

**Faculty of Engineering & Technology**

**Department of Electrical and Computer Engineering**

**Artificial Intelligence**

**ENCS3340**

***Report of Project***

***Search Algorithms for Route Navigation***

**Prepared by:**

**Maysam Khatib 1190207 Majd Abubaha 1190069**

**Section #4**

**Instructor: Dr. Yazan Abu Farha**

**Date: 17/5/2022**

# **Abstract**

This project tries to put what we've learned about search algorithms including breadth, depth, uniform, greedy, and a\* algorithms into practice.

The purpose of this project is to use search algorithms to determine the most efficient route connecting cities in Palestine.

**Contents**

[**1.** **Abstract** 2](#_Toc103724184)

[**2.** **Theory** 5](#_Toc103724185)

[Breadth First Search 5](#_Toc103724186)

[Depth First Search 5](#_Toc103724187)

[Greedy Search 5](#_Toc103724188)

[Uniform Cost Search 6](#_Toc103724189)

[A\* Search 7](#_Toc103724190)

[**3.** **Procedure and Discussion** 8](#_Toc103724191)

[createGraphs function 8](#_Toc103724192)

[H1 & H2 functions 9](#_Toc103724193)

[getPath function 9](#_Toc103724194)

[greedyBestFirstSearch functions 10](#_Toc103724195)

[breadthFirstSearch function 10](#_Toc103724196)

[depthFirstSearch function 11](#_Toc103724197)

[Uniform function 11](#_Toc103724198)

[A\* function 12](#_Toc103724199)

[**4.** **Results** 13](#_Toc103724200)

[**5.** **Appendix** 14](#_Toc103724201)

# **Theory**

## Breadth First Search

There are several methods for traversing graphs. BFS is the most often utilized method.

BFS is a graph traversal technique in which you start at a certain node (source or beginning node) and traverse the graph layer by layer, investigating the neighbors (nodes which are directly connected to source node). Then you must go to the next-level neighbor nodes.

Before moving on to the next layer of nodes in BFS, you must traverse all of the nodes in the previous layer.

## Depth First Search

The DFS algorithm is a backtracking recursive algorithm. It entails conducting extensive searches of all nodes, either forward if possible or backward if necessary.

Backtracking is travelling backwards on the same path when there are no more nodes to traverse. On the current path, all nodes will be visited until all unvisited nodes have been walked, at which point the new path will be chosen.

The fundamental concept is this:

Choose a beginning node and stack all of its neighbors. To visit the next node, pop one from the stack and place all of its nearby nodes into a stack, then repeat until the stack is depleted, and make sure the nodes you visit are indicated. You won't be able to access the same node twice, also, you can wind yourself in an unending loop if you don't mark the nodes you visit and visit the same node several times.

## Greedy Search

A greedy algorithm, as the name implies, always makes the best decision available at the time. This means that it makes a locally optimum decision in the hopes of achieving a globally optimal result.

if we have an objective function that has to be optimized (maximized or minimized) at a specific place A greedy algorithm takes greedy decisions at each step in order to optimize the objective function. The Greedy algorithm only has one chance to find the best answer, therefore it never goes back and reverses the choice.

Greedy algorithms offer both benefits and drawbacks:

A greedy algorithm (or perhaps numerous greedy algorithms) for a problem is very simple to create. Furthermore, assessing the run time of greedy algorithms will be considerably easier than studying the run time of other approaches (like Divide and conquer). It's unclear whether the Divide and conquer strategy is speedy or sluggish. This is because the size of the issue shrinks as the number of sub-problems grows at each level of recursion.

Working considerably harder to grasp accuracy difficulties for greedy algorithms is the challenging part. Even if the algorithm is right, proving why it is correct is difficult. Proving the correctness of a greedy algorithm is more of an art than a science. It necessitates a great deal of imagination.

## Uniform Cost Search

Uniform Cost Search is the best algorithm for solving a search issue without using heuristics. It can solve any generic graph at the lowest possible cost. Uniform Cost Search, as the name implies, looks for branches with similar costs.

Uniform Cost Search requires the usage of a priority queue once again. Remember that under Depth First Search, the depth up to a certain node was the priority, and the path from the root to the node was the element saved. The priority queue utilized here is similar, with the cumulative cost up to the node determining the priority. Uniform Cost Search, unlike Depth First Search, prioritizes the lowest cumulative cost.

## A\* Search

It's a useful algorithm for determining the shortest path to follow when traversing a map. A\* was created as a graph traversal issue to aid in the development of a self-navigating robot. It is still a commonly used approach for traversing graphs.

