ST. FRANCIS INSTITUTE OF TECHNOLOGY

Mount Poinsur, S.V.P. Road, Borivli (West), Mumbai – 400103

Computer Engineering Department

LAB MANUAL

Experiment No. 4: A* Algorithm

Aim: To implement A* algorithm

Theory:

(a) A* Algorithm

A* Search algorithm is one of the best and popular techniques used in path-finding and graph traversals. A* Search algorithms, unlike other traversal techniques, have "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms.

A* algorithm

1. Initialize the open list.
2. Initialize the closed list put the starting node on the open list (you can leave its f at zero)
3. While the open list is not empty
    a. find the node with the least f on the open list, call it "q" .
    b. pop q off the open list.
    c. Generate q's 8 successors and set their parents to q.
    d. for each successor
        1. if successor is the goal, stop search
        2. else, compute both g and h for successor
           successor.g = q.g + distance between successor and q
           successor.h = distance from goal to successors
           successor.f = successor.g + successor.
        3. if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
        4. if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list end (for loop)
        5. push q on the closed list end (while loop)

(b)Properties of A* Algorithm

A* achieve optimality and completeness, two valuable properties of search algorithms. When a search algorithm has the property of optimality, it means it is guaranteed to find the best possible solution. When a search algorithm has the property of completeness, it means that if a solution to a given problem exists, the algorithm is guaranteed to find it.

(c) Advantage and Disadvantage of A* Algorithm

Advantages:

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

Disadvantages:

1. This algorithm is complete if the branching factor is finite and every action has fixed cost.
2. The speed execution of A* search is highly dependent on the accuracy of the heuristic algorithm that is used to compute h (n).

Start State

$$\begin{bmatrix}3 & 7 & 6\\5 & 1 & 2\\4 & \_ & 8\end{bmatrix} \quad f = 0+4$$

up (1+3)

$$\begin{bmatrix}3 & 7 & 6\\5 & \_ & 2\\4 & 1 & 8\end{bmatrix}$$

left (1+5)

$$\begin{bmatrix}3 & 7 & 6\\5 & 1 & 2\\\_ & 4 & 8\end{bmatrix}$$

right (1+5)

$$\begin{bmatrix}3 & 7 & 6\\5 & 1 & 2\\4 & 8 & \_\end{bmatrix}$$

up (2+3)

$$\begin{bmatrix}3 & \_ & 6\\5 & 7 & 2\\4 & 1 & 8\end{bmatrix}$$

left (2+3)

$$\begin{bmatrix}3 & 7 & 6\\\_ & 5 & 2\\4 & 1 & 8\end{bmatrix}$$

right (2+4)

$$\begin{bmatrix}3 & 7 & 6\\5 & 2 & \_\\4 & 1 & 8\end{bmatrix}$$

left (3+2)

$$\begin{bmatrix}\_ & 3 & 6\\5 & 7 & 2\\4 & 1 & 8\end{bmatrix}$$

right (3+4)

$$\begin{bmatrix}3 & 6 & \_\\5 & 7 & 2\\4 & 1 & 8\end{bmatrix}$$

down (4+1)

$$\begin{bmatrix}5 & 3 & 6\\\_ & 7 & 2\\4 & 1 & 8\end{bmatrix} \xrightarrow{\text{right}} \begin{bmatrix}5 & 3 & 6\\7 & \_ & 2\\4 & 1 & 8\end{bmatrix}$$

goal state $f = 5+0$

Experiment:

The aim of A* algorithm is to traverse from the start state to goal state given below in the figure. Calculate and store the f (n) for each intermediate state. Display the solution like (3*3 matrix, Operation, F (n)), like this show entire path.

