**EXPERTIMENT NO. 6**

**Aim:** To implement A\* search algorithm.

**Requirements:** Compatible version of python.

**Theory:**

The most widely known form of best-first search is called A∗ A search (pronounced “A-star ∗ SEARCH search”). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: f(n) = g(n) + h(n) . Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n . Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A∗ search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses g + h instead of g.

**Algorithm:**

1. make an openlist containing only the starting node
2. make an empty closed list
3. while (the destination node has not been reached):
4. consider the node with the lowest f score in the open list
5. if (this node is our destination node) :
6. we are finished
7. if not:
8. put the current node in the closed list and look at all of its neighbours
9. for (each neighbour of the current node):
   1. if (neighbour has lower g value than current and is in the closed list) :
      1. replace the neighbour with the new, lower, g value
      2. current node is now the neighbour's parent
   2. else if (current g value is lower and this neighbour is in the open list ) :
      1. replace the neighbour with the new, lower, g value
      2. change the neighbour's parent to our current node
   3. else if this neighbour is not in both lists:
      1. add it to the open list and set its g

**Implementation:**

from queue import PriorityQueue

def create\_path(parent,dest):

temp = []

while(dest):

temp.append((dest,parent[dest][0]))

dest = parent[dest][1]

return list(reversed(temp))

def gbfs(graph,source,dest,heu):

parent, close\_ls = {}, set()

open\_ls = PriorityQueue()

open\_ls.put((heu[source],source))

parent[source] = (heu[source], None)

total\_cost = {}

while(not open\_ls.empty()):

current\_cost, current = open\_ls.get()

if current in close\_ls:

continue

if current == dest:

return create\_path(parent,dest),current\_cost

close\_ls.add(current)

for cost,neighbour in graph[current]:

if neighbour in close\_ls:

continue

temp = cost+heu[neighbour]

if neighbour not in total\_cost:

total\_cost[neighbour] = temp

open\_ls.put((temp,neighbour))

parent[neighbour] = (temp,current)

elif temp < total\_cost[neighbour]:

total\_cost[neighbour] = temp

open\_ls.put((temp,neighbour))

parent[neighbour] = (temp,current)

return "path doen't exist"

graph = {

"Arad":[(140,"Sibiu"), (118,"Timisoara"),(75,"Zerind")],

"Sibiu": [(280,"Arad"),(239,"Fagaras"),(291,"Oradea"), (220,"RimnicuVilcea")],

"Timisoara": [(200,"RimnicuVilcea")],

"Zerind": [],

"Fagaras": [(338,"Sibiu"),(450,"Bucharest")],

"Oradea":[],

"RimnicuVilcea": [(366,"Craiova"),(317,"Pitesti"),(300,"Sibiu")],

"Bucharest": [(100,"Zerind")],

"Craiova":[],

"Pitesti":[(418,"Bucharest"),(455,"Craiova"),(414,"RimnicuVilcea")]

}

heu = {

"Arad": 366, "Bucharest":0, "Fagaras":176,

"Sibiu":253, "Timisoara": 329, "Zerind":374,

"Oradea":380, "RimnicuVilcea": 193,

"Craiova":160, "Pitesti":100,

}

inputs = [

(graph,"Arad","Bucharest",heu),

(graph,"Arad","RimnicuVilcea",heu),

(graph,"Arad","Sinaia",heu)

]

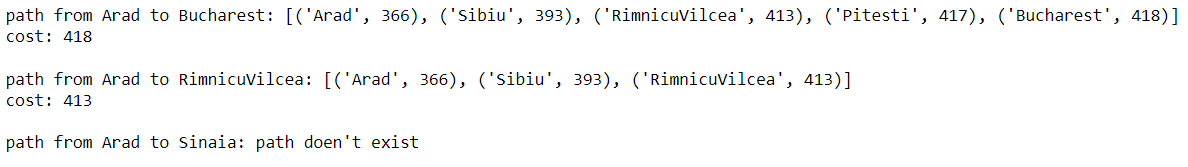
for x in inputs:

result = gbfs(\*x)

print(f"path from {x[1]} to {x[2]}: {result[0]}\ncost: {result[1]}\n")\

if len(result) <= 2 else print(f"path from {x[1]} to {x[2]}: {result}\n")

**Output:**

****

**Conclusion:** We have successfully implemented A\* search algorithm in python.