National University of Computer and Emerging Sciences

Lab Manual
Artificial Intelligence Lab

Department of Computer Science
FAST-NU, Lahore, Pakistan

Table of Contents

1 Objectives 3

2 Task Distribution 3

3 A-Star Algorithm 3

3.1 History 3

3.2 What is A-star Algorithm (A*)? 3

3.3 A* Algorithm Flow 4

Step by Step Approach: 5

3.4 A* Search and its Hueristic 5

3.5 Importance of Scale: 6

Problem Solving using A-StarAlgorithm 7

1 Objectives

After performing this lab, students shall be able to understand the following Python concepts and applications:

- ✓ A* Algorithm
- ✓ Problem solving using A* Algorithm

2 Task Distribution

Total Time	**170 Minutes**
A* Algorithm Introduction	25 Minutes
Application of A* Algorithm	25 Minutes
Exercise	120 Minutes
Online Submission	10 Minutes

3 A-Star Algorithm

3.1 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. This algorithm was called A1.
- In 1967 Bertram Raphael made dramatic improvements upon this algorithm, but failed to show optimality. He called this algorithm A2.
- Then in 1968 Peter E.Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only minor changes. His proof of the algorithm also included a section that showed that the new A2 algorithm was the best algorithm possible given the conditions. He thus named the algorithm in kleene star

3.2 What is A-star Algorithm (A*)?

- It is a searching algorithm that is used to find the shortest path between an initial and a final point.

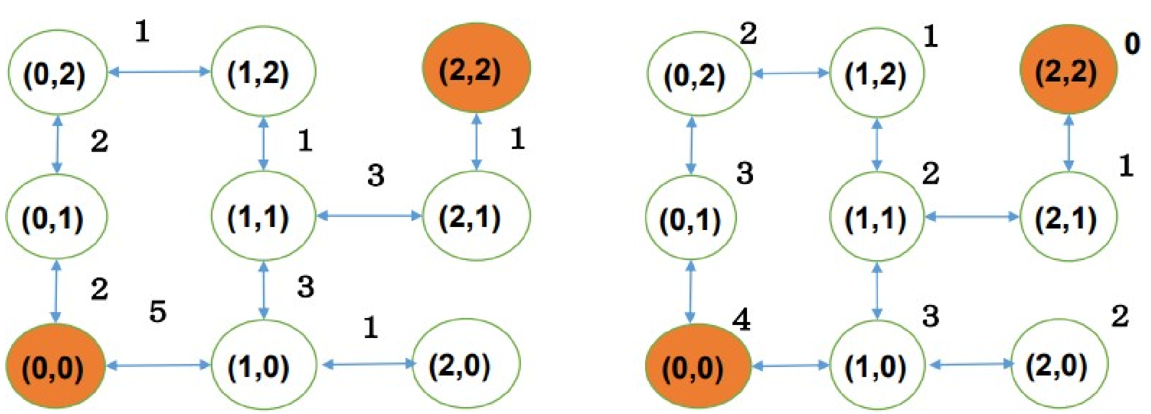
- It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

- It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

3.3 A* Algorithm Flow

Step 1: Put the initial node x0 and its cost F(x0)=H(x0) to the open list. • Step 2: Get a node x from the top of the open list. If the open list is 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. Put x to the closed list.

Step 4: For each x’ in S, find its cost – 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), put x’ along with the edge (x,x’) and the cost F to the open list.



Step by Step Approach:

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}

3.4 A* Search and its Hueristic

A* is like Dijkstra's algorithm in that it can be used to find a shortest path. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search. The secret to its success is that it combines the pieces of information that Dijkstra's algorithm uses (favouring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favouring vertices that are close to the goal). In the standard terminology used when talking about A*, 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. In the above diagram, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).

The heuristic can be used to control A*'s behaviour.

·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 a 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 a 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.

·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.

·At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role, and A* turns into Greedy Best-First-Search

A* is like Dijkstra's algorithm in that it can be used to find a shortest path. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search.

The secret to its success is that it combines the pieces of information that Dijkstra's algorithm uses (favouring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favouring vertices that are close to the goal). In the standard terminology used when talking about A*, 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.

In the above diagram, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).

The heuristic can be used to control A*'s behaviour.

