SNJB's Late Sau. K. B. Jain College of Engineering, Chandwad.

Department of Artificial Intelligence and Data Science

<u>Subject: Software Laboratory I (317523)</u>

Experiment No. (Group)

Title:

Date of Completion: ________________ **Date of Submission: ________________**

S.N.	Criteria	Possible Marks	Obtained Marks
1	**Active Participation**	9, 12, 15	
2	**Programming Correctness**	15, 20, 25	
3	**Timely Submission**	9, 12, 15	
4	**Documentation & Presentation**	15, 20, 25	
5	**Output ,Viva**	12, 16, 20	
		Total	

Date:__________________

Prof. K. S. Sagale
(Subject Teacher)

SNJB's K B Jain College of Engineering, Chandwad
Department of AI and DS
Lab: PG Lab (216)
Subject: <u>Software Laboratory I</u> (317523) (TE 2019 Pattern)

Rubrics to Assess Lab report for Practical Head: Practical Experiments

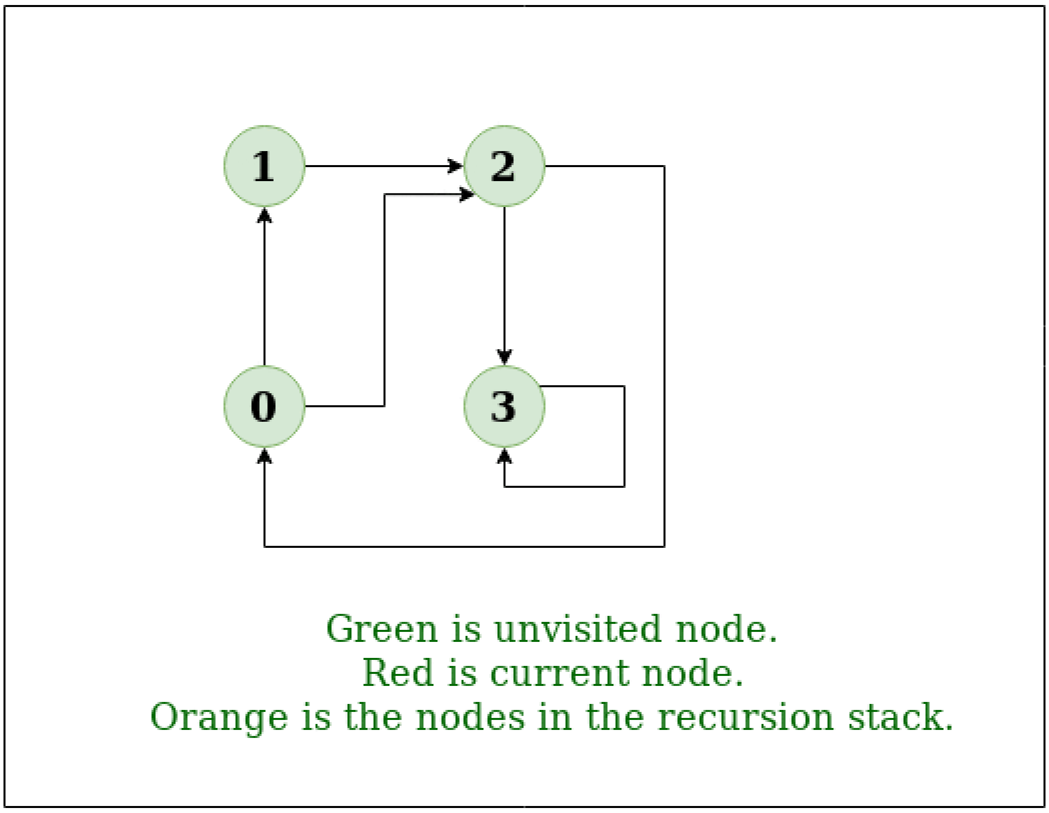
Sr. No.	Criteria	Satisfactory (2) 60 % (meets few expectations)	Good (3) 80 % (Meets all expectations)	Excellent (4) 100 % (exceeds expectations)	Max Marks (Obtained Marks)
1	**Active Participation**	**9**	**12**	**15**	**15 (9,12,15)**
		Active Performance in Conduction of Practical Experiment	Take lead to perform the practical. Helps Classmates.	Ready to work even after lab hour, spare time in lab in free hours or lunch break.	
2		**15**	**20**	**25**	**25(15,20,25)**
	Programming Correctness	Partial understanding of experiment and ample correctness in programming	Good understanding of experiment. If Can independently perform practical? With some guidance for error checking/correcting	Accurate understanding of experiment. Don't need anyone's Help to Perform practical. Questions are answered completely & correctly.	
3		**9**	**12**	**15**	**15 (9,12,15)**
	Timely Submission	Lab report submitted not as directed, but on time.	Lab report submitted as directed, and on time.	Lab report submitted as directed without mistake in time.	
4		**15**	**20**	**25**	**25(15,20,25)**
	Documentation & Presentation	Good representation of experimental stages/processes with examples with some mistakes may be.	Good presentation of all necessary details required for practical without any mistakes in it.	Proper and correct documentation and necessary presentation without any change.	
5		**12**	**16**	**20**	**20(12,16,20)**
	Output, Viva	Output is partially correct. Understanding is less. Few questions answered.	Output is Correct. Understanding is good, Can be improved.	Output is correct. All the concepts are clear. All questions are answered correctly.	
			<u>Marks out of 100</u>		

Title:

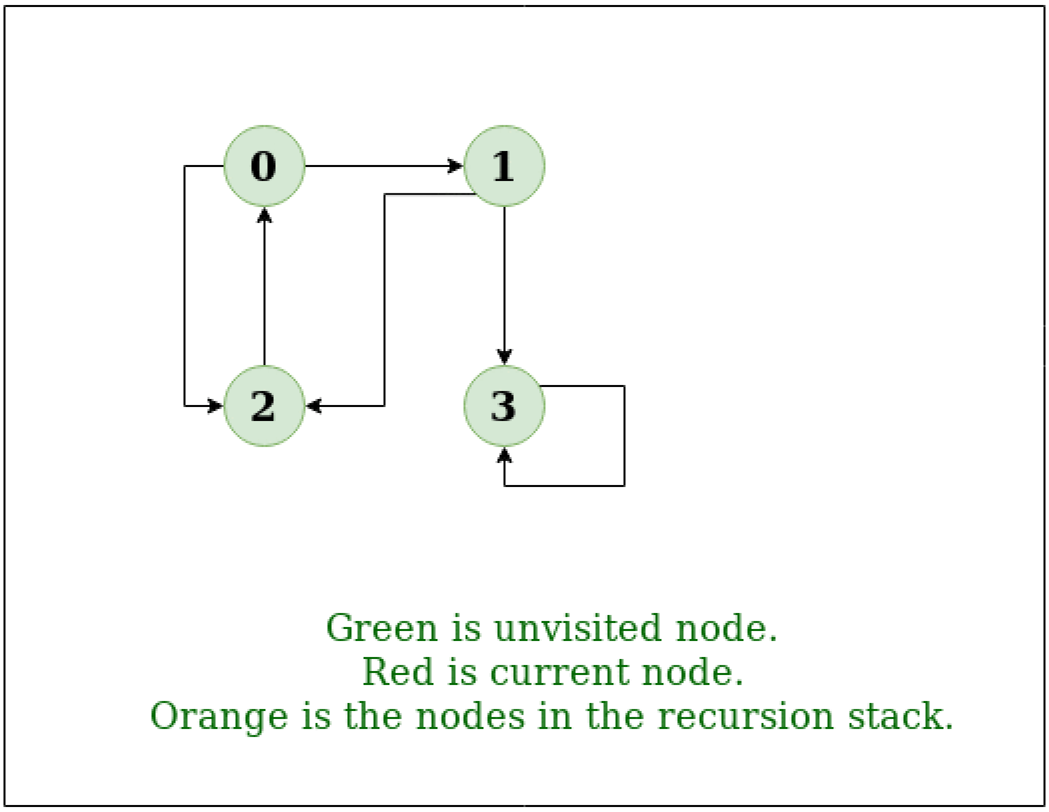
Implement depth first search algorithm and Breadth First Search algorithm. Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

Theory:

Depth First Search or DFS for a Graph
Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.
Example:
Input: n=4, e=6
0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3
Output: DFS from vertex 1 : 1 2 0 3
Explanation:
DFS Diagram:



Input: n = 4, e = 6
2 -> 0, 0 -> 2, 1 -> 2, 0 -> 1, 3 -> 3, 1 -> 3
Output: DFS from vertex 2 : 2 0 1 3
Explanation:
DFS Diagram:



DFS of Graph

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.
Follow the below steps to solve the problem:

- Create a recursive function that takes the index of the node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

Program:

```
// 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)
{
        adj[v].push_back(w); // Add w to v’s list.
}

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's 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;
}
```

Output:

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

Handling A Disconnected Graph:
This will happen by handling a corner case.

