## LAB MANUAL

**AI Lab**

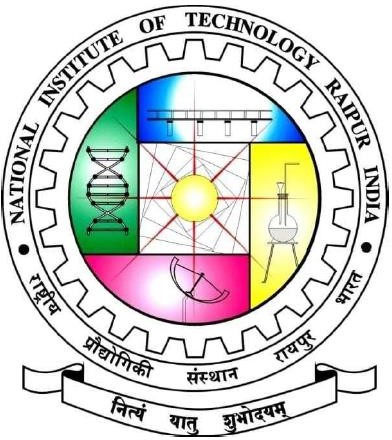
*Submitted By:*

Animesh Agrawal

Roll No.: **20118010**

Information Technology,

National Institute Of Technology Raipur



*Under the Supervision of:* **Dr. Satya Prakash Sahu** Associate Professor

Department of Information Technology National Institute of Technology, Raipur

# EXPERIMENT 1

**OBJECTIVE:** Implement the BFS algorithm to find the goal node from the tree

## DESCRIPTION:

Breadth first search is a general technique of traversing a graph. Breadth first search may use more memory but will always find the shortest path first. In this type of search the state space is represented in the form of a tree. The solution is obtained by traversing through the tree. The nodes of the tree represent the start value or starting state, various intermediate states and the final state. In this search a queue data structure is used and it is level by level traversal. Breadth first search expands nodes in order of their distance from the root. It is a path finding algorithm that is capable of always finding the solution if one exists. The solution which is found is always the optional solution. This task is completed in a very memory intensive manner. Each node in the search tree is expanded breadth wise at each level.

## ALGORITHM:

* Step 1: Place the root node inside the queue.
* Step 2: If the queue is empty then stop and return failure.
* Step 3: If the FRONT node of the queue is a goal node then stop and return success.
* Step 4: Remove the FRONT node from the queue. Process it and find all its neighbors that are in ready state then place them inside the queue in any order.
* Step 5: Go to Step 3.
* Step 6: Exit.

## COMPLEXITY:

* The time complexity of BFS is **O(V + E)**, where V is the number of nodes and E is the number of edges.
* The space complexity of BFS can be expressed as **O(V)**, where V is the number of vertices.

## CODE:

// Implement the BFS algorithm to find the goal node from the tree

#include <bits/stdc++.h>

**using namespace** std;

**class** Node{

**public**:

**int** data;

Node \*left = NULL; Node \*right = NULL; Node(**int** num){

data = num;

}

};

Node \*createTree(){

Node \*root = **new** Node(1); root->left = **new** Node(2); root->right = **new** Node(3);

root->left->left = **new** Node(4); root->left->right = **new** Node(5); root->right->right = **new** Node(6);

root->left->right->left = **new** Node(7); root->left->right->right = **new** Node(8); root->right->right->left = **new** Node(9); **return** root;

}

**bool** bfs(Node \*root, **int** target){

**if** (root == NULL){

**return false**;

}

queue<Node \*> q; q.push(root);

**while** (!q.empty()){

Node \*temp = q.front(); q.pop();

**if** (temp->data == target){

cout << "Number " << target << " found!" << endl;

**return true**;

}

**if** (temp->left != NULL){ q.push(temp->left);

}

**if** (temp->right != NULL){

q.push(temp->right);

}

}

cout << "Not Found!";

**return false**;

}

**int** main(){

Node \*root = createTree();

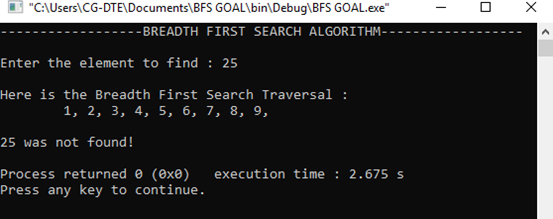
**int** x;

cout << "Enter the Number to find" << endl; cin >> x;

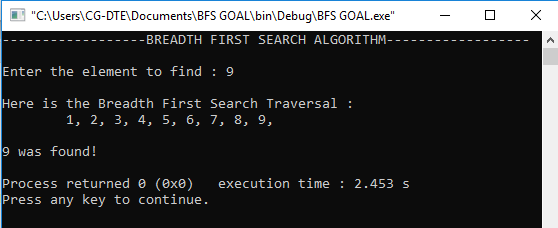
bfs(root, x);

}

Output 1:



Output 2:



# EXPERIMENT 2

**OBJECTIVE:** Implement the DFS algorithm to find the goal node from the tree

## DESCRIPTION:

DFS is also an important type of uniform search. DFS visits all the vertices in the graph. This type of algorithm always chooses to go deeper into the graph. After DFS visits all the reachable vertices from a particular source, it chooses one of the remaining undiscovered vertices and continues the search. DFS reminds the space limitation of breadth first search by always generating next a child of the deepest unexpanded nodded. The data structure stack or last in first out (LIFO) is used for DFS. One interesting property of DFS is that, the discovery and finish time of each vertex from a parenthesis structure. If we use one open parenthesis when a vertex is finished then the result is a properly nested set of parenthesis.

## ALGORITHM:

* Step 1: PUSH the starting node into the stack.
* Step 2: If the stack is empty then stop and return failure.
* Step 3: If the top node of the stack is the goal node, then stop and return success.
* Step 4: Else POP the top node from the stack and process it. Find all its neighbors that are in ready state and PUSH them into the stack in any order.
* Step 5: Go to step 3.
* Step 6: Exit.

## COMPLEXITY::

* The time complexity of the DFS algorithm is O(V+E), where V is the number of vertices and E is the number of edges in the graph.
* The space complexity of the DFS algorithm is O(V).

## CODE:

#include <bits/stdc++.h> **using namespace** std; **typedef long long** ll;

**class** Node{

**public**:

**int** data;

Node \*left = NULL; Node \*right = NULL; Node(**int** num){

data = num;

}

};

Node \*createTree(){

Node \*root = **new** Node(1); root->left = **new** Node(2); root->right = **new** Node(3);

root->left->left = **new** Node(4); root->left->right = **new** Node(5); root->right->right = **new** Node(6);

root->left->right->left = **new** Node(7); root->left->right->right = **new** Node(8); root->right->right->left = **new** Node(9); **return** root;

}

**void** dfs(Node \*root, **int** target){

**if** (root == NULL){

**return**;

}

**if** (root->data == target){

cout << "Number " << target << " found!" << endl;

}

**if** (root->left != NULL){ dfs(root->left, target);

}

**if** (root->right != NULL){ dfs(root->right, target);

}

}

**int** main(){

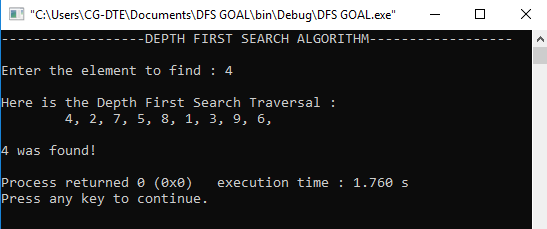
Node \*root = createTree();

**int** x;

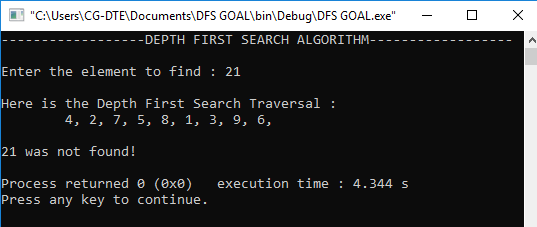
cout << "Enter the Number to find" << endl; cin >> x;

dfs(root, x);

}

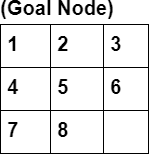
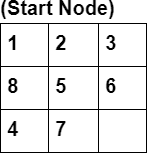
Output 1:

Output 2:



# EXPERIMENT 3

**OBJECTIVE:** Solve the 8 puzzle problem by using any search algorithm



## DESCRIPTION:

It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing numbers from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tiles in a sequence for getting the goal state.

## PROCEDURE:

The 8-puzzle problem belongs to the category of “sliding block puzzle” type of problem. The 8-puzzle is a square tray in which eight square tiles are placed. The remaining ninth square is uncovered. Each tile in the tray has a number on it. A tile that is adjacent to blank space can be slid into that space. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around. The control mechanisms for an 8-puzzle solver must keep track of the order in which operations are performed, so that the operations can be undone one at a time if necessary. The objective of the puzzles is to find a sequence of tile movements that leads from a starting configuration to a goal configuration.

