18CSC305J – ARTIFICIAL INTELLIGENCE LAB

Exp-5: Best First and A* Algorithm for any real world Problem

Submitted by-
Name:- Anannya P. Neog
Reg. No. :- RA1911003010367
Course :- Btech
Section :- F1
Branch:- Computer Science Engineering
Sem:- 6th Sem

AI LAB Ex – 5:- Best First and A* Algorithm for any real world Problem

Team Members:

- ✓ Richa - 357
- ✓ Anannya - 367
- ✓ Pushan - 371
- ✓ Ankit - 372
- ✓ Tanay - 377

Aim:

To implement Best First and A* Algorithm for any real world Problem

Objective:

The objective is to reach the goal from the initial state via the shortest path.

Best First Search Algorithm:

Best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function,

i.e. **f(n) = g(n).**

where, h(n) = estimated cost from node n to the goal.

The best first algorithm is implemented by the priority queue.

Time Complexity and Space Complexity

Best First Search is simply Breadth First Search (BFS) but with the nodes re-ordered by their heuristic value. In the worst case, the time and space complexity for Best First Search are the same as that of BFS: O(b^(d+1)) for Time and O(b^d) for Space.

A* Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a fitness number.

$$f(n) = g(n) + h(n)$$

where, f(n) = estimated cost of the cheapest solution

g(n) = cost to reach node n from start state

h(n) = cost to reach from node n to the goal node

Time Complexity

The time complexity of A* Search Algorithm depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: O(b^d), where b is the branching factor (the average number of successors per state). This assumes that a goal state exists at all, and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate.

Space Complexity

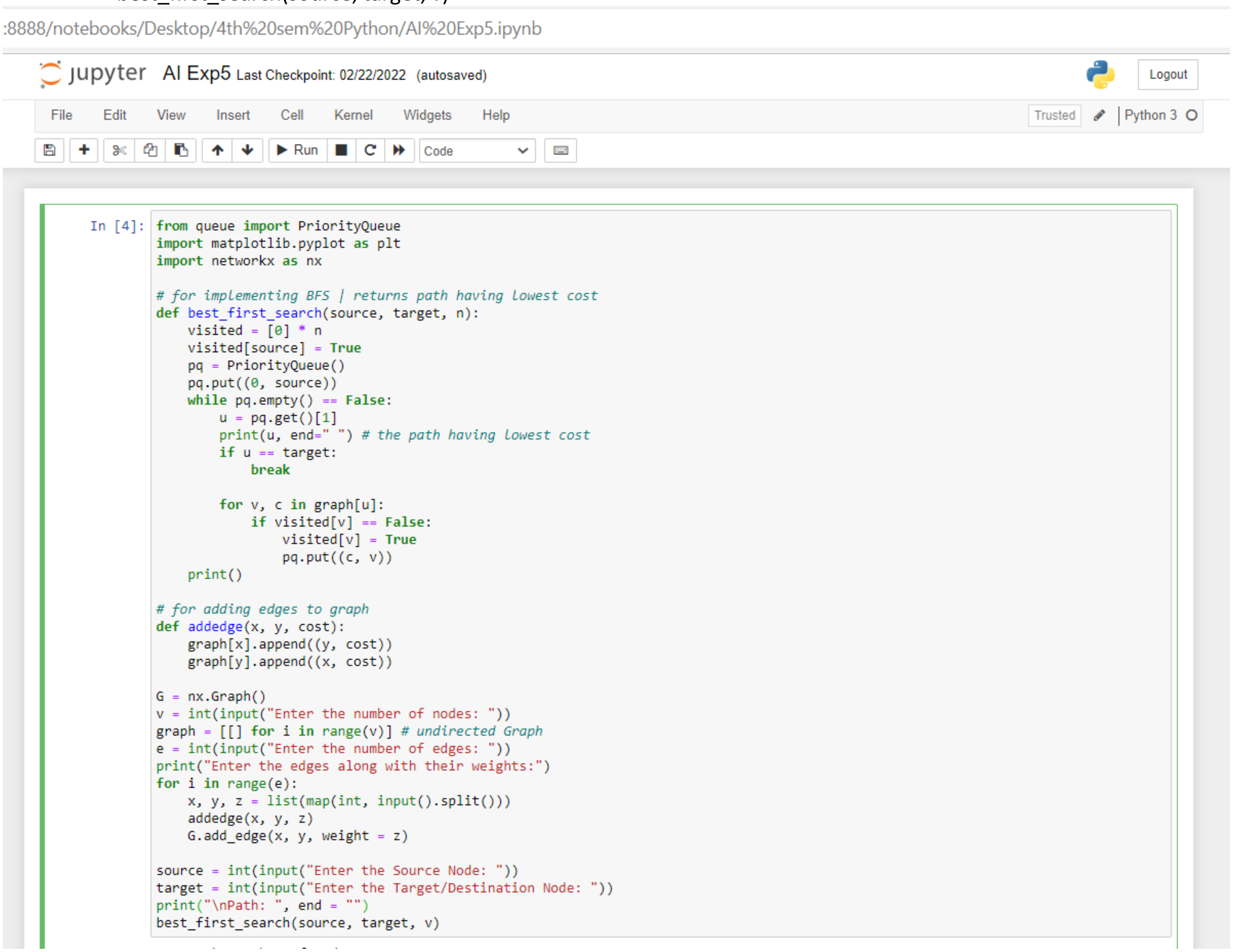
The space complexity of A* Search Algorithm is roughly the same as that of all other graph search algorithms i.e. O(b^d), as it keeps all generated nodes in memory.

