

Exp- 5 A* Algorithm and Best First Algorithm

Team- Automata lab

RA1911031010128-VEER VISWAJEET SWAMI

RA1911031010129-VIJAY RENGARAJ R

RA1911031010130-THINAKAR R

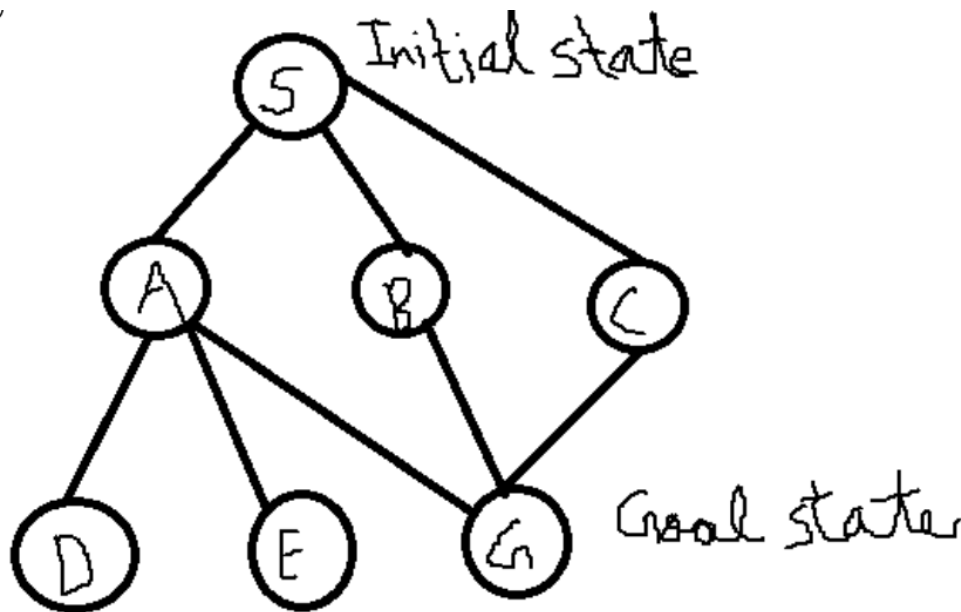
RA1911031010131-NAMARATA MISHRA

RA1911031010138-HUSNA QASIM

A* Algorithm

Problem chosen: Graph problem

Problem statement: The main aim of this problem is to reach the goal state 'G' from initial state 'S' and finding out the optimal path and visited nodes of the graph using A* search algorithm.



Code & Output:

```
tree = {'S': [['A', 1], ['B', 5], ['C', 8]],
```

```
'A': [['S', 1], ['D', 3], ['E', 7], ['G', 9]],  
'B': [['S', 5], ['G', 4]],  
'C': [['S', 8], ['G', 5]],  
'D': [['A', 3]],  
'E': [['A', 7]]}
```

```
tree2 = {'S': [['A', 1], ['B', 2]],  
        'A': [['S', 1]],  
        'B': [['S', 2], ['C', 3], ['D', 4]],  
        'C': [['B', 2], ['E', 5], ['F', 6]],  
        'D': [['B', 4], ['G', 7]],  
        'E': [['C', 5]],  
        'F': [['C', 6]]  
        }
```

```
heuristic = {'S': 8, 'A': 8, 'B': 4, 'C': 3, 'D': 5000, 'E': 5000, 'G': 0}
```

```
heuristic2 = {'S': 0, 'A': 5000, 'B': 2, 'C': 3, 'D': 4, 'E': 5000, 'F': 5000, 'G': 0}
```

```
cost = {'S': 0}      # total cost for nodes visited
```

```
def AStarSearch():
```

```
    global tree, heuristic
```

```
    closed = []      # closed nodes
```

```
opened = [['S', 8]] # opened nodes
```

```
'''find the visited nodes'''
```

```
while True:
```

```
    fn = [i[1] for i in opened] #  $fn = f(n) = g(n) + h(n)$ 
```

```
    chosen_index = fn.index(min(fn))
```

```
    node = opened[chosen_index][0] # current node
```

```
    closed.append(opened[chosen_index])
```

```
    del opened[chosen_index]
```

```
    if closed[-1][0] == 'G': # break the loop if node G has been found
```

```
        break
```

```
    for item in tree[node]:
```

```
        if item[0] in [closed_item[0] for closed_item in closed]:
```

```
            continue
```

```
            cost.update({item[0]: cost[node] + item[1]}) # add nodes to cost  
dictionary
```

```
            fn_node = cost[node] + heuristic[item[0]] + item[1] # calculate  $f(n)$  of  
current node
```

```
            temp = [item[0], fn_node]
```

```
            opened.append(temp) # store  $f(n)$  of current node in  
array opened
```

```
'''find optimal sequence'''
```

```
trace_node = 'G' # correct optimal tracing node, initialize as node  
G
```

```
optimal_sequence = ['G'] # optimal node sequence
```

```

for i in range(len(closed)-2, -1, -1):
    check_node = closed[i][0]      # current node
    if trace_node in [children[0] for children in tree[check_node]]:
        children_costs = [temp[1] for temp in tree[check_node]]
        children_nodes = [temp[0] for temp in tree[check_node]]

        '''check whether  $h(s) + g(s) = f(s)$ . If so, append current node to optimal
sequence
        change the correct optimal tracing node to current node'''
        if cost[check_node] + children_costs[children_nodes.index(trace_node)] ==
cost[trace_node]:
            optimal_sequence.append(check_node)
            trace_node = check_node
    optimal_sequence.reverse()      # reverse the optimal sequence

return closed, optimal_sequence

