Date: 28/01/2022

<u>Experiment #1</u>

Implementation of Camel and Banana Problem

Title:

To implement the Camel and Banana Problem in C++.

Problem Description:

A person has 3000 bananas and a camel. The person wants to transport the maximum number of bananas to a destination which is 1000 KMs away, using only the camel as a mode of transportation. The camel cannot carry more than 1000 bananas at a time and eats a banana every km it travels. What is the maximum number of bananas that can be transferred to the destination using only camel (no other mode of transportation is allowed).

Solution:

So we can conclude the following points from the question:

- We have a total of 3000 bananas.
- The destination is 1000KMs
- Only 1 mode of transport.
- Camel can carry a maximum of 1000 banana at a time.
- Camel eats a banana every km it travels.

So the solution over here is to have intermediate drop points, then, the camel can make several short trips in between. Also, we try to maintain the number of bananas at each point to be multiple of 1000.

Let’s have 2 drop points in between the source and destination. With 3000 bananas at the source. 2000 at a first intermediate point and 1000 at 2nd intermediate point.

Source——————IP1———————IP2—————————-Destination

(3000 b x km) (2000 b y km) (1000b z km)

————————> | ——————> |—————————>

<————————- | <————— |

————————-> | ——————>|

<———————— | |

————————-> | |

- To go from **Source** to **IP1** point camel has to take a total of 5 trips 3 forward and 2 backward. Since we have 3000 bananas to transport. So it has 5x bananas, as the distance between the **Source** and **IP1** is x km and the camel had 5 trips.

- The same way from **IP1** to **IP2,** camel has to take a total of 3 trips, 2 forward and 1 backward. Since we have 2000 bananas to transport. So it has 3y bananas, as the distance between **IP1** and **IP2** is y km and the camel had 3 trips.

- At last from **IP2** to the **Destination** only 1 forward move. From **IP2** to **Destination** its z bananas.

We now try to calculate the distance between the points:

3000 – 5x = 2000 so we get x = 200

2000-3y = 1000 so we get y = 332 but here the distance is also the number of bananas and it cannot be fraction so we take y =332 and at IP2 we have the number of bananas equal 1001, so its 2000-3y = 1001

So the remaining distance to the market is 1000 -x-y =z i.e.

1000-200-332 => z =468.

Now, there are 1001 bananas at IP2.

So from IP2 to the destination point camel eats 468 bananas. The remaining bananas are 1001-468=533.

So the maximum number of bananas that can be transferred is 533.

C++ Code (Input and Output):

camel_banana_prob.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

// Stores the overlapping state
int sol[1001][3001];

// Recursive function to find the maximum
// number of bananas that can be transferred
// to A distance
int countBanana(int A, int B, int C)
{
    // Case 1:
    // If count of bananas is less that the given distance
    if (B <= A) {
        return 0;
    }

    // Case 2:
    // If  count of bananas is less that camel's capacity
    if (B <= C) {
        return B - A;
    }

    // Case 3:
    // If distance = 0
    if (A == 0) {
        return B;
    }

```

```cpp
    if (sol[A][B] != -1) {
        return sol[A][B];
    }

    int maxCount = INT_MIN;

    int trip = B % C == 0 ? ((2 * B) / C) - 1
                          : ((2 * B) / C) + 1;

    for (int i = 1; i <= A; i++) {
        int curCount
            = countBanana(A - i, B - trip * i, C);

        if (curCount > maxCount) {
            maxCount = curCount;
            sol[A][B] = maxCount;
        }
    }

    return maxCount;
}

// Function to find the maximum number of
// bananas that can be transferred
int solution(int A, int B, int C)
{
    memset(sol, -1, sizeof(sol));
    return countBanana(A, B, C);
}
```

```cpp
int main()
{
    int A = 1000;
    int B = 3000;
    int C = 1000;
    cout << "Maximum number of bananas that can be transferred is " << solution(A, B, C);

    return 0;
}
```

```
Maximum number of bananas that can be transferred is 533

...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

The Camel and Banana problem is successfully implemented and tested in C++.

Date:11/02/2022

Experiment #2

Implementation of Vacuum Cleaner Problem

Aim :

To implement the Vacuum Cleaner Problem in Python.

Problem:

Vacuum cleaner problem is a well-known search problem for an agent which works on Artificial Intelligence. In this problem, our vacuum cleaner is our agent.

An agent can either perceive some information from the environment or can perform some actions on the environment. This work is done through the two main components of an agent: Sensors and Actuators. an agent can be anything that can be viewed as:

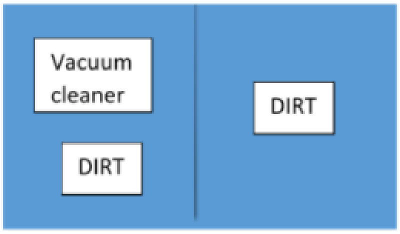
- Perceiving its environment through sensors
- Acting upon the environment through actuators

It is a goal-based agent, and the goal of this agent, which is the vacuum cleaner, is to clean up the whole area. So, in the classical vacuum cleaner problem, we have two rooms and one vacuum cleaner. There is dirt in both the rooms and it is to be cleaned. The vacuum cleaner is present in any one of these rooms. So, we have to reach a state in which both the rooms are clean and are dust free.

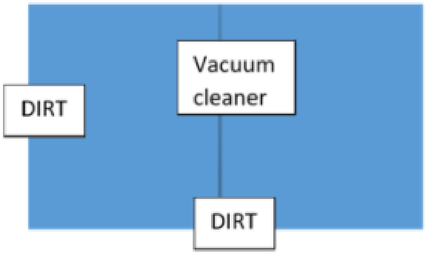
State Space Diagrams:

There are eight possible states possible in our vacuum cleaner problem. These can be well illustrated with the help of the following diagrams:

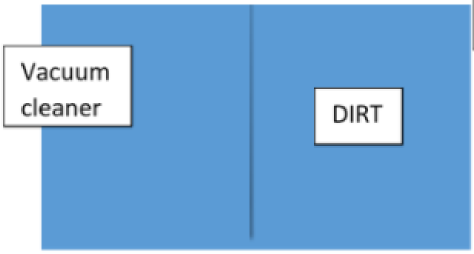
1



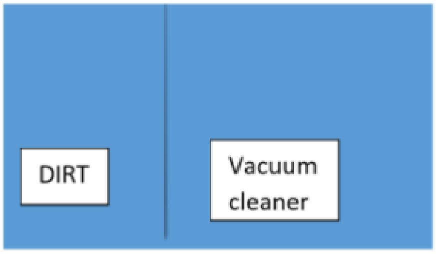
2



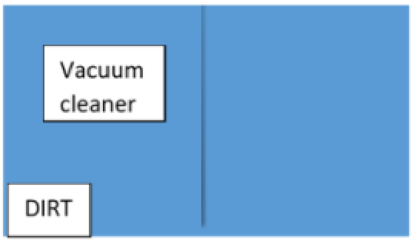
3



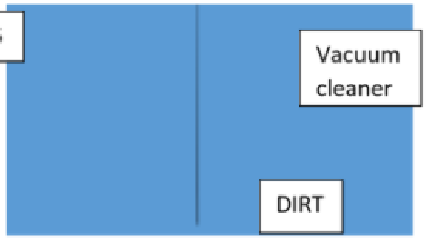
4



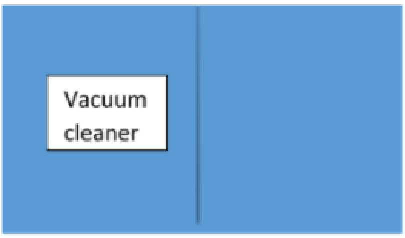
5



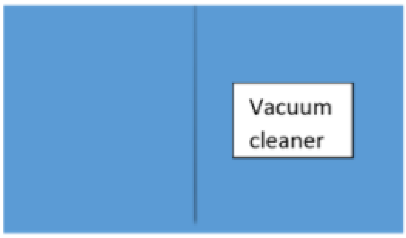
6



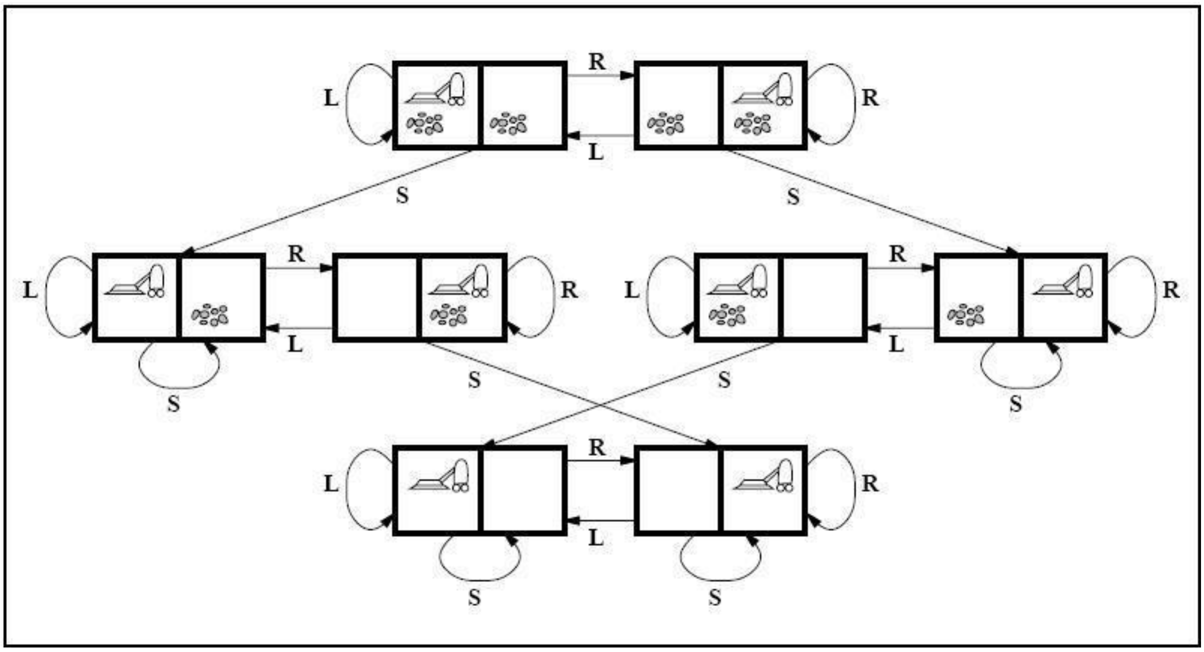
7



8



State Space Graph:



Problem Breakdown and Solution:

The states 1 and 2 are our initial states and state 7 and state 8 are our final states (goal states). This means that, initially, both the rooms are full of dirt and the **vacuum cleaner** can reside in any room. And to reach the final goal state, both the rooms should be clean and the **vacuum cleaner** again can reside in any of the two rooms.

The **vacuum cleaner** can perform the following functions: move left, move right, move forward, move backward and to suck dust. But as there are only two rooms in our problem, the vacuum cleaner performs only the following functions here: move left, move right and suck.

Here the performance of our agent (vacuum cleaner) depends upon many factors such as time taken in cleaning, the path followed in cleaning, the number of moves the agent takes in total, etc. But we consider two main factors for estimating the performance of the agent. They are:

1. **Search Cost:** How long the agent takes to come up with the solution.
2. **Path cost:** How expensive each action in the solution is.

function REFLEX-VACUUM-AGENT([*location*,*status*]) **returns** action

if *status* = *Dirty* **then return** *Suck*
else if *location* = *A* **then return** *Right*
else if *location* = *B* **then return** *Left*

Python Code (Input and Output):

```
def vacuum_world():
        # initializing goal_state
        # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum")
#user_input of location vacuum is placed
    status_input = input("Enter status of " + location_input) #user_input if location is dirty or clean
    status_input_complement = input("Enter status of other room")
    print("Initial Location Condition" + str(goal_state))
    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            # suck the dirt  and mark it as clean
            goal_state['A'] = '0'
            cost += 1                      #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")
```

```python
            if status_input_complement == '1':

                # if B is Dirty

                print("Location B is Dirty.")

                print("Moving right to the Location B. ")

                cost += 1                       #cost for moving right

                print("COST for moving RIGHT" + str(cost))

                # such the dirt and mark it as clean

                goal_state['B'] = '0'

                cost += 1                       #cost for such

                print("COST for SUCK " + str(cost))

                print("Location B has been Cleaned. ")

            else:

                print("No action" + str(cost))

                # suck and mark clean

                print("Location B is already clean.")

        if status_input == '0':

            print("Location A is already clean ")

            if status_input_complement == '1':# if B is Dirty

                print("Location B is Dirty.")

                print("Moving RIGHT to the Location B. ")

                cost += 1                       #cost for moving right
```

```python
                print("COST for moving RIGHT " + str(cost))

                # suck the dirt and mark it as clean

                goal_state['B'] = '0'

                cost += 1                   #cost for suck

                print("Cost for SUCK" + str(cost))

                print("Location B has been Cleaned. ")

            else:

                print("No action " + str(cost))

                print(cost)

                # suck and mark clean

                print("Location B is already clean.")

    else:

        print("Vacuum is placed in location B")

        # Location B is Dirty.

        if status_input == '1':

            print("Location B is Dirty.")

            # suck the dirt  and mark it as clean

            goal_state['B'] = '0'

            cost += 1  # cost for suck

            print("COST for CLEANING " + str(cost))

            print("Location B has been Cleaned.")
```

```python
            if status_input_complement == '1':

                # if A is Dirty

                print("Location A is Dirty.")

                print("Moving LEFT to the Location A. ")

                cost += 1  # cost for moving right

                print("COST for moving LEFT" + str(cost))

                # suck the dirt and mark it as clean

                goal_state['A'] = '0'

                cost += 1  # cost for suck

                print("COST for SUCK " + str(cost))

                print("Location A has been Cleaned.")

        else:

            print(cost)

            # suck and mark clean

            print("Location B is already clean.")

            if status_input_complement == '1':  # if A is Dirty

                print("Location A is Dirty.")

                print("Moving LEFT to the Location A. ")

                cost += 1  # cost for moving right

                print("COST for moving LEFT " + str(cost))

```

```
            # suck and mark clean

            print("Location B is already clean.")

            if status_input_complement == '1':  # if A is Dirty

                print("Location A is Dirty.")

                print("Moving LEFT to the Location A. ")

                cost += 1  # cost for moving right

                print("COST for moving LEFT " + str(cost))

                # suck the dirt and mark it as clean

                goal_state['A'] = '0'

                cost += 1  # cost for suck

                print("Cost for SUCK " + str(cost))

                print("Location A has been Cleaned. ")

            else:

                print("No action " + str(cost))

                # suck and mark clean

                print("Location A is already clean.")

    # done cleaning

    print("GOAL STATE: ")

    print(goal_state)

    print("Performance Measurement: " + str(cost))

vacuum_world()
```

Output:

```
Enter Location of Vacuum B
Enter status of B1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING 1
Location B has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT2
COST for SUCK 3
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

Result:

The Vacuum Cleaner problem is successfully implemented and tested in Python.

Date:18/02/2022

<u>Experiment #3</u>
Implementation of M Coloring Problem

Aim:-

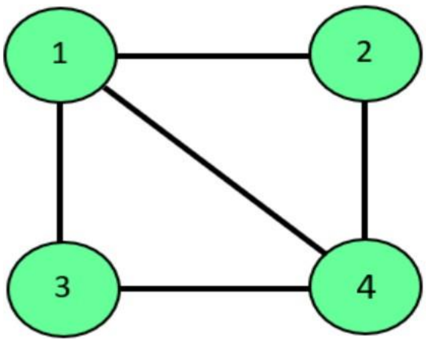
To implement the M coloring problem in C++.

Problem:-

Given an undirected graph and a number m, determine if the graph can be coloured with at most m colours such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.

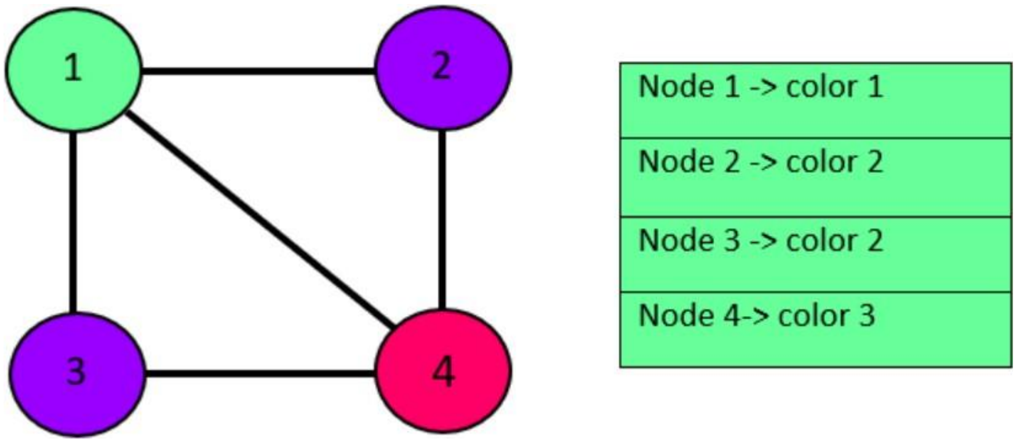
Example:-

Given an adjacency matrix of an undirected graph and a number M having a value of 3.



0	1	1	1
1	0	0	1
1	0	0	1
1	1	1	0

We've to check if we can color the above four vertices, such that no two adjacent vertices have the same color. After assigning the colors, the graph will look like this.



In this graph we can see that nodes 2 and 3 are not adjacent and have the same color, the remaining nodes have different colors. This is one possible solution. The m-coloring problem can have multiple solutions. Now, let’s move on to the solution approach of this problem.

Solution:-

Backtracking is a general algorithm for solving constraint satisfaction problems. It accomplishes this by constructing a solution incrementally, one component at a time, discarding any solutions that fail to satisfy the problem's criteria at any point in time. Generate all possible configurations of colors.

We'd assign a color to each vertex from 1 to m and see if it has a different color than its next vertex. If we obtain a configuration in which each node is colored from 1 to m, and adjacent vertices are of different colors, this will be the answer.

Time Complexity: O(M^V).
Reason: Choosing out of M given colors for V vertices will lead to an **O(M^V)** combination. So the time complexity is O(M^V).

Space Complexity: O(V).
Reason: The m_Coloring function will require O(V) space for the recursive call stack.
The above two approaches have the same time complexity regardless of different algorithms

Code:-

