Experiment – 5

Implementation Of Best First Search And A* Algorithm

RA1811030010023

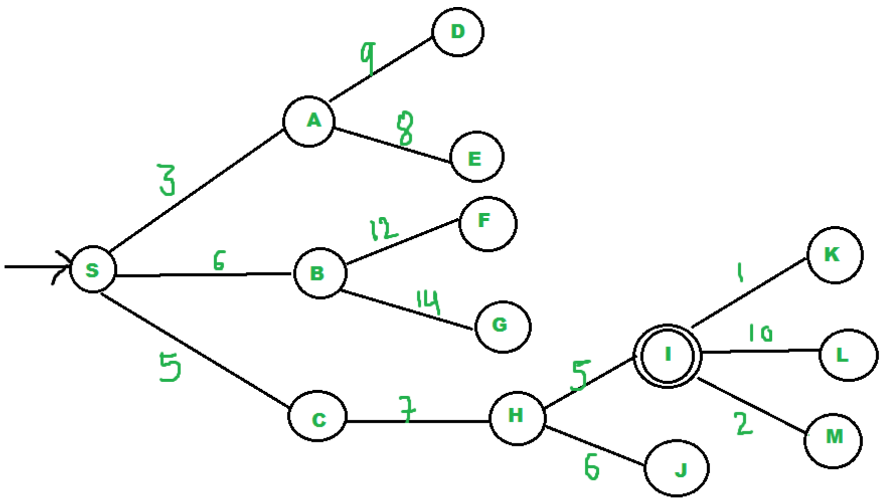
Sayan Pramanick

Best first search algorithm

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. We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

This pseudocode is adapted from below

```
Best-First-Search(Grah g, Node start)
    1) Create an empty PriorityQueue
       PriorityQueue pq;
    2) Insert "start" in pq.
       pq.insert(start)
    3) Until PriorityQueue is empty
          u = PriorityQueue.DeleteMin
          If u is the goal
             Exit
          Else
             Foreach neighbor v of u
                If v "Unvisited"
                    Mark v "Visited"
                    pq.insert(v)
             Mark u "Examined"
End procedure
```



Python Program

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]
def best_first_search(source, target, n):
    visited = [0] * n
    visited[0] = True
    pq = PriorityQueue()
    pq.put((0, source))
    while pq.empty() == False:
        u = pq.get()[1]
        print(u, end=" ")
        if u == target:
            break
        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()
def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
```

```
addedge(9, 13, 2)
source = 0
target = 9
best_first_search(source, target, v)
```

```
"C:\Users\Saransh Chauhan\PycharmProj
0 1 3 2 8 9

Process finished with exit code 0
```

A* Algorithm

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

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

where,
n = next node on the path

***g*(*n*)** = the cost of the path from the start node to n

***h*(*n*)** = a heuristic function that estimates the cost of the cheapest path from n to the goal

Python Program

```
from queue import PriorityQueue
class State(object):
    def __init__(self, value, parent, start=0, goal=0):
        self.children = []
        self.parent = parent
        self.value = value
        self.dist = 0
        if parent:
            self.start = parent.start
            self.goal = parent.goal
            self.path = parent.path[:]
```

```
            self.path.append(value)
        else:
            self.path = [value]
            self.start = start
            self.goal = goal
    def GetDistance(self):
        pass
    def CreateChildren(self):
        pass
class State_String(State):
    def __init__(self, value, parent, start=0, goal=0):
        super(State_String, self).__init__(value, parent, start, goal)
        self.dist = self.GetDistance()
    def GetDistance(self):
        if self.value == self.goal:
            return 0
        dist = 0
        for i in range(len(self.goal)):
            letter = self.goal[i]
            dist += abs(i - self.value.index(letter))
        return dist
    def CreateChildren(self):
        if not self.children:
            for i in range(len(self.goal) - 1):
                val = self.value
                val = val[:i] + val[i + 1] + val[i] + val[i + 2:]
                child = State_String(val, self)
                self.children.append(child)
class A_Star_Solver:
    def __init__(self, start, goal):
        self.path = []
        self.vistedQueue = []
        self.priorityQueue = PriorityQueue()
        self.start = start
        self.goal = goal
    def Solve(self):
        startState = State_String(self.start, 0, self.start, self.goal)
        count = 0
        self.priorityQueue.put((0, count, startState))
        while (not self.path and self.priorityQueue.qsize()):
            closesetChild = self.priorityQueue.get()[2]
            closesetChild.CreateChildren()
            self.vistedQueue.append(closesetChild.value)
            for child in closesetChild.children:
                if child.value not in self.vistedQueue:
                    count += 1
                    if not child.dist:
                        self.path = child.path
                        break
                    self.priorityQueue.put((child.dist, count, child))
```

```
        if not self.path:
            print("Goal Of  is not possible !" + self.goal)
        return self.path
if __name__ == "__main__":
    start1 = "hema"
    goal1 = "mahe"
    print("Starting....")
    a = A_Star_Solver(start1, goal1)
    a.Solve()
    for i in range(len(a.path)):
        print("{0}){1}".format(i, a.path[i]))
```

```
"C:\Users\Saransh Chauhan\PycharmProjects
Starting....
0)hema
1)hmea
2)mhea
3)mhae
4)mahe

Process finished with exit code 0
```

Result : Best first search and A* algorithm were hence studied and implemented.