Artificial Intelligence  
Lab Exercise 5  
Best First Search & A\* Algorithm

short line

Ashwin Prakash  
RA1911026010048

**Best First Search:**

The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

The two variants of Best First Search are **Greedy Best First Search** and **A\* Best First Search**. The Greedy BFS algorithm selects the path which appears to be the best, it can be known as the combination of depth-first search and breadth-first search. Greedy BFS makes use of the Heuristic function and search and allows us to take advantage of both algorithms.

**Aim:**

To implement and analyse the Best First Search Algorithm.

**Algorithm:**

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the ‘ordered’ OPEN list
3. Repeat the next steps until GOAL node is reached

* If OPEN list is empty, return False and exit the loop
* Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also, capture the information of the parent node
* If N is a GOAL node, then move the node to the Closed list and exit the loop returning ‘True’. The solution can be found by backtracking the path
* If N is not the GOAL node, expand node N to generate the ‘immediate’ next nodes or immediate children nodes linked to node N and add all those to the OPEN list
* Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

**Program:**

class Graph:

    def \_\_init\_\_(self, graph\_dict=None, directed=True):

        self.graph\_dict = graph\_dict or {}

        self.directed = directed

        if not directed:

            self.make\_undirected()

    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

    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

    def get(self, a, b=None):

        links = self.graph\_dict.setdefault(a, {})

        if b is None:

            return links

        else:

            return links.get(b)

    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)

class Node:

    def \_\_init\_\_(self, name:str, parent:str):

        self.name = name

        self.parent = parent

        self.g = 0

        self.h = 0

        self.f = 0

    # 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.position, self.f))

# Best-first search

def best\_first\_search(graph, heuristics, start, end):

    open = []

    closed = []

    start\_node = Node(start, None)

    goal\_node = Node(end, None)

    open.append(start\_node)

    while len(open) > 0:

        open.sort()

        current\_node = open.pop(0)

        closed.append(current\_node)

        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)

        for key, value in neighbors.items():

            neighbor = Node(key, current\_node)

            if(neighbor in closed):

                continue

            neighbor.g = current\_node.g + graph.get(current\_node.name, neighbor.name)

            neighbor.h = heuristics.get(neighbor.name)

            neighbor.f = neighbor.h

            if(add\_to\_open(open, neighbor) == True):

                open.append(neighbor)

    return None

def add\_to\_open(open, neighbor):

    for node in open:

        if (neighbor == node and neighbor.f >= node.f):

            return False

    return True

def main():

    graph = Graph()

    n = int(input('Enter No: of Graph Nodes: '))

    m = int(input('Enter No: of Connections: '))

    print('Enter the Graph Nodes (First Node, Second Node, Distance/Cost)')

    for \_ in range(m):

        f\_node, s\_node, cost = input().split()

        cost = int(cost)

        graph.connect(f\_node, s\_node, cost)

    heuristics = {}

    print('Enter the Heuristic Values (Node, Value)')

    for \_ in range(n):

        nnode, Hvalue = input().split()

        Hvalue = int(Hvalue)

        heuristics[nnode] = Hvalue

    graph.make\_undirected()

    start = input('Enter the Start state: ')

    goal = input('Enter the Goal state: ')