It prioritizes shorter pathways, making it an optimum and comprehensive algorithm. An optimum algorithm will discover the cheapest solution to a problem, but a full algorithm would locate all potential solutions.

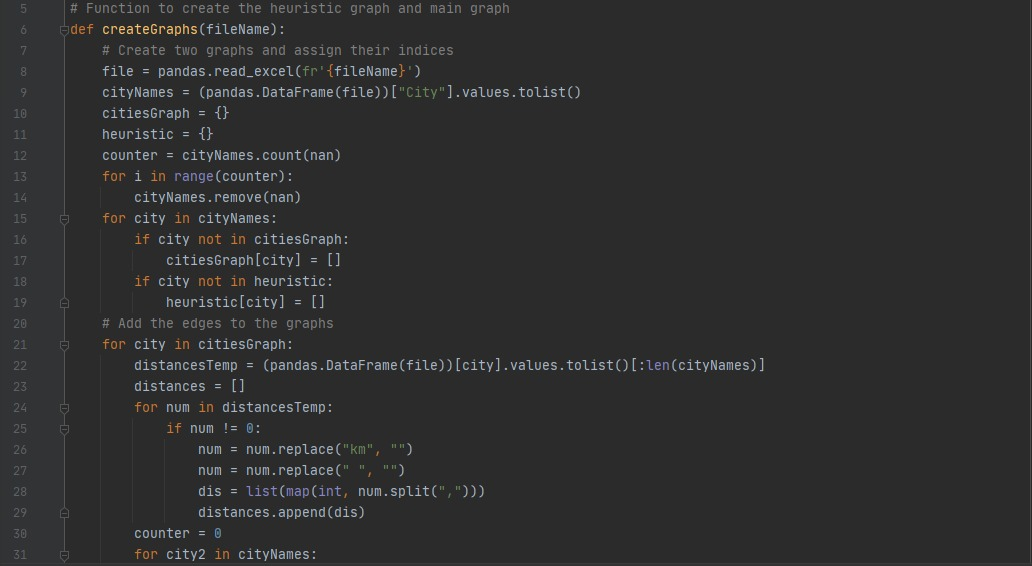
The use of weighted graphs in the algorithm's implementation is another feature that makes it so strong. The cost of each path or course of action is represented by numbers in a weighted graph. This implies the algorithms can determine the cheapest way and the most efficient route in terms of distance and time.

# **Procedure and Discussion**

## createGraphs function

In the beginning, we defined the primary graph for cities and another for the heuristic in the graph function. We use the names of the cities as a key for the graph of cities, and the value is an empty list, after reading the file and extracting the names of the cities.

Then we read the distances and deleted "km", the distances are either two or three numbers, the first for loop for each city alone, if we find three distances, we have an edge between them and we add them to the graph of cities, and if we only find two, we don't have an edge between them.



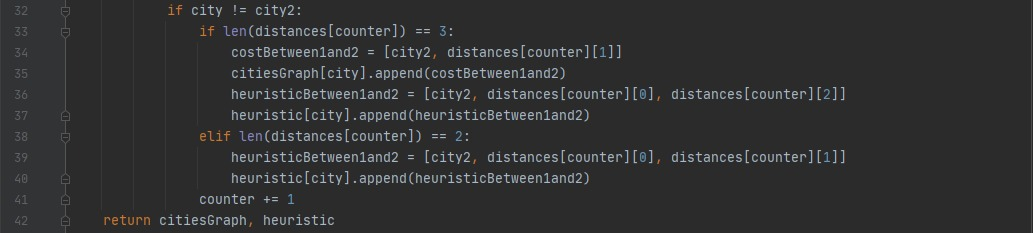


Figure 1. createGraph function

## H1 & H2 functions

If the user wants to use heuristic1 or heuristic2, then we return its value through the h1 and h2 functions.

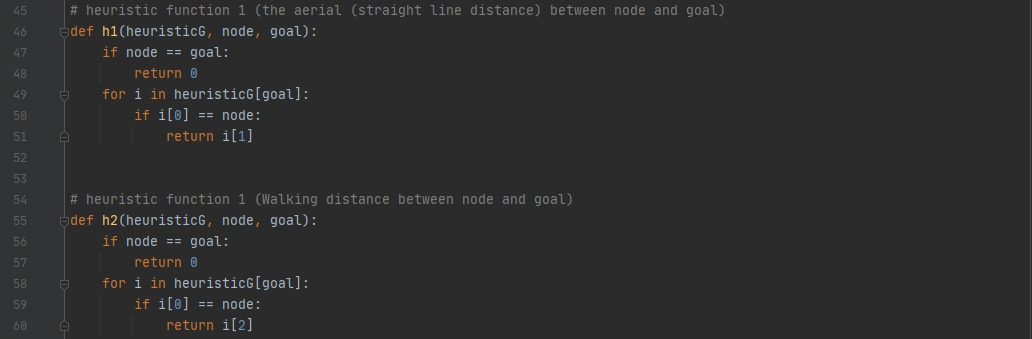


Figure 2. Heuristic functions

## getPath function

The purpose of the getPath function is to save the cities visited; we save each city from whichever point we arrived so that we can access the path; we save so that the son is the key and the father is the value, and we set the father to the start point zero so he quits the loop when reaching it. The beginning position is then saved in the bath, and the output is reflected.