Initial State

1	2	3
8		4
7	6	5

Goal State

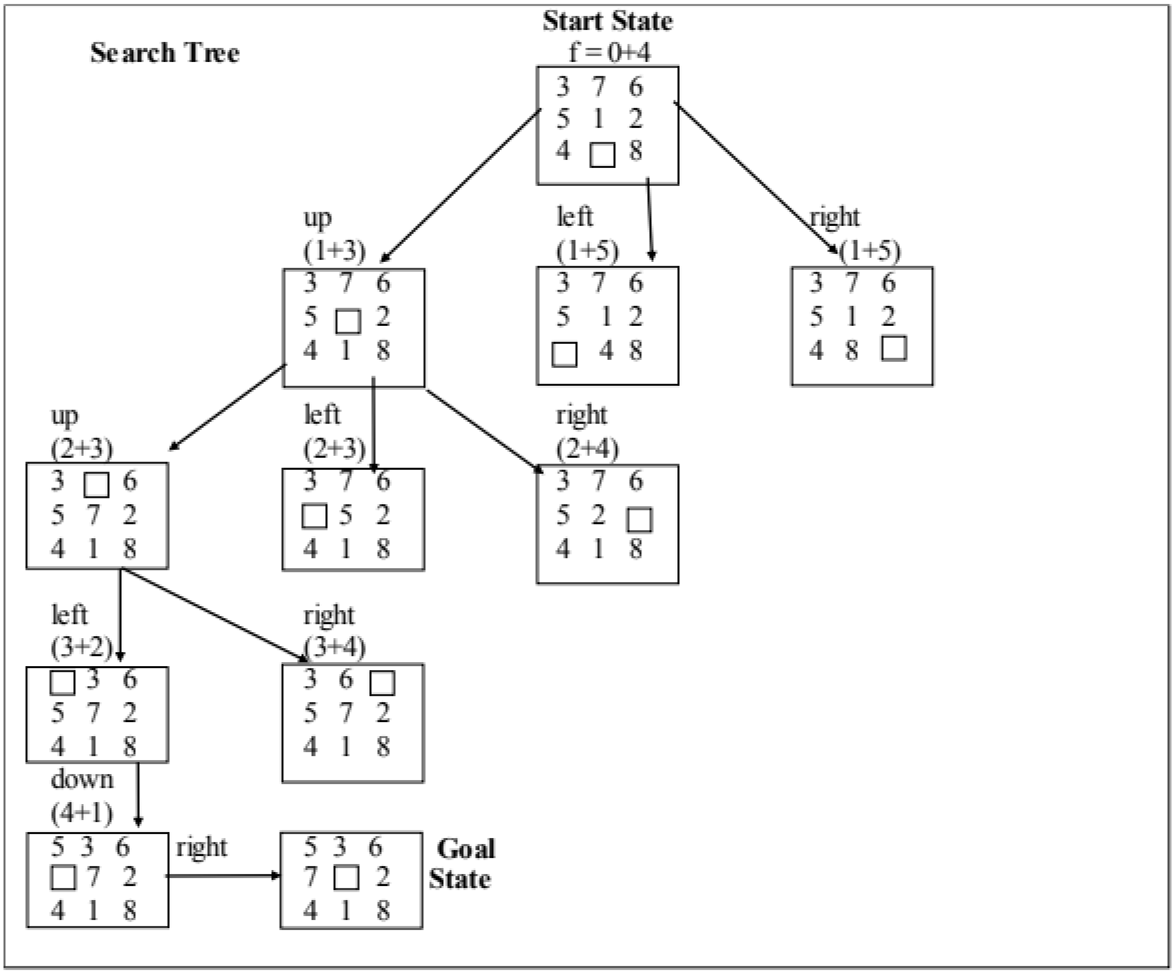
2	8	1
	4	3
7	6	5

Algorithmic Steps

Algorithm A* (with any h on search Graph)

- Input: an implicit search graph problem with cost on the arcs
- Output: the minimal cost path from start node to a goal node.
  - 1. Put the start node s on OPEN.
  - 2. If OPEN is empty, exit with failure
  - 3. Remove from OPEN and place on CLOSED a node n having minimum f.
  - 4. If n is a goal node exit successfully with a solution path obtained by tracing back the pointers from n to s.
  - 5. Otherwise, expand n generating its children and directing pointers from each child node to n.
    - For every child node n' do
      - evaluate h(n') and compute f(n') = g(n') +h(n')= g(n)+c(n,n')+h(n')
      - If n' is already on OPEN or CLOSED compare its new f with the old f. If the new value is higher, discard the node. Otherwise, replace old f with new f and reopen the node.
      - Else, put n' with its f value in the right order in OPEN
  - 6. Go to step 2.