## BRANCH AND BOUND:

The search for an answer node can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an answer node. It is similar to the backtracking technique but uses a BFS- like search.

There are basically three types of nodes involved in Branch and Bound

1. **Live node** is a node that has been generated but whose children have not yet been generated.
2. **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
3. **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

## Cost function:

Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as

**C(X) = g(X) + h(X)** where

g(X) = cost of reaching the current node from the root h(X) = cost of reaching an answer node from X.

## The ideal Cost function for an 8-puzzle Algorithm:

We assume that moving one tile in any direction will have a 1 unit cost. Keeping that in mind, we define a cost function for the 8-puzzle algorithm as below:

c(x) = f(x) + h(x) where

f(x) is the length of the path from root to x (the number of moves so far) and

h(x) is the number of non-blank tiles not in their goal position (the number of mis- placed tiles). There are at least h(x) moves to transform state x to a goal state

## CODE:

#include <bits/stdc++.h> **using namespace** std; #define N 3

**struct** Node{

Node \*parent; **int** mat[N][N]; **int** x, y;

**int** cost;

**int** level;

};

**int** printMatrix(**int** mat[N][N]){

**for** (**int** i = 0; i < N; i++){

**for** (**int** j = 0; j < N; j++) printf("%d ", mat[i][j]);

}

**return** 0;

}

Node \*newNode(**int** mat[N][N], **int** x, **int** y, **int** newX,

**int** newY, **int** level, Node \*parent){ Node \*node = **new** Node;

node->parent = parent;

memcpy(node->mat, mat, **sizeof** node->mat); swap(node->mat[x][y], node->mat[newX][newY]); node->cost = INT\_MAX;

node->level = level; node->x = newX; node->y = newY; **return** node;

}

**int** row[] = {1, 0, -1, 0};

**int** col[] = {0, -1, 0, 1};

**int** calculateCost(**int** initial[N][N], **int** final[N][N]){

**int** count = 0;

**for** (**int** i = 0; i < N; i++)

**for** (**int** j = 0; j < N; j++)

**if** (initial[i][j] && initial[i][j] != final[i][j]) count++;

**return** count;

}

**int** isSafe(**int** x, **int** y){

**return** (x >= 0 && x < N && y >= 0 && y < N);

}

**void** printPath(Node \*root){

**if** (root == NULL)

**return**; printPath(root->parent); printMatrix(root->mat);

printf("\n");

}

**struct** comp{

**bool operator**()(**const** Node \*lhs, **const** Node \*rhs) **const**{ **return** (lhs->cost + lhs->level) > (rhs->cost + rhs->level);

}};

**void** solve(**int** initial[N][N], **int** x, **int** y,

**int** final[N][N]){

priority\_queue<Node \*, std::vector<Node \*>, comp> pq; Node \*root = newNode(initial, x, y, x, y, 0, NULL); root->cost = calculateCost(initial, final); pq.push(root);

**while** (!pq.empty()){ Node \*min = pq.top(); pq.pop();

**if** (min->cost == 0){ printPath(min); **return**;

}

**for** (**int** i = 0; i < 4; i++){

**if** (isSafe(min->x + row[i], min->y + col[i])){ Node \*child = newNode(min->mat, min->x,

min->y, min->x + row[i], min->y + col[i],

min->level + 1, min); child->cost = calculateCost(child->mat, final); pq.push(child);

}}

}

}

**int** main()

{

**int** initial[N][N] =

{

{1, 2, 3},

{8, 5, 6},

{4, 7, 0}};

**int** final[N][N] =

{

{1, 2, 3},

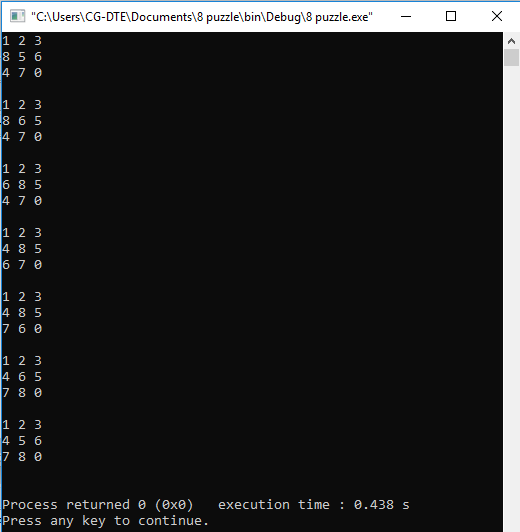
{4, 5, 6},

{7, 8, 0}};

**int** x = 1, y = 2; solve(initial, x, y, final);

}

Output :



# EXPERIMENT 4

**OBJECTIVE:** Solve the Classical Water Jug problem of 3L jug and 4L jug to measure exactly 2L in the 3L jug.

## DESCRIPTION:

Some jugs are given which should have non-calibrated properties. At least any one of the jugs should have been filled with water. Then the process through which we can divide the whole water into different jugs according to the question can be called a water jug problem.

## PROCEDURE:

Suppose that you are given 3 jugs A,B,C with capacities 8,5 and 3 liters respectively but are not calibrated (i.e. no measuring mark will be there). Jug A is filled with 8 liters of water. By a series of pouring back and forth among the 3 jugs, divide the 8 liters into 2 equal parts i.e. 4 liters in jug A and 4 liters in jug B. How? In this problem, the start state is that the jug A will contain 8 liters of water whereas jug B and jug C will be empty. The production rules involve filling a jug with some amount of water, taking from the jug A. The search will be finding the sequence of production rules which transforms the initial state to the final state. The state space for this problem can be described by a set of ordered pairs of three variables (A, B, C) where variable A represents the 8 liter jug, variable B represents the 5 liter and variable C represents the 3 liters jug respectively. You are given an m liter jug and a n liter jug. Both the jugs are initially empty. The jugs don’t have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n. (X, Y) corresponds to a state where X refers to the amount of water in Jug1 and Y refers to the amount of water in Jug2 Determine the path from the initial state (xi, yi) to the final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

1. Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
2. Fill a Jug, (0, 0)->(X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

Here, we keep exploring all the different valid cases of the states of water in the jug simultaneously until and unless we reach the required target water.

At any given state we can do either of the following operations:

1. Fill a jug
2. Empty a jug
3. Transfer water from one jug to another until either of them gets completely filled or empty.

## COMPLEXITY:

* Time Complexity: O(n\*m).
* Space Complexity: O(n\*m) Where n and m are the quantity of jug1 and jug2 respectively.

## CODE:

#include <bits/stdc++.h> **using namespace** std; **typedef** pair<**int**, **int**> pii;

**void** printpath(map<pii, pii> mp, pii u){

**if** (u.first == 0 && u.second == 0){ cout << 0 << " " << 0 << endl; **return**;

}

printpath(mp, mp[u]);

cout << u.first << " " << u.second << endl;

}

**void** BFS(**int** a, **int** b, **int** target){ map<pii, **int**> m;

**bool** isSolvable = **false**; map<pii, pii> mp; queue<pii> q; q.push(make\_pair(0, 0));

**while** (!q.empty()){

**auto** u = q.front(); q.pop();

**if** (m[u] == 1)

**continue**;

**if** ((u.first > a || u.second > b || u.first < 0 || u.second <0))

**continue**;

m[{u.first, u.second}] = 1;

**if** (u.first == target || u.second == target){ isSolvable = **true**;

printpath(mp, u);

**if** (u.first == target){

**if** (u.second != 0)

cout << u.first << " " << 0 << endl;

}

**else**{

**if** (u.first != 0)

cout << 0 << " " << u.second << endl;

}

**return**;

}

**if** (m[{u.first, b}] != 1){

q.push({u.first, b});

mp[{u.first, b}] = u;

}

**if** (m[{a, u.second}] != 1){

q.push({a, u.second});

mp[{a, u.second}] = u;

}

**int** d = b - u.second;

**if** (u.first >= d){

**int** c = u.first - d;

**if** (m[{c, b}] != 1){

q.push({c, b});

mp[{c, b}] = u;

}

}

**else**{

**int** c = u.first + u.second;

**if** (m[{0, c}] != 1){

q.push({0, c});

mp[{0, c}] = u;

}

}

d = a - u.first;

**if** (u.second >= d){

**int** c = u.second - d;

**if** (m[{a, c}] != 1){

q.push({a, c});

mp[{a, c}] = u;