```

```

if __name__ == '__main__':
    visited_nodes, optimal_nodes = AStarSearch()
    print('visited nodes: ' + str(visited_nodes))
    print('optimal nodes sequence: ' + str(optimal_nodes))

```

Online Python Compiler (Interp... x (92) CS540 L3 Python: A Star Se... x Home Page - Select or create a... x Untitled7 - Jupyter Notebook x +

localhost:8889/notebooks/Untitled7.ipynb?kernel_name=python3

Apps Gmail YouTube Maps Global NetAcad Ins... Reading list

jupyter Untitled7 Last Checkpoint: 23 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

```
In [1]: tree = {'S': [['A', 1], ['B', 5], ['C', 8]],
              'A': [['S', 1], ['D', 3], ['E', 7], ['G', 9]],
              'B': [['S', 5], ['G', 4]],
              'C': [['S', 8], ['G', 5]],
              'D': [['A', 3]],
              'E': [['A', 7]]

tree2 = {'S': [['A', 1], ['B', 2]],
        'A': [['S', 1]],
        'B': [['S', 2], ['C', 3], ['D', 4]],
        'C': [['B', 2], ['E', 5], ['F', 6]],
        'D': [['B', 4], ['G', 7]],
        'E': [['C', 5]],
        'F': [['C', 6]]
        }

heuristic = {'S': 8, 'A': 8, 'B': 4, 'C': 3, 'D': 5000, 'E': 5000, 'G': 0}
heuristic2 = {'S': 0, 'A': 5000, 'B': 2, 'C': 3, 'D': 4, 'E': 5000, 'F': 5000, 'G': 0}
cost = {'S': 0} # total cost for nodes visited

def AStarSearch():
    global tree, heuristic
    closed = [] # closed nodes
    opened = [['S', 8]] # opened nodes

    '''find the visited nodes'''
    while True:
        fn = [i[1] for i in opened] # fn = f(n) = g(n) + h(n)
        chosen_index = fn.index(min(fn))
        node = opened[chosen_index][0] # current node
        closed.append(opened[chosen_index])
        del opened[chosen_index]
        if closed[-1][0] == 'G': # break the loop if node G has been found
            break
        for item in tree[node]:
            if item[0] in [closed_item[0] for closed_item in closed]:
                continue
            cost.update(item[0]: cost[node] + item[1]) # add nodes to cost dictionary
            fn_node = cost[node] + heuristic[item[0]] + item[1] # calculate f(n) of current node
            temp = [item[0], fn_node] # store f(n) of current node in array opened
            opened.append(temp)

    '''find optimal sequence'''
    trace_node = 'G' # correct optimal tracing node, initialize as node G
    optimal_sequence = ['G'] # optimal node sequence
    for i in range(len(closed)-2, -1, -1):
        check_node = closed[i][0] # current node
        if trace_node in [children[0] for children in tree[check_node]]:
            children_costs = [temp[1] for temp in tree[check_node]]
            children_nodes = [temp[0] for temp in tree[check_node]]

            '''check whether h(s) + g(s) = f(s). If so, append current node to optimal sequence
            change the correct optimal tracing node to current node'''
            if cost[check_node] + children_costs[children_nodes.index(trace_node)] == cost[trace_node]:
                optimal_sequence.append(check_node)
                trace_node = check_node
    optimal_sequence.reverse() # reverse the optimal sequence

    return closed, optimal_sequence

if __name__ == '__main__':
    visited_nodes, optimal_nodes = AStarSearch()
    print('visited nodes: ' + str(visited_nodes))
    print('optimal nodes sequence: ' + str(optimal_nodes))
```