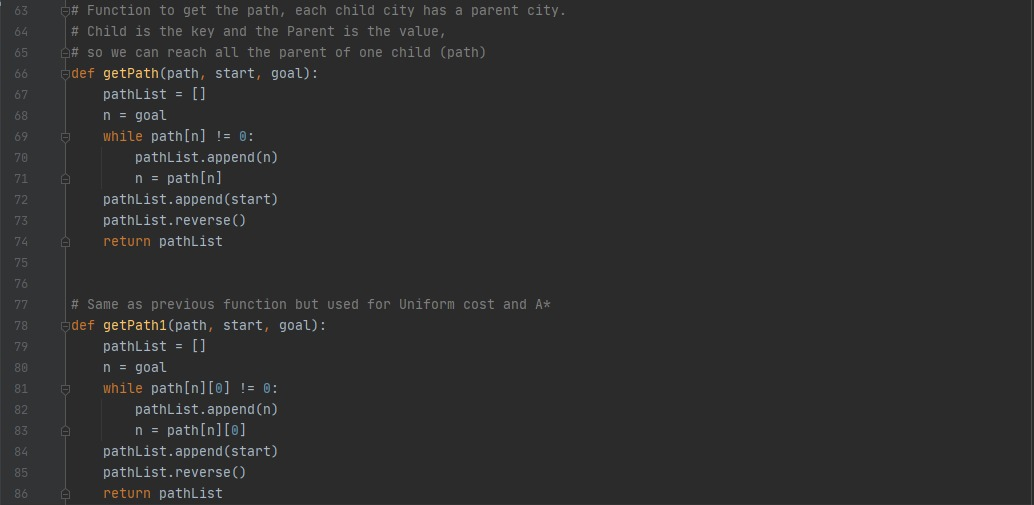


Figure 3. getPath function

## greedyBestFirstSearch functions

The nodes are arranged according to the heuristic value, with the lowest value always taken.

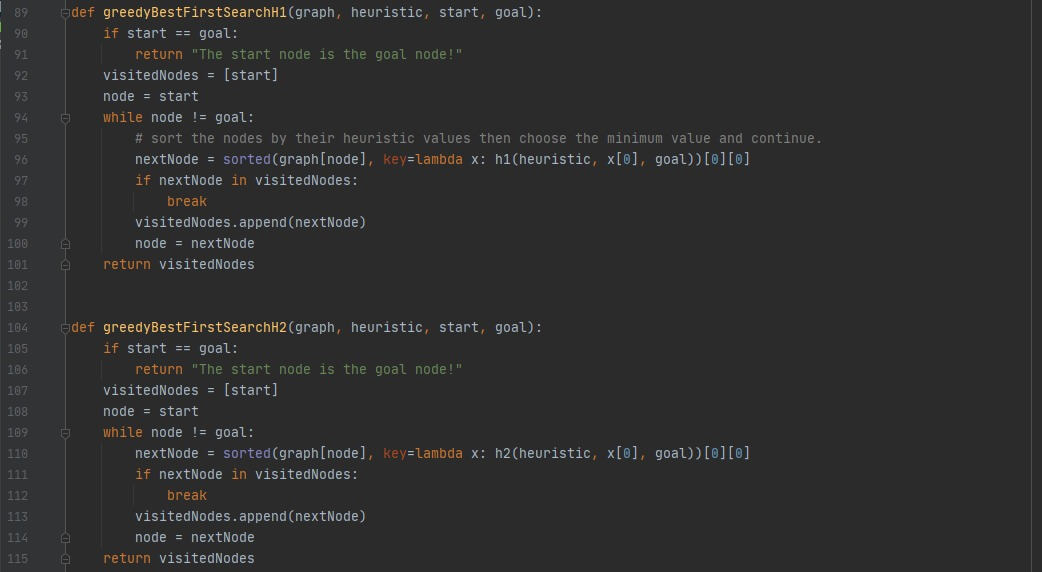


Figure 4. Greedy function

## breadthFirstSearch function

We start by alphabetizing the nodes and checking them one by one. Each one is absent from the visitors we keep track of. We keep them in order to contact their children afterwards. And we verify every node as we increase the depth level.