```
#include <bits/stdc++.h>
using namespace std;
// Number of vertices in the Adj_matrix
#define V 4
//Function to display
void Display(int color[])
{
cout << "The colors given to vertices are:"<<endl;
for (int i = 0; i < V; i++)
cout <<"Vertex "<<i+1<<" is given color:" << color[i]<<endl;
cout <<endl;
}
//Function to check constraints
bool satisfyConstraints(int v, bool Adj_matrix[V][V],int color[], int c)
{
for(int i = 0; i < V; i++)
{
    if (Adj_matrix[v][i] && c == color[i])
return false;
}
```

```
return true;

}
bool m_Coloring_Helper(bool Adj_matrix[V][V], int m, int color[], int v)
{
//If all vertices are assigned a color
if (v == V)
return true;
//Try different colors to vertex v
for(int c = 1; c <= m; c++)
{
if (satisfyConstraints(v, Adj_matrix, color, c))
{
color[v] = c;
//Assign colors to rest of the vertices
if (m_Coloring_Helper(Adj_matrix, m, color, v + 1) == true)
return true;
//Backtrack
color[v] = 0;
}
}
// If no color can be assigned
return false;
}
bool m_Coloring(bool Adj_matrix[V][V], int m)
{
// Initialize all color values as 0.
```

```cpp
int color[V];
for(int i = 0; i < V; i++)
{
color[i] = 0;
}
if (m_Coloring_Helper(Adj_matrix, m, color, 0) == false)
{
cout << "No such arrangement exists!!";
return false;
}
// Print the solution
Display(color);
return true;
}
int main()
{
// The adjacency matrix of the graph
    bool Adj_matrix[V][V] = {
{ 0, 1, 1, 1 },

{ 1, 0, 0, 1 },

{ 1, 0, 0, 1 },

{ 1, 1, 1, 0 },
};
// Number of colors
int m = 3;
m_Coloring(Adj_matrix, m);
return 0;
}
```

```
The colors given to vertices are:
Vertex 1 is given color:1
Vertex 2 is given color:2
Vertex 3 is given color:2
Vertex 4 is given color:3


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:-

The M coloring problem is successfully implemented and tested in C++.

Date:04/03/2022

Experiment #4

Implementation of Stepping Numbers

Aim: -

To implement Stepping numbers problem in C++.

Problem: -

Given two integers ‘n’ and ‘m’, find all the stepping numbers in range [n, m]. A number is called stepping number if all adjacent digits have an absolute difference of 1. 321 is a Stepping Number while 421 is not.

Example: -

```
Input: n = 0, m = 21
Output: 0 1 2 3 4 5 6 7 8 9 10 12 21

Input: n = 10, m = 15
Output: 10, 12
```

Solution: -

The idea is to use a Breadth First Search Traversal OR Depth First Search Traversal. Every node in the graph represents a stepping number; there will be a directed edge from a node U to V if V can be transformed from U. (U and V are Stepping Numbers) A Stepping Number V can be transformed from U in following manner.

lastDigit refers to the last digit of U (i.e. U % 10)

An adjacent number V can be:

U*10 + lastDigit + 1 (Neighbor A)

U*10 + lastDigit – 1 (Neighbor B)

By applying above operations a new digit is appended to U, it is either lastDigit-1 or lastDigit+1, so that the new number V formed from U is also a Stepping Number.

Therefore, every Node will have at most 2 neighboring Nodes.

Edge Cases: When the last digit of U is 0 or 9

Case 1: lastDigit is 0 : In this case only digit '1' can be appended.

Case 2: lastDigit is 9 : In this case only digit '8' can be appended.

Every single digit number is considered as a stepping Number, so bfs traversal for every digit will give all the stepping numbers starting from that digit.

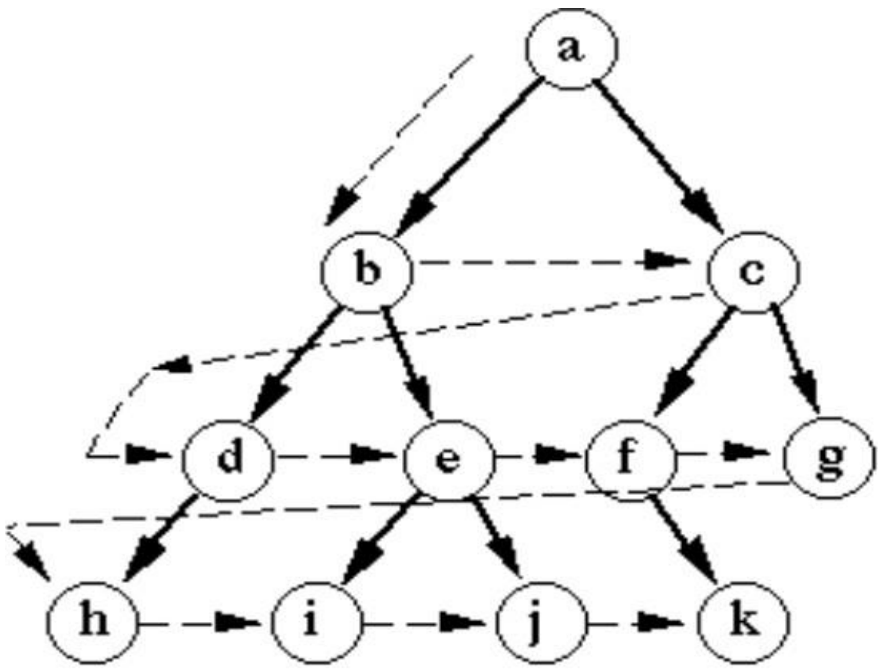
Do a bfs/dfs traversal for all the numbers from [0,9].

Time Complexity of BFS = O(V+E) where V is vertices and E is edges.

Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

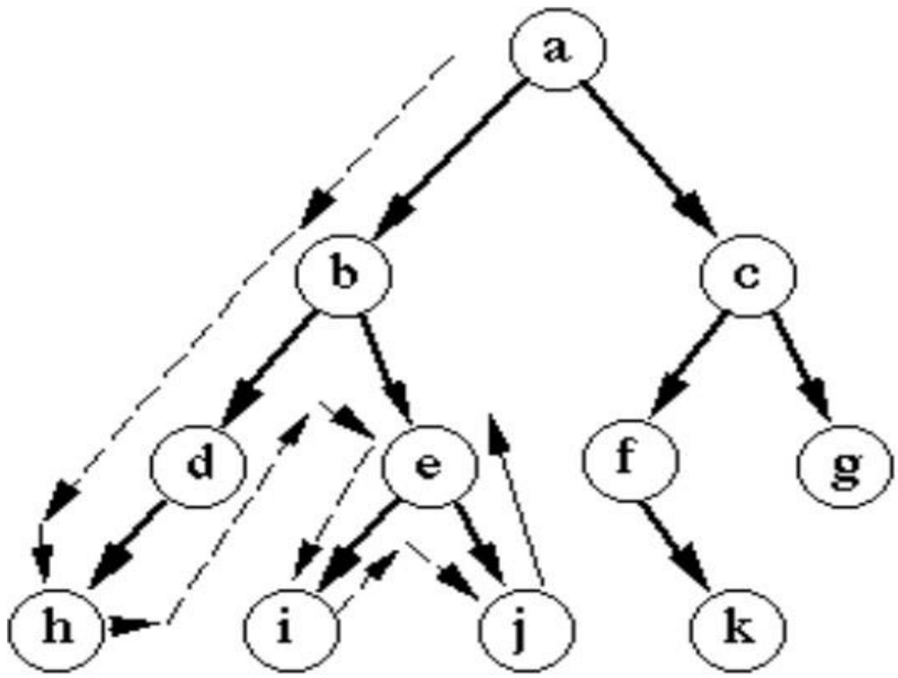
State -Space Tree: -

BFS: -



Breadth-first search

DFS:-



Depth-first search

Code: -

BFS: -

```
#include<bits/stdc++.h>
using namespace std;

void bfs(int n, int m, int num)
{
    queue<int> q;

    q.push(num);

    while (!q.empty())
    {

        int stepNum = q.front();
        q.pop();

        if (stepNum <= m && stepNum >= n)
            cout << stepNum << " ";

        if (num == 0 || stepNum > m)
            continue;

        int lastDigit = stepNum % 10;

        int stepNumA = stepNum * 10 + (lastDigit- 1);
        int stepNumB = stepNum * 10 + (lastDigit + 1);

        if (lastDigit == 0)
            q.push(stepNumB);

        else if (lastDigit == 9)
            q.push(stepNumA);
```

```
        else
        {
            q.push(stepNumA);
            q.push(stepNumB);
        }
    }
}

void displaySteppingNumbers(int n, int m)
{

    for (int i = 0 ; i <= 9 ; i++)
        bfs(n, m, i);
}

int main()
{
    int n = 0, m = 21;

    displaySteppingNumbers(n,m);

    return 0;
}
```

```
For the range 0 to 21:- 0 1 10 12 2 21 3 4 5 6 7 8 9
```

DFS: -

```cpp
#include<bits/stdc++.h>
using namespace std;

void dfs(int n, int m, int stepNum)
{

    if (stepNum <= m && stepNum >= n)
        cout << stepNum << " ";

    if (stepNum == 0 || stepNum > m)
        return ;

    int lastDigit = stepNum % 10;

    int stepNumA = stepNum*10 + (lastDigit-1);
    int stepNumB = stepNum*10 + (lastDigit+1);


    if (lastDigit == 0)
        dfs(n, m, stepNumB);


    else if(lastDigit == 9)
        dfs(n, m, stepNumA);
    else
    {
        dfs(n, m, stepNumA);
        dfs(n, m, stepNumB);
    }
}
```

```cpp
}


void displaySteppingNumbers(int n, int m)
{

    for (int i = 0 ; i <= 9 ; i++)
        dfs(n, m, i);
}

int main()
{
    int n = 0, m = 21;

    displaySteppingNumbers(n,m);
    return 0;
}
```

```
For the range 0 to 21:- 0 1 10 12 2 21 3 4 5 6 7 8 9
```

Result: -

The Stepping Numbers problem is successfully implemented and tested in C++.

Date:11/03/2022

Experiment #5

Developing Best first search and A* algorithm for real world problem

Aim:

To develop and implement the A* algorithm and the best first search [BFS] for real world problems.

A* Algorithm-

To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances.
We can consider a 2D Grid having several obstacles and we start from a source cell (colored red below) to reach towards a goal cell (colored green below)

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

i) if successor is the goal, stop search

ii) else, compute both **g** and **h** for successor
successor.**g** = q.**g** + distance between successor and q
successor.**h** = distance from goal to successor (This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)

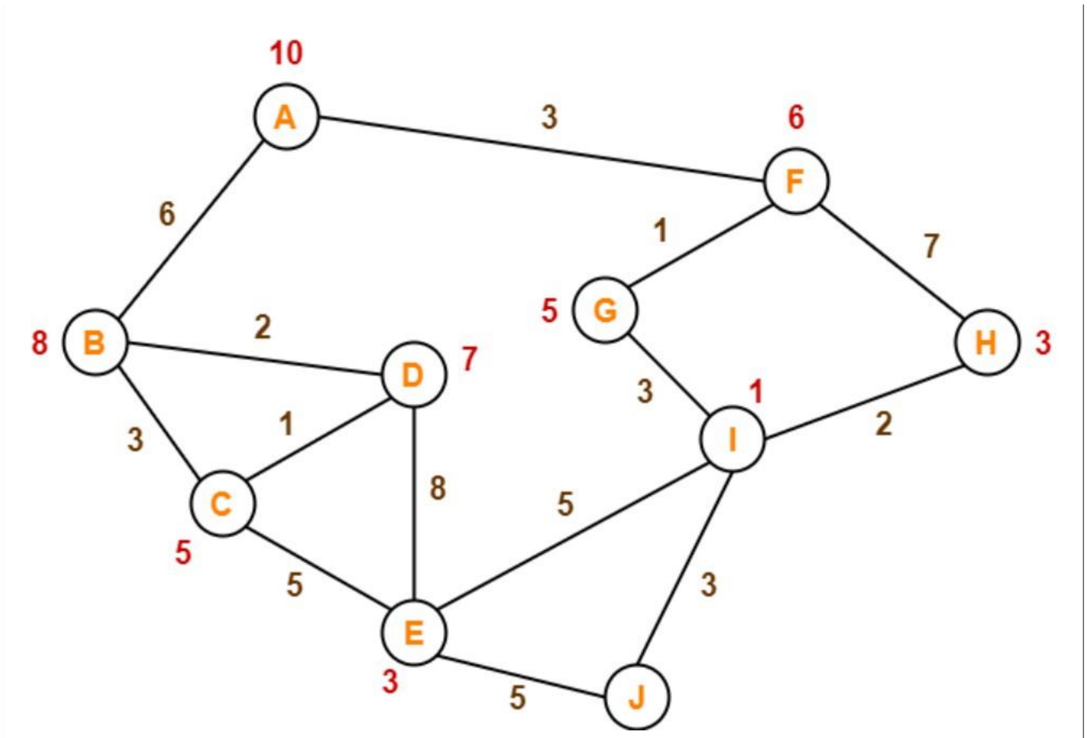
successor.**f** = successor.**g** + successor.**h**

iii) if a node with the same position as successor is in the OPEN list which has a lower **f** than successor, skip this successor

iV) 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)

e) push q on the closed list
end (while loop)