Type here to search

Online Python Compiler (Interp... x (92) CS540 L3 Python: A Star Se... x Home Page - Select or create a... x Untitled7 - Jupyter Notebook x +

localhost:8889/notebooks/Untitled7.ipynb?kernel_name=python3

Apps Gmail YouTube Maps Global NetAcad Ins... Reading list

jupyter Untitled7 Last Checkpoint: 23 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

```
'''find the visited nodes'''
while True:
    fn = [i[1] for i in opened] # fn = f(n) = g(n) + h(n)
    chosen_index = fn.index(min(fn))
    node = opened[chosen_index][0] # current node
    closed.append(opened[chosen_index])
    del opened[chosen_index]
    if closed[-1][0] == 'G': # break the loop if node G has been found
        break
    for item in tree[node]:
        if item[0] in [closed_item[0] for closed_item in closed]:
            continue
        cost.update(item[0]: cost[node] + item[1]) # add nodes to cost dictionary
        fn_node = cost[node] + heuristic[item[0]] + item[1] # calculate f(n) of current node
        temp = [item[0], fn_node] # store f(n) of current node in array opened
        opened.append(temp)

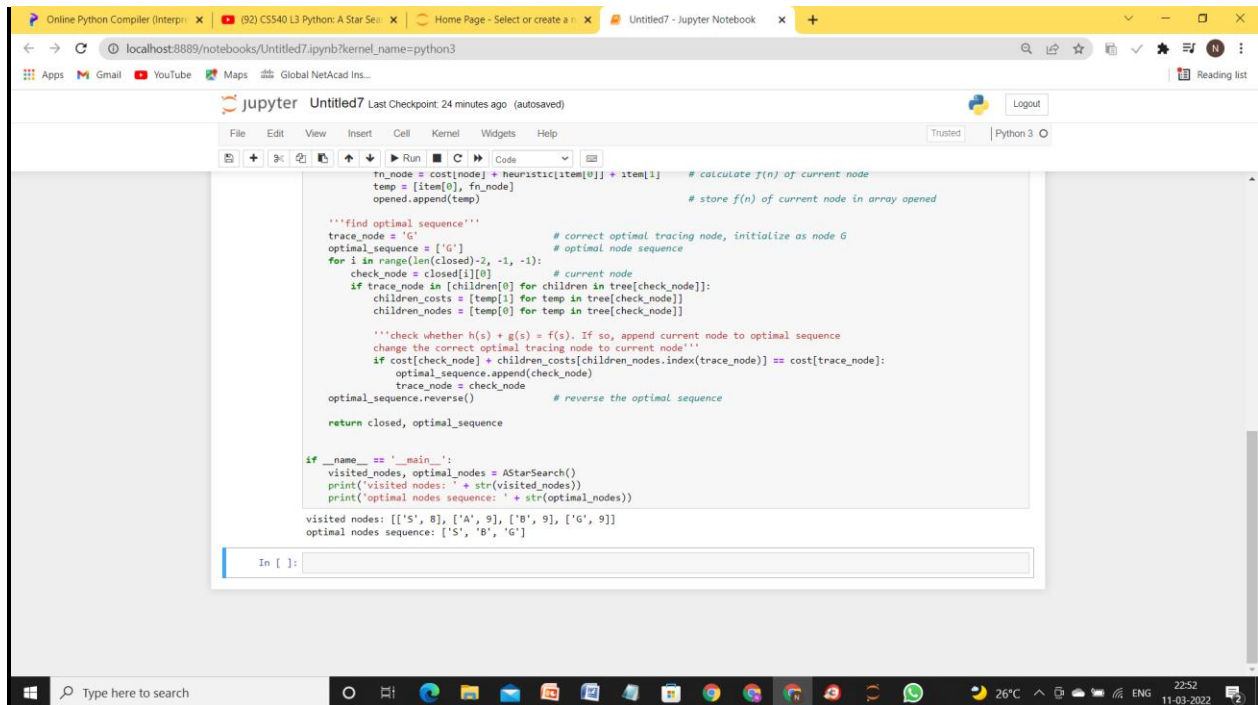
'''find optimal sequence'''
trace_node = 'G' # correct optimal tracing node, initialize as node G
optimal_sequence = ['G'] # optimal node sequence
for i in range(len(closed)-2, -1, -1):
    check_node = closed[i][0] # current node
    if trace_node in [children[0] for children in tree[check_node]]:
        children_costs = [temp[1] for temp in tree[check_node]]
        children_nodes = [temp[0] for temp in tree[check_node]]

        '''check whether h(s) + g(s) = f(s). If so, append current node to optimal sequence
        change the correct optimal tracing node to current node'''
        if cost[check_node] + children_costs[children_nodes.index(trace_node)] == cost[trace_node]:
            optimal_sequence.append(check_node)
            trace_node = check_node
optimal_sequence.reverse() # reverse the optimal sequence

return closed, optimal_sequence

if __name__ == '__main__':
    visited_nodes, optimal_nodes = AStarSearch()
    print('visited nodes: ' + str(visited_nodes))
    print('optimal nodes sequence: ' + str(optimal_nodes))
```