The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex, as in a Disconnected graph. To do a complete DFS traversal of such graphs, run DFS from all unvisited nodes after a DFS. The recursive function remains the same.

Follow the below steps to solve the problem:

- Create a recursive function that takes the index of the node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.
- Run a loop from 0 to the number of vertices and check if the node is unvisited in the previous DFS, then call the recursive function with the current node.

Program:

```
// C++ program to print DFS
// traversal for a given
// graph
#include <bits/stdc++.h>
using namespace std;

class Graph {

        // A function used by DFS
        void DFSUtil(int v);

public:
        map<int, bool> visited;
        map<int, list<int> > adj;
        // function to add an edge to graph
        void addEdge(int v, int w);

        // prints DFS traversal of the complete graph
        void DFS();
};

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(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])
                        DFSUtil(*i);
}

// The function to do DFS traversal. It uses recursive
// DFSUtil()
void Graph::DFS()
{
        // Call the recursive helper function to print DFS
        // traversal starting from all vertices one by one
        for (auto i : adj)
                if (visited[i.first] == false)
                        DFSUtil(i.first);
}

// Driver's Code
int main()
{
        // Create a graph given in the above diagram
        Graph g;
        g.addEdge(0, 1);
        g.addEdge(0, 9);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(9, 3);

        cout << "Following is Depth First Traversal \n";

        // Function call
        g.DFS();

        return 0;
}
```
Output:

```
Following is Depth First Traversal

0 1 2 3 9
```

Breadth First Search or BFS for a Graph

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree (See method 2 of this post).

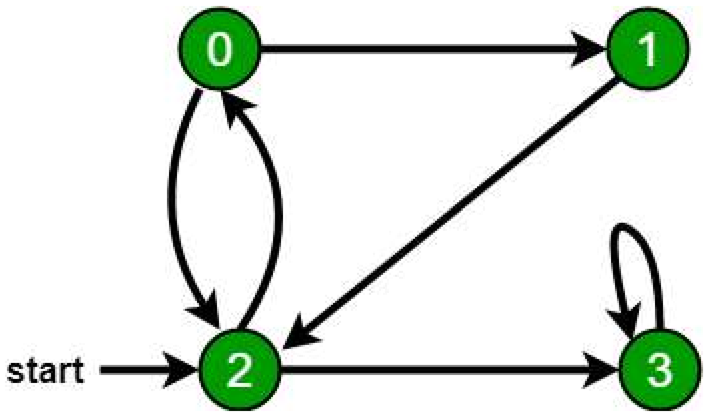
The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.

Example:

In the following graph, we start traversal from vertex 2.



When we come to **vertex 0**, we look for all adjacent vertices of it.

- 2 is also an adjacent vertex of 0.
- If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process.

There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :

2, 3, 0, 1
2, 0, 3, 1

Implementation of BFS traversal:

Follow the below method to implement BFS traversal.

- Declare a queue and insert the starting vertex.
- Initialize a visited array and mark the starting vertex as visited.
- Follow the below process till the queue becomes empty:
    - Remove the first vertex of the queue.
    - Mark that vertex as visited.
    - Insert all the unvisited neighbours of the vertex into the queue.

Program:

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<bits/stdc++.h>
using namespace std;

// This class represents a directed graph using
// adjacency list representation
```

```
class Graph
{
        int V; // No. of vertices

        // Pointer to an array containing adjacency
        // lists
        vector<list<int>> adj;
public:
        Graph(int V); // Constructor

        // 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)
{
        adj[v].push_back(w); // Add w to v's list.
}

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 a adjacent has not been visited,
                    // then mark it visited and enqueue it
                    for (auto adjecent: adj[s])
                    {
                              if (!visited[adjecent])
                              {
                                        visited[adjecent] = true;
                                        queue.push_back(adjecent);
                              }
                    }
          }
}

// Driver program to test methods of graph class
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
```

ESTD - 1928

SNJB's Late Sau. K. B. Jain College of Engineering, Chandwad.

Department of Artificial Intelligence and Data Science

Subject: Software Laboratory I (317523)

Experiment No. (Group)

Title:

Date of Completion: ________________ **Date of Submission: ________________**

S.N.	Criteria	Possible Marks	Obtained Marks
1	**Active Participation**	9, 12, 15	
2	**Programming Correctness**	15, 20, 25	
3	**Timely Submission**	9, 12, 15	
4	**Documentation & Presentation**	15, 20, 25	
5	**Output ,Viva**	12, 16, 20	
		Total	

Date:__________________

Prof. K. S. Sagale
(Subject Teacher)

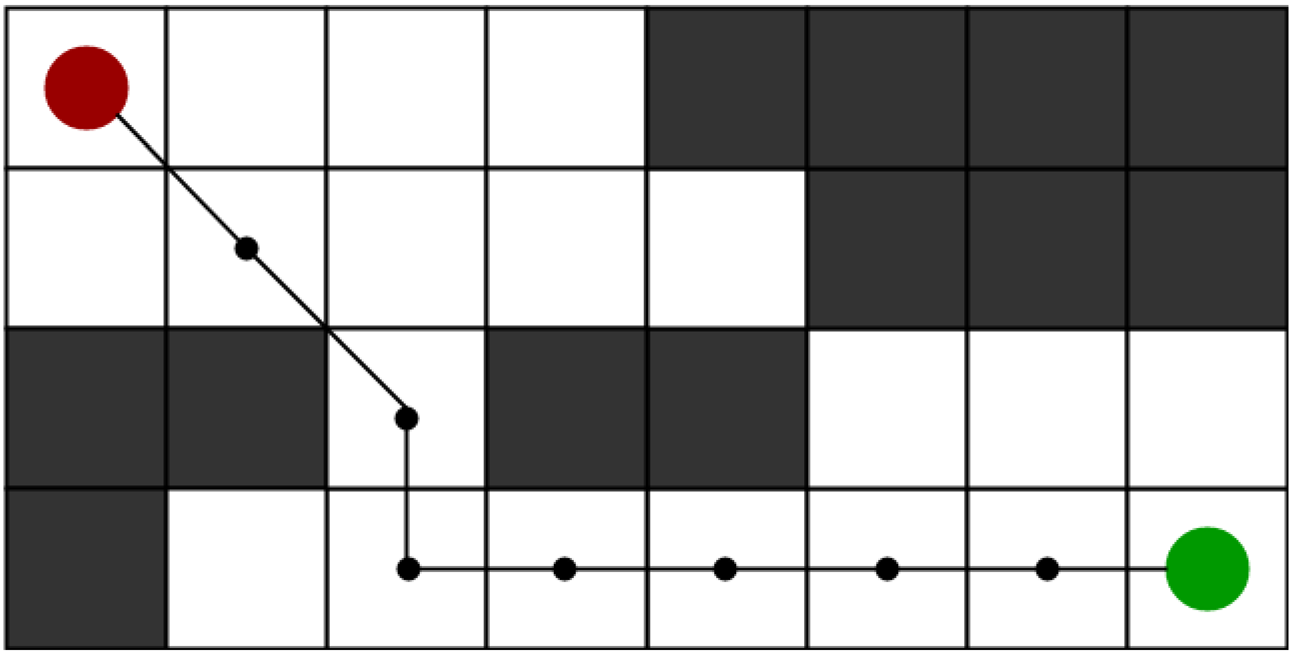
Title:

Implement A star (A*) Algorithm for any game search problem.

Theory:

A* Search 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)



What is A* Search Algorithm?

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Explanation

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue. What A* Search Algorithm does is that at each step it picks the node according to a value-'f' which is a parameter equal to the sum of two other parameters – 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell. We define 'g' and 'h' as simply as possible below

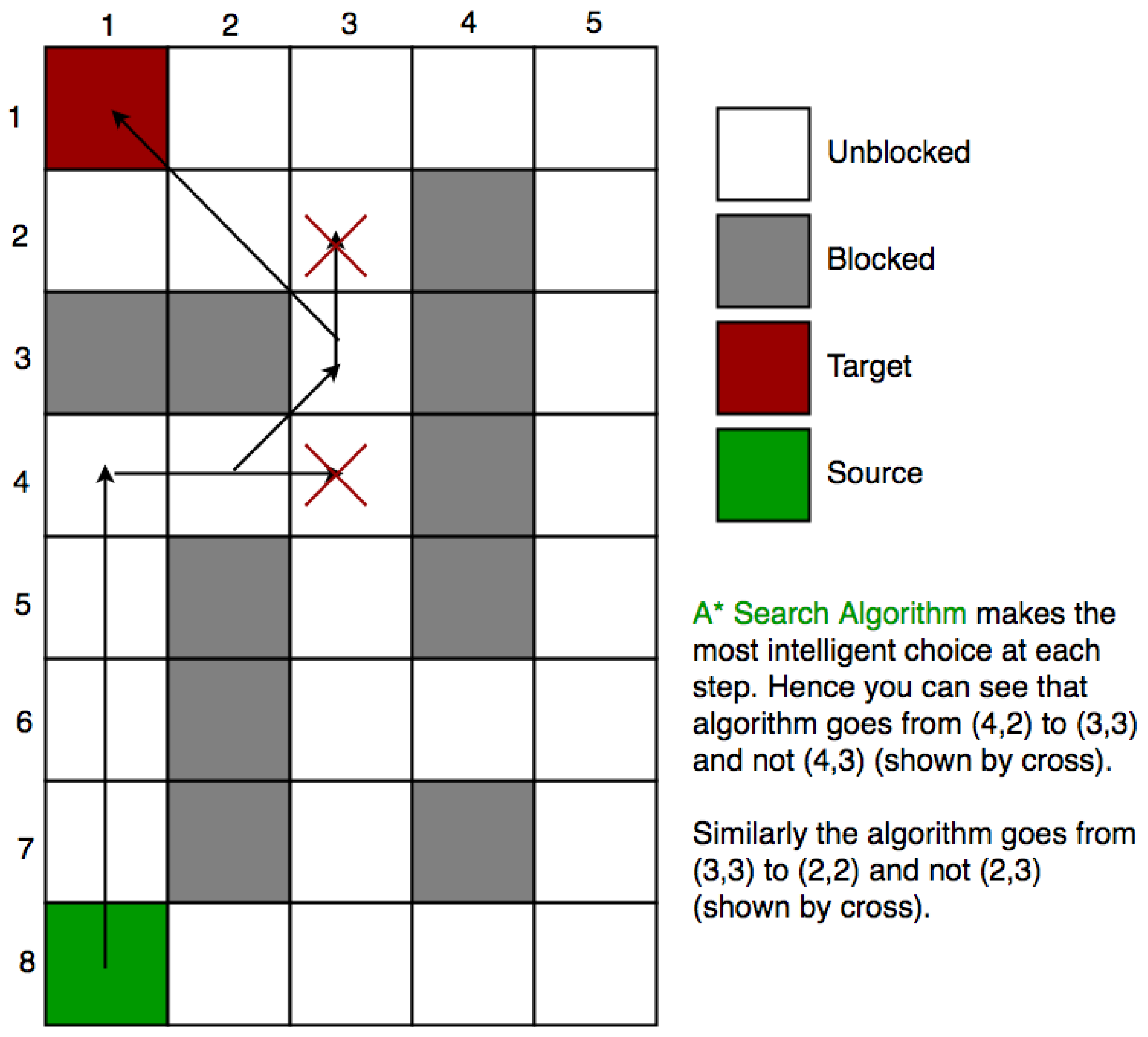
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there. h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this 'h' which are discussed in the later sections.
Algorithm
We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

// A* Search 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)
```

