Experiment No. 4: A* Algorithm

Dwayne Fernandes
BE CMPN-A
Roll.No : 35 pid : 172033

AIM : To implement A* algorithm

THEORY:

A*Algorithm: A* Algorithm is one of the best and popular techniques used for path finding and graph traversals.

A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.

It is essentially a best first search algorithm.

Working

- It maintains a tree of paths originating at the start node.
- It extends those paths one edge at a time.
- It continues until its termination criterion is satisfied.

A* Algorithm extends the path that minimizes the following function-

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

where f'(n) – It is estimated cost of the cheapest solution through n.

h'(n) – It is an estimate of the additional cost of getting from the current node to a goal node.

g'(n) – It is an estimate of getting from the initial node to the current node.

A* Algorithm is given as follow:

1. Put the initial node a list OPEN.
2. If (OPEN is empty) or (OPEN=GOAL) then terminate search
3. Removes the first node from OPEN. Call this node a.
4. If (a=GOAL) then terminate with SUCCESS.
5. Else if node a has successors, generate all of them. Estimate the fitness number of the successors by totaling the evaluation function value and the cost function value. Sort the list by fitness number.
6. Name the new list is CLOSED.

7. Replace OPEN with CLOSED.

8. Go to step 2.

Properties of A* Algorithm

Admissibility

- In A* algorithm related to pathfinding, a heuristic function is said to be admissible if it never overestimates the cost of reaching the goal, i.e. the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path .
- h is admissible when h(n) <= h*(n) holds. Using an admissible heuristic guarantees that the first solution found will be an optimal one. Therefore, A* is admissible.
- A search algorithm is admissible, if :- for any graph, it always terminates in an optimal path from initial state to goal state, if path exists.
- If heuristic function h is underestimate of actual value from current state to goal state, then it is called admissible function. (h(n)<=h*(n))
- Alternatively we can say that A* always terminates with the optimal path in case
- h(x) is an admissible heuristic function.

Monotonicity

- A heuristic function h is monotone if for all states Xi and Xj such that Xj is successor of Xi h(Xi)– h(Xj) ≤ cost (Xi, Xj) where, cost (Xi, Xj) actual cost of going from
- Xi to Xj
- The monotone property says:
- That search space which is every where locally consistent with heuristic function employed i.e., reaching each state along the shortest path from its ancestors.
- With monotonic heuristic, if a state is rediscovered, it is not necessary to check whether the new path is shorter.

• Each monotonic heuristic is admissible

– A cost function f(n) is monotone. if f(n) ≤f(succ(n)),for all n .

• For any admissible cost function f, we can construct a monotone admissible function.

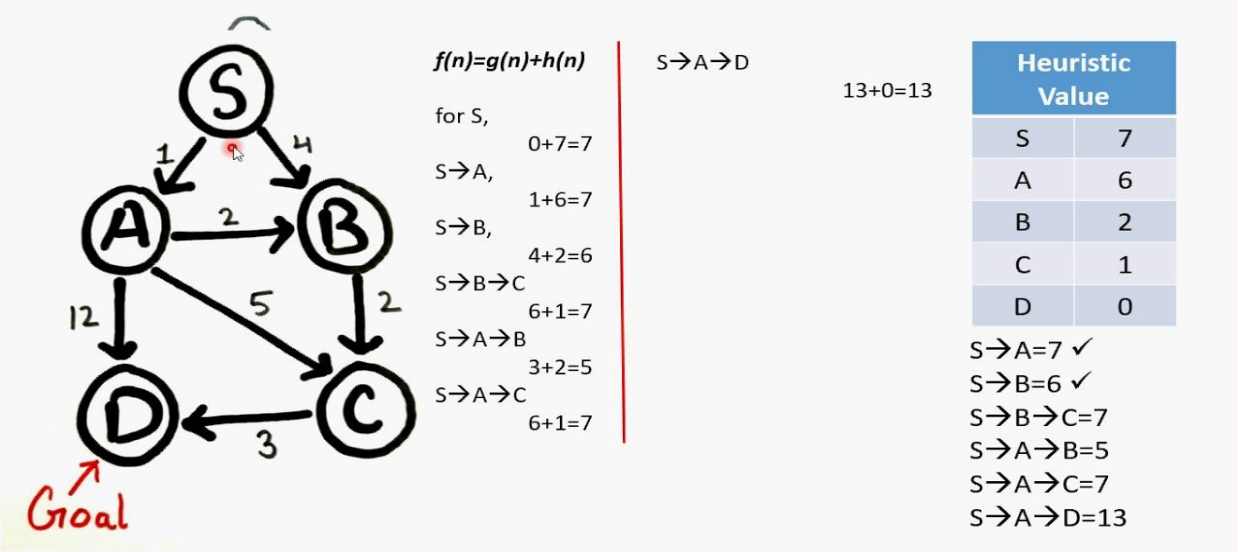
Advantages:

- It is complete and optimal.
- It is the best one from other techniques.
- It is used to solve very complex problems.
- It is optimally ecient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*.

Disadvantages:

- This algorithm is complete if the branching factor is nite and every action has x ed cost.
- The speed execution of A* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute h(n).

