Practical -1

1) C++ program to print DFS traversal from

```
// a given vertex in a  given graph
#include <bits/stdc++.h>

Using namespace std;

// Graph class represents a directed graph
// using adjacency list representation

Class Graph {

Public:

    Map<int, bool> visited;

    Map<int, list<int> > adj;



    // Function to add an edge to graph

    Void addEdge(int v, int w);



    // DFS traversal of the vertices

    // reachable from v
```

```
   Void DFS(int v);
};


Void Graph::addEdge(int v, int w)
{

   // Add w to v's list.

   Adj[v].push_back(w);
}


Void Graph::DFS(int v)
{

   // Mark the current node as visited and

   // print it

   Visited[v] = true;

   Cout << v << " ";



   // Recur for all the vertices adjacent
```

```
    // to this vertex

    List<int>::iterator i;

    For (i = adj[v].begin(); i != adj[v].end(); ++i)

        If (!visited[*i])

            DFS(*i);
}

// Driver code

Int main()
{

    // Create a graph given in the above diagram

    Graph g;

    g.addEdge(0, 1);

    g.addEdge(0, 2);

    g.addEdge(1, 2);
```

```
    g.addEdge(2, 0);

    g.addEdge(2, 3);

    g.addEdge(3, 3);


    cout << “Following is Depth First Traversal”

            “ (starting from vertex 2) \n”;


    // Function call

    g.DFS(2);


    return 0;
}

// improved by Vishnudev C
```

Output

Following is Depth First Traversal (starting from vertex 2)

2 0 1 3

Practical -2

```
// C++ code to print BFS traversal from a given
// source vertex
```

```
#include <bits/stdc++.h>
Using namespace std;
// This class represents a directed graph using
// adjacency list representation
Class Graph {
   // No. Of vertices
   Int V;
   // Pointer to an array containing adjacency lists
   Vector<list<int> > adj;
Public:
   // Constructor
   Graph(int V);
   // Function to add an edge to graph
   Void addEdge(int v, int w);
   // Prints BFS traversal from a given source s
   Void BFS(int s);
};
Graph::Graph(int V)
{
   This->V = V;
   Adj.resize(V);
}
Void Graph::addEdge(int v, int w)
{
   // Add w to v’s list.
   Adj[v].push_back(w);
}
```

```
Void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    Vector<bool> visited;
    Visited.resize(V, false);
    // Create a queue for BFS
    List<int> queue;
    // Mark the current node as visited and enqueue it
    Visited[s] = true;
    Queue.push_back(s);
    While (!queue.empty()) {
        // Dequeue a vertex from queue and print it
        S = queue.front();
        Cout << s << " ";
        Queue.pop_front();
        // Get all adjacent vertices of the dequeued
        // vertex s.
        // If an adjacent has not been visited,
        // then mark it visited and enqueue it
        For (auto adjacent : adj[s]) {
            If (!visited[adjacent]) {
                Visited[adjacent] = true;
                Queue.push_back(adjacent);
            }
        }
    }
}
```

```
// Driver code
Int main()
{
   // Create a graph given in the above diagram
   Graph g(4);
   g.addEdge(0, 1);
   g.addEdge(0, 2);
   g.addEdge(1, 2);
   g.addEdge(2, 0);
   g.addEdge(2, 3);
   g.addEdge(3, 3);
   cout << "Following is Breadth First Traversal "
       << "(starting from vertex 2) \n";
   g.BFS(2);
   return 0;
}
```

Output

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

Practical -3

```
// 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;
    /*
     Generating all the 8 successor of this cell
         N.W   N   N.E
           \   |   /
            \  |  /
         W----Cell----E
              / | \
            /   |  \
         S.W    S   S.E
     Cell→Popped Cell (i, j)
     N →  North       (i-1, j)
     S →  South       (i+1, j)
     E →  East        (i, j+1)
     W →  West           (i, j-1)
     N.E→ North-East  (i-1, j+1)
     N.W→ North-West  (i-1, j-1)
```

```
 S.E→ South-East  (i+1, j+1)
 S.W→ South-West  (i+1, j-1)*/
// To store the 'g', 'h' and 'f' of the 8 successors
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 it isn't on the open list, add it to
        // the open list. Make the current square
        // the parent of this square. Record the
        // f, g, and h costs of the square cell
        //              OR
        // If it is on the open list already, check
        // to see if this path to that square is
        // better, using 'f' cost as the measure.
        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 it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
    //                OR
    // If it is on the open list already, check
    // to see if this path to that square is
    // better, using 'f' cost as the measure.
    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 it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
    //              OR
    // If it is on the open list already, check
    // to see if this path to that square is
    // better, using 'f' cost as the measure.
    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;
    }
    // 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 it isn't on the open list, add it to
        // the open list. Make the current square
        // the parent of this square. Record the
```

```
        // f, g, and h costs of the square cell
        //                OR
        // If it is on the open list already, check
        // to see if this path to that square is
        // better, using 'f' cost as the measure.
        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;
}
// 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 it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
    //                OR
    // If it is on the open list already, check
    // to see if this path to that square is
    // better, using 'f' cost as the measure.
    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);
        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 it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
    //             OR
    // If it is on the open list already, check
    // to see if this path to that square is
    // better, using 'f' cost as the measure.
    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;
   }
   // 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 it isn't on the open list, add it to
      // the open list. Make the current square
      // the parent of this square. Record the
```

```
        // f, g, and h costs of the square cell
        //                OR
        // If it is on the open list already, check
        // to see if this path to that square is
        // better, using 'f' cost as the measure.
        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;
    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 it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
    //              OR
    // If it is on the open list already, check
    // to see if this path to that square is
    // better, using 'f' cost as the measure.
    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;
              }
            }
          }
        }
        // When the destination cell is not found and the open
        // list is empty, then we conclude that we failed to
        // reach the destination cell. This may happen when the
        // there is no way to destination cell (due to
        // blockages)
        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);
}
```

Practical-4

```
#include <algorithm>
#include <iostream>
#include <vector>
// Generates neighbors of x
Std::vector<int> generate_neighbors(int x)
{
    // TODO: implement this function
}
Int hill_climbing(int (*f)(int), int x0)
```

```
{
   Int x = x0; // initial solution
   While (true) {
      Std::vector<int> neighbors = generate_neighbors(
         x); // generate neighbors of x
      int best_neighbor = *std::max_element(
         neighbors.begin(), neighbors.end(),
         [f](int a, int b) {
            Return f(a) < f(b);
         }); // find the neighbor with the highest
            // function value
      If (f(best_neighbor)
         <= f(x)) // if the best neighbor is not better
                // than x, stop
         Return x;
      X = best_neighbor; // otherwise, continue with the
                     // best neighbor
   }
}
Int main()
{
   // Example usage
   Int x0 = 1;
   Int x = hill_climbing([](int x) { return x * x; }, x0);
   Std::cout << "Result: " << x << std::endl;
   Return 0;
}
```