Type here to search



```
Online Python Compiler (Interp... x (92) C5540 L3 Python: A Star Se... x Home Page - Select or create a... x Untitled7 - Jupyter Notebook x +
localhost:8889/notebooks/Untitled7.jpynb?kernel_name=python3
jupyter Untitled7 Last Checkpoint: 24 minutes ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run Code
fn_node = cost[node] + heuristic[item[u]] + item[l] # calculate f(n) of current node
temp = [item[0], fn_node] # store f(n) of current node in array opened
opened.append(temp)

'''find optimal sequence'''
trace_node = 'G' # correct optimal tracing node, initialize as node G
optimal_sequence = ['G'] # optimal node sequence
for i in range(len(closed)-2, -1, -1): # current node
    check_node = closed[i][0]
    if trace_node in [children[0] for children in tree[check_node]]:
        children_costs = [temp[l] for temp in tree[check_node]]
        children_nodes = [temp[0] for temp in tree[check_node]]

        '''check whether h(s) + g(s) = f(s). If so, append current node to optimal sequence
        change the correct optimal tracing node to current node'''
        if cost[check_node] + children_costs[children_nodes.index(trace_node)] == cost[trace_node]:
            optimal_sequence.append(check_node)
            trace_node = check_node

    optimal_sequence.reverse() # reverse the optimal sequence
return closed, optimal_sequence

if __name__ == '__main__':
    visited_nodes, optimal_nodes = AStarSearch()
    print('visited nodes: ' + str(visited_nodes))
    print('optimal nodes sequence: ' + str(optimal_nodes))

visited nodes: [['S', 8], ['A', 9], ['B', 9], ['G', 9]]
optimal nodes sequence: ['S', 'B', 'G']

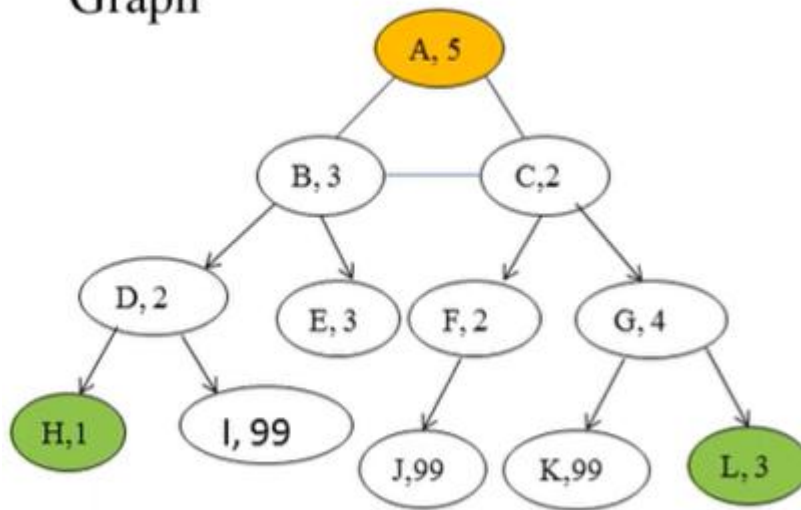
In [ ]:
```

