AI Lab Report No 1 - Group 16

Bhargavan, Sudarsan; Pan, Jiaqi; Hechehouche, Hacene

September 23, 2017

Introduction

This report is turned in as an observation for the first labroratory assignment which was tasked with guiding a **Delivery Man** through a gird with packages to be picked up and delivered to their respective destinations.

The Delivery Man has to traverse through a square grid with changing traffic conditions after every turn, while pickingup and delivering the packages and make sure that the overall cost is minimal. An A* algorithm is used to aid the delivery man in finding the next best move and the algorithm also has to adapt to the constantly changing traffic conditions to find the optimal route with minimal over all cost.

The main objective of this lab assignment is to dicuss the advantages and limitations of the A* algorithm by performing multiple simulations over the Delivery Man program, and also to implement and optimize the A* Algorithm, & various other strategies to find the most cost effective solution.

Tools Used

- R Language
- R Markdown
- R Studio
- Git & Github

A Theoritical Overview of A* Algorithm

A* Algorithm

A* Search Algorithm is a combination of **Dijkstra's Shortest Path Algorithm** and **Best First Search Algorithm**.

There are three important values used in A* search algorithm:

i. $f(n)$: estimated shortest path from start to goal via n
ii. $g(n)$: length of shortest path to n found so far
iii. $h(n)$: heuristic: estimated distance from n to goal

$f(n)$ is given by the following equation

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

In **Dijkstra's Shortest Path Algorithm**, we choose the next state only based on the $g(n)$ value, and it chooses to go to the state with the least cost of getting to there. In other words, Dijkstra's Shortest Path Algorithm is a special case of A* algorithm where the heuristic value $h(n)$ always equal to **0**.

In **Best First Search Algorithm**, we visit next state only based on heuristics function $f(n) = h$ with lowest heuristic value. It doesn't take into consideration the actual cost of the shortest path found so far to get to that state. All it cares about is that which next state from the current state has lowest heuristics.

In **A* Algorithm**, we decide next state based on the value of $f(n) = h(n) + g(n)$. We always choose the node with the minimum $f(n)$ value to visit next. Therefore A* algorithm chooses next state considering both its heuristics and real cost of getting to that state.

A* algorithm also guarantees that when we have visited the goal state, we have found the shortest path to the goal state. And then we can backtrack to the starting state and find out the shortest path and the minimum actual cost for the shortest.

Optimality Conditions

Not all heuristics value can guarantee us the shortest path. This leads to the discussion of optimality conditions: Let $h*(n)$ denotes the actual shortest path from n to goal.

Below are the definitions for optimistic and monotonic:

i. The heuristic is admissible or optimistic, as it will never overestimate the cost, or $h(n) \leq h*(n)$ for all nodes n.
ii. The heuristic is monotonic, that is, if $h(ni) <= h(ni+1)$, then $h*(ni) <= h*(ni+1)$.

Optimality conditions for A* using tree search and A* using graph search are different.

i. If $h(n)$ is admissible (optimistic), A* using tree search is optimal.
ii. If $h(n)$ is monotonic and therefore optimistic, A* using graph search(keep track of visited node and never revisit them) is optimal.

Only when the optimality conditions are met can A* algorithm guarantees us the shortest path instead of just giving us one of the shortest path.

Choice of Heuristic

The performance of A* algorithm highly depends on the choice of the heuristic function $h(n)$. There is a trade-off between running speed and accuracy of finding shortest path. We need to choose suitable heuristic functions according to our tasks. Below are three situations where heuristics are close to, below or higher than real cost:

In general, if the heuristics we find are very close to the real cost, the algorithm will figure out the shortest or one of the shortest paths quickly. And if $h(n)$ is exactly equal to the cost of moving from n to the goal, then A* will only follow the best path and never expand anything else, making it very fast.

And If $h(n)$ is always lower than (or equal to) the cost of moving from n to the goal, then A* is guaranteed to find a shortest path. The lower $h(n)$ is, the more node A* expands, making it slower. In one extreme case, we set $h = 0$, then the A* algorithm is reduced to Dijkstra Algorithm.

If $h(n)$ is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.

Properties of A* Algorithm

The main drawback of A* algorithm is its memory requirement because the entire frontier set must be saved and the visited set also need to be stored in graph search.