So suppose as in the below figure if we want to reach the target cell from the source cell, then the A* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristics.



A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Heuristics
We can calculate g but how to calculate h ?
We can do things.
A) Either calculate the exact value of h (which is certainly time consuming).
OR
B) Approximate the value of h using some heuristics (less time consuming).
We will discuss both of the methods.
A) Exact Heuristics –
We can find exact values of h, but that is generally very time consuming.

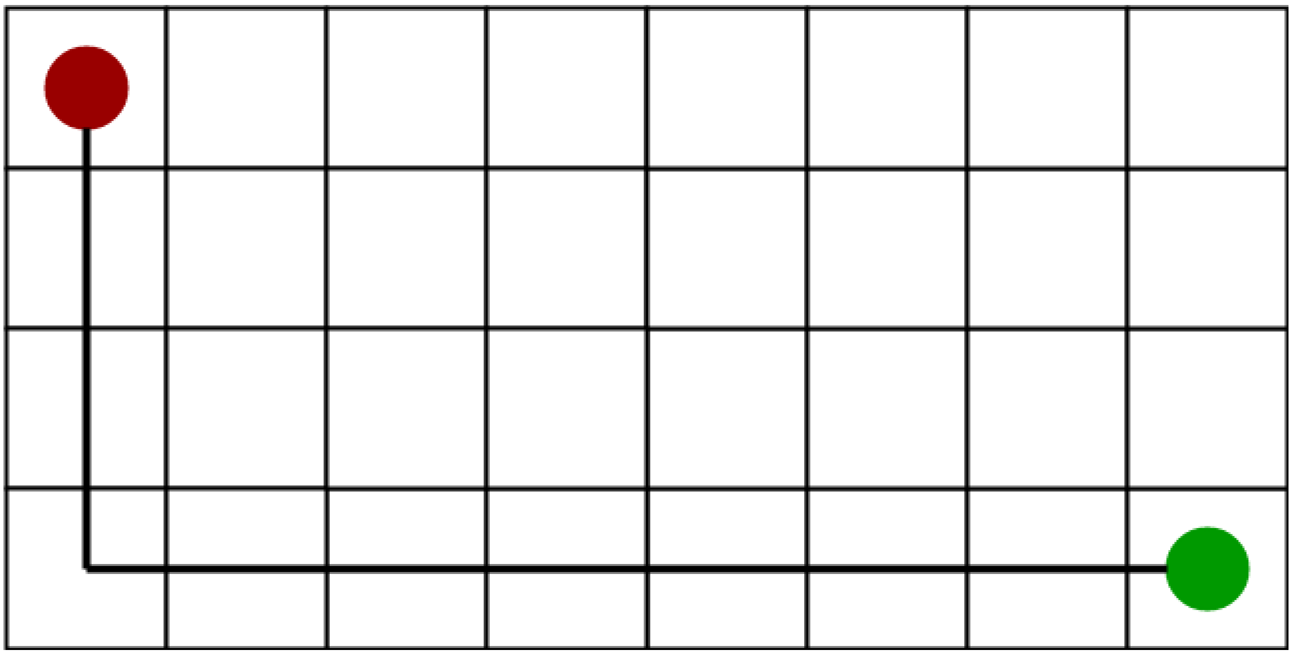
Below are some of the methods to calculate the exact value of h.
1) Pre-compute the distance between each pair of cells before running the A* Search Algorithm.
2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance
B) Approximation Heuristics –
There are generally three approximation heuristics to calculate h –
1) Manhattan Distance –

- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

h = abs (current_cell.x – goal.x) +
abs (current_cell.y – goal.y)

- When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



2) Diagonal Distance-

- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

dx = abs(current_cell.x – goal.x)

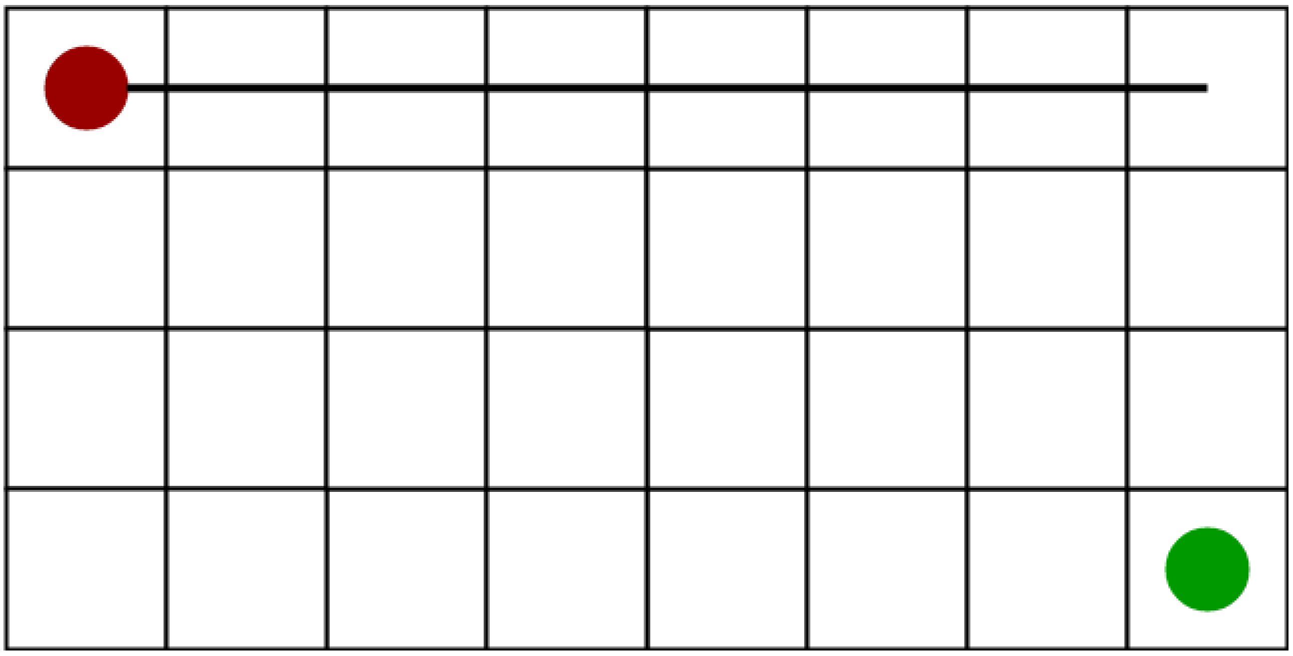
dy = abs(current_cell.y – goal.y)

h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)

where D is length of each node(usually = 1) and D2 is diagonal distance between each node (usually = sqrt(2)).

- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

The Diagonal Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



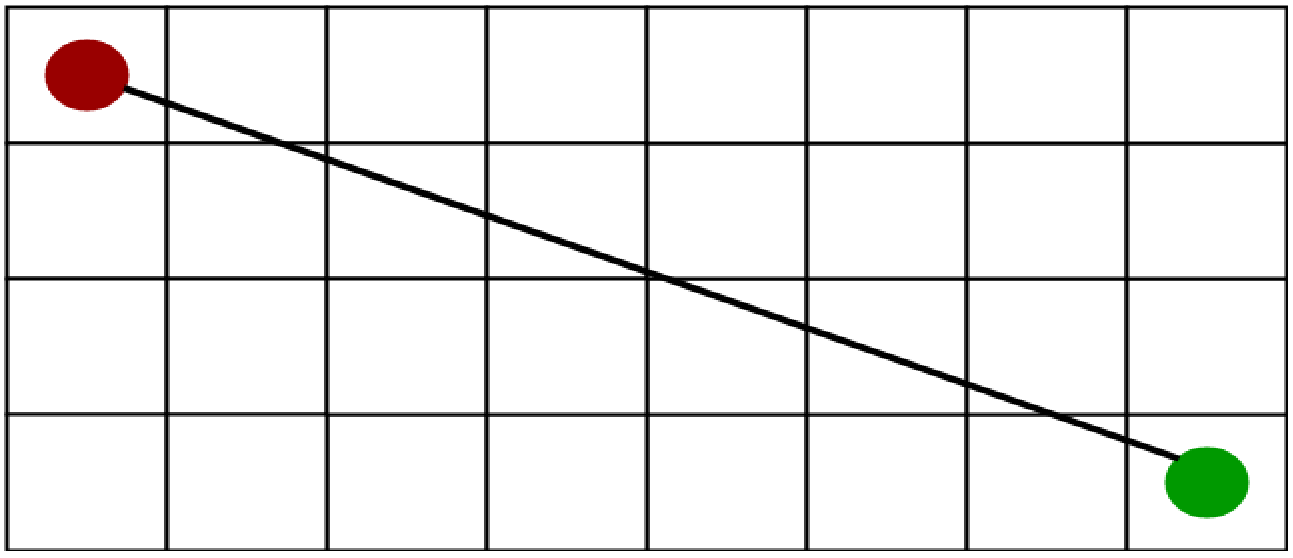
3) Euclidean Distance-

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

h = sqrt ((current_cell.x – goal.x)2 +
(current_cell.y – goal.y)2)

- When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



Relation (Similarity and Differences) with other algorithms-
Dijkstra is a special case of A* Search Algorithm, where h = 0 for all nodes.

Program:

```
// 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);
}
```

Limitations

Although being the best path finding algorithm around, A* Search Algorithm doesn’t produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h.

SNJB's Late Sau. K. B. Jain College of Engineering, Chandwad.

Department of Artificial Intelligence and Data Science

Subject: Software Laboratory I (317523)

Experiment No. (Group)

Title:

Date of Completion: ________________ **Date of Submission:** ________________

S.N.	**Criteria**	**Possible Marks**	**Obtained Marks**
1	**Active Participation**	9, 12, 15	
2	**Programming Correctness**	15, 20, 25	
3	**Timely Submission**	9, 12, 15	
4	**Documentation & Presentation**	15, 20, 25	
5	**Output ,Viva**	12, 16, 20	
		Total	

Date:__________________

Prof. K. S. Sagale
(Subject Teacher)

Title:

Implement Alpha-Beta Tree search for any game search problem.

Theory:

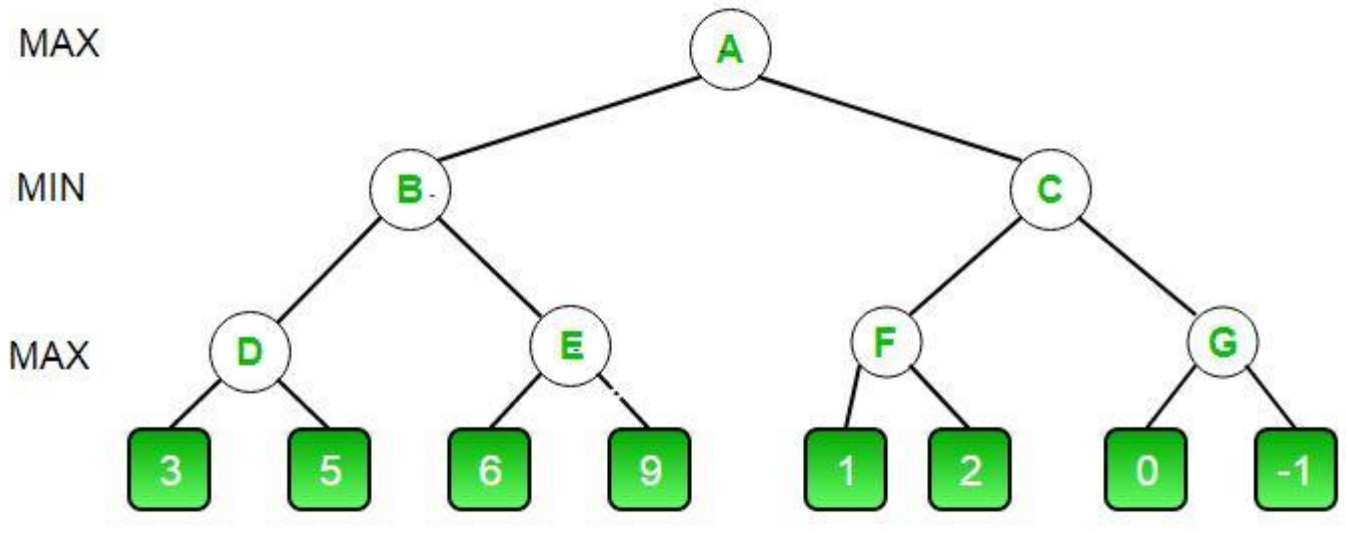
Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning)

Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.Let's define the parameters alpha and beta. **Alpha** is the best value that the **maximizer** currently can guarantee at that level or above. **Beta** is the best value that the **minimizer** currently can guarantee at that level or above.

Pseudocode :

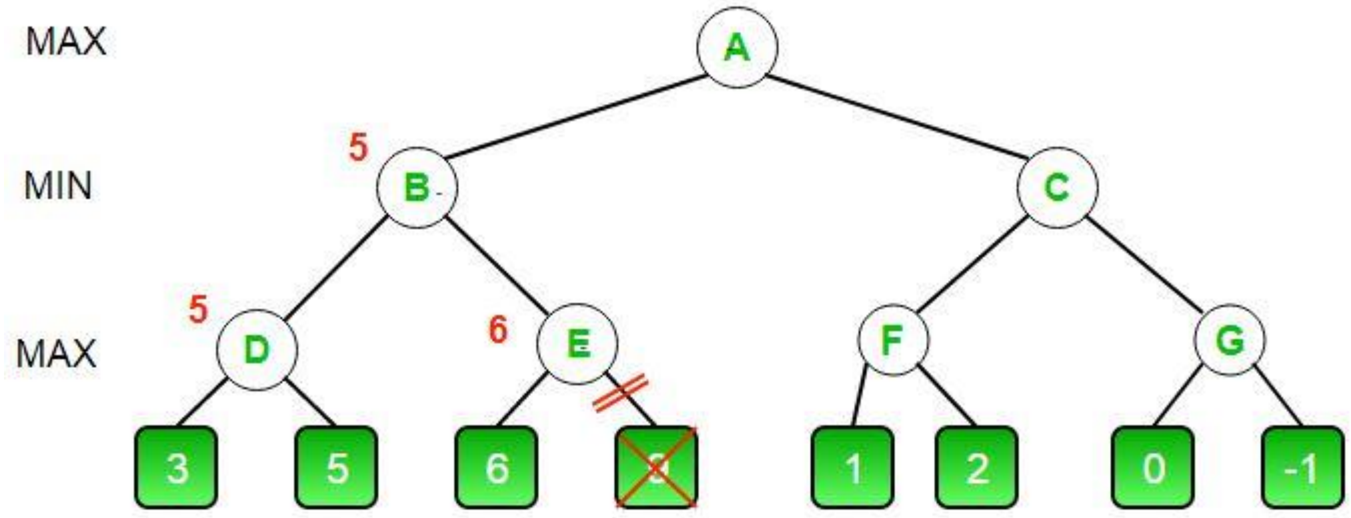
```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

  if node is a leaf node :
    return value of the node

  if isMaximizingPlayer :
    bestVal = -INFINITY
    for each child node :
      value = minimax(node, depth+1, false, alpha, beta)
      bestVal = max( bestVal, value)
      alpha = max( alpha, bestVal)
      if beta <= alpha:
        break
    return bestVal

  else :
    bestVal = +INFINITY
    for each child node :
      value = minimax(node, depth+1, true, alpha, beta)
      bestVal = min( bestVal, value)
      beta = min( beta, bestVal)
      if beta <= alpha:
        break
    return bestVal


// Calling the function for the first time.
minimax(0, 0, true, -INFINITY, +INFINITY)
```

Let's make above algorithm clear with an example.



