0.1 A* Algorithm

- *Algorithm:* Determine the shortest path between the start point and the end point (algo. 1)
- *Input:* The start point and goal point, a graph $G\langle V, E\rangle$
- *Complexity:* $\mathcal{O}((b^{\epsilon})^{d})$ for time complexity, $\mathcal{O}(|V|)$ for the worst space complexity
- *Data structure compatibility:* Priority Queue
- *Common applications:* Path planning for robots, path planning for character in games

A* Algorithm

A* Algorithm is a graph traversal and path search algorithm used in weighted graphs. The algorithm starts from a start point, aims to find a path with the smallest cost to the given goal point.

Description

Detail

We denote openSet as the set of nodes that have already been visited. Obviously, at first only the start point is in the openSet. At each loop, A* algorithm takes the node with the smallest path cost from the openSet. It determines the paths to extend the openSet and then goes into the next loop until it reaches the goal point. The path can be found by tracing path back from the goal point to the start point.

Determine the path cost

Different from Dijkstra Algorithm, the path cost $f(n)$ is calculated as

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

where $g(n)$ represents the exact cost of the path from the start point to the vertex n, and $h(n)$ represents the heuristic estimated cost from vertex n to the goal.

Determin the heuristic

The performance of A* algorithm is affected by the choice of heuristic. It works as follows.

- If $h(n) = 0$, A^* Algorithm actually becomes Dijkstra Algorithm, which is guaranteed to find a shortest path.
- If $h(n)$ is lower than (or equal to) the cost of moving from n to the goal, it's also guaranteed to find a shortest path. However, the lower $h(n)$ is, the more A^* Algorithm will expand, the slower it will be.
- If $h(n)$ is exactly the cost of moving from n to the goal, A^* Algorithm will follow the best path, and speed up the performance.
- If $h(n)$ is greater than (or equal to) the cost of moving from n to the goal, A^* is not sure to find a shortest path, but it does take less time.

- If $h(n)$ is relatively higher than $g(n)$, this turns into a Greedy Best-First-Search.

There is a problem that will considered in practical path planning but not discussed here. It is the choice between the compute speed and the accuracy of the shortest path. In such situation, the path needed is not necessarily to be the shortest path.

One way to construct a heuristic is to precompute the length of the shortest path between every pair of points. Such way is not feasible in practical use where the graph could be extremely large. However, there are ways of approximation.

The method that is commonly used in game design is to precompute the shortest path between any pair of waypoints. By waypoints, I mean a point along a path. For example, in many real-time strategy games, players can manually add path-specific waypoints by clicking. Then, use a new heuristic $h^{'}$ that estimates the cost from any location to its nearby waypoints.

The final heuristic can be

$$h(n) = h^{'}(n, w_1) + distance(w_1, w_2) + h^{'}(w_2, goal)$$

Specified for my implementation

For simplicity, the graph in my implementation is square grid. The heuristic for square grid is the Manhattan distance between two points. [3]

The Manhattan distance between points x_1, y_1 and x_2, y_2 is calculated as

$$D = |x_1 - x_2| + |y_1 - y_2|$$

Time complexity

Denote b as the branching factor, namely maximum number of successors of any node, d as the depth of the shallowest goal node, namely the the length of the path from the start to the goal. [1] At first glance, the complexity can be expressed as $\mathcal{O}(b^d)$. However, b is not constant during the algorithm.

For problems with constant step costs, the effective branching factor b can be analyzed in the absolute error or the relative error of the heuristic. The abosolute error is $\Delta \equiv h^* - h$, relative error $\epsilon \equiv (h^* - h)/h^*$ [2], where h^* is the actual cost from the start to the goal. The absolute error ϵ is at least proportional to the pathcost h^*, so ϵ is constant or increasing. The time complexity is written as

$$T(n) = \mathcal{O}((b^{\epsilon})^d)$$

Pseudocode

Algorithm 1: A* Algorithm(G,*start*,*goal*)

Input : The start point *start*, the goal point *goal*, a graph *graph*
Output: The shortest path and its distance

```
Function heuristic((a,b)):
    return the cost of the cheapest path from a to b.  /* e.g. Manhattan distance for square grid */
end
Push start into a priority queue P with its pathCost 0 as the key
while P is not empty do
    Pop the top of P and save it as x
    if x is the goal point then
        break
    end if
    for each x's neighbor point n in the graph that is not blocked do
        tmpCost = x.pathCost + n.cost
        if n is not visited or tmpCost < n.cost then
            n.cost = tmpCost
            val = tmpCost + heuristic (n, end)
            Push n into P with val as the key
            n.prev = x
        end if
    end for
end while
itr = goal
path = []
while itr ≠ start do
    Push itr to the front of path
    itr = itr.prev
end while
return path, end.pathCost
```

References.

[1] Peter Norvig and Stuart J. Russell. “Infrastructure for search algorithms”. In: *Artificial Intelligence: A Modern Approach*. Pearson, 2009, p. 80 (cit. on p. 2).

[2] Peter Norvig and Stuart J. Russell. “Infrastructure for search algorithms”. In: *Artificial Intelligence: A Modern Approach*. Pearson, 2009, p. 98 (cit. on p. 2).

[3] Amit Patel. *Amit’s Thoughts on Pathfinding*. `http://theory.stanford.edu/~amitp/GameProgramming/index.html`. Accessed November 13, 2019 (cit. on p. 2).