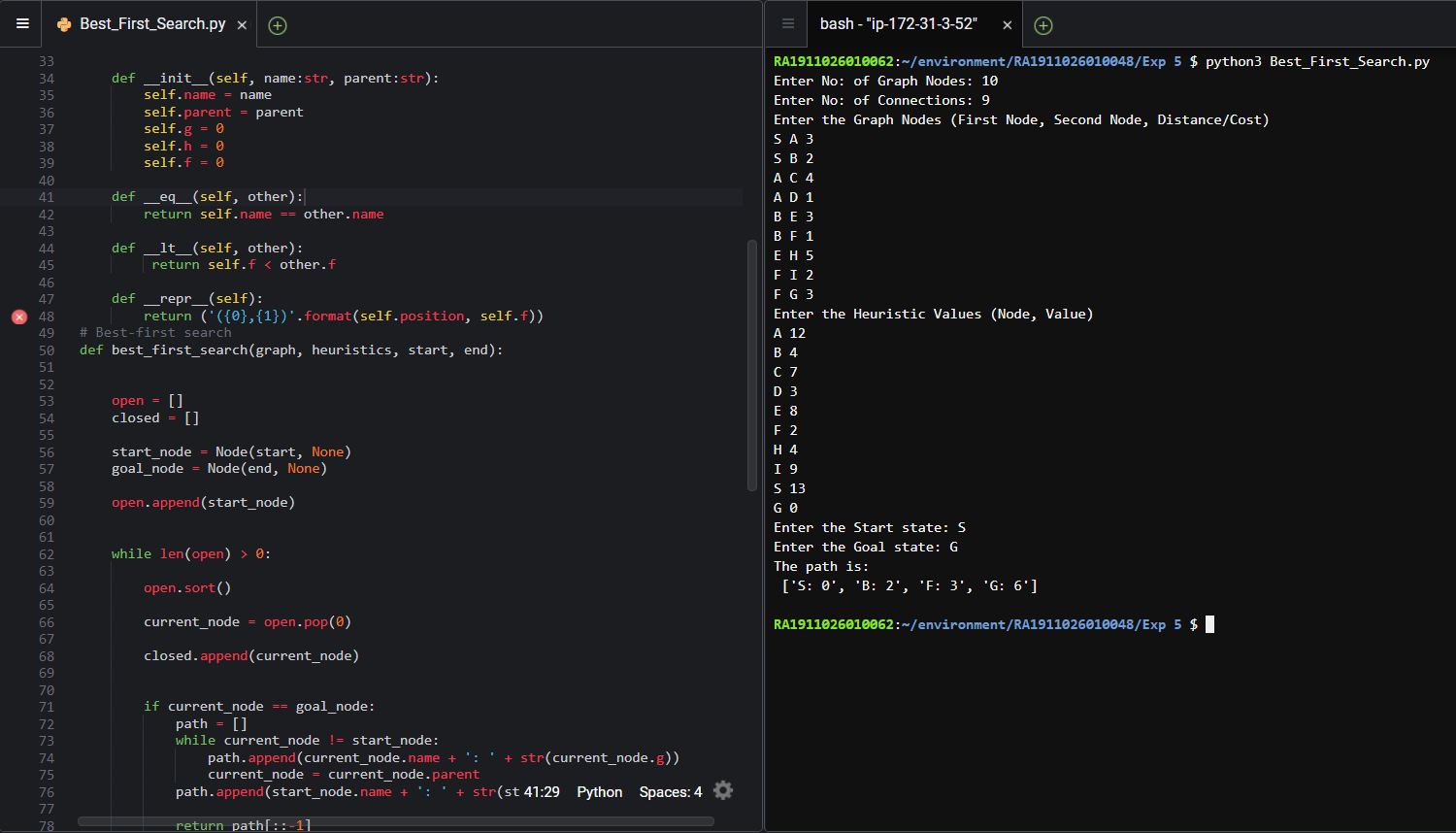
    path = best\_first\_search(graph, heuristics, start, goal)

    print('The path is: \n',path)

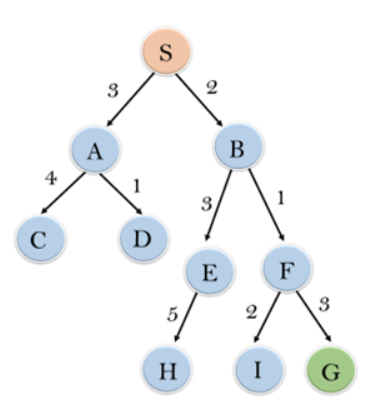
    print()

if \_\_name\_\_ == "\_\_main\_\_": main()

**Output:**



**Observations:**

****

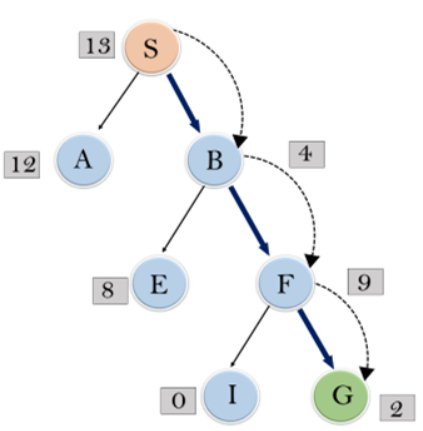
Heuristic Values

****

Here, the Start state is **S** and the Goal state is **G**. The start state is inserted into OPEN list. The first value in OPEN list is popped, and adds its immediate children node, sorted according to heuristic values, to the OPEN list. The popped value is inserted into CLOSED list. The above step is repeated until we reach the Goal state after which backtracking gives us the path travelled.

|  |  |  |
| --- | --- | --- |
| Open | Closed | Iterations |
| B, A | S | Start |
| A  F, E, A | S, B | 1 |
| E, A  G, E, A | S, B, F | 2 |
|  | S, B, F, G | Stop |

Path Traversed: **S🡪B🡪 F🡪 G**



**Inference:**

Best First Search can switch between DFS and BFS to get a greedy approach to the solution. On an average it performs better than DFS but in worst case, it has the time complexity of **O(bm)**, Where, m is the maximum depth of the search space.