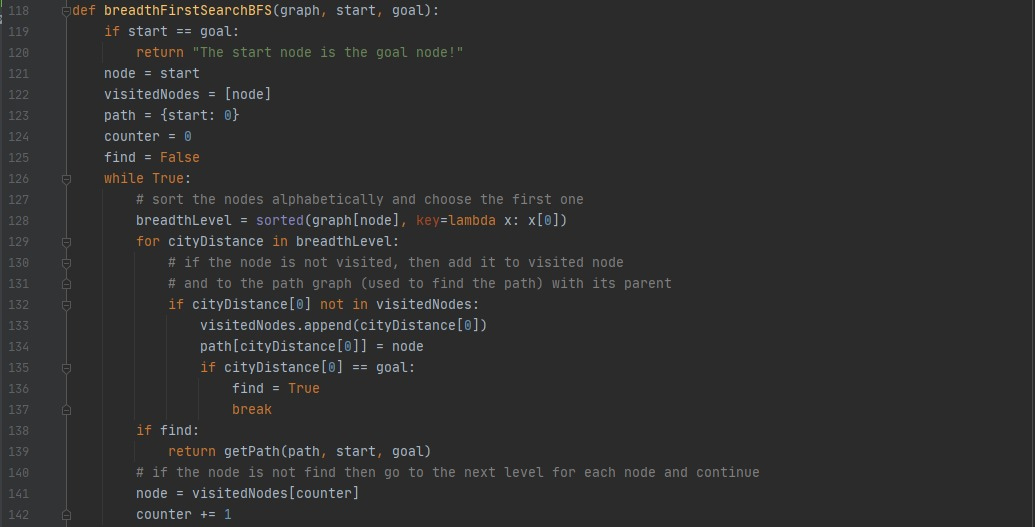


Figure 5. BFS function

## depthFirstSearch function

We utilize the stack in depth, and we add the nodes to the stack in sequence, from Z to A, since the function accepts nodes from the last node on the stack.

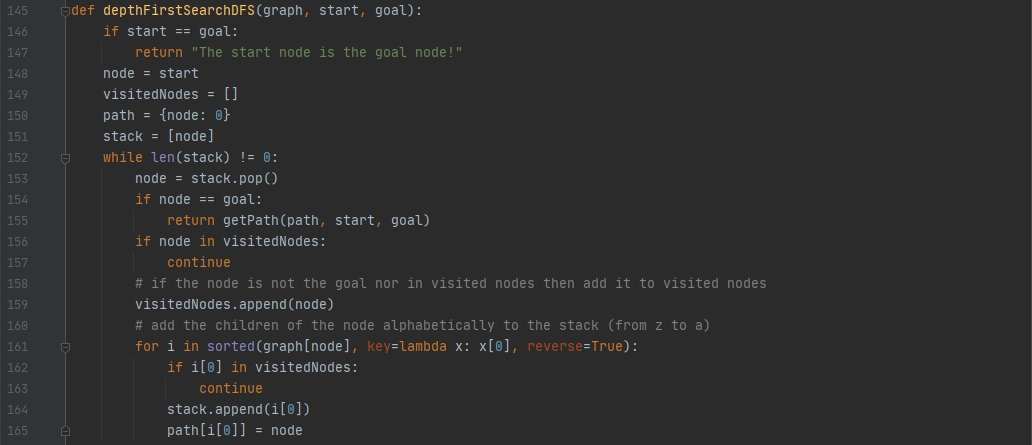


Figure 6. DFS function

## Uniform function

We put the start node with the parent and cost in the path, then we do the same in queue; we sort the queue by cost, and then we choose the first one that is less costly than the others. If one of these two requirements is satisfied, we enter the conditional phrase and enter the new cost and father values.

