

**Lab Report-05**

**Course Title: Artificial Intelligence**

**Course Code:** **CSE 366**

**Semester: Fall 2021**

**Section No:** **01**

**Submitted By**

**Name: Syeda Tamanna Sheme**

**ID:2018-2-60-010**

**Submitted To**

**Md Al-Imran**

**Lecturer**

**Department of Computer Science & Engineering**

**Date of Submission:** **1 December, 2021**

# **Lab-05: A\* Algorithm**

# **Theory**

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). Python is named after a TV Show called ëMonty Pythonís Flying Circusí and not after Python-the snake.

Python 3.0 was released in 2008. Although this version is supposed to be backward incompatibles, later on many of its important features have been backported to be compatible with version 2.7.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable.

**Characteristics of Python**

Following are important characteristics of python −

* It supports functional and structured programming methods as well as OOP.
* It provides very high-level dynamic data types and supports dynamic type checking.
* It supports automatic garbage collection.
* It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python is one of the most widely used language over the web.

## A\* Search Algorithm

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. Another aspect that makes A\* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

## The Basic Concept of A\* Algorithm: A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently. All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route. Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as: f(n) = g(n) + h(n), where :

g(n) = cost of traversing from one node to another. This will vary from node to node

h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost

### **A\* Search Algorithms Steps:**

**Step1:** Place the starting node in the OPEN list.

**Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

**Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

**Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

**Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

**Step 6:** Return to **Step 2**.

**The Pseudo-Code of the Algorithm goes like this:**

let the openList equal empty list of nodes

let the closedList equal empty list of nodes

put the startNode on the openList (leave it's f at zero)

while the openList is not empty

    let the currentNode equal the node with the least f value

    remove the currentNode from the openList

    add the currentNode to the closedList

    if currentNode is the goal

        You've found the end!

    let the children of the currentNode equal the adjacent nodes

    for each child in the children

        if child is in the closedList

            continue to beginning of for loop

        child.g = currentNode.g + distance between child and current

        child.h = distance from child to end

        child.f = child.g + child.h

        if child.position is in the openList's nodes positions

            if the child.g is higher than the openList node's g

                continue to beginning of for loop

        add the child to the openList

**A\* Algorithm implementation in Python (Source Code)**

**###First Method:**

**def aStar(start, stop):  
 open\_set = set(start)  
 closed\_set = set()  
 g = {}  
 parents = {}  
  
 g[start] = 0  
 parents[start] = start  
  
 while len(open\_set) > 0:  
 n = None  
  
 for v in open\_set:  
 if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):  
 n = v  
  
 if n == stop or graph\_nodes[n] == None:  
 pass  
 else:  
 for(m, weight) in get\_neighbors(n):  
  
 if m not in open\_set and m not in closed\_set:  
 open\_set.add(m)  
 parents[m] = n  
 g[m] = g[n] + weight**

**else:  
 if g[m] > g[n] + weight:  
  
 g[m] = g[n] + weight  
 parents[m] = n  
  
  
 if m in closed\_set:  
 closed\_set.remove(m)  
 open\_set.add(m)  
  
 if n == None:  
 print("Path does not exist")  
 return None  
  
 if n == stop:  
 path = []  
  
 while parents[n] != n:  
 path.append(n)  
 n = parents[n]  
 path.append(start)  
 path.reverse()  
  
 print("Path found: {}".format(path))  
 return path  
  
 open\_set.remove(n)  
 closed\_set.add(n)  
 print("Path does not exist")  
 return None  
  
  
def get\_neighbors(v):  
 if v in graph\_nodes:  
 return graph\_nodes[v]  
 else:  
 return None**