**Result:**

We have implemented the Best First Search algorithm and analysed it.

**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 solves 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.

**Aim:**

To implement and analyse the A\* Search algorithm.

**Algorithm:**

1. Place the starting node in the OPEN list.
2. Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
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
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.
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.
6. Repeat steps from 2 - 6 until we reach the goal state

**Program:**

class Graph:

    def \_\_init\_\_(self, graph\_dict=None, directed=True):

        self.graph\_dict = graph\_dict or {}

        self.directed = directed

        if not directed:

            self.make\_undirected()

    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

    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

    def get(self, a, b=None):

        links = self.graph\_dict.setdefault(a, {})

        if b is None:

            return links

        else:

            return links.get(b)

    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)

class Node:

    =

    def \_\_init\_\_(self, name:str, parent:str):

        self.name = name

        self.parent = parent

        self.g = 0

        self.h = 0

        self.f = 0

    # 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):

    open = []

    closed = []

    start\_node = Node(start, None)

    goal\_node = Node(end, None)

    open.append(start\_node)

    while len(open) > 0:

        open.sort()

        current\_node = open.pop(0)

        closed.append(current\_node)

        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)

        for key, value in neighbors.items():

            neighbor = Node(key, current\_node)

            if(neighbor in closed):

                continue

            neighbor.g = current\_node.g + graph.get(current\_node.name, neighbor.name)

            neighbor.h = heuristics.get(neighbor.name)

            neighbor.f = neighbor.g + neighbor.h

            if(add\_to\_open(open, neighbor) == True):

                open.append(neighbor)

    return None

def add\_to\_open(open, neighbor):

    for node in open:

        if (neighbor == node and neighbor.f > node.f):

            return False

    return True

def main():

    # Create a graph

    graph = Graph()

    n = int(input('Enter No: of Graph Nodes: '))

    m = int(input('Enter No: of Connections: '))

    print('Enter the Graph Nodes (First Node, Second Node, Distance/Cost)')

    for \_ in range(m):

        f\_node, s\_node, cost = input().split()

        cost = int(cost)

        graph.connect(f\_node, s\_node, cost)

    heuristics = {}

    print('Enter the Heuristic Values (Node, Value)')

    for \_ in range(n):

        nnode, Hvalue = input().split()

        Hvalue = int(Hvalue)

        heuristics[nnode] = Hvalue

    graph.make\_undirected()

    start = input('Enter the Start state: ')

    goal = input('Enter the Goal state: ')