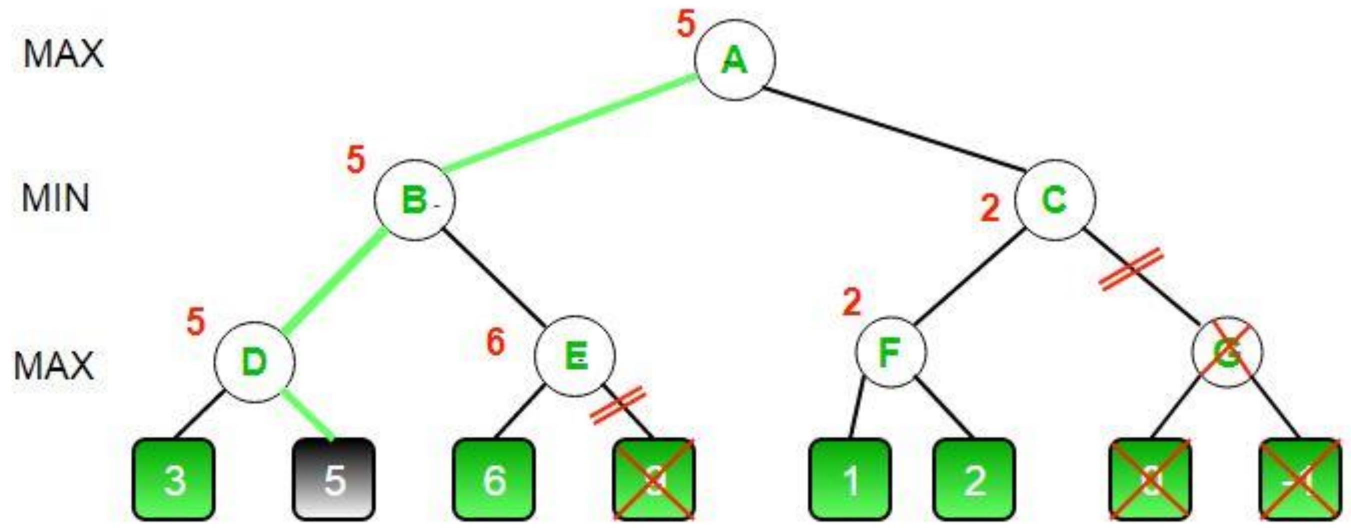
- The initial call starts from **A**. The value of alpha here is **-INFINITY** and the value of beta is **+INFINITY**. These values are passed down to subsequent nodes in the tree. At **A** the maximizer must choose max of **B** and **C**, so **A** calls **B** first
- At **B** it the minimizer must choose min of **D** and **E** and hence calls **D** first.
- At **D**, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at **D** is max(-INF, 3) which is 3.
- To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.
- **D** now looks at its right child which returns a value of 5.At **D**, alpha = max(3, 5) which is 5. Now the value of node **D** is 5
- **D** returns a value of 5 to **B**. At **B**, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. **B** now calls **E** to see if he can get a lower value than 5.
- At **E** the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at **B** and that is what **B** passed down to **E**
- Now **E** looks at its left child which is 6. At **E**, alpha = max(-INF, 6) which is 6. Here the condition becomes true. beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and **E** returns 6 to **B**
- Note how it did not matter what the value of **E**'s right child is. It could have been +INF or -INF, it still wouldn't matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of **B**. This way we dint have to look at that 9 and hence saved computation time.
- **E** returns a value of 6 to **B**. At **B**, beta = min(5, 6) which is 5.The value of node **B** is also 5

So far this is how our game tree looks. The 9 is crossed out because it was never computed.



- **B** returns 5 to **A**. At **A**, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. **A** now calls **C** to see if it can get a higher value than 5.
- At **C**, alpha = 5 and beta = +INF. **C** calls **F**
- At **F**, alpha = 5 and beta = +INF. **F** looks at its left child which is a 1. alpha = max(5, 1) which is still 5.
- **F** looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5
- **F** returns a value of 2 to **C**. At **C**, beta = min(+INF, 2). The condition beta <= alpha becomes true as beta = 2 and alpha = 5. So it breaks and it does not even have to compute the entire sub-tree of **G**.
- The intuition behind this break off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub tree.
- **C** now returns a value of 2 to **A**. Therefore the best value at **A** is max(5, 2) which is a 5.
- Hence the optimal value that the maximizer can get is 5

This is how our final game tree looks like. As you can see **G** has been crossed out as it was never computed.



Program:

```
// C++ program to demonstrate
// working of Alpha-Beta Pruning
#include<bits/stdc++.h>
using namespace std;

// Initial values of
// Alpha and Beta
const int MAX = 1000;
const int MIN = -1000;

// Returns optimal value for
// current player(Initially called
// for root and maximizer)
int minimax(int depth, int nodeIndex,
                        bool maximizingPlayer,
                        int values[], int alpha,
                        int beta)
{

        // Terminating condition. i.e
        // leaf node is reached
        if (depth == 3)
                return values[nodeIndex];

        if (maximizingPlayer)
        {
                int best = MIN;

                // Recur for left and
                // right children
                for (int i = 0; i < 2; i++)
                {
```

```
                int val = minimax(depth + 1, nodeIndex * 2 + i,
                                                false, values, alpha, beta);
                best = max(best, val);
                alpha = max(alpha, best);

                // Alpha Beta Pruning
                if (beta <= alpha)
                        break;
            }
            return best;
        }
        else
        {
            int best = MAX;

            // Recur for left and
            // right children
            for (int i = 0; i < 2; i++)
            {
                int val = minimax(depth + 1, nodeIndex * 2 + i,
                                                true, values, alpha, beta);
                best = min(best, val);
                beta = min(beta, best);

                // Alpha Beta Pruning
                if (beta <= alpha)
                        break;
            }
            return best;
        }
}
```

```
// Driver Code
int main()
{
        int values[8] = { 3, 5, 6, 9, 1, 2, 0, -1 };
        cout <<"The optimal value is : "<< minimax(0, 0, true, values, MIN, MAX);;
        return 0;
}
```

Output:

```
The optimal value is : 5
```

ESTD - 1928

SNJB's Late Sau. K. B. Jain College of Engineering, Chandwad.

Department of Artificial Intelligence and Data Science

Subject: Software Laboratory I (317523)

Experiment No. (Group)

Title:

Date of Completion: ________________ **Date of Submission: ________________**

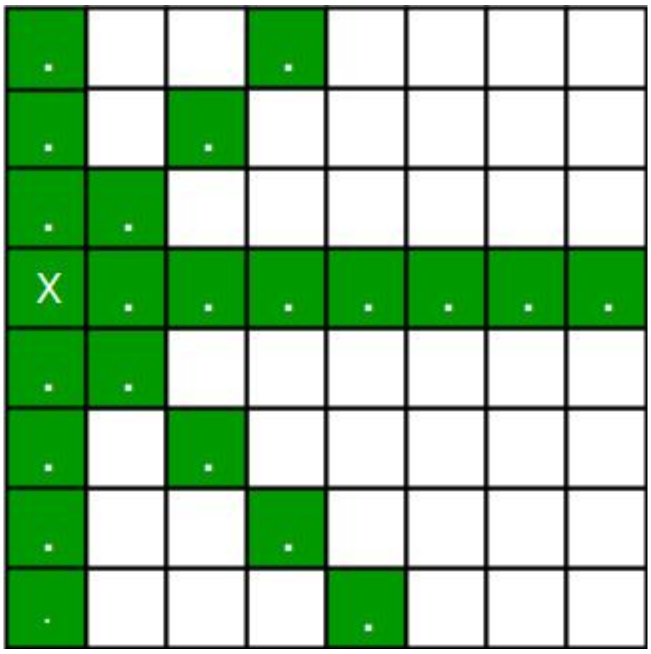
S.N.	Criteria	Possible Marks	Obtained Marks
1	**Active Participation**	9, 12, 15	
2	**Programming Correctness**	15, 20, 25	
3	**Timely Submission**	9, 12, 15	
4	**Documentation & Presentation**	15, 20, 25	
5	**Output ,Viva**	12, 16, 20	
		Total	