}

}

**else**{

**int** c = u.first + u.second;

**if** (m[{c, 0}] != 1){

q.push({c, 0});

mp[{c, 0}] = u;

}

}

**if** (m[{u.first, 0}] != 1){

q.push({u.first, 0});

mp[{u.first, 0}] = u;

}

**if** (m[{0, u.second}] != 1){

q.push({0, u.second});

mp[{0, u.second}] = u;

}

}

**if** (!isSolvable)

cout << "No solution";

}

**int** main(){

**int** Jug1 = 4, Jug2 = 3, target = 2;

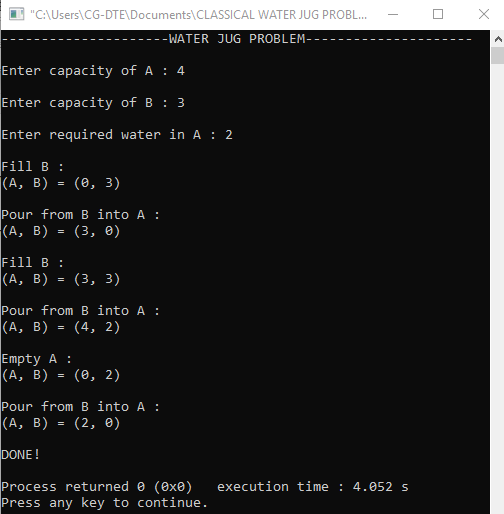
cout << "Path from initial state to solution state" << endl;

BFS(Jug1, Jug2, target);

**return** 0;

}

Output :



# EXPERIMENT 5

**OBJECTIVE:** Implement the simple hill climbing for finding the goal node.

## DESCRIPTION:

Hill climbing search algorithm is simply a loop that continuously moves in the direction of increasing value. It stops when it reaches a “peak” where no neighbor has higher value. This algorithm is considered to be one of the simplest procedures for implementing heuristic search. The hill climbing comes from that idea if you are trying to find the top of the hill and you go up direction from wherever you are. This heuristic combines the advantages of both depth first and breadth first searches into a single method

## ALGORITHM:

1. Evaluate the initial state. If it is a goal state,then stop and return success.
2. Else, continue with the starting state as considering it as a current state.
3. Continue Step 4 until a solution is found i.e. until there are no new operators left to be applied in the current state.

4.

* 1. Select an operator that has not been yet applied to the current state and apply it produces a new state.
  2. Procedure to evaluate a new state.
     1. If the current state is a goal state, then stop and return to success.
     2. If it is not a goal state but it is better than the current state, then make it the current state and proceed further.
     3. If it is not better than the current state, then continue in the loop until a solution is found.

5. Exit.

## ADVANTAGES:

* Hill climbing technique is useful in job shop scheduling, automatic programming, circuit designing, and vehicle routing and portfolio management.
* It is also helpful to solve pure optimization problems where the objective is to find the best state according to the objective function.
* It requires much less conditions than other search techniques.

## DISADVANTAGES:

The question that remains on hill climbing search is whether this hill is the highest hill possible. Unfortunately without further extensive exploration, this question cannot be answered. This technique works but as it uses local information that’s why it can be

fooled. The algorithm doesn’t maintain a search tree, so the current node data structure needs only record the state and its objective function value. It assumes that local improvement will lead to global improvement.

## CODE:

#include <bits/stdc++.h>

**using namespace** std;

**int** calcCost(**int** arr[], **int** N){

**int** c = 0;

**for** (**int** i = 0; i < N; i++){

**for** (**int** j = i + 1; j < N; j++){

**if** (arr[j] < arr[i]){ c++;

}

}

}

**return** c;

}

**void** swap(**int** arr[], **int** i, **int** j){

**int** tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;

}

**int** main(){

**int** N;

cout << "----------SIMPLE HILL CLIMBING \n\n";

cout << "Enter the number of elements : "; cin >> N;

**int** arr[N];

cout << "\nEnter Elements : ";

**for** (**int** i = 0; i < N; i++){ cin >> arr[i];

}

cout << endl;

**int** bestCost = calcCost(arr, N), newCost, swaps = 0;

**while** (bestCost > 0){

**for** (**int** i = 0; i < N - 1; i++){ swap(arr, i, i + 1);

newCost = calcCost(arr, N);

**if** (bestCost > newCost){

cout << "After swap " << ++swaps << " : ";

**for** (**int** i = 0; i < N; i++){ cout << arr[i] << ' ';

}

cout << "\n\n"; bestCost = newCost;

}

**else**{

swap(arr, i, i + 1);

}

}

}

cout << "Final answer\n";

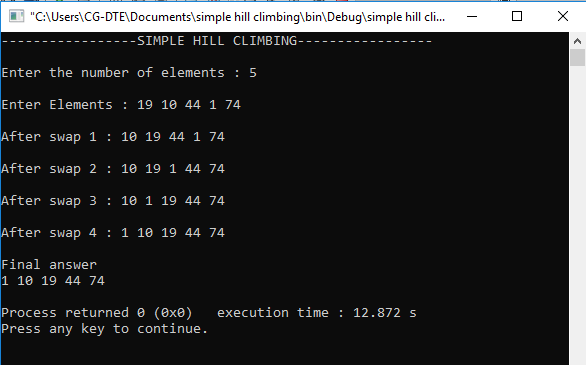
**for** (**int** i = 0; i < N; i++){ cout << arr[i] << ' ';

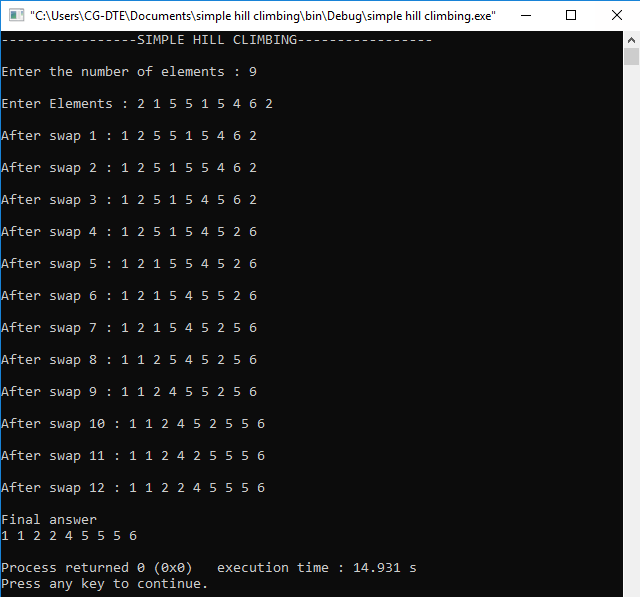
}

cout << endl;

**return** 0;

}

 Output 1:

Output 2:

# EXPERIMENT 6

**OBJECTIVE:** Implement the steepest hill climbing for finding the goal node

## DESCRIPTION:

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called steepest-ascent hill climbing or gradient search. Notice that this contrasts with the basic method in which the first state that is better than the current state is selected.

## ALGORITHM: STEEPEST- HILL CLIMBING:

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state:
   1. Let SUCC be a state such that any possible successor of the current state will be better than SUCC.
   2. For each operator that applies to the current state do
      1. Apply the operator and generate a new state.
      2. Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.
      3. If the SUCC is better than the current state, then set the current state to SUCC.