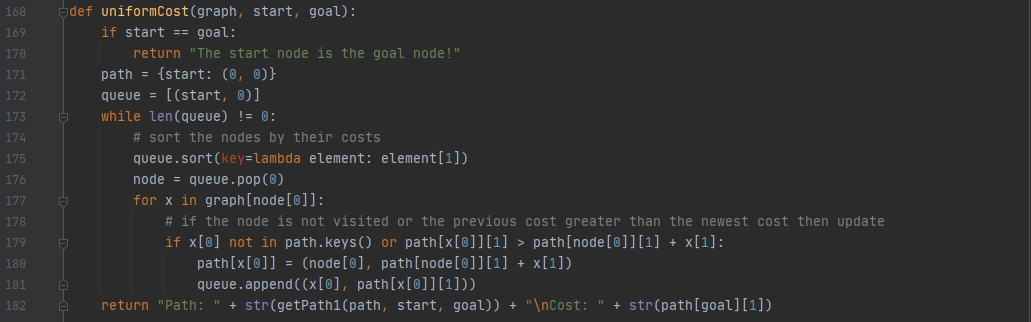
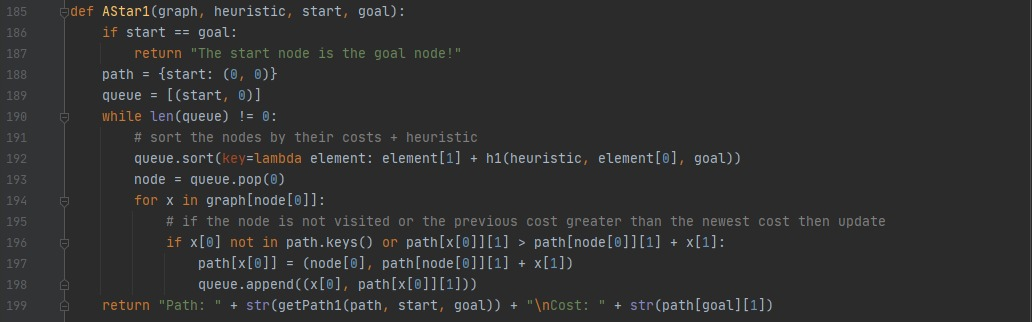


Figure 7. uniform cost function

## A\* function

The A\* algorithm follows the same description as the uniform algorithm, with the exception that the heuristic value is added to the cost.