Codes:

Best First Search Algorithm:

```
from queue import PriorityQueue
import matplotlib.pyplot as plt
import networkx as nx

# for implementing BFS | returns path having lowest cost
def best_first_search(source, target, n):
    visited = [0] * n
    visited[source] = True
    pq = PriorityQueue()
    pq.put((0, source))
    while pq.empty() == False:
        u = pq.get()[1]
        print(u, end=" ") # the path having lowest cost
        if u == target:
            break

        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()

# for adding edges to graph
```

```
def addedge(x, y, cost):
   graph[x].append((y, cost))
   graph[y].append((x, cost))

G = nx.Graph()
v = int(input("Enter the number of nodes: "))
graph = [[] for i in range(v)] # undirected Graph
e = int(input("Enter the number of edges: "))
print("Enter the edges along with their weights:")
for i in range(e):
   x, y, z = list(map(int, input().split()))
   addedge(x, y, z)
   G.add_edge(x, y, weight = z)

source = int(input("Enter the Source Node: "))
target = int(input("Enter the Target/Destination Node: "))
print("\nPath: ", end = "")
best_first_search(source, target, v)
```



:8888/notebooks/Desktop/4th%20sem%20Python/AI%20Exp5.ipynb

jupyter AI Exp5 Last Checkpoint: 02/22/2022 (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3

Run Code

In [4]:

```
from queue import PriorityQueue
import matplotlib.pyplot as plt
import networkx as nx

# for implementing BFS | returns path having lowest cost
def best_first_search(source, target, n):
    visited = [0] * n
    visited[source] = True
    pq = PriorityQueue()
    pq.put((0, source))
    while pq.empty() == False:
        u = pq.get()[1]
        print(u, end=" ") # the path having lowest cost
        if u == target:
            break

        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()

# for adding edges to graph
def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))

G = nx.Graph()
v = int(input("Enter the number of nodes: "))
graph = [[] for i in range(v)] # undirected Graph
e = int(input("Enter the number of edges: "))
print("Enter the edges along with their weights:")
for i in range(e):
    x, y, z = list(map(int, input().split()))
    addedge(x, y, z)
    G.add_edge(x, y, weight = z)

source = int(input("Enter the Source Node: "))
target = int(input("Enter the Target/Destination Node: "))
print("\nPath: ", end = "")
best_first_search(source, target, v)
```