## CODE:

#include <bits/stdc++.h> **using namespace** std; **typedef** pair<**int**, **int**> pi; vector<vector<pi>> graph;

**void** addedge(**int** x, **int** y, **int** cost){ graph[x].push\_back(make\_pair(cost, y)); graph[y].push\_back(make\_pair(cost, x));

}

**void** steep\_hill\_climb(**int** source, **int** target, **int** n){ vector<**bool**> visited(n, **false**); priority\_queue<pi, vector<pi>, greater<pi>> pq; pq.push(make\_pair(0, source));

**int** s = source; visited[s] = **true**; **while** (!pq.empty()){

**int** x = pq.top().second; cout << x << " "; pq.pop();

**if** (x == target)

**break**;

**for** (**int** i = 0; i < graph[x].size(); i++){

**if** (!visited[graph[x][i].second]){ visited[graph[x][i].second] = **true**; pq.push(make\_pair(graph[x][i].first,

graph[x][i].second));

}}}

}

**int** main(){

**int** v = 14; graph.resize(v); 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); **int** source = 0; **int** target = 6;

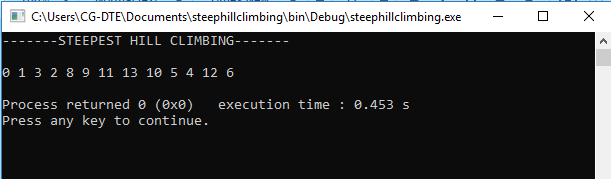
cout << "-------STEEPEST HILL CLIMBING \n\n";

steep\_hill\_climb(source, target, v); cout << endl;

**return** 0;

}

**Output :**



**EXPERIMENT 07**

**OBJECTIVE:** Implement the A\* algorithm for finding the goal node for OR Graph.

**DESCRIPTION**: A\* is a cornerstone name of many AI systems and has been used since it was developed in 1968 by Peter Hart; Nils Nilsson and Bertram Raphael. It is the combination of Dijkstra’s algorithm and Best first search. It can be used to solve many kinds of problems. A\* search finds the shortest path through a search space to goal state using heuristic function. This technique finds minimal cost solutions and is directed to a goal state called A\* search. In A\*, the \* is written for optimality purpose. The A\* algorithm also finds the lowest cost path between the start and goal state, where changing from one state to another requires some cost. A\* requires heuristic function to evaluate the cost of path that passes through the particular state. This algorithm is complete if the branching factor is finite and every action has fixed cost. A\* requires heuristic function to evaluate the cost of path that passes through the particular state. The implementation of A\* algorithm is 8-puzzle game.

It can be defined by following formula.

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

Where,

g (n): The actual cost path from the start state to the current state.

h (n): The actual cost path from the current state to goal state.

f (n): The actual cost path from the start state to the goal state.

For the implementation of A\* algorithm we will use two arrays namely OPEN and

CLOSE.

● **OPEN**: An array which contains the nodes that has been generated but has not

been yet examined.

● **CLOSE**: An array which contains the nodes that have been examined.

**ALGORITHM:**

● Step 1: Place the starting node into OPEN and find its f (n) value.

● Step 2: Remove the node from OPEN, having smallest f (n) value. If it is a goal

node then stop and return success.

● Step 3: Else remove the node from OPEN, find all its successors.

● Step 4: Find the f (n) value of all successors; place them into OPEN and place

the removed node into CLOSE.

● Step 5: Go to Step-2.

● Step 6: Exit.

**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 efficient, 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 finite and every action has

fixed cost.

● The speed execution of A\* search is highly dependant on the accuracy of the

heuristic algorithm that is used to compute h (n).

● It has complexity problems.

**CODE:**

#include <list>

#include <algorithm>

#include <iostream>

#include <stdio.h>

class point

{

public:

    point(int a = 0, int b = 0)

    {

        x = a;

        y = b;

    }

    bool operator==(const point &o)

    {

        return o.x == x && o.y == y;

    }

    point operator+(const point &o)

    {

        return point(o.x + x, o.y + y);

    }

    int x, y;

};

class map

{

public:

    map()

    {

        char t[8][8] =

            {

                {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 1, 1, 1, 0}, {0, 0, 1, 0, 0, 0, 1, 0}, {0, 0, 1, 0, 0, 0, 1, 0}, {0, 0, 1, 1, 1, 1, 1, 0}, {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0}};

        w = h = 8;

        for (int r = 0; r < h; r++)

            for (int s = 0; s < w; s++)

                m[s][r] = t[r][s];

    }

    int operator()(int x, int y)

    {

        return m[x][y];

    }

    char m[8][8];

    int w, h;

};

class node

{

public:

    bool operator==(const node &o)

    {

        return pos == o.pos;

    }

    bool operator==(const point &o)

    {

        return pos == o;

    }

    bool operator<(const node &o)

    {

        return dist + cost < o.dist + o.cost;

    }

    point pos, parent;

    int dist, cost;

};

class aStar

{

public:

    aStar()

    {

        neighbours[0] = point(-1, -1);

        neighbours[1] = point(1, -1);

        neighbours[2] = point(-1, 1);

        neighbours[3] = point(1, 1);

        neighbours[4] = point(0, -1);

        neighbours[5] = point(-1, 0);

        neighbours[6] = point(0, 1);

        neighbours[7] = point(1, 0);

    }

    int calcDist(point &p)

    {

        int x = end.x - p.x, y = end.y - p.y;

        return (x \* x + y \* y);

    }

    bool isValid(point &p)

    {

        return (p.x > -1 && p.y > -1 && p.x < m.w && p.y < m.h);

    }

    bool existPoint(point &p, int cost)

    {

        std::list<node>::iterator i;

        i = std::find(closed.begin(), closed.end(), p);

        if (i != closed.end())

        {

            if ((\*i).cost + (\*i).dist < cost)

                return true;

            else

                {

                    closed.erase(i);

                    return false;

                }

        }

        i = std::find(open.begin(), open.end(), p);

        if (i != open.end())

        {

            if ((\*i).cost + (\*i).dist < cost)

                return true;

            else

            {

                open.erase(i);

                return false;

            }

        }

        return false;

    }

    bool fillOpen(node &n)

    {

        int stepCost, nc, dist;

        point neighbour;

        for (int x = 0; x < 8; x++)

        {

            stepCost = x < 4 ? 1 : 1;

            neighbour = n.pos + neighbours[x];

            if (neighbour == end)

                return true;

            if (isValid(neighbour) && m(neighbour.x, neighbour.y) != 1)

            {

                nc = stepCost + n.cost;

                dist = calcDist(neighbour);

                if (!existPoint(neighbour, nc + dist))

                {

                    node m;

                    m.cost = nc;

                    m.dist = dist;

                    m.pos = neighbour;

                    m.parent = n.pos;

                    open.push\_back(m);

                }

            }

        }

        return false;

    }

    bool search(point &s, point &e, map &mp)

    {

        node n;

        end = e;

        start = s;

        m = mp;

        n.cost = 0;

        n.pos = s;

        n.parent = 0;

        n.dist = calcDist(s);

        open.push\_back(n);

        while (!open.empty())

        {

            // open.sort();

            node n = open.front();

            open.pop\_front();

            closed.push\_back(n);

            if (fillOpen(n))

                return true;

        }

        return false;

    }

    int path(std::list<point> &path)

    {

        path.push\_front(end);

        int cost = 1 + closed.back().cost;

        path.push\_front(closed.back().pos);

        point parent = closed.back().parent;

        for (std::list<node>::reverse\_iterator i = closed.rbegin(); i !=

                                                                    closed.rend();

             i++)

        {

            if ((\*i).pos == parent && !((\*i).pos == start))

            {

                path.push\_front((\*i).pos);

                parent = (\*i).parent;

            }

        }

        path.push\_front(start);

        return cost;

    }

    map m;

    point end, start;

    point neighbours[8];

    std::list<node> open;

    std::list<node> closed;

};

int main(int argc, char \*argv[])

{

    map m;

    point s, e(7, 7);

    aStar as;

    std::cout << “---------------A\*algorithm---------------------";

        std::cout

              << "\n\n";

    if (as.search(s, e, m))

    {

        std::list<point> path;

        int c = as.path(path);

        for (int y = -1; y < 9; y++)

        {

            for (int x = -1; x < 9; x++)

            {

                if (x < 0 || y < 0 || x > 7 || y > 7 || m(x, y) == 1)

                    std::cout << '0';

                else

                {

                    if (std::find(path.begin(), path.end(), point(x, y)) != path.end())

                        std::cout << "x";

                    else

                        std::cout << ".";

                }

}

            std::cout << "\n";

        }

        std::cout << "\nPath cost " << c << ": ";

        for (std::list<point>::iterator i = path.begin(); i != path.end(); i++)

        {

            std::cout << "(" << (\*i).x << ", " << (\*i).y << ") ";

        }

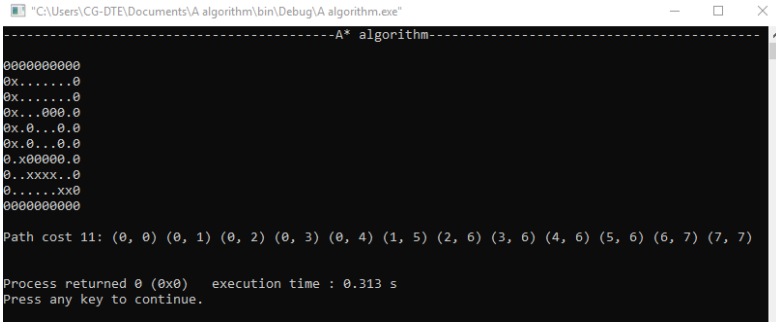
    }

    std::cout << "\n\n";

    return 0;

}

**OUTPUT:**

****

**EXPERIMENT 8**

**OBJECTIVE:** Solve the classical missionary & cannibals problem by any search algorithm.

**DESCRIPTION:** Three missionaries and three cannibals find themselves on one side of a river.they have agreed that they would all like to get to the other side.But the missionaries are not sure what else the cannibals have agreed to.so the missionaries want to manage the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals who are on the same side.the only boat available holds only two people at a time.How can everyone get across the river without the missionaries risking being eaten?So for solving the problem and to find out the solution on different states is called the Missionaries and Carnival Problem.