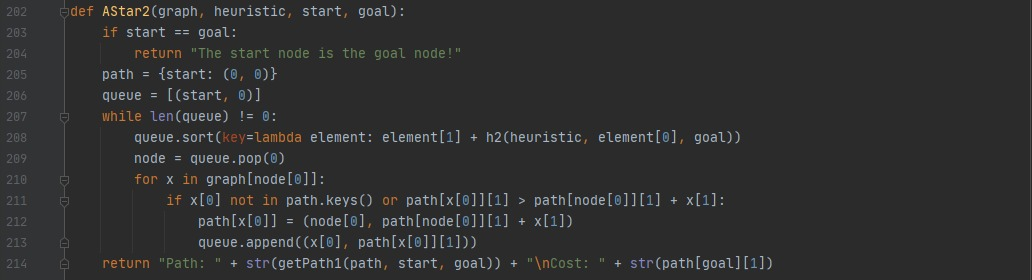
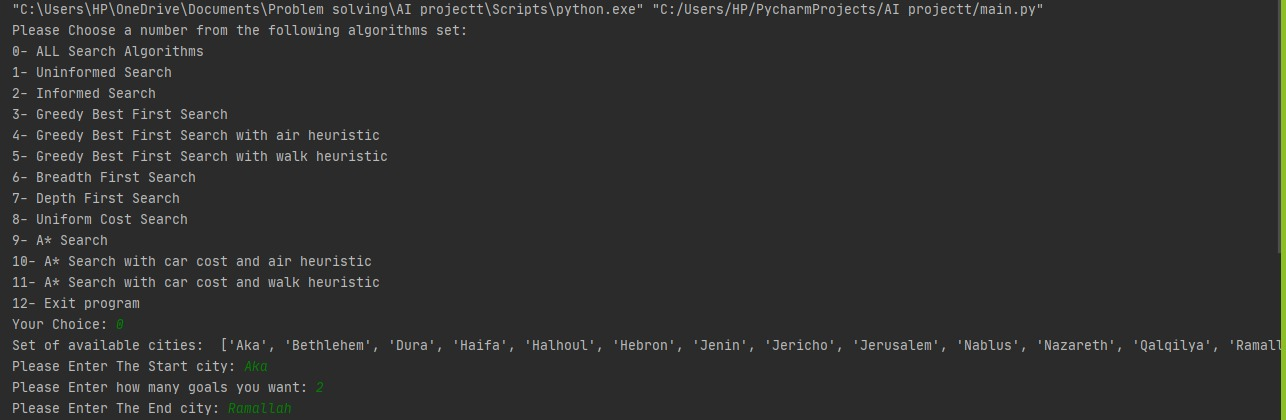
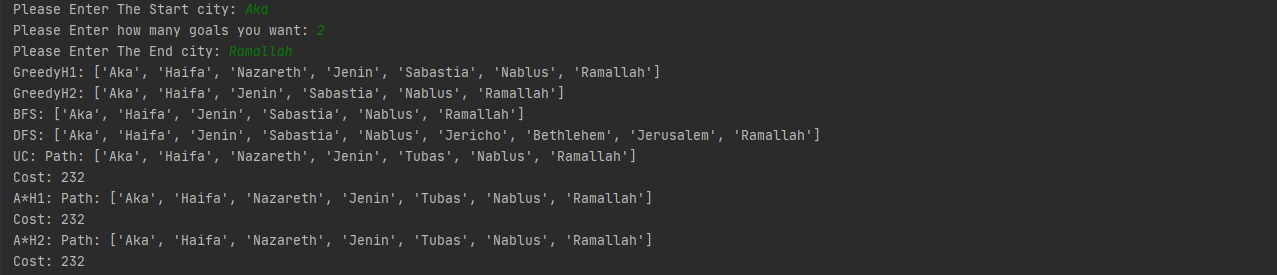
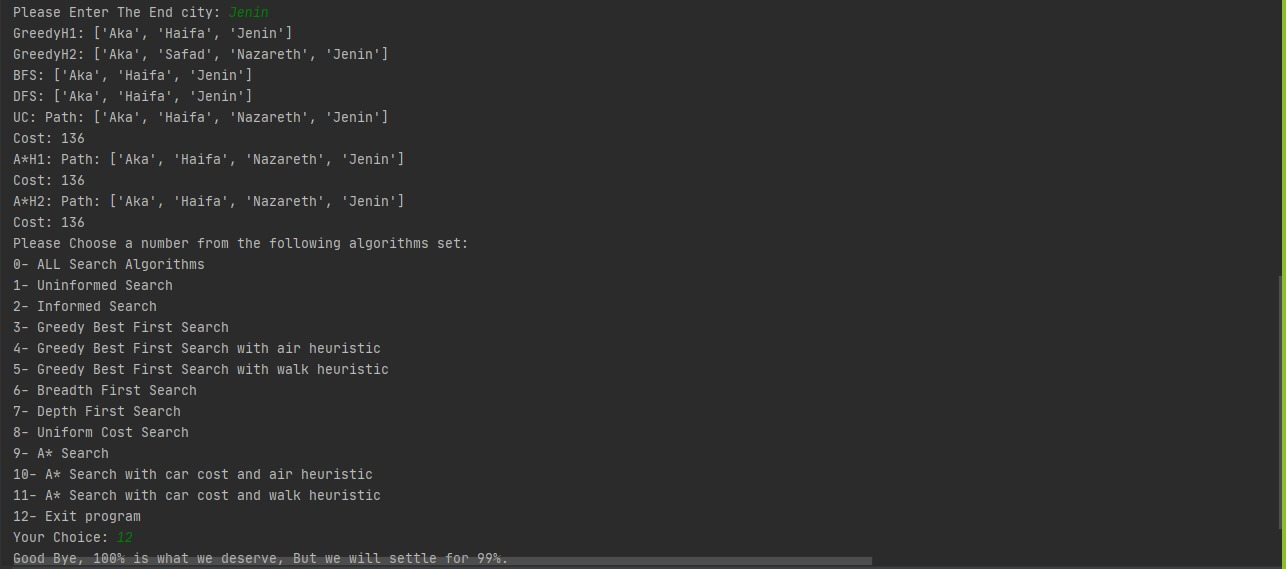


Figure 7. A\* function

# **Results**





****

# **Appendix**

1. import pandas  
   from numpy import nan  
     
     
   # Function to create the heuristic graph and main graph  
   def createGraphs(fileName):  
    # Create two graphs and assign their indices  
    file = pandas.read\_excel(fr'{fileName}')  
    cityNames = (pandas.DataFrame(file))["City"].values.tolist()  
    citiesGraph = {}  
    heuristic = {}  
    counter = cityNames.count(nan)  
    for i in range(counter):  
    cityNames.remove(nan)  
    for city in cityNames:  
    if city not in citiesGraph:  
    citiesGraph[city] = []  
    if city not in heuristic:  
    heuristic[city] = []  
    # Add the edges to the graphs  
    for city in citiesGraph:  
    distancesTemp = (pandas.DataFrame(file))[city].values.tolist()[:len(cityNames)]  
    distances = []  
    for num in distancesTemp:  
    if num != 0:  
    num = num.replace("km", "")  
    num = num.replace(" ", "")  
    dis = list(map(int, num.split(",")))  
    distances.append(dis)  
    counter = 0  
    for city2 in cityNames:  
    if city != city2:  
    if len(distances[counter]) == 3:  
    costBetween1and2 = [city2, distances[counter][1]]  
    citiesGraph[city].append(costBetween1and2)  
    heuristicBetween1and2 = [city2, distances[counter][0], distances[counter][2]]  
    heuristic[city].append(heuristicBetween1and2)  
    elif len(distances[counter]) == 2:  
    heuristicBetween1and2 = [city2, distances[counter][0], distances[counter][1]]  
    heuristic[city].append(heuristicBetween1and2)  
    counter += 1  
    return citiesGraph, heuristic  
     