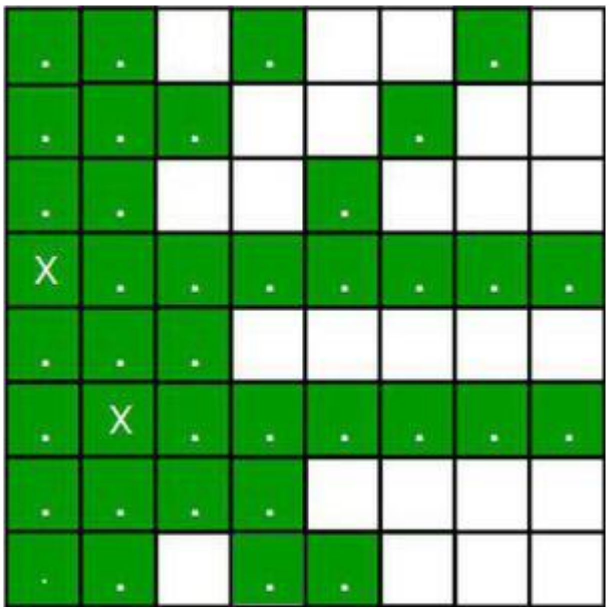
Date:__________________

Prof. K. S. Sagale
(Subject Teacher)

Title:
Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.

Theory:
N Queen Problem using Branch And Bound
The N queens puzzle is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.
The backtracking Algorithm for N-Queen is already discussed here. In a backtracking solution, we backtrack when we hit a dead end. In Branch and Bound solution, after building a partial solution, we figure out that there is no point going any deeper as we are going to hit a dead end.
Let's begin by describing the backtracking solution. "The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false."



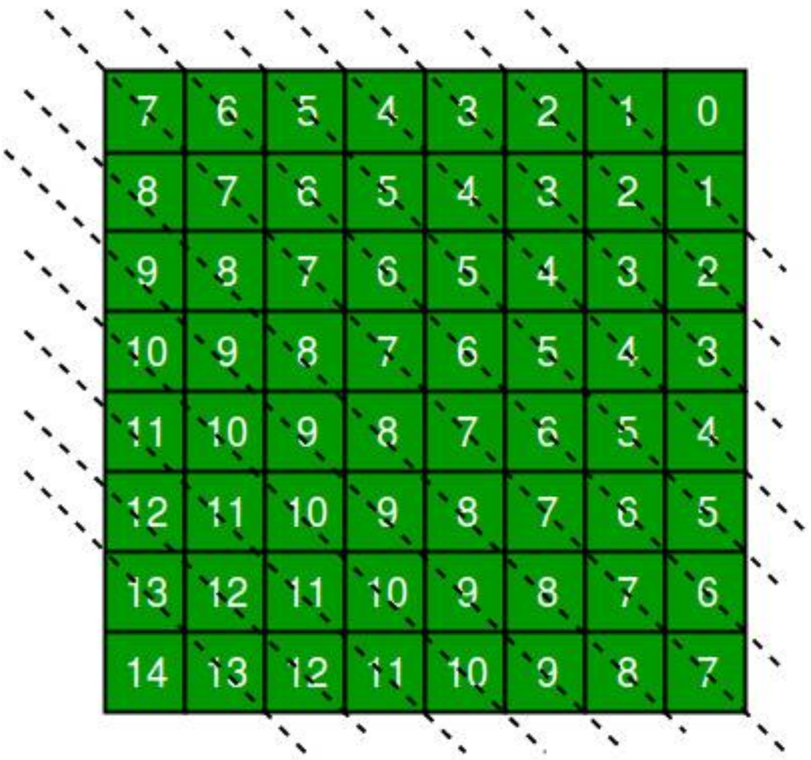


Placing 1st queen on (3, 0) and 2nd queen on (5, 1)
For the 1st Queen, there are total 8 possibilities as we can place 1st Queen in any row of first column. Let's place Queen 1 on row 3.
After placing 1st Queen, there are 7 possibilities left for the 2nd Queen. But wait, we don't really have 7 possibilities. We cannot place Queen 2 on rows 2, 3 or 4 as those cells are under attack from Queen 1. So, Queen 2 has only 8 – 3 = 5 valid positions left.
After picking a position for Queen 2, Queen 3 has even fewer options as most of the cells in its column are under attack from the first 2 Queens.
We need to figure out an efficient way of keeping track of which cells are under attack. In previous solution we kept an 8-by-8 Boolean matrix and update it each time we placed a queen, but that required linear time to update as we need to check for safe cells.

Basically, we have to ensure 4 things:
1. No two queens share a column.
2. No two queens share a row.
3. No two queens share a top-right to left-bottom diagonal.
4. No two queens share a top-left to bottom-right diagonal.

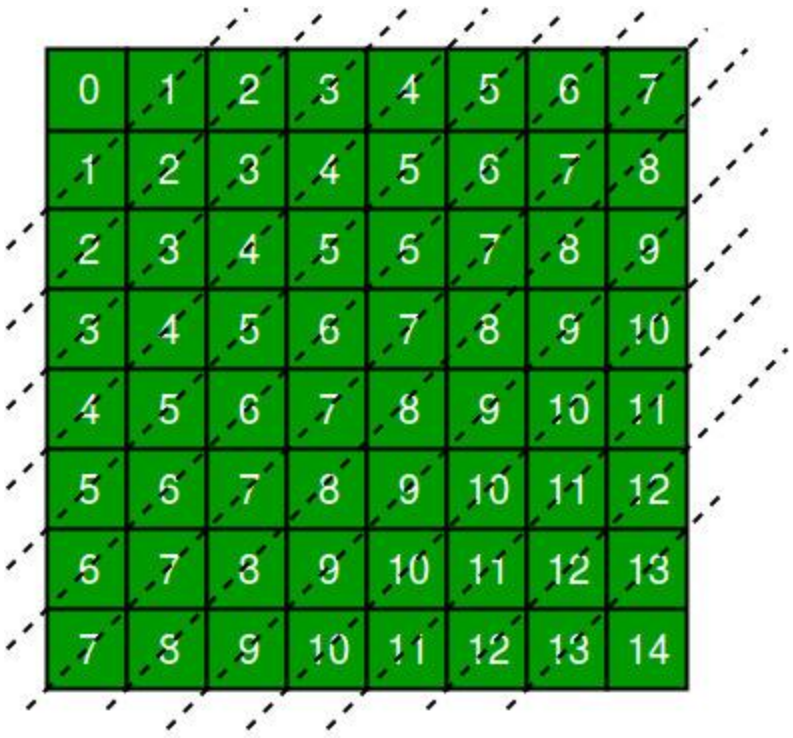
Number 1 is automatic because of the way we store the solution. For number 2, 3 and 4, we can perform updates in O(1) time. The idea is to keep three Boolean arrays that tell us which rows and which diagonals are occupied.
Lets do some pre-processing first. Let's create two N x N matrix one for / diagonal and other one for \ diagonal. Let's call them slashCode and backslashCode respectively. The trick is to fill them in such a way that two queens sharing a same /diagonal will have the same value in matrix slashCode, and if they share same \-diagonal, they will have the same value in backslashCode matrix.

For an N x N matrix, fill slashCode and backslashCode matrix using below formula –
slashCode[row][col] = row + col
backslashCode[row][col] = row – col + (N-1)
Using above formula will result in below matrices



7	6	5	4	3	2	1	0
8	7	6	5	4	3	2	1
9	8	7	6	5	4	3	2
10	9	8	7	6	5	4	3
11	10	9	8	7	6	5	4
12	11	10	9	8	7	6	5
13	12	11	10	9	8	7	6
14	13	12	11	10	9	8	7

r -c + 7



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

r + c

The ‘N – 1’ in the backslash code is there to ensure that the codes are never negative because we will be using the codes as indices in an array.Now before we place queen i on row j, we first check whether row j is used (use an array to store row info). Then we check whether slash code (j + i) or backslash code (j – i + 7) are used (keep two arrays that will tell us which diagonals are occupied). If yes, then we have to try a different location for queen i. If not, then we mark the row and the two diagonals as used and recurse on queen i + 1. After the recursive call returns and before we try another position for queen i, we need to reset the row, slash code and backslash code as unused again, like in the code from the previous notes.