**def heuristic(n):  
 h\_dist = {  
  
 'A' : 11,  
 'B' : 6,  
 'C' : 99,  
 'D' : 1,  
 'E' : 7,  
 'G' : 0,  
 }  
 return h\_dist[n]  
  
graph\_nodes = {  
  
 'A' : [('B', 2), ('E', 3)],  
 'B' : [('C' , 1), ('G', 9)],  
 'C' : None,  
 'E' : [('D', 6)],  
 'D' : [('G', 1)],  
 }  
  
aStar('A', 'G')**

## Output:

## <Path found: ['A', 'E', 'D', 'G']

**###Second Method:**

**# This class represent a graph  
class Graph:  
 # Initialize the class  
 def \_\_init\_\_(self, graph\_dict=None, directed=True):  
 self.graph\_dict = graph\_dict or {}  
 self.directed = directed  
 if not directed:  
 self.make\_undirected()  
 # Create an undirected graph by adding symmetric edges  
 def make\_undirected(self):  
 for a in list(self.graph\_dict.keys()):  
 for (b, dist) in self.graph\_dict[a].items():  
 self.graph\_dict.setdefault(b, {})[a] = dist  
 # Add a link from A and B of given distance, and also add the inverse link if the graph is undirected  
 def connect(self, A, B, distance=1):  
 self.graph\_dict.setdefault(A, {})[B] = distance  
 if not self.directed:  
 self.graph\_dict.setdefault(B, {})[A] = distance  
 # Get neighbors or a neighbor  
 def get(self, a, b=None):  
 links = self.graph\_dict.setdefault(a, {})  
 if b is None:  
 return links  
 else:  
 return links.get(b)  
 # Return a list of nodes in the graph  
 def nodes(self):  
 s1 = set([k for k in self.graph\_dict.keys()])  
 s2 = set([k2 for v in self.graph\_dict.values() for k2, v2 in v.items()])  
 nodes = s1.union(s2)  
 return list(nodes)  
# This class represent a node  
class Node:  
 # Initialize the class  
 def \_\_init\_\_(self, name:str, parent:str):  
 self.name = name  
 self.parent = parent  
 self.g = 0 # Distance to start node  
 self.h = 0 # Distance to goal node  
 self.f = 0 # Total cost  
 # Compare nodes  
 def \_\_eq\_\_(self, other):  
 return self.name == other.name  
 # Sort nodes  
 def \_\_lt\_\_(self, other):  
 return self.f < other.f  
 # Print node  
 def \_\_repr\_\_(self):  
 return ('({0},{1})'.format(self.name, self.f))  
# A\* search  
def astar\_search(graph, heuristics, start, end):  
  
 # Create lists for open nodes and closed nodes  
 open = []  
 closed = []  
 # Create a start node and an goal node  
 start\_node = Node(start, None)  
 goal\_node = Node(end, None)  
 # Add the start node  
 open.append(start\_node)  
  
 # Loop until the open list is empty  
 while len(open) > 0:  
 # Sort the open list to get the node with the lowest cost first  
 open.sort()  
 # Get the node with the lowest cost  
 current\_node = open.pop(0)  
 # Add the current node to the closed list  
 closed.append(current\_node)  
  