Solved Example:



Input: Start state and goal state (any 3*3 start and goal state can be given by user) Solution should be displayed like this:

(state1, Up, f (n) =4)

(state2, Up, f (n) =5)

(state3, left, f (n) =5)

(state4, down, f (n) =5)

(state5, right, f (n) =4)

Code:

```
class Node:
```

```
def __init__(self,data,level,fval):
    """ Initialize the node with the data, level of the node and the calculated fvalue """
    self.data = data
    self.level = level
    self.fval = fval

def generate_child(self):
    """ Generate child nodes from the given node by moving the blank space
        either in the four directions {up,down,left,right} """
    x,y = self.find(self.data,'_')
    """ val_list contains position values for moving the blank space in either of
        the 4 directions [up,down,left,right] respectively. """
    val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
    children = []
    for i in val_list:
        child = self.shuffle(self.data,x,y,i[0],i[1])
        if child is not None:
            child_node = Node(child,self.level+1,0)
            children.append(child_node)
    return children

def shuffle(self,puz,x1,y1,x2,y2):
    """ Move the blank space in the given direction and if the position value are out
        of limits the return None """
    if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
        temp_puz = []
        temp_puz = self.copy(puz)
        temp = temp_puz[x2][y2]
        temp_puz[x2][y2] = temp_puz[x1][y1]
        temp_puz[x1][y1] = temp
        return temp_puz
    else:
        return None


def copy(self,root):
    """ Copy function to create a similar matrix of the given node"""
    temp = []
    for i in root:
```

```
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        """ Specifically used to find the position of the blank space """
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j


class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self,start,goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
        #print("Possible Child:")
        hf=self.h(start.data,goal) #+start.level
        # print("Heuristic Function Value: ",hf, "\n")
        return hf

    def h(self,start,goal):
        """ Calculates the different between the given puzzles """
        temp = 0
```

```
    for i in range(0,self.n):
        for j in range(0,self.n):
#           print(start[i][j],end=" ")
            # if start[i][j]=='_':
            #     continue
            if start[i][j] != goal[i][j] and start[i][j] != '_':
                temp += 1
#       print("")
    return temp


def process(self):
    """ Accept Start and Goal Puzzle state"""
    print("Enter the start state matrix ")
    start = self.accept()
    print("Enter the goal state matrix ")
    goal = self.accept()

    start = Node(start,0,0)
    start.fval = self.f(start,goal)
    print("Heuristic Function Value(hn): ",start.fval, "\n")
    print("fn=",0,"+",start.fval)
    """ Put the start node in the open list"""
    self.open.append(start)
    gn=0
    cost=0
    while True:
        cur = self.open[0]
        print("Next State:")
        for i in cur.data:
            for j in i:
                print(j,end=" ")
            print("")
        """ If the difference between current and goal node is 0 we have reached the goal node"""
        if(self.h(cur.data,goal) == 0):
            break
        hn=100000
        for i in cur.generate_child():
            i.fval = self.f(i,goal)
```

```
                hn=min(hn,i.fval)
                self.open.append(i)
            self.closed.append(cur)
            if (gn>0):
                print("Heuristic Function Value(hn): ",hn, "\n")
                print("fn=",gn,"+",hn)
            gn+=1
            del self.open[0]

            """ sort the opne list based on f value """
            self.open.sort(key = lambda x:x.fval,reverse=False)

        print("Cost:",gn+hn-1, "\n")
print("N-Puzzle Problem")
N=int(input("Enter Value of N: "))
puz = Puzzle(N)
puz.process()
```

Output:

```
N-Puzzle Problem
Enter Value of N: 3
Enter the start state matrix
3 7 6
5 1 2
4 _ 8
Enter the goal state matrix
5 3 6
7 _ 2
4 1 8
Heuristic Function Value(hn):  4

fn= 0 + 4
Next State:
3 7 6
5 1 2
4 _ 8
Next State:
3 7 6
5 _ 2
4 1 8
Heuristic Function Value(hn):  3

fn= 1 + 3
Next State:
3 7 6
_ 5 2
4 1 8
Heuristic Function Value(hn):  3

fn= 2 + 3
Next State:
3 _ 6
5 7 2
4 1 8
Heuristic Function Value(hn):  2

fn= 3 + 2
Next State:
_ 3 6
5 7 2
4 1 8
Heuristic Function Value(hn):  1
```

```
fn= 3 + 2
Next State:
_ 3 6
5 7 2
4 1 8
Heuristic Function Value(hn):  1

fn= 4 + 1
Next State:
5 3 6
_ 7 2
4 1 8
Heuristic Function Value(hn):  0

fn= 5 + 0
Next State:
5 3 6
7 _ 2
4 1 8
Cost: 5
```

Conclusion: Implementation of A* algorithm is done.