Program:

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


// function for printing the solution
void printSol(vector<vector<int>>board)
{
for(int i = 0;i<N;i++){
	for(int j = 0;j<N;j++){
		cout<<board[i][j]<<" ";
	}
	cout<<"\n";
}
}

/* Optimized isSafe function
isSafe function to check if current row contains or current left diagonal or current right diagonal contains any queen or not if
yes return false
else return true
*/

bool isSafe(int row ,int col ,vector<bool>rows , vector<bool>left_digonals ,vector<bool>Right_digonals)
{

if(rows[row] == true || left_digonals[row+col] == true || Right_digonals[col-row+N-1] == true){
	return false;
}

return true;
}

// Recursive function to solve N-queen Problem
bool solve(vector<vector<int>>& board ,int col ,vector<bool>rows , vector<bool>left_digonals ,vector<bool>Right_digonals)
{
	// base Case : If all Queens are placed
	if(col>=N){
		return true;
	}

	/* Consider this Column and move in all rows one by one */
	for(int i = 0;i<N;i++)
```

```
        {
                if(isSafe(i,col,rows,left_digonals,Right_digonals) == true)
                {
                        rows[i] = true;
                        left_digonals[i+col] = true;
                        Right_digonals[col-i+N-1] = true;
                        board[i][col] = 1; // placing the Queen in board[i][col]

                                /* recur to place rest of the queens */

                        if(solve(board,col+1,rows,left_digonals,Right_digonals) == true){
                                return true;
                        }

                        // Backtracking
                        rows[i] = false;
                        left_digonals[i+col] = false;
                        Right_digonals[col-i+N-1] = false;
                        board[i][col] = 0; // removing the Queen from board[i][col]

                }
        }

                return false;
}


int main()
{
// Taking input from the user

cout<<"Enter the no of rows for the square Board : ";
cin>>N;


// board of size N*N
vector<vector<int>>board(N,vector<int>(N,0));


        // array to tell which rows are occupied
vector<bool>rows(N,false);

// arrays to tell which diagonals are occupied
vector<bool>left_digonals(2*N-1,false);
vector<bool>Right_digonals(2*N-1,false);


bool ans = solve(board , 0, rows,left_digonals,Right_digonals);
```

```
if(ans == true){

            // printing the solution Board
            printSol(board);
}
else{
            cout<<"Solution Does not Exist\n";
}
}
```

Input:
Enter the no of rows for the square Board : 8

Output :

```
 1  0  0  0  0  0  0  0

 0  0  0  0  0  0  1  0

 0  0  0  0  1  0  0  0

 0  0  0  0  0  0  0  1

 0  1  0  0  0  0  0  0

 0  0  0  1  0  0  0  0

 0  0  0  0  0  1  0  0

 0  0  1  0  0  0  0  0
```

SNJB's Late Sau. K. B. Jain College of Engineering, Chandwad.

Department of Artificial Intelligence and Data Science

Subject: Software Laboratory I (317523)

Experiment No. (Group)

Title:

Date of Completion: ________________ **Date of Submission:** ________________

S.N.	Criteria	Possible Marks	Obtained Marks
1	**Active Participation**	9, 12, 15	
2	**Programming Correctness**	15, 20, 25	
3	**Timely Submission**	9, 12, 15	
4	**Documentation & Presentation**	15, 20, 25	
5	**Output ,Viva**	12, 16, 20	
		Total	

Date:__________________

Prof. K. S. Sagale
(Subject Teacher)

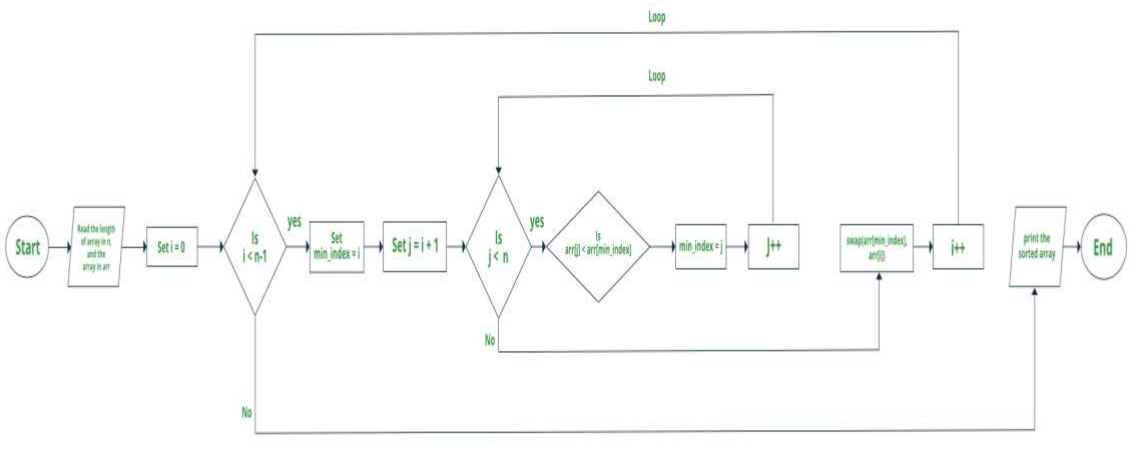
Title:
Implement Greedy search algorithm for Selection Sort.

Theory:
Selection Sort Algorithm
The **selection sort algorithm** sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.
The algorithm maintains two subarrays in a given array.

- The subarray which already sorted.
- The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Flowchart of the Selection Sort:



Flowchart for Selection Sort

How selection sort works?
Lets consider the following array as an example: **arr[] = {64, 25, 12, 22, 11}**
First pass:

- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where **64** is stored presently, after traversing whole array it is clear that **11** is the lowest value.

64	25	12	22	11

- Thus, replace 64 with 11. After one iteration **11**, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

11	25	12	22	64

Second Pass:

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

11	**25**	12	22	64

- After traversing, we found that **12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.

11	**12**	25	22	64

Third Pass:

- Now, for third place, where **25** is present again traverse the rest of the array and find the third least value present in the array.

11	12	**25**	22	64

- While traversing, **22** came out to be the third least value and it should appear at the third place in the array, thus swap **22** with element present at third position.

11	12	**22**	25	64

Fourth pass:

- Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
- As **25** is the 4th lowest value hence, it will place at the fourth position.

11	12	22	**25**	64

Fifth Pass:

- At last the largest value present in the array automatically get placed at the last position in the array
- The resulted array is the sorted array.

11	12	22	**25**	64

Follow the below steps to solve the problem:

- Initialize minimum value(min_idx) to location 0.
- Traverse the array to find the minimum element in the array.
- While traversing if any element smaller than min_idx is found then swap both the values.
- Then, increment min_idx to point to the next element.
- Repeat until the array is sorted.

Program:

```
// C++ program for implementation of
// selection sort
#include <bits/stdc++.h>
using namespace std;

//Swap function
void swap(int *xp, int *yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}

void selectionSort(int arr[], int n)
{
	int i, j, min_idx;

	// One by one move boundary of
	// unsorted subarray
	for (i = 0; i < n-1; i++)
	{

		// Find the minimum element in
		// unsorted array
		min_idx = i;
		for (j = i+1; j < n; j++)
		if (arr[j] < arr[min_idx])
			min_idx = j;

		// Swap the found minimum element
		// with the first element
		if(min_idx!=i)
			swap(&arr[min_idx], &arr[i]);
	}
}

//Function to print an array
void printArray(int arr[], int size)
{
	int i;
	for (i=0; i < size; i++)
		cout << arr[i] << " ";
	cout << endl;
}

// Driver program to test above functions
int main()
{
	int arr[] = {64, 25, 12, 22, 11};
	int n = sizeof(arr)/sizeof(arr[0]);
	selectionSort(arr, n);
	cout << "Sorted array: \n";
	printArray(arr, n);
	return 0;
}
```

Output:

```
Sorted array:

11 12 22 25 64
```

SNJB's Late Sau. K. B. Jain College of Engineering, Chandwad.

Department of Artificial Intelligence and Data Science

Subject: Software Laboratory I (317523)

Experiment No. (Group)

Title:

Date of Completion: ________________ **Date of Submission: ________________**

S.N.	Criteria	Possible Marks	Obtained Marks
1	**Active Participation**	9, 12, 15	
2	**Programming Correctness**	15, 20, 25	
3	**Timely Submission**	9, 12, 15	
4	**Documentation & Presentation**	15, 20, 25	
5	**Output ,Viva**	12, 16, 20	
		Total	

Date:__________________

Prof. K. S. Sagale
(Subject Teacher)

Title:
Mini Project: Implement Information management.

Theory:
Problem Statement:
Write a program to build a simple Software for Student Information Management System which can perform the following operations:

1. Store the First name of the student.
2. Store the Last name of the student.
3. Store the unique Roll number for every student.
4. Store the CGPA of every student.
5. Store the courses registered by the student.

Approach:
The idea is to form an individual functions for every operation. All the functions are unified together to form software.
1. Add Student Details: Get data from user and add a student to the list of students. While adding the students into the list, check for the uniqueness of the roll number.
2. Find the student by the given roll number: This function is to find the student record for the given roll number and print the details.
3. Find the student by the given first name: This function is to find all the students with the given first name and print their details.
4. Find the students registered in a course: This function is to find all the students who have registered for a given course.
5. Count of Students: This function is to print the total number of students in the system
6. Delete a student: This function is to delete the student record for the given roll number.
7. Update Student: This function is to update the student records. This function does not ask for new details for all fields but the user should be able to pick and choose what he wants to update.

Program:
```
// Function to add the student into the database
void add_student()
{

        printf("Add the Students Details\n");
        printf("-------------------------\n");
        printf("Enter the first name of student\n");