Best First Algorithm

Problem chosen: Graph problem

Problem statement: A successor list is given ,i.e., a dictionary is given with start value as 'A', goal value as 'H', global closed list, success as "True", failure as "False" and state as "Failure" initially. The main aim of this problem is to reach the goal value from start value using best first search algorithm.

Graph



Code & Output:

```
SuccList = { 'A':[['B',3],['C',2]], 'B':[['A',5],['C',2],['D',2],['E',3]],  
'C':[['A',5],['B',3],['F',2],['G',4]], 'D':[['H',1],['I',99]], 'F': [['J',99]], 'G':[['K',99],['L',3]] }
```

```
Start='A'
```

```
Goal='E'
```

```
Closed = list()
```

```
SUCCESS=True
```

```
FAILURE=False
```

```
State=FAILURE
```

```
def MOVEGEN(N):
```

```
    New_list=list()
```

```
    if N in SuccList.keys():
```

```
        New_list=SuccList[N]
```

```
    return New_list
```

```
def GOALTEST(N):
```

```
    if N == Goal:
```

```
        return True
```

```
    else:
```

```
        return False
```

```
def APPEND(L1,L2):
```

```
    New_list=list(L1)+list(L2)
```

```
    return New_list
```

```
def SORT(L):
```

```
    L.sort(key = lambda x: x[1])
```

```
    return L
```

```
def BestFirstSearch():
```

```
    OPEN=[[Start,5]]
```

```
    CLOSED=list()
```

```
    global State
```

```
    global Closed
```

```
    while (len(OPEN) != 0) and (State != SUCCESS):
```

```
        print("-----")
```



```
N= OPEN[0]
print("N=",N)
del OPEN[0] #delete the node we picked
```

```
if GOALTEST(N[0])==True:
```

```
    State = SUCCESS
```

```
    CLOSED = APPEND(CLOSED,[N])
```

```
    print("CLOSED=",CLOSED)
```

```
else:
```

```
    CLOSED = APPEND(CLOSED,[N])
```

```
    print("CLOSED=",CLOSED)
```

```
    CHILD = MOVEGEN(N[0])
```

```
    print("CHILD=",CHILD)
```

```
    for val in CLOSED:
```

```
        if val in CHILD:
```

```
            CHILD.remove(val)
```

```
    for val in OPEN:
```

```
        if val in CHILD:
```

```
            CHILD.remove(val)
```

```
    OPEN = APPEND(CHILD,OPEN) #append movegen elements to
```

```
OPEN
```

```
    print("Unsorted OPEN=",OPEN)
```

```
    SORT(OPEN)
```

```
    print("Sorted OPEN=",OPEN)
```