**PROCEDURE:** Let us take an example. Initially a boatman, Grass, Tiger and Goat is present at the left bank of the river and want to cross it. The only boat available is one capable of carrying 2 objects of portions at a time. The condition of safe crossing is that at no time the tiger present with goat, the goat present with the grass at the either side of the river. How they will cross the river? The objective of the solution is to find the sequence of their transfer from one bank of the river to the other using the boat sailing through the river satisfying these constraints.

**COMMENTS:**

● This problem requires a lot of space for its state implementation.

● It takes a lot of time to search the goal node.

● The production rules at each level of state are very strict

**CODE:**

#include <bits/stdc++.h>

using namespace std;

struct StateSTR

{

    int Mlhs;         // nr missionaries on LHS of river

    int Clhs;         // nr cannibals on LHS of river

    int pos;          // boat on LHS (0) or RHS(1) of river

    int Mrhs;         // nr missionaries on RHS of river

    int Crhs;         // nr cannibals on RHS of river

    StateSTR \*parent; // pointer to parent state

    int opUsed;

    bool operator==(const StateSTR &rhs) const

    {

        return ((Mlhs == rhs.Mlhs) && (Clhs == rhs.Clhs) &&

                (Mrhs == rhs.Mrhs) && (Crhs == rhs.Crhs) && (pos == rhs.pos));

    }

};

ostream &operator<<(ostream &out, const StateSTR &s)

{

    out << "Mlhs:" << s.Mlhs << endl;

    out << "Clhs:" << s.Clhs << endl;

    out << "Boat:" << s.pos << endl;

    out << "Mrhs:" << s.Mrhs << endl;

    out << "Crhs:" << s.Crhs << endl

        << endl;

    return out;

}

bool validState(StateSTR \*S)

{

    if (((\*S).Clhs < 0) || ((\*S).Clhs > 3))

        return false;

    if (((\*S).Crhs < 0) || ((\*S).Crhs > 3))

        return false;

    if (((\*S).Mlhs < 0) || ((\*S).Mlhs > 3))

        return false;

    if (((\*S).Mrhs < 0) || ((\*S).Mrhs > 3))

        return false;

    if (((\*S).pos != 0) && ((\*S).pos != 1))

        return false;

    if ((((\*S).Clhs > (\*S).Mlhs) && ((\*S).Mlhs > 0)) ||

        (((\*S).Crhs > (\*S).Mrhs) && ((\*S).Mrhs > 0)))

        return false;

    return true;

}

StateSTR \*nextState(StateSTR \*Z, const int j)

{

    StateSTR \*S = new StateSTR();

    (\*S) = (\*Z);

    (\*S).opUsed = j;

    switch (j)

    {

    case 0:

    {

        (\*S).pos -= 1;

        (\*S).Mlhs += 0;

        (\*S).Clhs += 1;

        (\*S).Mrhs -= 0;

        (\*S).Crhs -= 1;

    }

    break;

    case 1:

    {

        (\*S).pos -= 1;

        (\*S).Mlhs += 0;

        (\*S).Clhs += 2;

        (\*S).Mrhs -= 0;

        (\*S).Crhs -= 2;

    }

    break;

    case 2:

    {

        (\*S).pos -= 1;

        (\*S).Mlhs += 1;

        (\*S).Clhs += 0;

        (\*S).Mrhs -= 1;

        (\*S).Crhs -= 0;

    }

    break;

    case 3:

    {

        (\*S).pos -= 1;

        (\*S).Mlhs += 2;

        (\*S).Clhs += 0;

        (\*S).Mrhs -= 2;

        (\*S).Crhs -= 0;

    }

    break;

    case 4:

    {

        (\*S).pos -= 1;

        (\*S).Mlhs += 1;

        (\*S).Clhs += 1;

        (\*S).Mrhs -= 1;

        (\*S).Crhs -= 1;

    }

    break;

    case 5:

    {

        (\*S).pos += 1;

        (\*S).Mrhs += 0;

        (\*S).Crhs += 1;

        (\*S).Mlhs -= 0;

        (\*S).Clhs -= 1;

    }

    break;

    case 6:

    {

        (\*S).pos += 1;

        (\*S).Mrhs += 0;

        (\*S).Crhs += 2;

        (\*S).Mlhs -= 0;

        (\*S).Clhs -= 2;

    }

    break;

    case 7:

    {

        (\*S).pos += 1;

        (\*S).Mrhs += 1;

        (\*S).Crhs += 0;

        (\*S).Mlhs -= 1;

        (\*S).Clhs -= 0;

    }

    break;

    case 8:

    {

        (\*S).pos += 1;

        (\*S).Mrhs += 2;

        (\*S).Crhs += 0;

        (\*S).Mlhs -= 2;

        (\*S).Clhs -= 0;

    }

    break;

    case 9:

    {

        (\*S).pos += 1;

        (\*S).Mrhs += 1;

        (\*S).Crhs += 1;

        (\*S).Mlhs -= 1;

        (\*S).Clhs -= 1;

    }

    break;

    }

    return S;

}

bool notFound(StateSTR \*Y, list<StateSTR \*> OPEN, list<StateSTR \*> CLOSED)

{

    list<StateSTR \*>::iterator itr1 = OPEN.begin();

    list<StateSTR \*>::iterator itr2 = CLOSED.begin();

    for (; itr1 != OPEN.end(); itr1++)

        if ((\*(\*itr1)) == (\*Y))

            break;

    for (; itr2 != CLOSED.end(); itr2++)

        if ((\*(\*itr2)) == (\*Y))

            break;

    if ((itr1 == OPEN.end()) && (itr2 == CLOSED.end()))

        return true;

    return false;

}

void addChildren(list<StateSTR \*> &OPEN, list<StateSTR \*> &CLOSED,

                 StateSTR \*Y)

{

    StateSTR \*tState;

    for (int i = 0; i < 10; i++)

    {

        tState = nextState(Y, i);

        if ((validState(tState)) && (notFound(tState, OPEN, CLOSED)))

        {

            (\*tState).parent = Y;

            OPEN.push\_front(tState);

        }

        else

            delete tState;

    }

    return;

}

void printOP(int n)

{

    switch (n)

    {

    case 0:

        cout << "C(0,1,0)" << endl;

        break;

    case 1:

        cout << "C(0,2,0)" << endl;

        Break;

    case 2:

        cout << "C(1,0,0)" << endl;

        break;

    case 3:

        cout << "C(2,0,0)" << endl;

        break;

    case 4:

        cout << "C(1,1,0)" << endl;

        break;

    case 5:

        cout << "C(0,1,1)" << endl;

        break;

    case 6:

        cout << "C(0,2,1)" << endl;

        break;

    case 7:

        cout << "C(1,0,1)" << endl;

        break;

    case 8:

        cout << "C(2,0,1)" << endl;

        break;

    case 9:

        cout << "C(1,1,1)" << endl;

        break;

    }

}

int main()

{

    cout << "-----------MISSIONARY AND CANNIBALS PROBLEM-----------\n";

    cout << "\nMISSIONARIES AND CANNIBALS";

    bool searchResult = false;

    stack<int> opsUsed;

    StateSTR START =

        {

            3,

            3,

            0,

            0,

            0,

            NULL,

            -1};

    StateSTR GOAL =

        {

            0,

            0,

            1,

            3,

            3,

            NULL};

    StateSTR \*X;

    StateSTR \*tempState;

    list<StateSTR \*> OPEN;

    list<StateSTR \*> CLOSED;

