National University of Computer and Emerging Sciences

Artificial Intelligence

Lab 6

Fast School of Computing

FAST-NU, Lahore, Pakistan

Objectives

- A* Algorithm

1. History

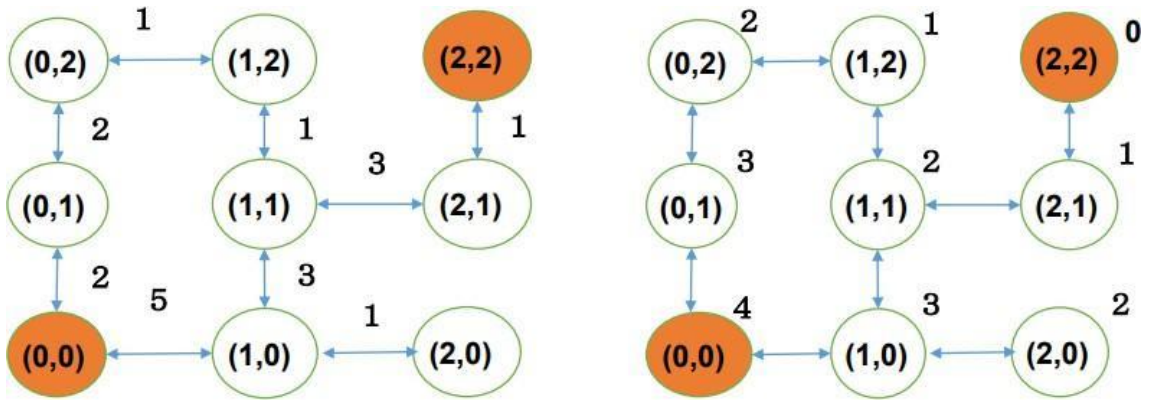
- As early as 1962, John Holland's work on adaptive systems laid the foundation for later developments.
- In 1964, Nils Nilsson invented a heuristic-based approach to increase the speed of Dijkstra's algorithm, known as A1.
- In 1967, Bertram Raphael made significant improvements upon this algorithm but failed to demonstrate optimality, naming it A2.
- Then in 1968, Peter E. Hart introduced an argument proving A2's optimality when using a consistent heuristic with minor changes. He named the algorithm A*.

*1.2 What is A-Star Algorithm (A)?**

- A* is a searching algorithm used to find the shortest path between an initial and final point, often utilized for map traversal.
- Initially designed for graph traversal to aid in building robots finding their own course, A* remains widely popular for graph traversal.
- It prioritizes searching for shorter paths first, making it an optimal and complete algorithm.

1.3 A Algorithm Flow*

- **Step 1:** Place the initial node x0 and its cost F(x0) = H(x0) in the open list.
- **Step 2:** Retrieve a node x from the top of the open list. If empty, stop with failure. If x is the target node, stop with success.
- **Step 3:** Expand x to get a set S of child nodes. Place x in the closed list.
- **Step 4:** For each x' in S:
    - If x' is in the closed list but the new cost is smaller than the old one, move x' to the open list and update the edge (x, x') and the cost.
    - Else, if x' is in the open list but the new cost is smaller than the old one, update the edge (x, x') and the cost.
    - Else (if x' is not in the open list nor in the closed list), place x' along with the edge (x, x') and the cost F in the open list.

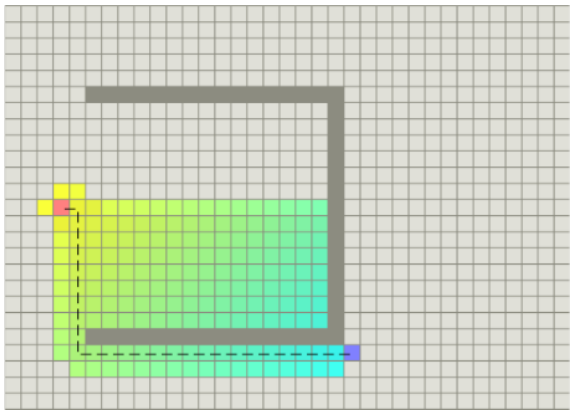


Steps	Open List	Closed List
0	{(0,0), 4}	--
1	{(0,1),5} {(1,0),8}	{(0,0),4}
2	{(0,2),6} {(1,0),8}	{(0,0),4} {(0,1),5}
3	{(1,2),6} {(1,0),8}	{(0,0),4} {(0,1),5} {(0,2),6}
4	{(1,0),8} {(1,1),8}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}}
5	{(1,1),8} {(2,0),8}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8}
6	{(2,0),8} {(2,1),10}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8} {(1,1),8}
7	{(2,1),10}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8} {(1,1),8} {(2,0),8}
8	{(2,2),10}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8} {(1,1),8} {(2,0),8} {(2,1),10}
9	(2,2)=target node	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8} {(1,1),8} {(2,0),8} {(2,1),10}

*1.4 A * Search and its Heuristic*

- A* combines aspects of Dijkstra's algorithm (favoring vertices close to the starting point) and Greedy Best-First-Search (favoring vertices close to the goal).
- In standard terminology, g(n) represents the exact cost of the path from the starting point to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal.
- The heuristic can be used to control A*'s behavior:
    - At one extreme, if h(n) is 0, then only g(n) plays a role, and A* turns into Dijkstra's Algorithm, which is guaranteed to find the shortest path.
    - 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 the shortest path. The lower h(n) is, the more node A* expands, making it slower.
    - 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.

Although You can't make this happen in all cases, you can make it exact in some special cases. It's nice to know that given perfect information, A* will behave perfectly.