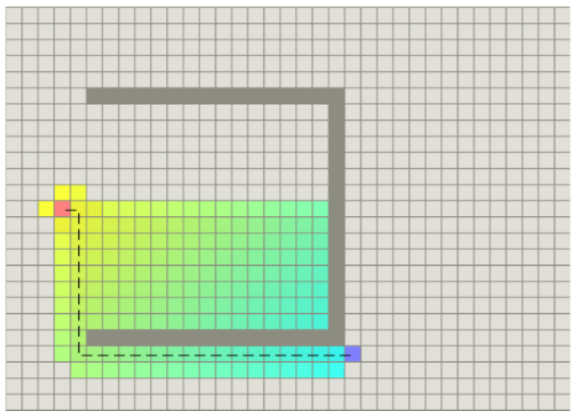
·At one extreme, if h(n) is 0, then only g(n) plays a role, and A* turns into Dijkstra's algorithm, which is guaranteed to find a shortest path.

·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 a shortest path. The lower h(n) is, the more node A* expands, making it slower.

48 ·If h(n) is exactly equal to the cost of moving from n to the goal, then A* will only
49 follow the best path and never expand anything else, making it very fast. Although
50 you can’t make this happen in all cases, you can make it exact in some special cases.
51 It’s nice to know that given perfect information, A* will behave perfectly.
52 ·If h(n) is sometimes greater than the cost of moving from n to the goal, then A* is not
53 guaranteed to find a shortest path, but it can run faster.
54 ·At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role,
55 and A* turns into Greedy Best-First-Search
56 A* is like Dijkstra’s algorithm in that it can be used to find a shortest path. A* is like Greedy
57 Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as
58 Greedy Best-First-Search.
59 The secret to its success is that it combines the pieces of information that Dijkstra’s algorithm
60 uses (favouring vertices that are close to the starting point) and information that Greedy Best-
61 First-Search uses (favouring vertices that are close to the goal). In the standard terminology
62 used when talking about A*, g(n) represents the exact cost of the path from the starting point
63 to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal.
64 In the above diagram, the yellow (h) represents vertices far from the goal and teal (g)
65 represents vertices far from the starting point. A* balances the two as it moves from the
66 starting point to the goal. Each time through the main loop, it examines the vertex n that has
67 the lowest f(n) = g(n) + h(n).
68 The heuristic can be used to control A*’s behaviour.
69 ·At one extreme, if h(n) is 0, then only g(n) plays a role, and A* turns into Dijkstra’s
70 algorithm, which is guaranteed to find a shortest path.
71 ·If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then
72 A* is guaranteed to find a shortest path. The lower h(n) is, the more node A* expands,
73 making it slower.
74 ·If h(n) is exactly equal to the cost of moving from n to the goal, then A* will only
75 follow the best path and never expand anything else, making it very fast. Although
76 you can’t make this happen in all cases, you can make it exact in some special cases.
77 It’s nice to know that given perfect information, A* will behave perfectly.
78 ·If h(n) is sometimes greater than the cost of moving from n to the goal, then A* is not

79 guaranteed to find a shortest path, but it can run faster.
80 ·At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role,
81 and A* turns into Greedy Best-First-Search
82 A* is like Dijkstra's algorithm in that it can be used to find a shortest path. A* is like Greedy
83 Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as
84 Greedy Best-First-Search

A* is like Dijkstra's algorithm in that it can be used to find a shortest path. A* is like GreedyBest-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast asGreedy Best-First-Search.



The secret to its success is that it combines the pieces of information that Dijkstra's algorithmuses (favouring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favouring vertices that are close to the goal). In the standard terminologyused when talking about A*, g(n) represents the exact cost of the path from the starting pointto any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal.

In the above diagram, the yellow (h) represents vertices far from the goal and teal (g)represents vertices far from the starting point. A* balances the two as it moves from thestarting point to the goal. Each time through the main loop, it examines the vertex n that hasthe lowest f(n) = g(n) + h(n)

The heuristic can be used to control A*'s behaviour.

- At one extreme, if h(n) is 0, then only g(n) plays a role, and A* turns into Dijkstra'salgorithm, which is guaranteed to find a shortest path.
- If h(n) is always lower than (or equal to) the cost of moving from n to the goal, thenA* is guaranteed to find a 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 onlyfollow the best path and never expand anything else, making it very fast. Althoughyou 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.