STATE SPACE DIAGRAM:-



Code:

```cpp
// A C++ Program to implement A* Search Algorithm
#include <bits/stdc++.h>
using namespace std;

#define ROW 9
#define COL 10

// Creating a shortcut for int, int pair type
typedef pair<int, int> Pair;

// Creating a shortcut for pair<int, pair<int, int>> type
typedef pair<double, pair<int, int> > pPair;

// A structure to hold the necessary parameters
struct cell {
    // Row and Column index of its parent
    // Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
    int parent_i, parent_j;
    // f = g + h
    double f, g, h;
};

// A Utility Function to check whether given cell (row, col)
// is a valid cell or not.
bool isValid(int row, int col)
{
    // Returns true if row number and column number
    // is in range
    return (row >= 0) && (row < ROW) && (col >= 0)
        && (col < COL);
}

// A Utility Function to check whether the given cell is
// blocked or not
bool isUnBlocked(int grid[][COL], int row, int col)
```

```
{
    // Returns true if the cell is not blocked else false
    if (grid[row][col] == 1)
        return (true);
    else
        return (false);
}

// A Utility Function to check whether destination cell has
// been reached or not
bool isDestination(int row, int col, Pair dest)
{
    if (row == dest.first && col == dest.second)
        return (true);
    else
        return (false);
}

// A Utility Function to calculate the 'h' heuristics.
double calculateHValue(int row, int col, Pair dest)
{
    // Return using the distance formula
    return ((double)sqrt(
        (row - dest.first) * (row - dest.first)
        + (col - dest.second) * (col - dest.second)));
}

// A Utility Function to trace the path from the source
// to destination
void tracePath(cell cellDetails[][COL], Pair dest)
{
    printf("\nThe Path is ");
    int row = dest.first;
    int col = dest.second;

    stack<Pair> Path;

    while (!(cellDetails[row][col].parent_i == row
             && cellDetails[row][col].parent_j == col)) {
        Path.push(make_pair(row, col));
        int temp_row = cellDetails[row][col].parent_i;
        int temp_col = cellDetails[row][col].parent_j;
        row = temp_row;
        col = temp_col;
    }

    Path.push(make_pair(row, col));
    while (!Path.empty()) {
        pair<int, int> p = Path.top();
        Path.pop();
        printf("-> (%d,%d) ", p.first, p.second);
    }

    return;
}

// A Function to find the shortest path between
// a given source cell to a destination cell according
// to A* Search Algorithm
void aStarSearch(int grid[][COL], Pair src, Pair dest)
{
    // If the source is out of range
    if (isValid(src.first, src.second) == false) {
        printf("Source is invalid\n");
        return;
    }

    // If the destination is out of range
    if (isValid(dest.first, dest.second) == false) {
        printf("Destination is invalid\n");
```

```
        return;
    }

    // Either the source or the destination is blocked
    if (isUnBlocked(grid, src.first, src.second) == false
        || isUnBlocked(grid, dest.first, dest.second)
            == false) {
        printf("Source or the destination is blocked\n");
        return;
    }

    // If the destination cell is the same as source cell
    if (isDestination(src.first, src.second, dest)
        == true) {
        printf("We are already at the destination\n");
        return;
    }

    // Create a closed list and initialise it to false which
    // means that no cell has been included yet This closed
    // list is implemented as a boolean 2D array
    bool closedList[ROW][COL];
    memset(closedList, false, sizeof(closedList));

    // Declare a 2D array of structure to hold the details
    // of that cell
    cell cellDetails[ROW][COL];

    int i, j;

    for (i = 0; i < ROW; i++) {
        for (j = 0; j < COL; j++) {
            cellDetails[i][j].f = FLT_MAX;
            cellDetails[i][j].g = FLT_MAX;
            cellDetails[i][j].h = FLT_MAX;
            cellDetails[i][j].parent_i = -1;
            cellDetails[i][j].parent_j = -1;
        }
    }

    // Initialising the parameters of the starting node
    i = src.first, j = src.second;
    cellDetails[i][j].f = 0.0;
    cellDetails[i][j].g = 0.0;
    cellDetails[i][j].h = 0.0;
    cellDetails[i][j].parent_i = i;
    cellDetails[i][j].parent_j = j;

    /*
    Create an open list having information as-
    <f, <i, j>>
    where f = g + h,
    and i, j are the row and column index of that cell
    Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
    This open list is implemented as a set of pair of
    pair.*/
    set<pPair> openList;

    // Put the starting cell on the open list and set its
    // 'f' as 0
    openList.insert(make_pair(0.0, make_pair(i, j)));

    // We set this boolean value as false as initially
    // the destination is not reached.
    bool foundDest = false;

    while (!openList.empty()) {
        pPair p = *openList.begin();

        // Remove this vertex from the open list
```

```
        openList.erase(openList.begin());

        // Add this vertex to the closed list
        i = p.second.first;
        j = p.second.second;
        closedList[i][j] = true;
        double gNew, hNew, fNew;

        //----------- 1st Successor (North) ------------

        // Only process this cell if this is a valid one
        if (isValid(i - 1, j) == true) {
            // If the destination cell is the same as the
            // current successor
            if (isDestination(i - 1, j, dest) == true) {
                // Set the Parent of the destination cell
                cellDetails[i - 1][j].parent_i = i;
                cellDetails[i - 1][j].parent_j = j;
                printf("The destination cell is found\n");
                tracePath(cellDetails, dest);
                foundDest = true;
                return;
            }
            // If the successor is already on the closed
            // list or if it is blocked, then ignore it.
            // Else do the following
            else if (closedList[i - 1][j] == false
                     && isUnBlocked(grid, i - 1, j)
                            == true) {
                gNew = cellDetails[i][j].g + 1.0;
                hNew = calculateHValue(i - 1, j, dest);
                fNew = gNew + hNew;
                if (cellDetails[i - 1][j].f == FLT_MAX
                    || cellDetails[i - 1][j].f > fNew) {
                    openList.insert(make_pair(
                        fNew, make_pair(i - 1, j)));

                    // Update the details of this cell
                    cellDetails[i - 1][j].f = fNew;
                    cellDetails[i - 1][j].g = gNew;
                    cellDetails[i - 1][j].h = hNew;
                    cellDetails[i - 1][j].parent_i = i;
                    cellDetails[i - 1][j].parent_j = j;
                }
            }
        }

        //----------- 2nd Successor (South) ------------

        // Only process this cell if this is a valid one
        if (isValid(i + 1, j) == true) {
            // If the destination cell is the same as the
            // current successor
            if (isDestination(i + 1, j, dest) == true) {
                // Set the Parent of the destination cell
                cellDetails[i + 1][j].parent_i = i;
                cellDetails[i + 1][j].parent_j = j;
                printf("The destination cell is found\n");
                tracePath(cellDetails, dest);
                foundDest = true;
                return;
            }
            // If the successor is already on the closed
            // list or if it is blocked, then ignore it.
            // Else do the following
            else if (closedList[i + 1][j] == false
                     && isUnBlocked(grid, i + 1, j)
                            == true) {
                gNew = cellDetails[i][j].g + 1.0;
                hNew = calculateHValue(i + 1, j, dest);
```

```
                fNew = gNew + hNew;
                if (cellDetails[i + 1][j].f == FLT_MAX
                    || cellDetails[i + 1][j].f > fNew) {
                    openList.insert(make_pair(
                        fNew, make_pair(i + 1, j)));
                    // Update the details of this cell
                    cellDetails[i + 1][j].f = fNew;
                    cellDetails[i + 1][j].g = gNew;
                    cellDetails[i + 1][j].h = hNew;
                    cellDetails[i + 1][j].parent_i = i;
                    cellDetails[i + 1][j].parent_j = j;
                }
            }
        }

        //----------- 3rd Successor (East) ------------

        // Only process this cell if this is a valid one
        if (isValid(i, j + 1) == true) {
            // If the destination cell is the same as the
            // current successor
            if (isDestination(i, j + 1, dest) == true) {
                // Set the Parent of the destination cell
                cellDetails[i][j + 1].parent_i = i;
                cellDetails[i][j + 1].parent_j = j;
                printf("The destination cell is found\n");
                tracePath(cellDetails, dest);
                foundDest = true;
                return;
            }

            // If the successor is already on the closed
            // list or if it is blocked, then ignore it.
            // Else do the following
            else if (closedList[i][j + 1] == false
```

```
                     && isUnBlocked(grid, i, j + 1)
                            == true) {
                    gNew = cellDetails[i][j].g + 1.0;
                    hNew = calculateHValue(i, j + 1, dest);
                    fNew = gNew + hNew;
                    if (cellDetails[i][j + 1].f == FLT_MAX
                        || cellDetails[i][j + 1].f > fNew) {
                        openList.insert(make_pair(
                            fNew, make_pair(i, j + 1)));

                        // Update the details of this cell
                        cellDetails[i][j + 1].f = fNew;
                        cellDetails[i][j + 1].g = gNew;
                        cellDetails[i][j + 1].h = hNew;
                        cellDetails[i][j + 1].parent_i = i;
                        cellDetails[i][j + 1].parent_j = j;
                    }
                }
            }

            //----------- 4th Successor (West) ------------

            // Only process this cell if this is a valid one
            if (isValid(i, j - 1) == true) {
                // If the destination cell is the same as the
                // current successor
                if (isDestination(i, j - 1, dest) == true) {
                    // Set the Parent of the destination cell
                    cellDetails[i][j - 1].parent_i = i;
                    cellDetails[i][j - 1].parent_j = j;
                    printf("The destination cell is found\n");
                    tracePath(cellDetails, dest);
                    foundDest = true;
                    return;
                }
```

```
                     && isUnBlocked(grid, i, j - 1)
                            == true) {
                    gNew = cellDetails[i][j].g + 1.0;
                    hNew = calculateHValue(i, j - 1, dest);
                    fNew = gNew + hNew;
                    if (cellDetails[i][j - 1].f == FLT_MAX
                        || cellDetails[i][j - 1].f > fNew) {
                        openList.insert(make_pair(
                            fNew, make_pair(i, j - 1)));

                        // Update the details of this cell
                        cellDetails[i][j - 1].f = fNew;
                        cellDetails[i][j - 1].g = gNew;
                        cellDetails[i][j - 1].h = hNew;
                        cellDetails[i][j - 1].parent_i = i;
                        cellDetails[i][j - 1].parent_j = j;
                    }
                }
            }

            //----------- 5th Successor (North-East)
            //------------

            // Only process this cell if this is a valid one
            if (isValid(i - 1, j + 1) == true) {
                // If the destination cell is the same as the
                // current successor
                if (isDestination(i - 1, j + 1, dest) == true) {
                    // Set the Parent of the destination cell
                    cellDetails[i - 1][j + 1].parent_i = i;
                    cellDetails[i - 1][j + 1].parent_j = j;
                    printf("The destination cell is found\n");
                    tracePath(cellDetails, dest);
                    foundDest = true;
                    return;
```

```
    else if (closedList[i - 1][j - 1] == false
            && isUnBlocked(grid, i - 1, j - 1)
                == true) {
        gNew = cellDetails[i][j].g + 1.414;
        hNew = calculateHValue(i - 1, j - 1, dest);
        fNew = gNew + hNew;
        if (cellDetails[i - 1][j - 1].f == FLT_MAX
            || cellDetails[i - 1][j - 1].f > fNew) {
            openList.insert(make_pair(
                fNew, make_pair(i - 1, j - 1)));
            // Update the details of this cell
            cellDetails[i - 1][j - 1].f = fNew;
            cellDetails[i - 1][j - 1].g = gNew;
            cellDetails[i - 1][j - 1].h = hNew;
            cellDetails[i - 1][j - 1].parent_i = i;
            cellDetails[i - 1][j - 1].parent_j = j;
        }
    }
}

//----------- 7th Successor (South-East)
//------------

// Only process this cell if this is a valid one
if (isValid(i + 1, j + 1) == true) {
    // If the destination cell is the same as the
    // current successor
    if (isDestination(i + 1, j + 1, dest) == true) {
        // Set the Parent of the destination cell
        cellDetails[i + 1][j + 1].parent_i = i;
        cellDetails[i + 1][j + 1].parent_j = j;
        printf("The destination cell is found\n");
        tracePath(cellDetails, dest);
        foundDest = true;
        return;
```

```
    else if (closedList[i + 1][j + 1] == false
            && isUnBlocked(grid, i + 1, j + 1)
                == true) {
        gNew = cellDetails[i][j].g + 1.414;
        hNew = calculateHValue(i + 1, j + 1, dest);
        fNew = gNew + hNew;
        if (cellDetails[i + 1][j + 1].f == FLT_MAX
            || cellDetails[i + 1][j + 1].f > fNew) {
            openList.insert(make_pair(
                fNew, make_pair(i + 1, j + 1)));

            // Update the details of this cell
            cellDetails[i + 1][j + 1].f = fNew;
            cellDetails[i + 1][j + 1].g = gNew;
            cellDetails[i + 1][j + 1].h = hNew;
            cellDetails[i + 1][j + 1].parent_i = i;
            cellDetails[i + 1][j + 1].parent_j = j;
        }
    }
}

//----------- 8th Successor (South-West)
//------------

// Only process this cell if this is a valid one
if (isValid(i + 1, j - 1) == true) {
    // If the destination cell is the same as the
    // current successor
    if (isDestination(i + 1, j - 1, dest) == true) {
        // Set the Parent of the destination cell
        cellDetails[i + 1][j - 1].parent_i = i;
        cellDetails[i + 1][j - 1].parent_j = j;
        printf("The destination cell is found\n");
        tracePath(cellDetails, dest);
        foundDest = true;
```

```
                cellDetails[i + 1][j - 1].parent_j = j;
                printf("The destination cell is found\n");
                tracePath(cellDetails, dest);
                foundDest = true;
                return;
            }

            // If the successor is already on the closed
            // list or if it is blocked, then ignore it.
            // Else do the following
            else if (closedList[i + 1][j - 1] == false
                     && isUnBlocked(grid, i + 1, j - 1)
                            == true) {
                gNew = cellDetails[i][j].g + 1.414;
                hNew = calculateHValue(i + 1, j - 1, dest);
                fNew = gNew + hNew;
                if (cellDetails[i + 1][j - 1].f == FLT_MAX
                    || cellDetails[i + 1][j - 1].f > fNew) {
                    openList.insert(make_pair(
                        fNew, make_pair(i + 1, j - 1)));

                    // Update the details of this cell
                    cellDetails[i + 1][j - 1].f = fNew;
                    cellDetails[i + 1][j - 1].g = gNew;
                    cellDetails[i + 1][j - 1].h = hNew;
                    cellDetails[i + 1][j - 1].parent_i = i;
                    cellDetails[i + 1][j - 1].parent_j = j;
                }
            }
        }
    }
    if (foundDest == false)
        printf("Failed to find the Destination Cell\n");

    return;
}
```

```
            else if (closedList[i - 1][j + 1] == false
                     && isUnBlocked(grid, i - 1, j + 1)
                            == true) {
                gNew = cellDetails[i][j].g + 1.414;
                hNew = calculateHValue(i - 1, j + 1, dest);
                fNew = gNew + hNew;

                if (cellDetails[i - 1][j + 1].f == FLT_MAX
                    || cellDetails[i - 1][j + 1].f > fNew) {
                    openList.insert(make_pair(
                        fNew, make_pair(i - 1, j + 1)));

                    // Update the details of this cell
                    cellDetails[i - 1][j + 1].f = fNew;
                    cellDetails[i - 1][j + 1].g = gNew;
                    cellDetails[i - 1][j + 1].h = hNew;
                    cellDetails[i - 1][j + 1].parent_i = i;
                    cellDetails[i - 1][j + 1].parent_j = j;
                }
            }
        }

        //----------- 6th Successor (North-West)
        //------------

        // Only process this cell if this is a valid one
        if (isValid(i - 1, j - 1) == true) {
            // If the destination cell is the same as the
            // current successor
            if (isDestination(i - 1, j - 1, dest) == true) {
                // Set the Parent of the destination cell
                cellDetails[i - 1][j - 1].parent_i = i;
                cellDetails[i - 1][j - 1].parent_j = j;
                printf("The destination cell is found\n");
                tracePath(cellDetails, dest);
```

```
    }
    if (foundDest == false)
        printf("Failed to find the Destination Cell\n");

    return;
}

// Driver program to test above function
int main()
{
    /* Description of the Grid-
    1--> The cell is not blocked
    0--> The cell is blocked */
    int grid[ROW][COL]
        = { { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },
            { 1, 1, 1, 0, 1, 1, 1, 0, 1, 1 },
            { 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 },
            { 0, 0, 1, 0, 1, 0, 0, 0, 0, 1 },
            { 1, 1, 1, 0, 1, 1, 1, 0, 1, 0 },
            { 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 },
            { 1, 0, 0, 0, 0, 1, 0, 0, 0, 1 },
            { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },
            { 1, 1, 1, 0, 0, 0, 1, 0, 0, 1 } };

    // Source is the left-most bottom-most corner
    Pair src = make_pair(8, 0);

    // Destination is the left-most top-most corner
    Pair dest = make_pair(0, 0);

    aStarSearch(grid, src, dest);

    return (0);
}
```