3.5 Importance of Scale

- A* computes f(n) = g(n) + h(n). To ensure accurate path finding, g(n) and h(n) should be at the same scale.
- If g(n) and h(n) are measured in different units, A* may not perform optimally, leading to suboptimal paths or slower execution.

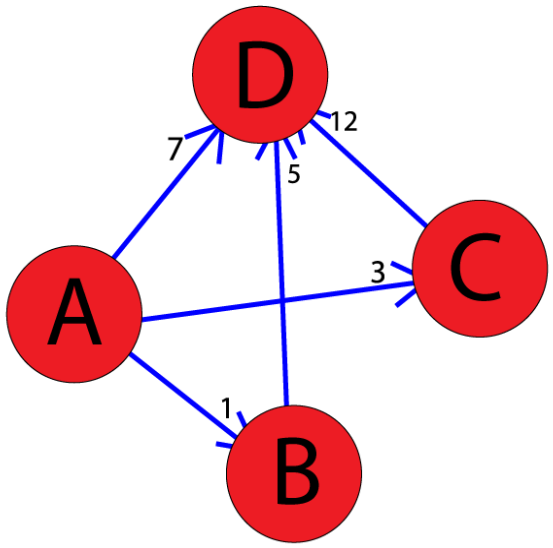
Step by Step Approach

Problem Solving using A-Star Algorithm

Famous problems which use A*Algorithm:

Shortest Path Problem

The graph is represented with an adjacency list, where the keys represent graph nodes, and the values contain a list of edges with the the corresponding neighboring nodes.



Here you'll find the A* algorithm implemented

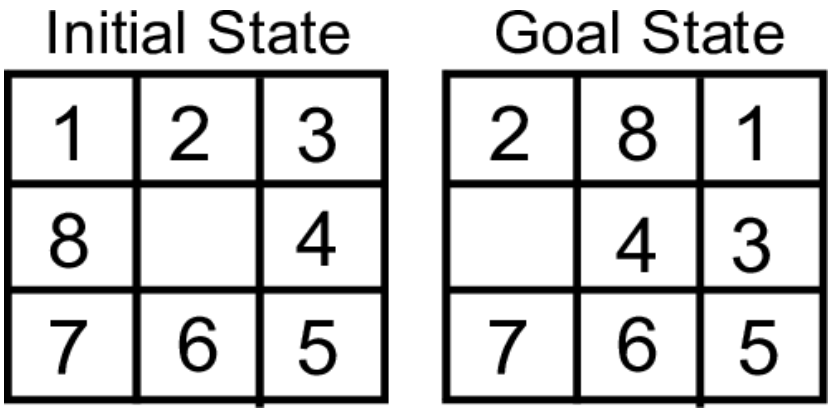
```
from collections import deque

class Graph:
    # example of adjacency list (or rather map) #
    adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)], # 'B':
    [('D', 5)],
    # 'C': [('D', 12)] #
    }

    def___init___(self, adjacency_list):
        self.adjacency_list = adjacency_list

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

    # heuristic function with 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_node, stop_node):
    # open_list is a list of nodes which have been visited, but who's neighbors # haven't all
    been inspected, starts off with the start node
    # closed_list is a list of nodes which have been visited # and
    who's neighbors have been inspected
    open_list = set([start_node])
    closed_list = set([])

    # g contains current distances from start_node to all other nodes # the
    default value (if it's not found in the map) is +infinity
    g = {}

    g[start_node] = 0

    # parents contains an adjacency map of all nodes parents =
    {}
    parents[start_node] = start_node

    while len(open_list) > 0: n =
        None

        # find a node with the lowest value of f() - evaluation function for v in
        open_list:
            if n == None or g[v] + self.h(v) < g[n] + self.h(n): n = v;

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

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node if n ==
        stop_node:
            reconst_path = []

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

            reconst_path.append(start_node)

            reconst_path.reverse()
```

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

        # for all neighbors of the current node do for
        (m, weight) in self.get_neighbors(n):
            # if the current node isn't in both open_list and closed_list # add it
            to open_list and note n as it's parent
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight

            # otherwise, check if it's quicker to first visit n, then m # and if
            it is, update parent data and g data
            # and if the node was in the closed_list, move it to open_list else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n

                    if m in closed_list:
                        closed_list.remove(m)
                        open_list.add(m)

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

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

To run this code, adjacency_list =
{
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')

Output will be:
Path found: ['A', 'B', 'D']
['A', 'B', 'D']
```

Exercise: You have to implement the 8-Puzzle problem using A* Algorithm.

N-Puzzle or sliding puzzle is a popular puzzle that consists of N tiles where N can be 8, 15, 24, and so on. In our example N = 8. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns. Eg. 15-Puzzle will have 4 rows and 4 columns and an 8-Puzzle will have 3 rows and 3 columns.
The puzzle consists of N tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. You have to solve the puzzle by moving the tiles one by one in the single empty space and thus achieving the Goal configuration.



Rules for solving the puzzle:

Instead of moving the tiles in the empty space, we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions viz.,

1. Up
2. Down
3. Right or
4. Left

The empty space cannot move diagonally and can take only one step at a time (i.e. move the empty space one position at a time).

You can read more about solving the 8-Puzzle problem here.

Submission Instructions

Always read the submission instructions carefully.

- Rename your Jupyter notebook to your roll number and download the notebook as **.ipynb** extension.
- To download the required file, go to **File->Download .ipynb**
- Only submit the **.ipynb** file. DO NOT **zip** or **rar** your submission file.
- Submit this file on Google Classroom under the relevant assignment.

Late submissions will not be accepted