A* algorithm

Sunday, January 12, 2020 19:48

For the glory of God

What is A* algorithm ?

- A* is a path search algorithm that is often used in computer science due to its properties (e.g. optimal efficiency)
  - ↳ More specifically, it is an informed search algorithm, meaning that it is formulated in terms of weighted graphs; starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost.
- A* was created as part of the Shakey project which had the aim of building a mobile robot that could plan its own actions.
- The Stanford Research Institute (SRI) research team first published the algorithm in 1968.

Why A* algorithm ?

a) Introduction

- Let's say that we have a problem in which we need to find the shortest path from one node in the network to another.
  - It is well known that A* algorithm is the best algorithm for this class of problems.
  - Then why ? In order for us to answer the question, we may first need to take a look the other algorithms (e.g. Dijkstra)

b) Dijkstra's algorithm (Breadth First Search)

- The algorithm works by visiting vertices in the graph starting with the object's starting point.
- It then repeatedly examines the closest not yet examined vertex, adding its vertices to the set of vertices to be examined.
- It expands outwards from the starting point until it reaches the goal.
- The algorithm is guaranteed to find the shortest path from the starting point to the goal as long as none of edges have a negative cost.
  - ↳ If there is an edge having a negative cost, we need to employ Bellman-Ford algorithm. (For more details, see the data structure hand-written note.)

c) The Greedy best search algorithm (Depth First Search)

- It works in a similar way compared to Dijkstra's algorithm, except that it has some estimate (called a heuristic) of how far from the goal any vertex is.
- Instead of selecting the vertex closest to the starting point, it selects the vertex closest to the goal.
- The algorithm is not guaranteed to find the shortest path; however, it runs much quicker because it uses the heuristic function to guide its way towards the goal very quickly.
  - For example, if the goal is to the south of the starting position, the algorithm will tend to focus on paths that lead southwards.

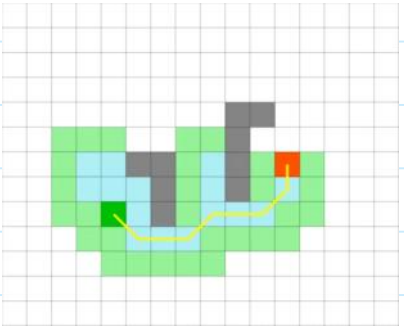
d) A* algorithm

- The algorithm was developed to combine heuristic approach like as the Greedy best search and formal approach like as Dijkstra.
- It is using the sum of two distances as follows ; thus $f(n) = g(n) + h(n)$
  - ① $g(n)$: the distance from the starting point
  - ② $h(n)$: the distance to the goal (by heuristic, e.g. Euclidean distance)
- The secret to its success is that it combines the pieces of information ;
  - favoring vertices that are close to the starting point
  - favoring vertices that are close to the goal
- The algorithm is guaranteed to find the shortest path if the properties are satisfied.

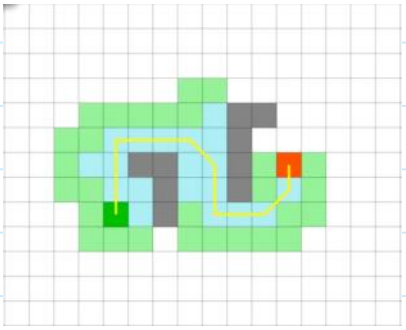
  e.g. The heuristic is never larger than the true distance.
- Because it is fairly flexible and can be used in a wide range of contexts, it's considered as the most popular choice for pathfinding.

e) Algorithm example

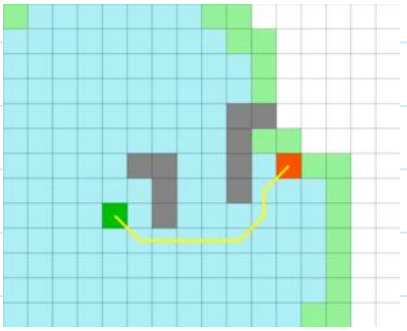
- The following example shows how they are different in the particular example.



A* algorithm



The Greedy best search



Dijkstra's algorithm

What are properties of A* algorithm ?

a) Introduction

- As mentioned, A* algorithm is guaranteed to find the shortest path if properties are satisfied.

  ↳ Then what are the properties ?
- The properties are as follows ;
  - Admissibility, Completeness, Consistency, and optimal efficiency

b) Completeness

- On finite graphs with non-negative edge weights, A* is guaranteed to terminate and is complete.
- In other words, it will always find a solution if one exists.

c) Admissibility

· When A* terminates its search, it has found a path from start to goal.

↳ which is based on the completeness

· Let's say that we found the solution path.

- If the heuristic is admissible, the solution is an optimum.

- Then the question is 'what does it mean, admissible?'

: Estimated distance between any node x and the goal is less than or equal to the actual distance.

" Heuristic (x, goal) ≤ Actual (x, goal) "

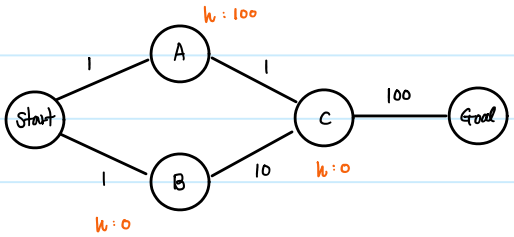
↳ This is a mathematical definition of admissibility.

· Then we are good?