Output:

```
Enter the number of nodes: 14
Enter the number of edges: 13
Enter the edges along with their weights:
0 1 3
0 2 6
0 3 5
1 4 9
1 5 8
2 6 12
2 7 14
3 8 7
8 9 5
8 10 6
9 11 1
9 12 10
9 13 2
Enter the Source Node: 0
Enter the Target/Destination Node: 9

Path: 0 1 3 2 8 9
```

Graph:

```
print("Graph:\n")
pos = nx.spring_layout(G, seed=7)  # positions for all nodes - seed for reproducibility

# nodes
nx.draw_networkx_nodes(G, pos, node_size=350)

# edges
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_edges(G, pos, alpha=0.5, edge_color="r")

# labels
nx.draw_networkx_labels(G, pos, font_size=20)

ax = plt.gca()
ax.margins(0.08)
plt.axis("off")
plt.tight_layout()
plt.show()
```

localhost:8888/notebooks/Desktop/4th%20sem%20Python/AI%20Exp5.ipynb

jupyter AI Exp5 Last Checkpoint: 02/22/2022 (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3

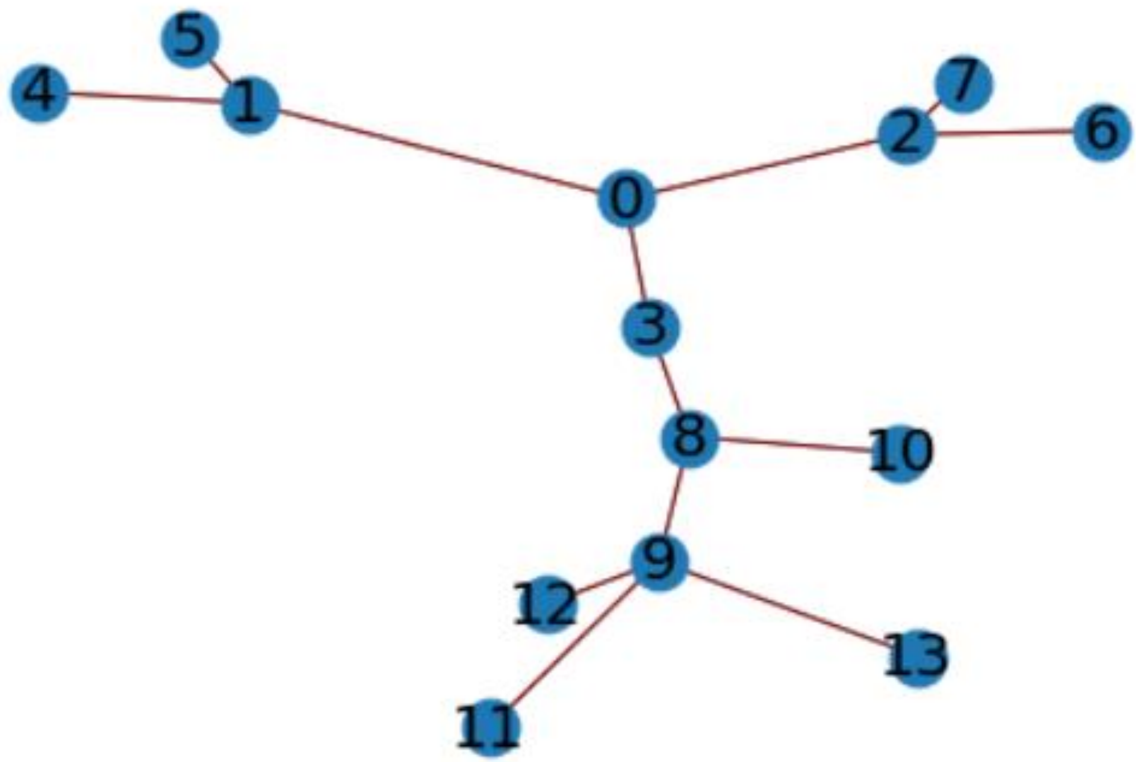
Run Code

In [8]:

```
print("Graph:\n")
pos = nx.spring_layout(G, seed=7)  # positions for all nodes - seed for reproducibility

# nodes
nx.draw_networkx_nodes(G, pos, node_size=350)

# edges
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_edges(G, pos, alpha=0.5, edge_color="r")

# labels
nx.draw_networkx_labels(G, pos, font_size=20)

ax = plt.gca()
ax.margins(0.08)
plt.axis("off")
plt.tight_layout()
plt.show()
```