        // First name of the student
        st[i].fname = "Rahul";
        printf("Enter the last name of student\n");

        // Last name of the student
        st[i].lname = "Kumar";
        printf("Enter the Roll Number\n");

        // Roll Number of the student
        st[i].roll = 1;
        printf("Enter the CGPA you obtained\n");

        // CGPA of the student
        st[i].cgpa = 8;
        printf("Enter the course ID"
                " of each course\n");

        // Storing the courses every student
        // is enrolled in
        for (int j = 0; j < 5; j++) {
                st[i].cid[j] = j;
        }
        i = i + 1;
}


// C program for the implementation of
// menu driven program for student
// management system
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Variable to keep track of
// number of students
int i = 0;

// Structure to store the student
struct sinfo {
        char fname[50];
        char lname[50];
        int roll;
        float cgpa;
        int cid[10];
} st[55];

// Function to add the student
void add_student()
{

        printf("Add the Students Details\n");
        printf("-------------------------\n");
        printf("Enter the first "
                "name of student\n");
        scanf("%s", st[i].fname);
        printf("Enter the last name"
                " of student\n");
        scanf("%s", st[i].lname);
        printf("Enter the Roll Number\n");
        scanf("%d", &st[i].roll);
        printf("Enter the CGPA "
                "you obtained\n");
        scanf("%f", &st[i].cgpa);
        printf("Enter the course ID"
                " of each course\n");
        for (int j = 0; j < 5; j++) {
                scanf("%d", &st[i].cid[j]);
        }
        i = i + 1;
}

// Function to find the student
// by the roll number
void find_rl()
{
        int x;
        printf("Enter the Roll Number"
                " of the student\n");
        scanf("%d", &x);
        for (int j = 1; j <= i; j++) {
                if (x == st[i].roll) {
                        printf(
                                "The Students Details are\n");
                        printf(
                                "The First name is %s\n",
                                st[i].fname);
                        printf(
                                "The Last name is %s\n",
                                st[i].lname);
                        printf(
                                "The CGPA is %f\n",
                                st[i].cgpa);
                        printf(
                                "Enter the course ID"
                                " of each course\n");
                }
                for (int j = 0; j < 5; j++) {
                        printf(
                                "The course ID are %d\n",
                                st[i].cid[j]);
                }
                break;
        }
}

// Function to find the student
// by the first name
```

```
void find_fn()
{
	char a[50];
	printf("Enter the First Name"
		" of the student\n");
	scanf("%s", a);
	int c = 0;

	for (int j = 1; j <= i; j++) {
		if (!strcmp(st[j].fname, a)) {

			printf(
				"The Students Details are\n");
			printf(
				"The First name is %s\n",
				st[i].fname);
			printf(
				"The Last name is %s\n",
				st[i].lname);
			printf(
				"The Roll Number is %d\n ",
				st[i].roll);
			printf(
				"The CGPA is %f\n",
				st[i].cgpa);
			printf(
				"Enter the course ID of each course\n");

			for (int j = 0; j < 5; j++) {
				printf(
					"The course ID are %d\n",
					st[i].cid[j]);
			}
			c = 1;
		}
		else
			printf(
				"The First Name not Found\n");
	}
}

// Function to find
// the students enrolled
// in a particular course
void find_c()
{
	int id;
	printf("Enter the course ID \n");
	scanf("%d", &id);
	int c = 0;

	for (int j = 1; j <= i; j++) {
		for (int d = 0; d < 5; d++) {
			if (id == st[j].cid[d]) {

				printf(
					"The Students Details are\n");
				printf(
					"The First name is %s\n",
					st[i].fname);
				printf(
					"The Last name is %s\n",
					st[i].lname);
				printf(
					"The Roll Number is %d\n ",
					st[i].roll);
				printf(
					"The CGPA is %f\n",
					st[i].cgpa);

				c = 1;

				break;
			}
			else
				printf(
					"The First Name not Found\n");
		}
	}
}

// Function to print the total
// number of students
void tot_s()
{
	printf("The total number of"
		" Student is %d\n",
		i);
	printf("\n you can have a "
		"max of 50 students\n");
	printf("you can have %d "
		"more students\n",
		50 - i);
}

// Function to delete a student
// by the roll number
void del_s()
```

```
{
        int a;
        printf("Enter the Roll Number"
                " which you want "
                "to delete\n");
        scanf("%d", &a);
        for (int j = 1; j <= i; j++) {
                if (a == st[j].roll) {
                        for (int k = j; k < 49; k++)
                                st[k] = st[k + 1];
                        i--;
                }
        }
        printf("The Roll Number"
                " is removed Successfully\n");
}

// Function to update a students data
void up_s()
{

        printf("Enter the roll number"
                " to update the entry : ");
        long int x;
        scanf("%ld", &x);
        for (int j = 0; j < i; j++) {
                if (st[j].roll == x) {
                        printf("1. first name\n"
                                "2. last name\n"
                                "3. roll no.\n"
                                "4. CGPA\n"
                                "5. courses\n");
                        int z;
                        scanf("%d", &z);
                        switch (z) {
                        case 1:
                                printf("Enter the new "
                                        "first name : \n");
                                scanf("%s", st[j].fname);
                                break;
                        case 2:
                                printf("Enter the new "
                                        "last name : \n");
                                scanf("%s", st[j].lname);
                                break;
                        case 3:
                                printf("Enter the new "
                                        "roll number : \n");
                                scanf("%d", &st[j].roll);
                                break;
                        case 4:
                                printf("Enter the new CGPA : \n");
                                scanf("%f", &st[j].cgpa);
                                break;
                        case 5:
                                printf("Enter the new courses \n");
                                scanf(
                                        "%d%d%d%d%d", &st[j].cid[0],
                                        &st[j].cid[1], &st[j].cid[2],
                                        &st[j].cid[3], &st[j].cid[4]);
                                break;
                        }
                        printf("UPDATED SUCCESSFULLY.\n");
                }
        }
}

// Driver code
void main()

{
        int choice, count;
        while (i = 1) {
                printf("The Task that you "
                        "want to perform\n");
                printf("1. Add the Student Details\n");
                printf("2. Find the Student "
                        "Details by Roll Number\n");
                printf("3. Find the Student "
                        "Details by First Name\n");
                printf("4. Find the Student "
                        "Details by Course Id\n");
                printf("5. Find the Total number"
                        " of Students\n");
                printf("6. Delete the Students Details"
                        " by Roll Number\n");
                printf("7. Update the Students Details"
                        " by Roll Number\n");
                printf("8. To Exit\n");
                printf("Enter your choice to "
                        "find the task\n");
                scanf("%d", &choice);
                switch (choice) {
                case 1:
                        add_student();
                        break;
                case 2:
                        find_rl();
                        break;
                case 3:
                        find_fn();
                        break;
                case 4:
                        find_c();
                        break;
                case 5:
                        tot_s();
                        break;
                case 6:
                        del_s();
                        break;
                case 7:
                        up_s();
                        break;
                case 8:
                        exit(0);
                        break;
                }
        }
}
```

Output:

```
The Task that you want to perform
1. Add the Student Details
2. Find the Student Details by Roll Number
 3. Find the Student Details by First Name
4. Find the Student Details by Course Id
5. Find the Total number of Students
6. Delete the Students Details by Roll Number
7. Update the Students Details by Roll Number
8. TO Exit
Enter your choice to find the task
1
Add the Students Details
Enter the first name of student
Rahul
Enter the last name of student
Kumar
Enter the Roll Number 1
Enter the CGPA you obtained
8
Enter the course ID of each course
1 2 3 4 5

The Task that you want to perform
1. Add the Student Details
2. Find the Student Details by Roll Number
```

```
3. Find the Student Details by First Name
4. Find the Student Details by Course Id
5. Find the Total number of Students
6. Delete the Students Details by Roll Number
7. Update the Students Details by Roll Number
8. TO Exit
Enter your choice to find the task
2
Enter the Roll Number of the student
1
The Students Details are
The First name is Rahul
The Last name is Kumar
The CGPA is 8.000000
Enter the course ID of each course
The course ID are 1 2 3 4 5

The Task that you want to perform
1. Add the Student Details
2. Find the Student Details by Roll Number
3. Find the Student Details by First Name
4. Find the Student Details by Course Id
5. Find the Total number of Students
6. Delete the Students Details by Roll Number
7. Update the Students Details by Roll Number
8. TO Exit

Enter your choice to find the task
8
```