    OPEN.push\_front(&START);

    while (!OPEN.empty())

    {

        X = OPEN.front(); // stack-like operation

        OPEN.pop\_front();

        if ((\*X) == GOAL)

        {

            searchResult = true;

            break;

        }

        else

        {

            addChildren(OPEN, CLOSED, X);

            CLOSED.push\_back(X);

        }

    }

    // Display results

    if (searchResult == true)

    {

        cout << endl

             << endl

             << "PATH" << endl

             << endl;

        for (StateSTR \*p = X; p != NULL; p = (\*p).parent)

            opsUsed.push((\*p).opUsed);

    }

    while (!opsUsed.empty())

    {

        printOP(opsUsed.top());

        opsUsed.pop();

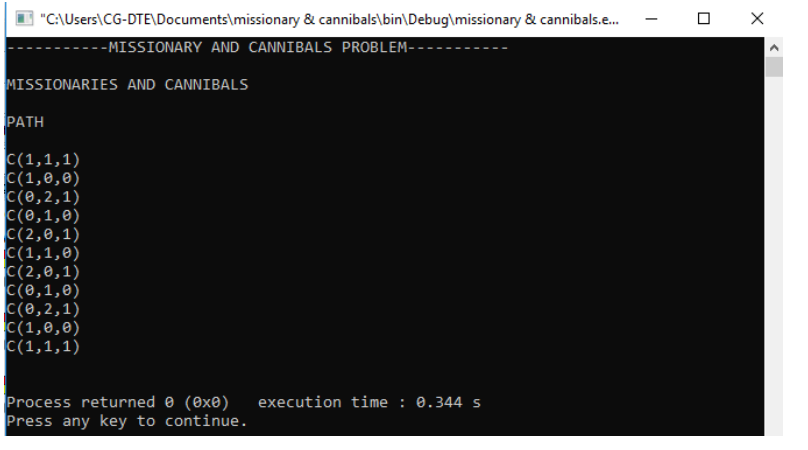
    }

    cout << endl;

    return 0;

}

**OUTPUT:**

****

**EXPERIMENT 9**

**OBJECTIVE:** Solve the classical Travelling Salesman Problem of AI by heuristic approach

**DESCRIPTION**: The traveling salesman problem is a classic problem in combinatorial optimization. This problem is to find the shortest path that a salesman should take to traverse through a list of cities and return to the origin city. The list of cities and the distance between each pair are provided. TSP is useful in various applications in real life such as planning or logistics. For example, a concert tour manager who wants to schedule a series of performances for the band must determine the shortest path for the tour to ensure reducing traveling costs and not making the band unnecessarily exhausted. This is an NP-hard problem. In simple words, it means you can not guarantee to find the shortest path within a reasonable time limit. This is not unique to TSP though. In real-world optimization problems, you frequently encounter problems for which you must find sub-optimal solutions instead of optimal ones.

**CODE:**

#include <stdio.h>

#include <iostream>

using namespace std;

int ary[10][10], completed[10], n, cost = 0;

void takeInput()

{

    int i, j;

    cout << "------------------TRAVELLING SALESMAN PROBLEM

        -- -- -- -- -- -- -- -- --";

        cout

         << "\n\nEnter the number of villages:\n";

    cin >> n;

    cout << "\nEnter the Cost Matrix\n";

    for (i = 0; i < n; i++)

    {

        cout << "\nEnter Elements of Row " << i + 1 << " :\n";

        for (j = 0; j < n; j++)

            cin >> ary[i][j];

        completed[i] = 0;

    }

    cout << "\n\nThe cost list is:\n";

    for (i = 0; i < n; i++)

    {

        cout << "\n";

        for (j = 0; j < n; j++)

            cout << "\t" << ary[i][j];

    }

}

int least(int c)

{

    int i, nc = 999;

    int min = 999, kmin;

    for (i = 0; i < n; i++)

    {

        if ((ary[c][i] != 0) && (completed[i] == 0))

            if (ary[c][i] + ary[i][c] < min)

            {

                min = ary[i][0] + ary[c][i];

                kmin = ary[c][i];

                nc = i;

            }

    }

    if (min != 999)

        cost += kmin;

    return nc;

}

void mincost(int city)

{

    int i, ncity;

    completed[city] = 1;

    cout << city + 1 << "--->";

    ncity = least(city);

    if (ncity == 999)

    {

        ncity = 0;

        cout << ncity + 1;

        cost += ary[city][ncity];

        return;

    }

    mincost(ncity);

}

int main()

{

    takeInput();

    cout << "\n\nThe Path is:\n";

    mincost(0); // passing 0 because starting vertex

    cout << "\n\nMinimum cost is :\n"

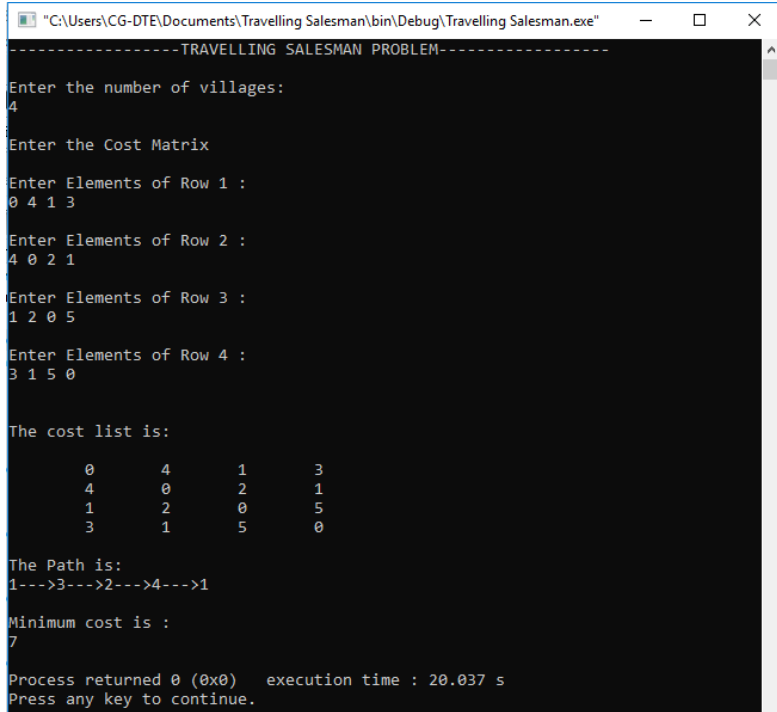
         << cost;

    cout << "\n";

    return 0;

}

**OUTPUT:**

****

**EXPERIMENT 10**

**OBJECTIVE:** Write a program to search any goal given an input graph using AO\* algorithm in AO graph

**DESCRIPTION:**

(And-Or) Graph The Depth first search and Breadth first search given earlier for

OR trees or graphs can be easily adopted by AND-OR graph. The main difference

lies in the way termination conditions are determined, since all goals following an

AND nodes must be realized; where as a single goal node following an OR node

will do. So for this purpose we are using AO\* algorithm.

Like A\* algorithm here we will use two arrays and one heuristic function.

OPEN: It contains the nodes that has been traversed but yet not been marked

solvable or unsolvable.

CLOSE: It contains the nodes that have already been processed.

h(n):The distance from current node to goal node.

**ALGORITHM:**

● Step 1: Place the starting node into OPEN.

● Step 2: Compute the most promising solution tree say T0.

● Step 3: Select a node n that is both on OPEN and a member of T0.

Remove it from OPEN and place it in CLOSE

● Step 4: If n is the terminal goal node then leveled n as solved and leveled

all the ancestors of n as solved. If the starting node is marked as solved

then success and exit.

● Step 5: If n is not a solvable node, then mark n as unsolvable. If starting

node is marked as unsolvable, then return failure and exit.

● Step 6: Expand n. Find all its successors and find their h (n) value, push

them into OPEN.

● Step 7: Return to Step 2.

● Step 8: Exit.