Output:

Graph:



A* Search Algorithm:

```
# graph class
class Graph:

  # init 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 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 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 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)

# node class
class Node:

    # init 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):

    # lists for open nodes and closed nodes
    open = []
    closed = []

    # a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)

    # add start node
    open.append(start_node)

    # loop until the open list is empty
    while len(open) > 0:


        open.sort()                                  # sort open list to get the node with the lowest cost first
        current_node = open.pop(0)                   # get node with the lowest cost
        closed.append(current_node)                  # add current node to the closed list
```

```
        # 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 path[::-1]


        neighbors = graph.get(current_node.name)    # get neighbours

        # loop neighbors
        for key, value in neighbors.items():
            neighbor = Node(key, current_node)      # create neighbor node
            if(neighbor in closed):                 # check if the neighbor is in the closed list
                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

# create a graph
graph = Graph() # user-based input for edges will be updated in the upcoming days
# create graph connections (Actual distance)
graph.connect('Frankfurt', 'Wurzburg', 111)
graph.connect('Frankfurt', 'Mannheim', 85)
graph.connect('Wurzburg', 'Nurnberg', 104)
graph.connect('Wurzburg', 'Stuttgart', 140)
graph.connect('Wurzburg', 'Ulm', 183)
graph.connect('Mannheim', 'Nurnberg', 230)
graph.connect('Mannheim', 'Karlsruhe', 67)
graph.connect('Karlsruhe', 'Basel', 191)
graph.connect('Karlsruhe', 'Stuttgart', 64)
graph.connect('Nurnberg', 'Ulm', 171)
graph.connect('Nurnberg', 'Munchen', 170)
graph.connect('Nurnberg', 'Passau', 220)
graph.connect('Stuttgart', 'Ulm', 107)
```

```
graph.connect('Basel', 'Bern', 91)
graph.connect('Basel', 'Zurich', 85)
graph.connect('Bern', 'Zurich', 120)
graph.connect('Zurich', 'Memmingen', 184)
graph.connect('Memmingen', 'Ulm', 55)
graph.connect('Memmingen', 'Munchen', 115)
graph.connect('Munchen', 'Ulm', 123)
graph.connect('Munchen', 'Passau', 189)
graph.connect('Munchen', 'Rosenheim', 59)
graph.connect('Rosenheim', 'Salzburg', 81)
graph.connect('Passau', 'Linz', 102)
graph.connect('Salzburg', 'Linz', 126)
# make graph undirected, create symmetric connections
graph.make_undirected()
# create heuristics (straight-line distance, air-travel distance)
heuristics = {}
heuristics['Basel'] = 204
heuristics['Bern'] = 247
heuristics['Frankfurt'] = 215
heuristics['Karlsruhe'] = 137
heuristics['Linz'] = 318
heuristics['Mannheim'] = 164
heuristics['Munchen'] = 120
heuristics['Memmingen'] = 47
heuristics['Nurnberg'] = 132
heuristics['Passau'] = 257
heuristics['Rosenheim'] = 168
heuristics['Stuttgart'] = 75
heuristics['Salzburg'] = 236
heuristics['Wurzburg'] = 153
heuristics['Zurich'] = 157
heuristics['Ulm'] = 0
# run the search algorithm
path = astar_search(graph, heuristics, 'Frankfurt', 'Ulm')
print("Path:", path)
```

localhost:8888/notebooks/Desktop/4th%20sem%20Python/AI%20Exp5.ipynb

jupyter AI Exp5 Last Checkpoint: 02/22/2022 (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted Python 3

Run Code

In [4]:
```
# graph class
class Graph:

    # init 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 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 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 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)
```

localhost:8888/notebooks/Desktop/4th%20sem%20Python/AI%20Exp5.ipynb

jupyter AI Exp5 Last Checkpoint: 02/22/2022 (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted Python 3