- If h(n) is sometimes greater than the cost of moving from n to the goal, then A* is notguaranteed to find a shortest path, but it can run faster.
- At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role,and A* turns into Greedy Best-First-Search.

You need to arrange three boxes labeled as A, B and C. State space graph of the problem is shown below.

So we have an interesting situation in that we can decide what we want to get out of A*. Atexactly the right point, we'll get shortest paths really quickly. If we're too low, then we'll continue to get shortest paths, but it'll slow down. If we're too high, then we give up shortestpaths, but A* will run faster.

3.5 Importance of Scale:

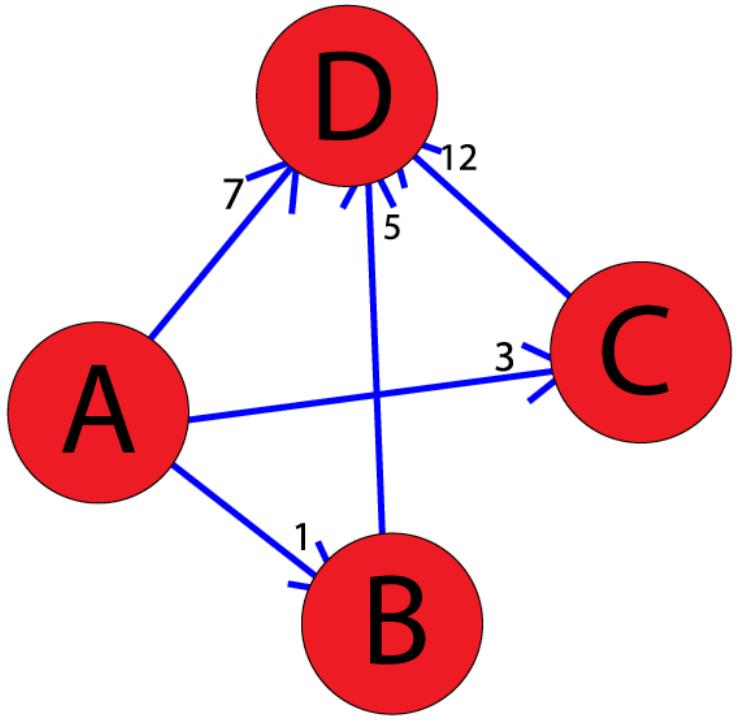
A* computes f(n) = g(n) + h(n). To add two values, those two values need to be at the samescale. If g(n) is measured in hours and h(n) is measured in meters, then A* is going toconsider g or h too much or too little, and you either won't get as good paths or you A* willrun slower than it could

Problem Solving using A-StarAlgorithm

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,

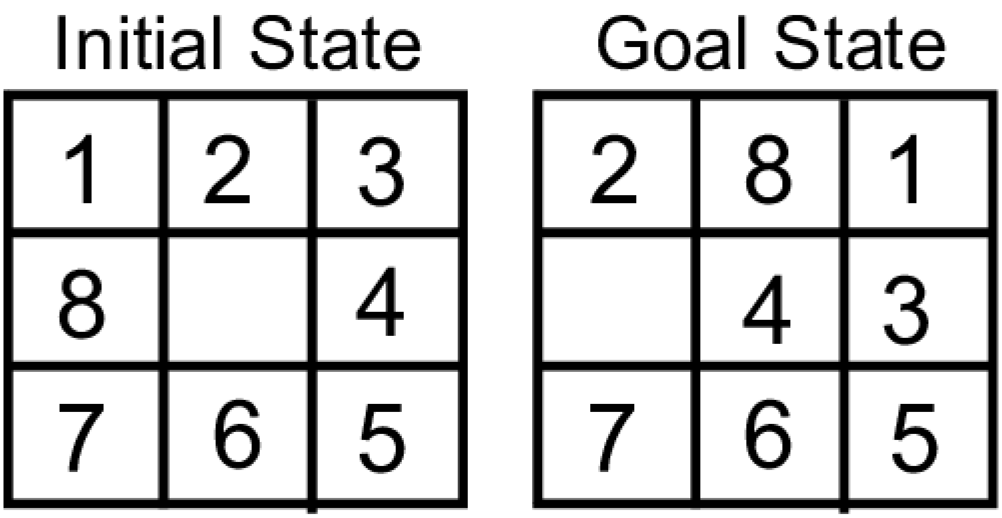
```
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

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.