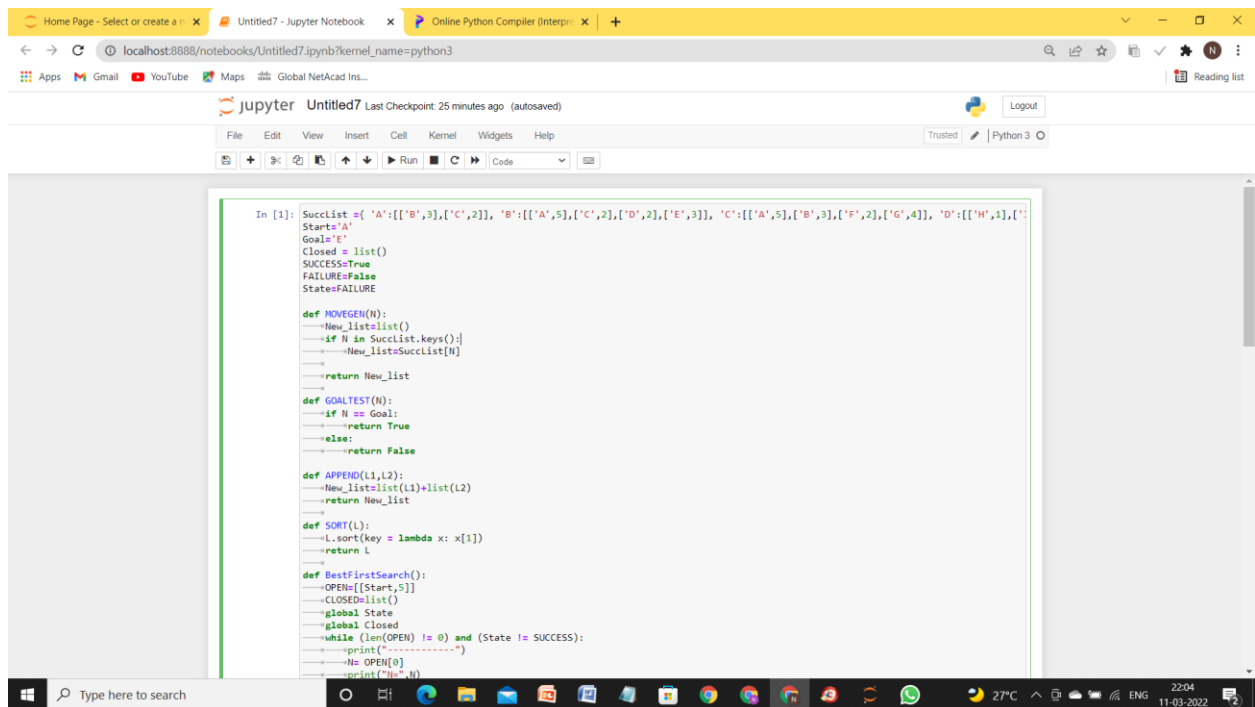
Closed=CLOSED

return State

#Driver Code

result=BestFirstSearch() #call search algorithm

print(Closed,result)