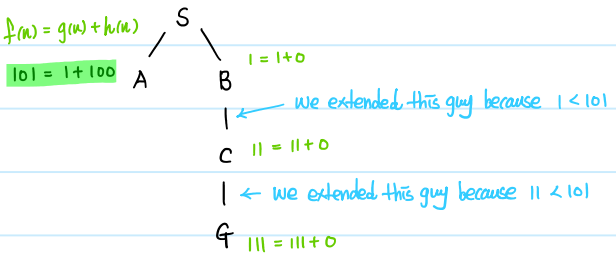
- No! we need one more property, which is consistency.

· Let's take an example that shows why we need one more stronger property.

e.g. MIT AI Class example (Instructor : prof. Patrick Winston)



; Let us simulate A* algorithm and see what happens.



∴ S → B → C → G ; Hence, this is obviously not the shortest path.

↳ This is because we would keep extending A, thus; S → A → C → G (102)

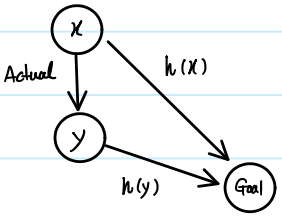
d) Consistency

· A heuristic is consistent if it satisfies " | Heuristic (x, goal) - Heuristic (y, goal) | ≤ Actual (x, y) "

- What it means is as follows;

: The distance between x and goal ; minus ; the distance between some other node y ; and take the absolute value of that ; that has to be less than or equal to the actual distance between x and y.

- It is a sort of the triangle inequality.



; where $h(x) \leq Actual + h(y)$

$\Leftrightarrow h(x) - h(y) \leq Actual$

e) Optimal efficiency

- Rina Dechter and Judea Pearl proved that :
  - A* is optimally efficient with respect to all A*-like search algorithms if the A* is both admissible and consistent.
- In other words,
  - There is no other algorithm that can find the shortest path between a pair of nodes on the network in a smaller number of paths expansion.

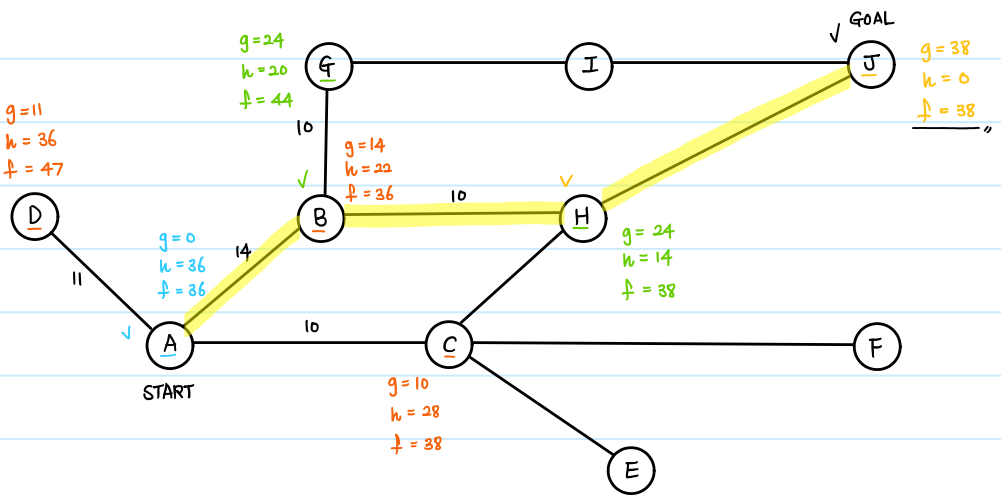
How does A* algorithm work ?

- First of all, keep in mind that there are a number of implementation ways that can affect the performance of an A* algorithm.
- The following Pseudo code is one of the examples, which is from Wiki.

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        total_path.prepend(current)
        current := cameFrom[current]
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n).
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

- Now, let us take a canonical example to get better understanding.

- Let's say that the following graph is given and we want to find the shortest path. (A → B → H → J)



Step 1) OPEN = [A], CLOSE = [∅]

· To begin with, note that there are two sets, namely OPEN and CLOSE.

- The OPEN set contains those nodes that are candidates for examining.

- The CLOSE set contains those nodes that have already been examined.

· Initially { the OPEN set contains only one element ; starting point.
the CLOSE set is empty.

· The initial cost is as follows ;

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

$= 0 + 36$ → Assume that we use the Euclidean distance as the heuristic function. (e.g. $h = \sqrt{30^2 + 20^2} = 36$)

$= 36$

Step 2) OPEN = [B, C, D], CLOSE = [A]

· Note that the OPEN queue keeps track of nodes that can be visited.

- For this case, since B/C/D are connected from A, the queue added them.

· Since the node A is done, it is removed from the OPEN queue and added to the CLOSE queue.

· The cost for each node is computed as can be seen.

Step 3) OPEN = [H, C, G, D], CLOSE = [A, B]

· You can imagine that there is a mechanism that repeatedly pulls out the best node in the OPEN queue, namely priority queue.

- For this case, the next visited node is B because the node has the lowest value.

- Note that the mechanism remembers the previous node. (A → B) ; let's say it is a parent node.

Step 4) OPEN = [J, C, G, D], CLOSE = [A, B, H]

· We notice that the node H and C have the same cost values.

- which one would the A* algorithm choose ?

: The rule is to choose a node with the smallest heuristic value.

↳ That's why the node H is removed from the priority queue for this case. (and then add J into the OPEN queue)

Step 5) OPEN = [C, G, D], CLOESE = [A, B, H, J], Current node = [J]

· The current node is the goal ; hence, the algorithm is done.

· As a final note, keep in mind that A* won't work if there is a situation in which the start and goal are not connected by the graph.