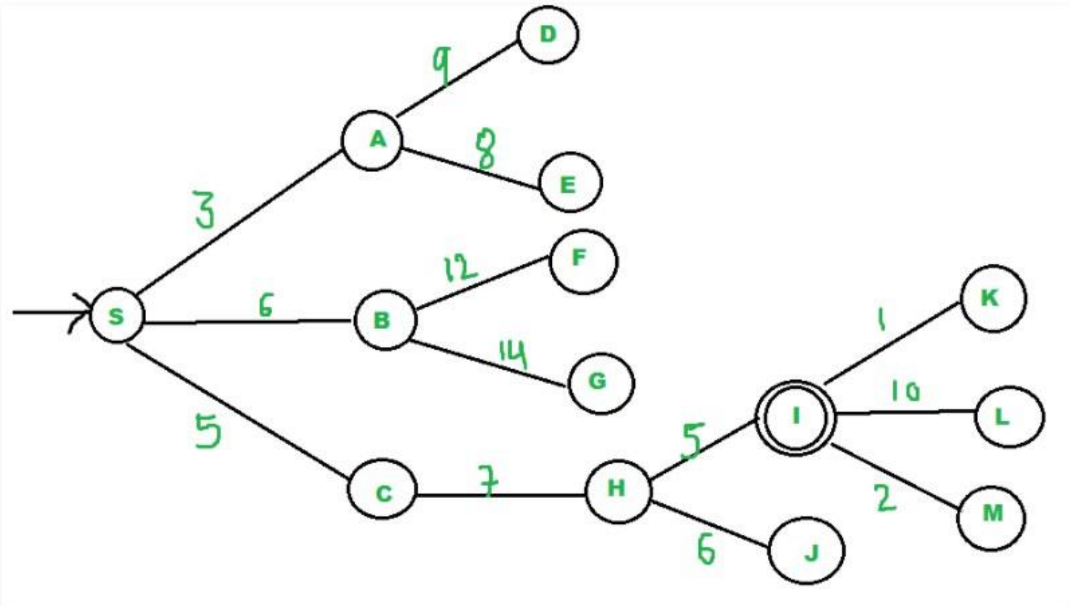
Output:

```
The destination cell is found

The Path is -> (8,0) -> (7,0) -> (6,0) -> (5,0) -> (4,1) -> (3,2) -> (2,1) -> (1,0) -> (0,0)

...Program finished with exit code 0
Press ENTER to exit console.
```

Best First Search-

To approximate the shortest possible road distance from city S to city I by visiting the least number of cities in the way, i.e. the least number of steps.