**CODE:**

#include <bits/stdc++.h>

using namespace std;

struct node

{

    int data;

    vector<vector<node \*> \*> v;

    bool mark;

    bool solved;

};

int edge\_cost = 0;

void insert(node \*root)

{

    cout << "\nEnter data of node : ";

    cin >> root->data;

    cout << "\nEnter number of OR nodes for value " << root->data << " : ";

    int or\_no;

    cin >> or\_no;

    for (int i = 0; i < or\_no; i++)

    {

        vector<node \*> \*ans = new vector<node \*>;

        cout << "\nEnter number of AND nodes for " << i + 1 << " or node for

            value "<<root->data<<" : ";

                                     int and\_no;

        cin >> and\_no;

        for (int j = 0; j < and\_no; j++)

        {

            node \*n = new node;

            n->solved = false;

            n->mark = false;

            insert(n);

            (\*ans).push\_back(n);

        }

        root->v.push\_back(ans);

    }

}

void aostar(node \*root)

{

    vector<node \*> \*min\_ans = new vector<node \*>;

    (\*min\_ans).push\_back(root);

    while (!root->solved)

    {

        node \*next\_node = root;

        stack<node \*> st;

        while (next\_node && next\_node->mark)

        {

            if ((next\_node->v).size() == 0)

            {

                root->solved = true;

                return;

            }

            int cost = INT\_MAX;

            st.push(next\_node);

            for (unsigned int i = 0; i < next\_node->v.size(); i++)

            {

                vector<node \*> \*ans = (next\_node->v)[i];

                vector<node \*> ans\_v = \*ans;

                int temp\_cost = 0;

                for (unsigned int j = 0; j < (ans\_v.size()); j++)

                {

                    node \*n = ans\_v[j];

                    temp\_cost += n->data;

                }

                if (temp\_cost < cost)

                {

                    min\_ans = ans;

                    cost = temp\_cost;

                }

            }

            vector<node \*> min\_ans\_v = \*min\_ans;

            next\_node = NULL;

            for (unsigned int j = 0; j < min\_ans\_v.size(); j++)

            {

                if (min\_ans\_v[j]->mark)

                {

                    next\_node = min\_ans\_v[j];

                    break;

                }

            }

        }

        vector<node \*> min\_ans\_v = \*min\_ans;

        for (unsigned int j = 0; j < min\_ans\_v.size(); j++)

        {

            node \*n = min\_ans\_v[j];

            cout << "Exploring : " << n->data << endl;

            int final\_cost = INT\_MAX;

            if (n->v.size() == 0)

            {

                n->mark = true;

            }

            else

            {

                for (unsigned int i = 0; i < n->v.size(); i++)

                {

                    vector<node \*> \*ans = (n->v)[i];

                    vector<node \*> ans\_v = \*ans;

                    int temp\_cost = 0;

                    for (unsigned int j = 0; j < (ans\_v.size()); j++)

                    {

                        node \*n = ans\_v[j];

                        temp\_cost += n->data;

                        temp\_cost += edge\_cost;

                    }

                    if (temp\_cost < final\_cost)

                    {

                        final\_cost = temp\_cost;

                    }

                }

                n->data = final\_cost;

                n->mark = true;

            }

            cout << "Marked : " << n->data << endl;

        }

        for (int i = 0; i < 20; i++)

            cout << "=";

        cout << endl;

        while (!st.empty())

        {

            node \*n = st.top();

            cout << n->data << " ";

            st.pop();

            int final\_cost = INT\_MAX;

            for (unsigned int i = 0; i < n->v.size(); i++)

            {

                vector<node \*> \*ans = (n->v)[i];

                vector<node \*> ans\_v = \*ans;

                int temp\_cost = 0;

                for (unsigned int j = 0; j < (ans\_v.size()); j++)

                {

                    node \*n = ans\_v[j];

                    temp\_cost += n->data;

                    temp\_cost += edge\_cost;

                }

                if (temp\_cost < final\_cost)

                {

                    min\_ans = ans;

                    final\_cost = temp\_cost;

                }

            }

            n->data = final\_cost;

        }

        cout << endl;

        next\_node = root;

    }

}

void print(node \*root)

{

    if (root)

    {

        cout << root->data << " ";

        vector<vector<node \*> \*> vec = root->v;

        for (unsigned int i = 0; i < (root->v).size(); i++)

        {

            vector<node \*> \*ans = (root->v)[i];

            vector<node \*> ans\_v = \*ans;

            for (unsigned int j = 0; j < ans\_v.size(); j++)

            {

                node \*n = ans\_v[j];

                print(n);

            }

        }

    }

    return;

}

int main()

{

    cout << "--------------------------AO\* ALGORITHM--------------------------\n";

    node \*root = new node;

    root->solved = false;

    root->mark = false;

    insert(root);

    cout << endl;

    cout << "\nEnter the edge cost : ";

    cin >> edge\_cost;

    cout << endl;

    cout << "\nThe tree is as follows : ";

    print(root);

    cout << endl

         << endl;

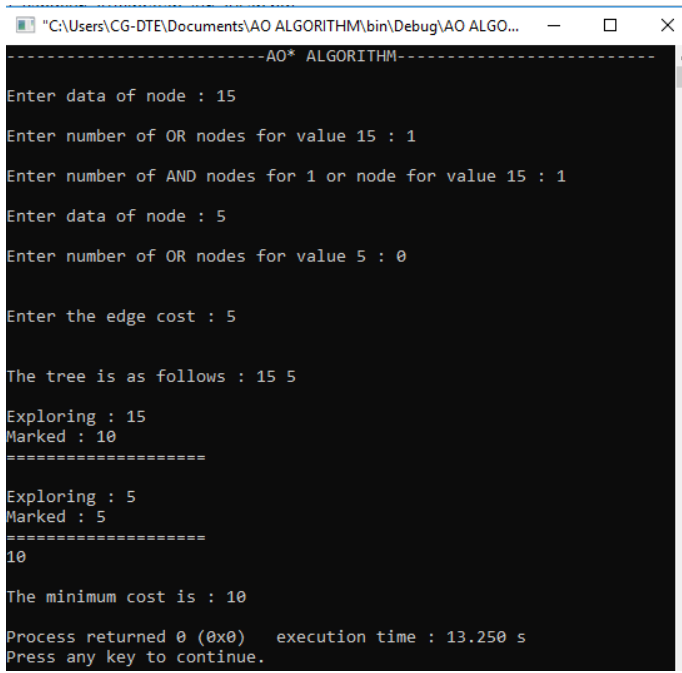
    aostar(root);

    cout << "\nThe minimum cost is : " << root->data << endl;

    return 0;

}

**OUTPUT:**

****

**EXPERIMENT 11**

**OBJECTIVE:** Implement the MINIMAX algorithm for any game playing.

**DESCRIPTION:**

Minimax is a kind of backtracking algorithm that is used in decision making and game

theory to find the optimal move for a player, assuming that your opponent also plays

optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe,

Backgammon, Mancala, Chess, etc.In Minimax the two players are called maximizer

and minimizer. The maximizer tries to get the highest score possible while the

minimizer tries to do the opposite and get the lowest score possible.Every board state

has a value associated with it. In a given state if the maximizer has upper hand then,

the score of the board will tend to be some positive value. If the minimizer has the upper

hand in that board state then it will tend to be some negative value. The values of the

board are calculated by some heuristics which are unique for every type of game.

Let us combine minimax and evaluation function to write a proper Tic-Tac-Toe AI

(Artificial Intelligence) that plays a perfect game.This AI will consider all possible

scenarios and makes the most optimal move.

**Finding the Best Move:**

We shall be introducing a new function called findBestMove(). This function evaluates

all the available moves using minimax() and then returns the best move the maximizer

can make. The pseudocode is as follows :

function findBestMove(board):

bestMove = NULL

for each move in board :

if current move is better than bestMove

bestMove = current move

return bestMove

**Minimax :**

To check whether or not the current move is better than the best move we take the help

of minimax() function which will consider all the possible ways the game can go and

returns the best value for that move, assuming the opponent also plays optimally

The code for the maximizer and minimizer in the minimax() function is similar to

findBestMove(), the only difference is, instead of returning a move, it will return a value.

Here is the pseudocode :

function minimax(board, depth, isMaximizingPlayer):

if current board state is a terminal state :

return value of the board

if isMaximizingPlayer :

bestVal = -INFINITY

for each move in board :

value = minimax(board, depth+1, false)

bestVal = max( bestVal, value)

return bestVal

else :

bestVal = +INFINITY

for each move in board :

value = minimax(board, depth+1, true)

bestVal = min( bestVal, value)

return bestVal

**Checking for GameOver state :**

To check whether the game is over and to make sure there are no moves left we use

isMovesLeft() function. It is a simple straightforward function which checks whether a

move is available or not and returns true or false respectively.Pseudocode is as follows:

function isMovesLeft(board):

for each cell in board:

if current cell is empty:

return true

return false

**Making our AI smarter :**

One final step is to make our AI a little bit smarter. Even though the following AI plays

perfectly, it might choose to make a move which will result in a slower victory or a faster

loss. Lets take an example and explain it.