Run Code

```
# node class
class Node:

    # init 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):

    # lists for open nodes and closed nodes
    open = []
    closed = []

    # a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)

    # add start node
    open.append(start_node)

    # loop until the open list is empty
    while len(open) > 0:
```

localhost:8888/notebooks/Desktop/4th%20sem%20Python/AI%20Exp5.ipynb

jupyter AI Exp5 Last Checkpoint: 02/22/2022 (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

```
        open.sort()                                  # sort open list to get the node with the lowest cost first
        current_node = open.pop(0)                   # get node with the lowest cost
        closed.append(current_node)                  # add current node to the closed list

        # 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 path[::-1]


        neighbors = graph.get(current_node.name)     # get neighbours

        # loop neighbors
        for key, value in neighbors.items():
            neighbor = Node(key, current_node)       # create neighbor node
            if(neighbor in closed):                  # check if the neighbor is in the closed list
                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
```

localhost:8888/notebooks/Desktop/4th%20sem%20Python/AI%20Exp5.ipynb

jupyter AI Exp5 Last Checkpoint: 02/22/2022 (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

```
# create a graph
graph = Graph() # user-based input for edges will be updated in the upcoming days
# create graph connections (Actual distance)
graph.connect('Frankfurt', 'Wurzburg', 111)
graph.connect('Frankfurt', 'Mannheim', 85)
graph.connect('Wurzburg', 'Nurnberg', 104)
graph.connect('Wurzburg', 'Stuttgart', 140)
graph.connect('Wurzburg', 'Ulm', 183)
graph.connect('Mannheim', 'Nurnberg', 230)
graph.connect('Mannheim', 'Karlsruhe', 67)
graph.connect('Karlsruhe', 'Basel', 191)
graph.connect('Karlsruhe', 'Stuttgart', 64)
graph.connect('Nurnberg', 'Ulm', 171)
graph.connect('Nurnberg', 'Munchen', 170)
graph.connect('Nurnberg', 'Passau', 220)
graph.connect('Stuttgart', 'Ulm', 107)
graph.connect('Basel', 'Bern', 91)
graph.connect('Basel', 'Zurich', 85)
graph.connect('Bern', 'Zurich', 120)
graph.connect('Zurich', 'Memmingen', 184)
graph.connect('Memmingen', 'Ulm', 55)
graph.connect('Memmingen', 'Munchen', 115)
graph.connect('Munchen', 'Ulm', 123)
graph.connect('Munchen', 'Passau', 189)
graph.connect('Munchen', 'Rosenheim', 59)
graph.connect('Rosenheim', 'Salzburg', 81)
graph.connect('Passau', 'Linz', 102)
graph.connect('Salzburg', 'Linz', 126)
```

localhost:8888/notebooks/Desktop/4th%20sem%20Python/AI%20Exp5.ipynb

jupyter AI Exp5 Last Checkpoint: 02/22/2022 (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted Python 3

Run Code

```
# make graph undirected, create symmetric connections
graph.make_undirected()
# create heuristics (straight-line distance, air-travel distance)
heuristics = {}
heuristics['Basel'] = 204
heuristics['Bern'] = 247
heuristics['Frankfurt'] = 215
heuristics['Karlsruhe'] = 137
heuristics['Linz'] = 318
heuristics['Mannheim'] = 164
heuristics['Munchen'] = 120
heuristics['Memmingen'] = 47
heuristics['Nurnberg'] = 132
heuristics['Passau'] = 257
heuristics['Rosenheim'] = 168
heuristics['Stuttgart'] = 75
heuristics['Salzburg'] = 236
heuristics['Wurzburg'] = 153
heuristics['Zurich'] = 157
heuristics['Ulm'] = 0
# run the search algorithm
path = astar_search(graph, heuristics, 'Frankfurt', 'Ulm')
print("Path:", path)
```

Output:

```
Path: ['Frankfurt: 0', 'Wurzburg: 111', 'Ulm: 294']
```