) = h(v) + g(v)] - where h is the heuristic and g is the cost so far. Note that if you use a non-informed heuristic function: h(v) = 0 for each v: you get that A* chooses which vertex to develop next according to the "so far cost" (g(v)) only, same as Dijkstra's algorithm - so if h(v) = 0, A* defaults to Dijkstra's Algorithm.

Held–Karp algorithm

The Held–Karp algorithm is a dynamic programming algorithm proposed in 1962 to solve the Traveling Salesman Problem (TSP). TSP is an extension of the Hamiltonian circuit problem. The problem can be described as: find a tour of N cities in a country (assuming all cities to be visited are reachable), the tour should (a) visit every city just once, (b) return to the starting point and (c) be of minimum distance. Every subpath of a path of minimum distance is itself of minimum distance. It computes the solutions of all subproblems starting with the smallest. Whenever computing a solution requires solutions for smaller problems using the above recursive equations, look up these solutions which are already computed. To compute a minimum distance tour, use the final equation to generate the 1st node, and repeat for the other nodes. For this problem, we cannot know which subproblems we need to solve, so we solve them all. This algorithm cannot be used as it takes too much time to calculate all the possibilities.