     
   # heuristic function 1 (the aerial (straight line distance) between node and goal)  
   def h1(heuristicG, node, goal):  
    if node == goal:  
    return 0  
    for i in heuristicG[goal]:  
    if i[0] == node:  
    return i[1]  
     
     
   # heuristic function 1 (Walking distance between node and goal)  
   def h2(heuristicG, node, goal):  
    if node == goal:  
    return 0  
    for i in heuristicG[goal]:  
    if i[0] == node:  
    return i[2]  
     
     
   # Function to get the path, each child city has a parent city.  
   # Child is the key and the Parent is the value,  
   # so we can reach all the parent of one child (path)  
   def getPath(path, start, goal):  
    pathList = []  
    n = goal  
    while path[n] != 0:  
    pathList.append(n)  
    n = path[n]  
    pathList.append(start)  
    pathList.reverse()  
    return pathList  
     
     
   # Same as previous function but used for Uniform cost and A\*  
   def getPath1(path, start, goal):  
    pathList = []  
    n = goal  
    while path[n][0] != 0:  
    pathList.append(n)  
    n = path[n][0]  
    pathList.append(start)  
    pathList.reverse()  
    return pathList  
     
     
   def greedyBestFirstSearchH1(graph, heuristic, start, goal):  
    if start == goal:  
    return "The start node is the goal node!"  
    visitedNodes = [start]  
    node = start  
    while node != goal:  
    # sort the nodes by their heuristic values then choose the minimum value and continue.  
    nextNode = sorted(graph[node], key=lambda x: h1(heuristic, x[0], goal))[0][0]  
    if nextNode in visitedNodes:  
    break  
    visitedNodes.append(nextNode)  
    node = nextNode  
    return visitedNodes  
     
     
   def greedyBestFirstSearchH2(graph, heuristic, start, goal):  
    if start == goal:  
    return "The start node is the goal node!"  
    visitedNodes = [start]  
    node = start  
    while node != goal:  
    nextNode = sorted(graph[node], key=lambda x: h2(heuristic, x[0], goal))[0][0]  
    if nextNode in visitedNodes:  
    break  
    visitedNodes.append(nextNode)  
    node = nextNode  
    return visitedNodes  
     
     
   def breadthFirstSearchBFS(graph, start, goal):  
    if start == goal:  
    return "The start node is the goal node!"  
    node = start  
    visitedNodes = [node]  
    path = {start: 0}  
    counter = 0  
    find = False  
    while True:  
    # sort the nodes alphabetically and choose the first one  
    breadthLevel = sorted(graph[node], key=lambda x: x[0])  
    for cityDistance in breadthLevel:  
    # if the node is not visited, then add it to visited node  
    # and to the path graph (used to find the path) with its parent  
    if cityDistance[0] not in visitedNodes:  
    visitedNodes.append(cityDistance[0])  
    path[cityDistance[0]] = node  
    if cityDistance[0] == goal:  
    find = True  
    break  
    if find:  
    return getPath(path, start, goal)  
    # if the node is not find then go to the next level for each node and continue  
    node = visitedNodes[counter]  
    counter += 1  
     
     
   def depthFirstSearchDFS(graph, start, goal):  
    if start == goal:  
    return "The start node is the goal node!"  
    node = start  
    visitedNodes = []  
    path = {node: 0}  
    stack = [node]  
    while len(stack) != 0:  
    node = stack.pop()  
    if node == goal:  
    return getPath(path, start, goal)  
    if node in visitedNodes:  
    continue  
    # if the node is not the goal nor in visited nodes then add it to visited nodes  
    visitedNodes.append(node)  
    # add the children of the node alphabetically to the stack (from z to a)  
    for i in sorted(graph[node], key=lambda x: x[0], reverse=True):  
    if i[0] in visitedNodes:  
    continue  
    stack.append(i[0])  
    path[i[0]] = node  
     
     
   def uniformCost(graph, start, goal):  
    if start == goal:  
    return "The start node is the goal node!"  
    path = {start: (0, 0)}  
    queue = [(start, 0)]  
    while len(queue) != 0:  
    # sort the nodes by their costs  
    queue.sort(key=lambda element: element[1])  
    node = queue.pop(0)  
    for x in graph[node[0]]:  
    # if the node is not visited or the previous cost greater than the newest cost then update  
    if x[0] not in path.keys() or path[x[0]][1] > path[node[0]][1] + x[1]:  
    path[x[0]] = (node[0], path[node[0]][1] + x[1])  
    queue.append((x[0], path[x[0]][1]))  
    return "Path: " + str(getPath1(path, start, goal)) + "\nCost: " + str(path[goal][1])  
     