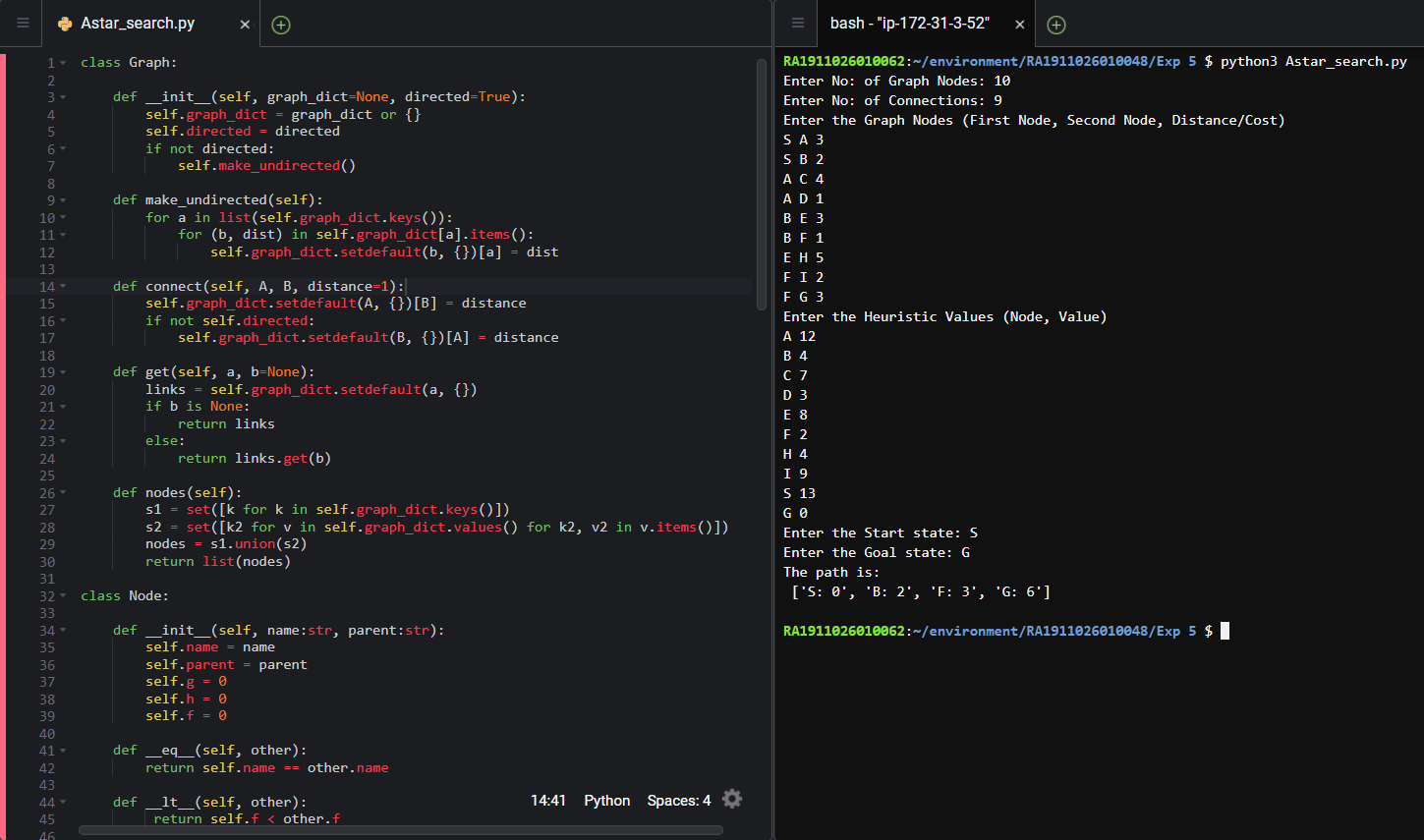
    path = astar\_search(graph, heuristics, start, goal)

    print('The path is: \n',path)

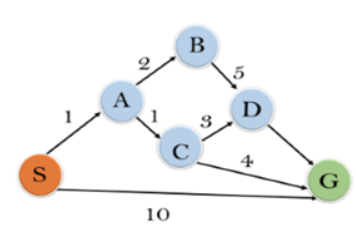
    print()

if \_\_name\_\_ == "\_\_main\_\_": main()

**Output:**

****

**Observations:**

****

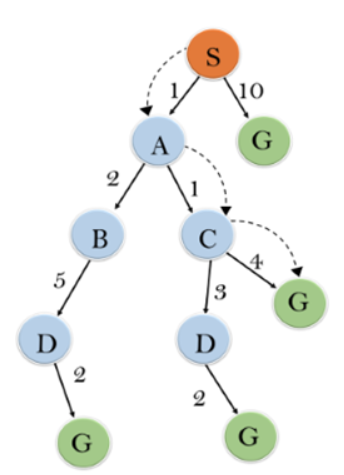
Heuristic Values



Here, the Start state is **S** and the Goal state is **G**. The start state is inserted into OPEN list. The first value in OPEN list is popped, and adds its immediate children node, sorted according to sum of cost and heuristic values(g(n) + h(n)), to the OPEN list. The popped value is inserted into CLOSED list. The above step is repeated until we reach the Goal state after which backtracking gives us the path travelled.

|  |  |  |
| --- | --- | --- |
| Open | Closed | Iterations |
| S, 5 | S | Start |
| S, A, 4  S, G, 10 | S | 1 |
| S, A, C, 4  S, A, B, 7  S, G, 10 | S, A | 2 |
| S, A, C, G, 6  S, A, C, D, 11  S, A, B, 7  S, G, 10 | S, A, C | 3 |
|  | S, A, C, G | Stop |

Path Travelled: **S🡪A🡪 C🡪 G**



**Inference:**

A\* algorithm returns the path which occurred first, and it does not search for all remaining paths. The efficiency of A\* algorithm depends on the quality of heuristic and it expands all nodes which satisfy the condition f(n).

The time complexity is **O(n)** in a grid and **O(bd)** in a graph/tree with a branching factor (b) and a depth (d). The branching factor is the average number of neighbor nodes that can be expanded from each node and the depth is the average number of levels in a graph/tree.

**Result:**

We have implemented the A\* Search algorithm and analysed it.