**PMAS Arid Agriculture University**

**University Institute of Information Technology**

**CS-632 Artificial Intelligence**

**Handout 05**

### What is A* Search Algorithm?

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

## Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

## Explanation

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

What A* Search Algorithm does is that at each step it picks the node according to a value-'f' which is a parameter equal to the sum of two other parameters – 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell.

We define 'g' and 'h' as simply as possible below

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this 'h' which are discussed in the later sections.

## Algorithm

We create two lists – Open List and Closed List

// A* Search Algorithm

1. Initialize the open list
2. Initialize the closed list
   put the starting node on the open
   list (you can leave its f at zero)
3. while the open list is not empty
   a. find the node with the least f on the open list, call it "q"
   b. pop q off the open list
   c. generate q's 8 successors and set their
      parents to q
   d. for each successor
      i. if successor is the goal, stop search
      ii. else, compute both g and h for successor

      successor.g = q.g + distance between

      successor and q

      successor.h = distance from goal to

      successor (This can be done using many

      ways, we will discuss three heuristics-

      Manhattan, Diagonal and Euclidean

      Heuristics)

      successor.f = successor.g + successor.h

      iii. if a node with the same position as

      successor is in the OPEN list which has a

      lower f than successor, skip this successor

      iv. if a node with the same position as

      successor is in the CLOSED list which has

      a lower f than successor, skip this successor

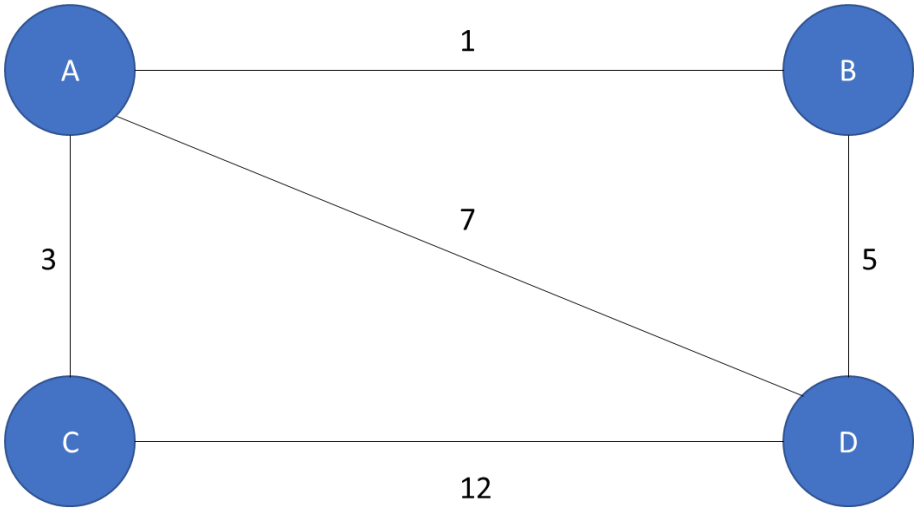
      otherwise, add the node to the open list

   end (for loop)

e. push q on the closed list

end (while loop)

## Graph:



## Implementation

```
from collections import deque

class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    # This is heuristic function which is having equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start, stop):
        # In this open_lst is a lisy of nodes which have been visited, but who's
        # neighbours haven't all been always inspected, It starts off with the start
  #node
        # And closed_lst is a list of nodes which have been visited
        # and who's neighbors have been always inspected
        open_lst = set([start])
        closed_lst = set([])

        # poo has present distances from start to all other nodes
        # the default value is +infinity
        poo = {}
        poo[start] = 0

        # par contains an adjac mapping of all nodes
```

```
        # par contains an adjac mapping of all nodes
        par = {}
        par[start] = start

        while len(open_lst) > 0:
            n = None

            # it will find a node with the lowest value of f() -
            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop
            # then we start again from start
            if n == stop:
                reconst_path = []

                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]

                reconst_path.append(start)

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all the neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
              # if the current node is not presentin both open_lst and closed_lst
                # add it to open_lst and note n as it's par
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    par[m] = n
                    poo[m] = poo[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update par data and poo data
```

```
                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update par data and poo data
                # and if the node was in the closed_lst, move it to open_lst
                else:
                    if poo[m] > poo[n] + weight:
                        poo[m] = poo[n] + weight
                        par[m] = n

                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)

            # remove n from the open_lst, and add it to closed_lst
            # because all of his neighbors were inspected
            open_lst.remove(n)
            closed_lst.add(n)

        print('Path does not exist!')
        return None

adjac_lis = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

## Output

```
A-star search
Path found: ['A', 'B', 'D']
```

## Code

```
from collections import deque


class Graph:

    def __init__(self, adjac_lis):

        self.adjac_lis = adjac_lis
```

```
  def get_neighbors(self, v):
    return self.adjac_lis[v]

  # This is heuristic function which is having equal values for all nodes
  def h(self, n):
    H = {
      'A': 1,
      'B': 1,
      'C': 1,
      'D': 1
    }

    return H[n]

  def a_star_algorithm(self, start, stop):
    # In this open_lst is a lisy of nodes which have been visited, but who's
    # neighbours haven't all been always inspected, It starts off with the start
#node
    # And closed_lst is a list of nodes which have been visited
    # and who's neighbors have been always inspected
    open_lst = set([start])
    closed_lst = set([])

    # poo has present distances from start to all other nodes
    # the default value is +infinity
    poo = {}
    poo[start] = 0

    # par contains an adjac mapping of all nodes
```

```
par = {}
par[start] = start

while len(open_lst) > 0:
    n = None

    # it will find a node with the lowest value of f() -
    for v in open_lst:
        if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
            n = v;

    if n == None:
        print('Path does not exist!')
        return None

    # if the current node is the stop
    # then we start again from start
    if n == stop:
        reconst_path = []

        while par[n] != n:
            reconst_path.append(n)
            n = par[n]

        reconst_path.append(start)

        reconst_path.reverse()

        print('\nA-star search\nPath found: {}'.format(reconst_path))
```

```
        return reconst_path

    # for all the neighbors of the current node do
    for (m, weight) in self.get_neighbors(n):
      # if the current node is not presentin both open_lst and closed_lst
        # add it to open_lst and note n as it's par
        if m not in open_lst and m not in closed_lst:
            open_lst.add(m)
            par[m] = n
            poo[m] = poo[n] + weight

        # otherwise, check if it's quicker to first visit n, then m
        # and if it is, update par data and poo data
        # and if the node was in the closed_lst, move it to open_lst
        else:
            if poo[m] > poo[n] + weight:
                poo[m] = poo[n] + weight
                par[m] = n

                if m in closed_lst:
                    closed_lst.remove(m)
                    open_lst.add(m)

    # remove n from the open_lst, and add it to closed_lst
    # because all of his neighbors were inspected
    open_lst.remove(n)
    closed_lst.add(n)

print('Path does not exist!')
```

```
        return None


adjac_lis = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

---