     
   def AStar1(graph, heuristic, start, goal):  
    if start == goal:  
    return "The start node is the goal node!"  
    path = {start: (0, 0)}  
    queue = [(start, 0)]  
    while len(queue) != 0:  
    # sort the nodes by their costs + heuristic  
    queue.sort(key=lambda element: element[1] + h1(heuristic, element[0], goal))  
    node = queue.pop(0)  
    for x in graph[node[0]]:  
    # if the node is not visited or the previous cost greater than the newest cost then update  
    if x[0] not in path.keys() or path[x[0]][1] > path[node[0]][1] + x[1]:  
    path[x[0]] = (node[0], path[node[0]][1] + x[1])  
    queue.append((x[0], path[x[0]][1]))  
    return "Path: " + str(getPath1(path, start, goal)) + "\nCost: " + str(path[goal][1])  
     
     
   def AStar2(graph, heuristic, start, goal):  
    if start == goal:  
    return "The start node is the goal node!"  
    path = {start: (0, 0)}  
    queue = [(start, 0)]  
    while len(queue) != 0:  
    queue.sort(key=lambda element: element[1] + h2(heuristic, element[0], goal))  
    node = queue.pop(0)  
    for x in graph[node[0]]:  
    if x[0] not in path.keys() or path[x[0]][1] > path[node[0]][1] + x[1]:  
    path[x[0]] = (node[0], path[node[0]][1] + x[1])  
    queue.append((x[0], path[x[0]][1]))  
    return "Path: " + str(getPath1(path, start, goal)) + "\nCost: " + str(path[goal][1])  
     
     
   Data = 'C:/Users/HP/Downloads/DB\_Cities.xlsx'  
   graphOfCities, heuristicGraph = createGraphs(Data)  
   while True:  
    # Print the menu and get the choice  
    choice = int(input("Please Choose a number from the following algorithms set:\n"  
    "0- ALL Search Algorithms\n"  
    "1- Uninformed Search\n"  
    "2- Informed Search\n"  
    "3- Greedy Best First Search\n"  
    "4- Greedy Best First Search with air heuristic\n"  
    "5- Greedy Best First Search with walk heuristic\n"  
    "6- Breadth First Search\n"  
    "7- Depth First Search\n"  
    "8- Uniform Cost Search\n"  
    "9- A\* Search\n"  
    "10- A\* Search with car cost and air heuristic\n"  
    "11- A\* Search with car cost and walk heuristic\n"  
    "12- Exit program\n"  
    "Your Choice: "))  
    if choice == 12:  
    print("Good Bye, 100% is what we deserve, But we will settle for 99%.")  
    break  
    print("Set of available cities: ", list(graphOfCities.keys()))  
    startNode = input("Please Enter The Start city: ")  
    while startNode not in graphOfCities.keys():  
    print("Please Enter an available city")  
    startNode = input()  
    number\_of\_Goals = int(input("Please Enter how many goals you want: "))  
    for number in range(number\_of\_Goals):  
    goalNode = input("Please Enter The End city: ")  
    while goalNode not in graphOfCities.keys():  
    print("Please Enter an available city")  
    goalNode = input()  
    if choice == 0 or choice == 2 or choice == 3 or choice == 4:  
    print("GreedyH1:", greedyBestFirstSearchH1(graphOfCities, heuristicGraph, startNode, goalNode))  
    if choice == 0 or choice == 2 or choice == 3 or choice == 4:  
    print("GreedyH2:", greedyBestFirstSearchH2(graphOfCities, heuristicGraph, startNode, goalNode))  
    if choice == 0 or choice == 1 or choice == 6:  
    print("BFS:", breadthFirstSearchBFS(graphOfCities, startNode, goalNode))  
    if choice == 0 or choice == 1 or choice == 7:  
    print("DFS:", depthFirstSearchDFS(graphOfCities, startNode, goalNode))  
    if choice == 0 or choice == 2 or choice == 8:  
    print("UC:", uniformCost(graphOfCities, startNode, goalNode))  
    if choice == 0 or choice == 2 or choice == 9 or choice == 10:  
    print("A\*H1:", AStar1(graphOfCities, heuristicGraph, startNode, goalNode))  
    if choice == 0 or choice == 2 or choice == 9 or choice == 11:  
    print("A\*H2:", AStar2(graphOfCities, heuristicGraph, startNode, goalNode))