Assume that there are 2 possible ways for X to win the game from a give board state.

Move A : X can win in 2 move

Move B : X can win in 4 moves

Our evaluation function will return a value of +10 for both moves A and B. Even though

the move A is better because it ensures a faster victory, our AI may choose B

sometimes. To overcome this problem we subtract the depth value from the evaluated

score. This means that in case of a victory it will choose a the victory which takes least

number of moves and in case of a loss it will try to prolong the game and play as many

moves as possible. So the new evaluated value will be

Move A will have a value of +10 – 2 = 8

Move B will have a value of +10 – 4 = 6

Now since move A has a higher score compared to move B our AI will choose move A

over move B. The same thing must be applied to the minimizer. Instead of subtracting

the depth we add the depth value as the minimizer always tries to get, as negative a

value as possible. We can subtract the depth either inside the evaluation function or

outside it. Anywhere is fine. I have chosen to do it outside the function. Pseudocode

implementation is as follows.

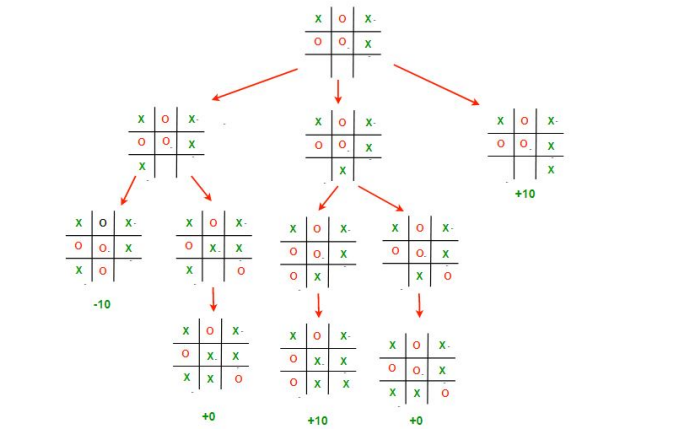
if maximizer has won:

return WIN\_SCORE – depth

else if minimizer has won:

return LOOSE\_SCORE + depth

**EXPLANATION: GAME TREE**

****

**CODE:**

#include <bits/stdc++.h>

using namespace std;

struct Move

{

    int row, col;

};

char player = 'x', opponent = 'o';

// This function returns true if there are moves remaining on the board.

// It returns false if there are no moves left to play.

bool isMovesLeft(char board[3][3])

{

    for (int i = 0; i < 3; i++)

        for (int j = 0; j < 3; j++)

            if (board[i][j] == '\_')

                return true;

    return false;

}

// This is the evaluation function

int evaluate(char b[3][3])

{

    // Checking for Rows for X or O victory.

    for (int row = 0; row < 3; row++)

    {

        if (b[row][0] == b[row][1] && b[row][1] == b[row][2])

        {

            if (b[row][0] == player)

                return +10;

            else if (b[row][0] == opponent)

                return -10;

        }

    }

    // Checking for Columns for X or O victory.

    for (int col = 0; col < 3; col++)

    {

        if (b[0][col] == b[1][col] && b[1][col] == b[2][col])

        {

            if (b[0][col] == player)

                return +10;

            else if (b[0][col] == opponent)

                return -10;

        }

    }

    // Checking for Diagonals for X or O victory.

    if (b[0][0] == b[1][1] && b[1][1] == b[2][2])

    {

        if (b[0][0] == player)

            return +10;

        else if (b[0][0] == opponent)

            return -10;

    }

    if (b[0][2] == b[1][1] && b[1][1] == b[2][0])

    {

        if (b[0][2] == player)

            return +10;

        else if (b[0][2] == opponent)

            return -10;

    }

    // Else if none of them have won then return 0

    return 0;

}

// This is the minimax function. It considers all the possible ways the game can go

and returns the value of the board int minimax(char board[3][3], int depth, bool isMax)

{

    int score = evaluate(board);

    // If Maximizer has won the game return his/her evaluated score

    if (score == 10)

        return score;

    // If Minimizer has won the game return his/her evaluated score

    if (score == -10)

        return score;

    // If there are no more moves and no winner then it is a tie

    if (isMovesLeft(board) == false)

        return 0;

    // If this maximizer's move

    if (isMax)

    {

        int best = -1000;

        // Traverse all cells

        for (int i = 0; i < 3; i++)

        {

            for (int j = 0; j < 3; j++)

            {

                // Check if cell is empty

                if (board[i][j] == '\_')

                {

                    // Make the move

                    board[i][j] = player;

                    // Call minimax recursively and

                    // choose the maximum value

                    best = max(best, minimax(board,

                                             depth + 1, !isMax));

                    // Undo the move

                    board[i][j] = '\_';

                }

            }

        }

        return best;

    }

    // If this minimizer's move

    else

    {

        int best = 1000;

        // Traverse all cells

        for (int i = 0; i < 3; i++)

        {

            for (int j = 0; j < 3; j++)

            {

                // Check if cell is empty

                if (board[i][j] == '\_')

                {

                    // Make the move

                    board[i][j] = opponent;

                    // Call minimax recursively and

                    // choose the minimum value

                    best = min(best, minimax(board,

                                             depth + 1, !isMax));

                    // Undo the move

                    board[i][j] = '\_';

                }

            }

        }

        return best;

    }

}

// This will return the best possible move for the player

Move findBestMove(char board[3][3])

{

    int bestVal = -1000;

    Move bestMove;

    bestMove.row = -1;

    bestMove.col = -1;

    // Traverse all cells, evaluate minimax function for all empty cells.

    // And return the cell with optimal value.

    for (int i = 0; i < 3; i++)

    {

        for (int j = 0; j < 3; j++)

        {

            // Check if cell is empty

            if (board[i][j] == '\_')

            {

                // Make the move

                board[i][j] = player;

                // compute evaluation function for this move

                int moveVal = minimax(board, 0, false);

                // Undo the move

                board[i][j] = '\_';

                // If the value of the current move is more

                than the best value, then update best if (moveVal > bestVal)

                {

                    bestMove.row = i;

                    bestMove.col = j;

                    bestVal = moveVal;

                }

            }

        }

    }

    printf("\nThe value of the best Move is : %d\n\n", bestVal);

    return bestMove;

}

// Driver code

int main()

{

    char board[3][3] =

        {

            {'x', 'o', 'x'},

            {'o', 'o', 'x'},

            {'\_', '\_', '\_'}};

    cout << "------------------------MINIMAX ALGORITHM------------------------";

    cout << "\n\n";

    cout << "The given Board is :\n\n";

    for (int i = 0; i < 3; ++i)

    {

        for (int j = 0; j < 3; ++j)

        {

            cout << '\t' << board[i][j] << ' ';

        }

        cout << endl

             << endl;

    }

    Move bestMove = findBestMove(board);

    printf("The Optimal Move is :\n\n");

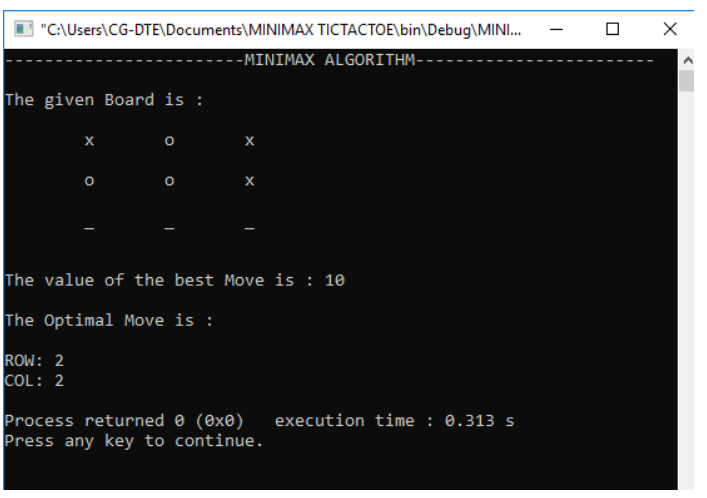
    printf("ROW: %d\n", bestMove.row);

    printf("COL: %d\n", bestMove.col);

    return 0;

}

**OUTPUT:**

****