STATE SPACE:



Algorithm:

Create 2 empty lists: OPEN and CLOSED.

Start from the initial node (say N) and put it in the ‘ordered’ OPEN list.

Repeat the next steps until GOAL node is reached

If OPEN list is empty, then EXIT the loop returning ‘False’.

Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also capture the information of the parent node.

If N is a GOAL node, then move the node to the Closed list and exit the loop returning ‘True’. The solution can be found by backtracking the path.

If N is not the GOAL node, expand node N to generate the ‘immediate’ next nodes linked to node N and add all those to the OPEN list.

Reorder the nodes in the OPEN list in ascending order according to an evaluation function.

This algorithm will traverse the shortest path first in the queue. The **time complexity** of the algorithm is given by **O(n*logn)** .

Code:

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

vector<vector<pi> > graph;

// Function for adding edges to 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));
}

// Function For Implementing Best First Search
// Gives output path having lowest cost
void best_first_search(int source, int target, int n)
{
    vector<bool> visited(n, false);
    // MIN HEAP priority queue
    priority_queue<pi, vector<pi>, greater<pi> > pq;
    // sorting in pq gets done by first value of pair
    pq.push(make_pair(0, source));
    int s = source;
    visited[s] = true;
    while (!pq.empty()) {
        int x = pq.top().second;
```

```
        // Displaying the path having lowest cost
        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));
            }
        }
    }
}

// Driver code to test above methods
int main()
{
    // No. of Nodes
    int v = 14;
    graph.resize(v);

    // The nodes shown in above example(by alphabets) are
    // implemented using integers addedge(x,y,cost);
    addedge(0, 1, 3);
    addedge(0, 2, 6);
```

```
    // The nodes shown in above example(by alphabets) are
    // implemented using integers addedge(x,y,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);

    int source = 0;
    int target = 9;

    // Function call
    best_first_search(source, target, v);

    return 0;
}
```

OUTPUT:-

input

```
0 1 3 2 8 9

...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

Both Best First Search and A* algorithm for real world problems were successfully developed and implemented using C++.

Date:28/03/2022

Experiment-6

Developing Mini-max algorithm on a real-world problem

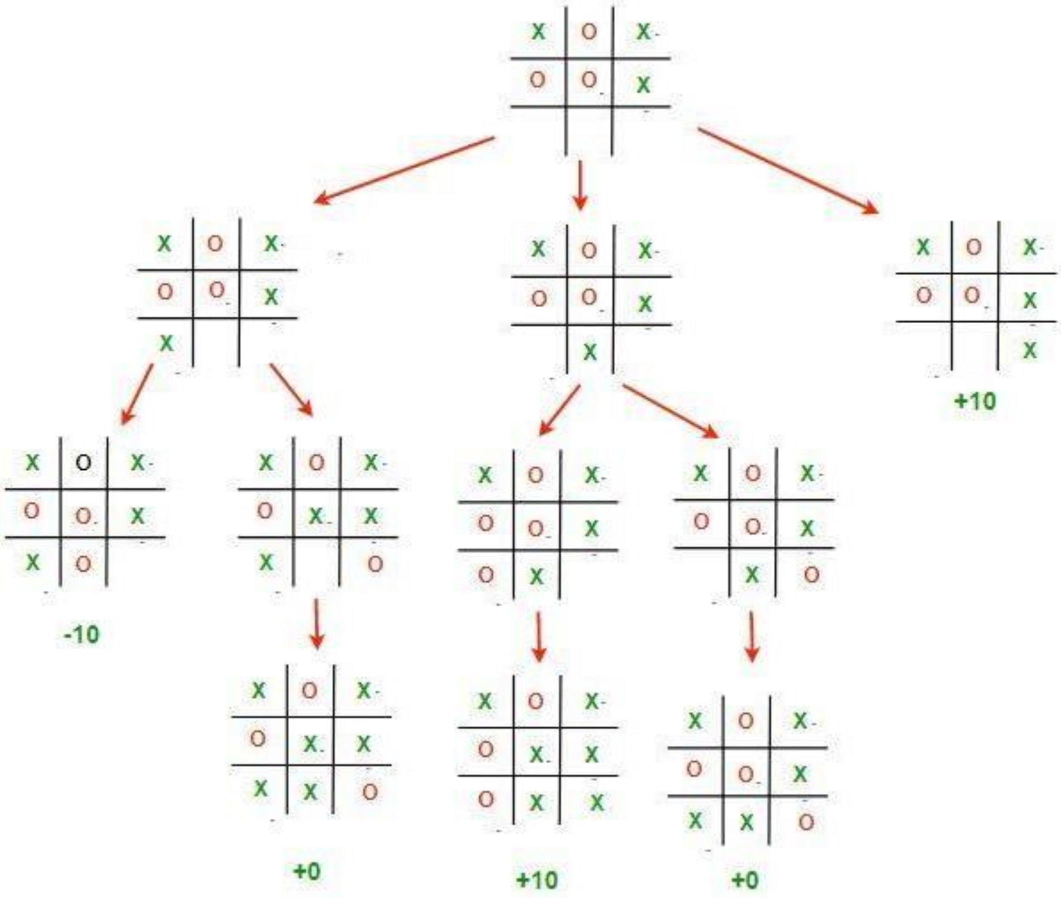
Aim:

To develop and implement the Min-max algorithm on tic-tac toe problem.

Algorithm:

- **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.
- **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.
- **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.
- **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.

Graph:



Code:

```python
# Python3 program to find the next optimal move for a player
player, opponent = 'x', 'o'

# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board) :

    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                return True
    return False

def evaluate(b) :

    # Checking for Rows for X or O victory.
    for row in range(3) :
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
            if (b[row][0] == player) :
                return 10
            elif(b[row][0] == opponent) :
                return -10

    # Checking for Columns for X or O victory.
    for col in range(3) :

        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

            if (b[0][col] == player) :
                return 10
            elif(b[0][col] == opponent) :
                return -10

    # Checking for Diagonals for X or O victory.
    if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

        if (b[0][0] == player) :
            return 10
        elif(b[0][0] == opponent) :
            return -10

    if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

        if (b[0][2] == player) :
            return 10
        elif(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
def minimax(board, depth, isMax) :
    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
```

```python
    # 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) :
        best = -1000

        # Traverse all cells
        for i in range(3) :
            for j in range(3) :

                # 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,
                                              not isMax) )

                    # Undo the move
                    board[i][j] = '_'
        return best

    # If this minimizer's move
    else :
        best = 1000

        # Traverse all cells
        for i in range(3) :
            for j in range(3) :

                # 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, not isMax))

                    # Undo the move
                    board[i][j] = '_'
        return best

# This will return the best possible move for the player
def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)

    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
    # value.
```

```python
    for i in range(3) :
        for j in range(3) :

            # Check if cell is empty
            if (board[i][j] == '_') :

                # Make the move
                board[i][j] = player

                # compute evaluation function for this
                # move.
                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 = (i, j)
                    bestVal = moveVal

    print("The value of the best Move is :", bestVal)
    print()
    return bestMove
# Driver code
board = [
    [ 'x', 'o', 'x' ],
    [ 'o', 'o', 'x' ],
    [ '_', '_', '_' ]
]

bestMove = findBestMove(board)

print("The Optimal Move is :")
print("ROW:", bestMove[0]+1, " COL:", bestMove[1]+1)
```

Output:

```
The value of the best Move is : 10

The Optimal Move is :
ROW: 3  COL: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

The Mini-max algorithm for real world problem was successfully developed and implemented using Python 3.

Date:04/04/2022

Experiment- 7

Implementation of Unification Algorithm

Aim:

To develop and implement the Unification Algorithm.

Algorithm:

Step. 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

a) If Ψ_1 or Ψ_2 are identical, then return NIL.

b) Else if $\Psi_1$is a variable,

a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

b. Else return { (Ψ_2/ Ψ_1)}.

c) Else if Ψ_2 is a variable,

a. If Ψ_2 occurs in Ψ_1 then return FAILURE,

b. Else return {(Ψ_1/ Ψ_2)}.

d) Else return FAILURE.

Step. 2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step. 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE. Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST). Step.6: Return SUBST.

Code:

```
def get_index_comma(string):
    """
    Return index of commas in string
    """

    index_list = list()
    # Count open parentheses
    par_count = 0

    for i in range(len(string)):
        if string[i] == ',' and par_count == 0:
            index_list.append(i)
        elif string[i] == '(':
            par_count += 1
        elif string[i] == ')':
            par_count -= 1

    return index_list


def is_variable(expr):
    """
    Check if expression is variable
    """

    for i in expr:
        if i == '(':
            return False

    return True


def process_expression(expr):
    """
    input:  - expression:
            'Q(a, g(x, b), f(y))'
    return: - predicate symbol:
            Q
            - list of arguments
            ['a', 'g(x, b)', 'f(y)']
    """

    # Remove space in expression
    expr = expr.replace(' ', '')
```

```python
    # Find the first index == '('
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break

    # Return predicate symbol and remove predicate symbol in expression
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')

    # Remove '(' in the first index and ')' in the last index
    expr = expr[1:len(expr) - 1]

    # List of arguments
    arg_list = list()

    # Split string with commas, return list of arguments
    indices = get_index_comma(expr)

    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])

    return predicate_symbol, arg_list


def get_arg_list(expr):
    """
    input:  expression:
            'Q(a, g(x, b), f(y))'
    return: full list of arguments:
            ['a', 'x', 'b', 'y']
    """

    _, arg_list = process_expression(expr)

    flag = True
    while flag:
        flag = False

        for i in arg_list:
            if not is_variable(i):
```

```
            flag = True
            _, tmp = process_expression(i)
            for j in tmp:
                if j not in arg_list:
                    arg_list.append(j)
            arg_list.remove(i)

    return arg_list


def check_occurs(var, expr):
    """
    Check if var occurs in expr
    """

    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True

    return False


def unify(expr1, expr2):
    """
    Unification Algorithm

    Step 1: If Ψ1 or Ψ2 is a variable or constant, then:
              a, If Ψ1 or Ψ2 are identical, then return NULL.
              b, Else if Ψ1 is a variable:
                  - then if Ψ1 occurs in Ψ2, then return False
                  - Else return (Ψ2 / Ψ1)
              c, Else if Ψ2 is a variable:
                  - then if Ψ2 occurs in Ψ1, then return False
                  - Else return (Ψ1 / Ψ2)
              d, Else return False

    Step 2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return False.

    Step 3: IF Ψ1 and Ψ2 have a different number of arguments, then return False.

    Step 4: Create Substitution list.

    Step 5: For i=1 to the number of elements in Ψ1.
              a, Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result into S.
              b, If S = False then returns False
              c, If S ≠ Null then append to Substitution list
```

```
Step 6: Return Substitution list.
"""

# Step 1:
if is_variable(expr1) and is_variable(expr2):
    if expr1 == expr2:
        return 'Null'
    else:
        return False
elif is_variable(expr1) and not is_variable(expr2):
    if check_occurs(expr1, expr2):
        return False
    else:
        tmp = str(expr2) + '/' + str(expr1)
        return tmp
elif not is_variable(expr1) and is_variable(expr2):
    if check_occurs(expr2, expr1):
        return False
    else:
        tmp = str(expr1) + '/' + str(expr2)
        return tmp
else:
    predicate_symbol_1, arg_list_1 = process_expression(expr1)
    predicate_symbol_2, arg_list_2 = process_expression(expr2)

    # Step 2
    if predicate_symbol_1 != predicate_symbol_2:
        return False
    # Step 3
    elif len(arg_list_1) != len(arg_list_2):
        return False
    else:
        # Step 4: Create substitution list
        sub_list = list()

        # Step 5:
        for i in range(len(arg_list_1)):
            tmp = unify(arg_list_1[i], arg_list_2[i])

            if not tmp:
                return False
            elif tmp == 'Null':
                pass
            else:
                if type(tmp) == list:
                    for j in tmp:
                        sub_list.append(j)
```

```
                    else:
                        sub_list.append(tmp)

            # Step 6
            return sub_list


if __name__ == '__main__':
    # Data 1
    f1 = 'p(b(A), X, f(g(Z)))'
    f2 = 'p(Z, f(Y), f(Y))'

    # Data 2
    # f1 = 'Q(a, g(x, a), f(y))'
    # f2 = 'Q(a, g(f(b), a), x)'

    # Data 3
    # f1 = 'Q(a, g(x, a, d), f(y))'
    # f2 = 'Q(a, g(f(b), a), x)'

    result = unify(f1, f2)
    if not result:
        print('Unification failed!')
    else:
        print('Unification successfully!')
        print(result)
```

Output:

```
Unification successfully!
['b(A)/Z', 'f(Y)/X', 'g(Z)/Y']
```

Result:

The Unification Algorithm has been successfully developed and implemented using Python 3.

Date:04/04/2022

Experiment-8

Implementation of Monty Hall Problem

Aim:

To develop and implement the Monty Hall Problem.

Monty Hall Problem:

Suppose that you're on a game show and you have three closed doors in front of you, where behind two of the doors there are goats, and behind the last one there's a car.

You guess one of the doors, and the game show hosts opens one of the doors you didn't pick, say the third door, revealing a goat and leaving your guess and the second door closed. The host then gives you the option to keep the door you have or switch to the second door. What do you do?

Your gut feeling might say that It's now a 50/50 shoot of getting the car, while in fact if you switch the odds are 2/3 in your favour. Let's take this step by step:

Before any guesses have been made, there is a 1/3 probability for each door, which adds up to 1, obviously. After one door is removed, the probability of the first door you picked has not changed, but you do gain more information about the other closed door. The probability must add up to 1 (if you had the option to open both of them), and therefore $1/3 + x = 1$, gives you a 2/3 probability if you switch. The reason why you change the probability if you switch (and not stay) is because you gain information about the closed door (the switch) when the host reveals one false door. Opening one door updates your information about the switch, while you haven't learned anything new about your original pick because it was never an option to open your original guess.

$$\overset{\text{Posterior}}{P(A|B)} = \frac{\overset{\text{Likelihood}}{P(B|A)}\overset{\text{Prior}}{P(A)}}{\underset{\text{Normalizing constant}}{P(B)}}$$

$$P(B) = \sum_{Y} P(B|A)P(A)$$

We have three doors, so the odds of the car being behind one of them is P(CarA), P(CarB), P(CarC), are all equal to 1/3. Now let’s shift our focus a little bit, and look at the probability that the host opens a certain door.

Let’s say that we pick door A, and door B is opened with a goat behind it. So, we look at the probability of the car being behind door A, given that the host opens door B.

P(CarA|B) = P(B|CarA)*P(CarA) /P(B)

We need to find what is the probability that the host opens door B when we picked door A.

P(B) = P(B|CarA)*P(CarA)+P(B|CarB)*P(CarB)+P(B|CarC)*P(CarC).

So, let’s break up this equation into three separate equations, and remember, we picked door A:

- P(B|CarA): the odds of opening door B if the car is behind door A, is a ½, since the host can in theory pick door B or C
- P(B|CarB): the odds of the host opening door B if the car is in B is zero, since he will never pick the door with the car to open.
- P(B|CarC): Finally, the odds of the host opening door B, given that we picked door A and the car is in door C has to be 1. It’s the only option.

This gives us:

$$P(CarA|B) = \frac{\frac{1}{2} * \frac{1}{3}}{\frac{1}{2} * \frac{1}{3} + 0 * \frac{1}{3} + 1 * \frac{1}{3}} = \frac{1}{3}$$

Similarly, if we do the same exercise where we pick A and the car is in C – P(CarC|B) we will get the probability of 2/3.

To prove this, we’ve added a code to test out the math for 10000 simulations.

Code:

```python
import numpy as np

# N Samples
N = 10000

#Define an array of the different doors with the car at random
cars = np.random.randint(0,high=3,size=N)+1
#define an array of the different doors, and picks at random
picks = np.random.randint(0,high=3,size=N)+1

#Counters for win if stay and switch
count_stay =0
count_switch =0


for i in range(N):
    #define array of 3 doors
    doors_round1 = [1,2,3]

    #First we have to remove both the car and the pick
    doors_round1.remove(picks[i])
    if cars[i] != picks[i]:
        doors_round1.remove(cars[i])

    #Will open one door at random.
    #If Cars and Picks are the same door, it can only choose one.
    open_door = doors_round1[np.random.randint(len(doors_round1))]

    doors_round2 = [1,2,3]
    doors_round2.remove(open_door)

    #Switch picks
    doors_round2.remove(picks[i])
    pick2 = doors_round2[0]


    if  cars[i] == picks[i]:
        count_stay = count_stay + 1

    if  cars[i] == pick2:
        count_switch = count_switch + 1

print("\nStay: %d"%(100*count_stay/N))
print("Switch: %d"%(100*count_switch/N))
```

Output:

```
Stay: 33
Switch: 66
```

Result:

The Monty Hall problem has been successfully developed and implemented using Bayes Theorem in Python 3.