Example:



CODE:

```
import numpy as np
import math
import copy
class Puzzle8:
  def __init__(self,start,goal):
       self.current = np.array(start)
       self.goal = np.array(goal)
       self.depth = 0
       self.emptyPosition = np.argwhere(self.current==0)[0]
       self.goalEmptyPosition = np.argwhere(self.goal==0)[0]
       self.height = len(self.current)
       self.width = len(self.current[0])
```

```
        self.prev = ''
    def _repr_(self):
        res = 'Current State :\n'
        for row in self.current:
         res+= ' '.join(map(str,row)) + '\n'
        return res
    def __cost(self,nextPosition,depth):
       h = (self.goal!=nextPosition).sum()
       if h>0:
          h-=1
       return h + depth

    def __matSwap(self, mat,i,j):
        mat[i[0]][i[1]] , mat[j[0]][j[1]] = mat[j[0]][j[1]] , mat[i[0]][i[1]
]
        return mat
    def __nextLevel(self):
      self.depth += 1
      costsDict = dict()
      resultMats = dict()
      avoidLoop = {'R':'L','L':'R','U':'D','D':'U'}
# Right
      if self.emptyPosition[1]<self.width-1 and self.prev != 'R':
        resultMats['R'] = self.__matSwap(copy.deepcopy(self.current)
                                  ,self.emptyPosition,self.emptyPosition+[0
,1])
        costsDict['R'] = self.__cost(resultMats['R'],self.depth)
      else:
        costsDict['R'] = math.inf
# Left
      if self.emptyPosition[1]>0 and self.prev != 'L':
        resultMats['L'] = self.__matSwap(copy.deepcopy(self.current)
                                  ,self.emptyPosition,self.emptyPosition+[0
,-1])
        costsDict['L'] = self.__cost(resultMats['L'],self.depth)
      else:
        costsDict['L'] = math.inf
# Up
      if self.emptyPosition[0]>0 and self.prev != 'U':
          resultMats['U'] = self.__matSwap(copy.deepcopy(self.current)
                                           ,self.emptyPosition,self.emptyPos
ition+[-1,0])
          costsDict['U'] = self.__cost(resultMats['U'],self.depth)
      else:
          costsDict['U'] = math.inf
```

```
# Down

    if self.emptyPosition[0]<self.height-1 and self.prev != 'D':
       resultMats['D'] = self.__matSwap(copy.deepcopy(self.current)
                                        ,self.emptyPosition,self.emptyPositi
on+[1,0])
       costsDict['D'] = self.__cost(resultMats['D'],self.depth)
    else:
        costsDict['D'] = math.inf
    direction, minCost = min(costsDict.items(),key=lambda x : x[1])

# print(self.prev,direction)
# print(costsDict)

    self.current = resultMats[direction]
    if self.prev != avoidLoop[direction]:
      self.prev = direction
    if direction == 'R':
      self.emptyPosition += [0,1]
    elif direction == 'L':
      self.emptyPosition += [0,-1]
    elif direction == 'U':
       self.emptyPosition += [-1,0]
    elif direction == 'D':
      self.emptyPosition += [1,0]
    return minCost

  def showMat(self,mat):
    for row in mat:
      print(*row)

  def astar(self):
    print("/// Starting Position///")
    self.showMat(self.current)
    print("\n/// A* Algorithm ///")
    directionMap = {'R':'Right','L':'Left','U':'Up','D':'Down'}
    stop = False
    while self.__cost(self.current,0) != 0:
        cost = self.__nextLevel()
       print(f"stage{self.depth},{directionMap[self.prev]}, f(n) ={self.de
pth}+{costself.depth}={cost}\n")
        if self.depth > 20:
            print("Too Many Iterations")
            stop = True
            break
    if not stop:
```

```
        print("Goal State Achieved")

start = [[3,7,6],[5,1,2],[4,0,8]]
goal = [[5,3,6],[7,0,2],[4 ,1 ,8]]


puzzle = Puzzle8(start,goal)
puzzle.astar()
```

OUTPUT:

///Starting Position///
[3 7 6]
[5 1 2]
[4 0 8]

/// A* Algorithm///
[3 7 6]
[5 0 2]
[4 1 8]
stage1,Up, f(n) = 1+ 2=3

[3 7 6]
[0 5 2]
[4 1 8]
stage2,Left, f(n) = 2+ 3=5

[3 7 6]
[5 0 2]
[4 1 8]
stage3,Left, f(n) = 3+ 2=5

[3 0 6]
[5 7 2]
[4 1 8]
stage4,Up, f(n) = 4+ 3=7

[0 3 6]
[5 7 2]
[4 1 8]
stage5,Left, f(n) = 5+ 2=7

[5 3 6]
[0 7 2]

[4 1 8]
stage6,Down, f(n) = 6+ 1=7

[5 3 6]
[7 0 2]
[4 1 8]
stage7,Right, f(n) = 7+ 0=7

Goal State Achieved

CONCLUSION :A* is brilliant when it comes to finding paths from one place to another. It also makes sure that it finds the paths which are the most efficient. A* is a very powerful algorithm with almost unlimited potential. However, it is only as good as its heuristic function, which can be highly variable considering the nature of a problem. It has found applications in many software systems, from Machine Learning and Search Optimization to game development where NPC characters navigate through complex terrain and obstacles to reach the player.