 # Check if we have reached the goal, return the path  
 if current\_node == goal\_node:  
 path = []  
 while current\_node != start\_node:  
 path.append(current\_node.name + ': ' + str(current\_node.g))  
 current\_node = current\_node.parent  
 path.append(start\_node.name + ': ' + str(start\_node.g))  
 # Return reversed path  
 return path[::-1]  
 # Get neighbours  
 neighbors = graph.get(current\_node.name)  
 # Loop neighbors  
 for key, value in neighbors.items():  
 # Create a neighbor node  
 neighbor = Node(key, current\_node)  
 # Check if the neighbor is in the closed list  
 if(neighbor in closed):  
 continue  
 # Calculate full path cost  
 neighbor.g = current\_node.g + graph.get(current\_node.name, neighbor.name)  
 neighbor.h = heuristics.get(neighbor.name)  
 neighbor.f = neighbor.g + neighbor.h  
 # Check if neighbor is in open list and if it has a lower f value  
 if(add\_to\_open(open, neighbor) == True):  
 # Everything is green, add neighbor to open list  
 open.append(neighbor)  
 # Return None, no path is found  
 return None  
# Check if a neighbor should be added to open list  
def add\_to\_open(open, neighbor):  
 for node in open:  
 if (neighbor == node and neighbor.f > node.f):  
 return False  
 return True  
# The main entry point for this module  
def main():  
 # Create a graph  
 graph = Graph()  
 # Create graph connections (Actual distance)  
 graph.connect('Arad', 'Zerind', 75)  
 graph.connect('Timisoara', 'Lugoj', 111)  
 graph.connect('Zerind', 'Oradea', 71)  
 graph.connect('Timisoara', 'Arad', 118)  
 graph.connect('RimnicuVilcea', 'Sibiu', 80)  
 graph.connect('Pitesti', 'RimnicuVilcea', 97)  
 graph.connect('Fagaras', 'Sibiu', 99)  
 graph.connect('Dobreta', 'Mehadia', 75)  
 graph.connect('Bucharest', 'Fagaras', 211)  
 graph.connect('Giurgiu', 'Bucharest', 90)  
 graph.connect('Pitesti', 'Craiova', 70)  
 graph.connect('Timisoara', 'Mehadia', 97)  
 graph.connect('Zerind', 'Sibiu', 151)  
 graph.connect('Fagaras', 'RimnicuVilcea', 80)  
 graph.connect('Pitesti', 'Bucharest', 101)  
 graph.connect('Dobreta', 'Giurgiu', 90)  
 graph.connect('RimnicuVilcea', 'Pitesti', 138)  
 graph.connect('Craiova', 'Bucharest',101 )  
 graph.connect('Sibiu', 'Craiova', 81)  
 graph.connect('Mehadia', 'Craiova', 120)  
 graph.connect('Fagarasu', 'Pitesti', 101)  
 # Make graph undirected, create symmetric connections  
 graph.make\_undirected()  
 # Create heuristics (straight-line distance, air-travel distance)  
 heuristics = {}  
 heuristics['Giurgiu'] = 204  
 heuristics['Pitesti'] = 101  
 heuristics['Fagaras'] = 211  
 heuristics['Craiova'] = 120  
 heuristics['Sibiu'] = 99  
 heuristics['RimnicuVilcea'] = 146  
 heuristics['Mehadia'] = 138  
 heuristics['Fagarasu'] = 47  
 heuristics['Timisoara'] = 132  
 heuristics['Passau'] = 257  
 heuristics['Zerind'] = 168  
 heuristics['Dobreta'] = 70  
 heuristics['Oradea'] = 75  
 heuristics['Bucharest'] = 0  
 # Run the search algorithm  
 path = astar\_search(graph, heuristics, 'RimnicuVilcea', 'Zerind')  
 print(path)  
 print()  
# Tell python to run main method  
if \_\_name\_\_ == "\_\_main\_\_": main()**

## Output:

## ['RimnicuVilcea: 0', 'Sibiu: 80', 'Zerind: 231']

### **Advantages:**

* A\* search algorithm is the best algorithm than other search algorithms.
* A\* search algorithm is optimal and complete.
* This algorithm can solve very complex problems.

**Time Complexity:** The time complexity of A\* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

**Space Complexity:** The space complexity of A\* search algorithm is **O(b^d)**

**Conclusion:** A\* is a very powerful algorithm with almost unlimited potential. However, it is only as good as its heuristic function, which can be highly variable considering the nature of a problem. It has found applications in many software systems, from Machine Learning and Search Optimization to game development where NPC characters navigate through complex terrain and obstacles to reach the player.

## Results

After doing the lab task and lab work we learn -A\* Search algorithm. Now we are able to do programs related to these topics

## Discussion

Python is a general-purpose, versatile and popular programming language. It’s great as a first language because it is concise and easy to read, and it is also a good language to have in any programmer’s stack as it can be used for everything from web development to software development and data science applications.

This lab task is a great introduction to both fundamental programming concepts and the Python programming language. Python 3 is the most up-to-date version of the language with many improvements made to increase the efficiency and simplicity of the code that we write. Dayby-day, python new version are realising and all the new versions have new features and these new features are better than previous one. So, finally I can say that python will be more user friendly in future.