But A* is optimally efficient as no algorithm with the same heuristic is guaranteed to expand fewer nodes. And it is complete in finite space as it is guaranteed to find a solution if there is one.

Reference: stackowerflow; Lecture Slides; Stanford CS Theory

A* Strategies Discussed

i. While implementing the A* algorithm we careated 3 lists viz;

ii. visited

iii. frontier

iv. neighbours.

v. We calcualted the heuristics by passing the predefined **offset**, **packages** and **goal** as parameters, then we add the starting point **start** list which is given by the equation

$$start = (X = x, Y = y, parent = NULL, f = h[x, y], g = 0, heuristic = h[x, y])$$

where X & Y are the x,y co-ordinates of the car, since we start at the origin the $parent$ is set to null, f is the total cost, g is the estimated cost.

i. We then append the value of start to the first element in the frontier set.

ii. Then when the frontier set is not empty we sort the frontier set based on increasing f value. Now the first element in the frontier set is expanded and it is deleted from the same and appended to the visited set.

iii. If the goal node is visited then we terminate.

iv. Now we expand the node and set its neighbours to the frontier set after all the attributes of the neighbours have been calculated. We then check if the neighbouring nodes are already visited.

v. We now find the next step by backtracking to the start, we end up with the shortest path.

Our Heuristic

i. We use Manhattan Distance to calculate our heuristic

$$|(x_i - x_j)| + |(y_i - y_j)|$$

i. Manhattan Distance Metric is the distance sum of all the edges (Vertical &/or Horizontal Path) between two points in a gird, as opposed to Diagonal Path.

ii. We store it in a matrix, so that the heuristics of every node is stored by mapping the coordinates to the matrix, Assigned to every node is a heuristic value generated by finding the Manhattan distance between the node and the goal.

iii. We used the following equation because

$$(|(x_i - x_j)| + |(y_i - y_j)|) * 1$$

We can only move on the grids, so the Manhattan distance best describes that feature. And the number of edges to go through to reach the goal generally reflects the costs.

i. Why is k
in $(|(x_i - x_j)| + |(y_i - y_j)|) * k$
set to 1 ? because the minimum cost for the roads is 1, we need it to be optimistic to reach optimality condition, so $k <= 1$
.

And if h
value close to real cost, the algorithm runs faster, so we set the k
to its max of all the possible values which is 1

i. We also experimented by setting $k = 2$
. By theory, it cannot guarantee us the shortest path, however it turns out there is no big difference. (When testing it 500 times, the averages of turns only
differ around 1).

ii. The possible explanation for this is the road conditions are always changing. We call A* every time we make a move and only move one step according to the shortest path found in current conditions. So in the long term, we cannot make sure the path we follow is the shortest taking into consideration the time variable. So the small cost difference between the shortest path and one of the shortest paths may be diluted by the time varying cost.

iii. But our goal is to minimize the total cost, or the turns, so we set $k = 1$
to make sure it's the best solution using to A* algorithm.

Tree Search vs. Graph Search

We choose graph search for this task.

The advantage for graph search is it keeps track of all the visited nodes so that it won't revisit them. This feature makes it runs faster than tree search.

The drawback regarding graph search is it consumes more memory space than tree search.

Regarding our problem, we only have a 10x10 grids, so memory is not the key factor here. But we implement the A* algorithm and test it by running 500 times, it takes minutes. So our goal is to make it faster. Therefore, we choose graph search instead tree search

---

Non A* Strategies Discussed

When discussing about other strategies we ended up with 2 cases viz;

i. **Manhattan Distance Metric** to calculate the total cost to the goal, & heuristic values and then **A* Algorithm** to traverse through the nodes.

ii. **A* Algorithm** twice, when the car is not carrying any packages, apply A* algorithm to all the packages that are left to find the nearest package to pickup, and then use A* algorithm to get there.

Case 1

why Manhattan + A* Algorithm ?

Since where the packages should be picked up and where they should be delivered is told before the delivery, we can plan the total route in advance. Since the number of edges we have to go through from pick up point to delivery point is fixed, what we need to do is calculate the distance from the starting point to the 1st pick up point plus the 1st delivery point to the 2nd pick up point… plus from the 4th delivery point to the 5th pick up point. Since we need to plan for the future, this time cost found by the A *function won't be helpful. Because we cannot use the current road conditions to predict the future cost. And also since in this case we have 5! = 120 possible orders to test, A* is not efficient to do so. So we simply use the Manhattan distance to do the planning.