The screenshot shows a Jupyter Notebook interface with a yellow header bar. The browser address bar shows 'localhost:8888/notebooks/Untitled7.ipynb?kernel_name=python3'. The Jupyter interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and code execution. The main area displays a Python script for a search algorithm. The script defines a dictionary 'SuccList' with keys 'A' through 'D', each containing a list of pairs. It also defines functions for moving elements, testing goals, appending lists, sorting, and performing a best-first search. The search function uses a while loop to explore the search space until it reaches a goal state or exhausts the open list. The script concludes with a call to 'BestFirstSearch()' and a print statement for the 'Closed' set and the 'result'.

```
In [1]: SuccList = { 'A':[['B',3],['C',2]], 'B':[['A',5],['C',2],['D',2],['E',3]], 'C':[['A',5],['B',3],['F',2],['G',4]], 'D':[['H',1],['I',2]]
Start='A'
Goal='E'
Closed = list()
SUCCESS=True
FAILURE=False
State=FAILURE

def MOVEGEN(N):
    New_list=list()
    if N in SuccList.keys():
        New_list=SuccList[N]
    return New_list

def GOALTEST(N):
    if N == Goal:
        return True
    else:
        return False

def APPEND(L1,L2):
    New_list=list(L1)+list(L2)
    return New_list

def SORT(L):
    L.sort(key = lambda x: x[1])
    return L

def BestFirstSearch():
    OPEN=[['Start',5]]
    CLOSED=list()
    global State
    global Closed
    while (len(OPEN) != 0) and (State != SUCCESS):
        print("-----")
        N= OPEN[0]
        print(N)
```

```
L.sort(key = lambda x: X[i])
return L

def BestFirstSearch():
    OPEN=[Start,S]
    CLOSED=list()
    global State
    global Closed
    while (len(OPEN) != 0) and (State != SUCCESS):
        print("-----")
        N= OPEN[0]
        print("N=",N)
        del OPEN[0] #Delete the node we picked
        if GOALTEST(N[0])==True:
            State = SUCCESS
            CLOSED = APPEND(CLOSED,[N])
            print("CLOSED=",CLOSED)
        else:
            CLOSED = APPEND(CLOSED,[N])
            print("CLOSED=",CLOSED)
            CHILD = MOVEGEN(N[0])
            print("CHILD=",CHILD)
            for val in CLOSED:
                if val in CHILD:
                    CHILD.remove(val)
            for val in OPEN:
                if val in CHILD:
                    CHILD.remove(val)
            OPEN = APPEND(CHILD,OPEN) #append movegen elements to OPEN
            print("Unsorted OPEN=",OPEN)
            SORT(OPEN)
            print("Sorted OPEN=",OPEN)
        Closed=CLOSED
    return State

#Driver Code
result=BestFirstSearch() #call search algorithm
print(Closed,result)
```

```
N= ['A', 5]
CLOSED= [['A', 5]]
CHILD= [['B', 3], ['C', 2]]
Unsorted OPEN= [['B', 3], ['C', 2]]
Sorted OPEN= [['C', 2], ['B', 3]]
-----
N= ['C', 2]
CLOSED= [['A', 5], ['C', 2]]
CHILD= [['A', 5], ['B', 3], ['F', 2], ['G', 4]]
Unsorted OPEN= [['F', 2], ['G', 4], ['B', 3]]
Sorted OPEN= [['F', 2], ['B', 3], ['G', 4]]
-----
N= ['F', 2]
CLOSED= [['A', 5], ['C', 2], ['F', 2]]
CHILD= [['J', 99]]
Unsorted OPEN= [['J', 99], ['B', 3], ['G', 4]]
Sorted OPEN= [['B', 3], ['G', 4], ['J', 99]]
-----
N= ['B', 3]
CLOSED= [['A', 5], ['C', 2], ['F', 2], ['B', 3]]
CHILD= [['A', 5], ['C', 2], ['D', 2], ['E', 3]]
Unsorted OPEN= [['D', 2], ['E', 3], ['G', 4], ['J', 99]]
Sorted OPEN= [['D', 2], ['E', 3], ['G', 4], ['J', 99]]
-----
N= ['D', 2]
CLOSED= [['A', 5], ['C', 2], ['F', 2], ['B', 3], ['D', 2]]
CHILD= [['H', 1], ['I', 99]]
Unsorted OPEN= [['H', 1], ['I', 99], ['E', 3], ['G', 4], ['J', 99]]
Sorted OPEN= [['H', 1], ['E', 3], ['G', 4], ['I', 99], ['J', 99]]
-----
N= ['H', 1]
CLOSED= [['A', 5], ['C', 2], ['F', 2], ['B', 3], ['D', 2], ['H', 1]]
CHILD= []
Unsorted OPEN= [['E', 3], ['G', 4], ['I', 99], ['J', 99]]
Sorted OPEN= [['E', 3], ['G', 4], ['I', 99], ['J', 99]]
-----
N= ['E', 3]
CLOSED= [['A', 5], ['C', 2], ['F', 2], ['B', 3], ['D', 2], ['H', 1], ['E', 3]]
[['A', 5], ['C', 2], ['F', 2], ['B', 3], ['D', 2], ['H', 1], ['E', 3]] True
```

RESULT:

The problem statements for both A* search and best first search(BFS) are solved.