At first, we think we only need to plan once before we get started and then just follow this order to deliver. However, when we run this algorithm, we find that because we use A* algorithm to decide next move, the car may accidentally pick up a package that doesn't follow our planned order. So we decide whenever the car is not carrying packages, it keeps planning the best route.

i. While using the **Manhattan Distance Metric** to calculate the heuristic values we generate a heuristic matrix h with dimension 10x10 and manhattan distance values as elements.

ii. Then we implemented the above discussed **A* Algorithm** to get the total cost to the goal and the next best node to traverse to with minimal projected cost.

iii. Now when looking into the order in which deliveries can be made we figured out that $5!$ possibilities were available, so we generated a permutations matrix with n rows and n! columns to house all these permuted orders.

example;

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 3 & 2 & 1 \\ 3 & 5 & 4 & 1 & 2 \end{bmatrix}$$

i. We then calculated cost for every order in the permutations matrix.

ii. Then we appended these cost values to the permutations matrix as a column with the corresponding order permutation and found the minimum value of the cost column, which helped us in picking the delivery order which wil only take very minimal turns to deliver all the packages.

6.We then used A* algorithm, which in turn used the car load as a condition to generate a matrix p which holds information about the next node to be traversed to (co-ordinates).

i. By examining the values of p the car can be moved based on conditions like; if the car is at $(0, 0)$ and the x co-ordinate of the car is less than the x co-ordinated of p move right.

Case 2

i. When we run the basic A* algorithm, we noticed that the car sometimes go to pick up the farther package instead of the nearest one and thus take a detour because it always follow a fixed delivery order. So we decided to let the car dynamically decide which package to pick up, and always pick up the nearest package.

ii. We have two possible ways of calculating the nearest distance. The first one is simply use Manhattan distance to see which is the nearest. However, since we want to calculate which is the nearest package currently, and since the word “nearest” means the minimum actual the cost of the shortest path there. So we decide to use A* algorithm to find out the cost of the shortest path to each not picked up packages when the car is not carrying any package. Although running the A* algorithm cost a lot of computational resources, it is practical and acceptable because we have at most 5 not picked up packages and we run the A* algorithm for at most 5 times.

iii. So, the first step we do is to find out the cost of the shortest path from the car to each of the not picked up packages. We modify the A* function we write before to make it returns both the cost of the shortest path and the next node along the shortest path. We stored this cost value along with the package ID in a matrix and then compare the cost of the shortest path, choose the package with the minimum value to go to next. After we decide which package to go to, we can just use the “next node” returned from A* function and decide the direction of the next move.

iv. We do this each time the car makes a move when it is not carrying a package to adjust to the changing traffic

conditions. When the car is carrying a package, we only run A* function once to find out the next node on the shortest path.

Observations

When we only use A* Algorithm and a fixed delivery order (1,2,3,4,5), the mean value is 205

500 simulations were run and a mean of these readings were calculated,

i. Case 1 had a mean value of 165
ii. Case 2 had a mean value of 178

Other Observations

i. We also observe that the road is getting more congested. When inspecting the code we find that when the road cost is one, it only has a possibility to go up and no possibility to go down, and when road cost is more than one it has 5% probability of going up and 5% probability of going down. So we calculate the average road costs each time they update and print that result out. We observe that, the average cost will go from 1 to 2 very quickly and then keep going up with a slowed down rate. All the packages are delivered when average cost reaches 4.

ii. So we think about when planning for the total route, we may let the car travel longer paths at beginning and then shorter paths afterwards. We want to apply this observation to our total planning by assigning weights to our distance calculation.

iii. Previously, Distance = d1+d2+d3+d4+d5 (where di denotes the (i-1)th delivery point to the ith pick up point). We try different weights for Distance = w1*d1+w2*d2+w3*d3+w4*d4+w5*d5 where w1<w2<w3<w4<w5.

iv. However, after we tried many combinations of weights, we find this method doesn't make significant improvement. We think it is because in the previous methods we didn't take into consideration the distance from pick up to delivery point. And a small modification in the weight doesn't generally change the shortest delivery order we find before.