# Group - A

# Practical No. 1

☆ Title :: Implement Depth First Search algorithm and Breadth First Search algorithm. Use an undirected graph and develop a recursive algorithm for searching all vertices of a graph or tree data structure

☆ Date of Completion ::

☆ Objective :: To identify all nodes that are reachable from a given starting node.

☆ Problem Statement :: To use a technique to find shortest path in graph or tree.

☆ Software & Hardware Requirements :: C++

☆ Theory ::

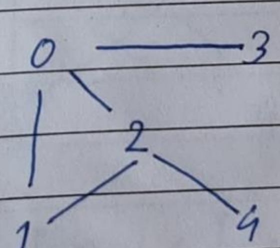
A) BFS :- It is one of traversing algorithm used in graphs. This algorithm is implemented using a queue data structures. In this algorithm, main focus is on vertices of graph. Select a starting node or vertex at first. Mark the starting node or vertex as visited & store it in queue. Then visit the vertices or nodes which are adjacent to the starting node. Mark them as visited and store these vertices or node in a queue. Repeat this process until all nodes or vertices are completely visited.

Advantages of BFS:

1) It can be useful in order to find whether the graph has connected components or not.

B) DFS :- It is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph. It works as follows:

1) Start by putting anyone of the graph's vertex on top of stack.
2) Take top item of stack & add it to visited list.
3) Create a list of vertices adjacent to that node. Add them, which aren't visited, to the top of stack.
4) Keep repeating steps 2 & 3 until the stack is empty.

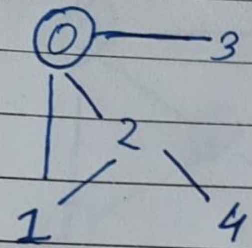



Visited

Stack

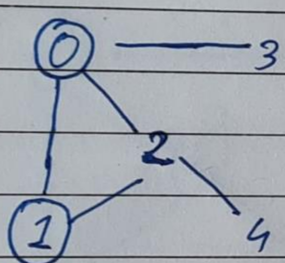
We start from vertex 0. The DFS algorithm starts by putting it in visited list and putting all its adjacent vertices in stack.

Teacher's Signature ____________



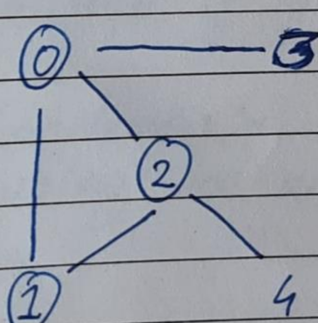

Visited : 0
Stack : 1 2 3

Next, we visit the element at top of stack i.e. 1. Since, 0 has already been visited, we visit 2 instead.




Visited : 0 1
stack : 2 3

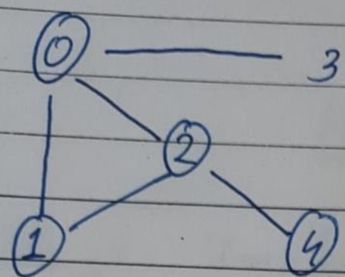
Visit the element at top of stack. Vertex 2 has been visited. Adjacent vertex is 4, so we add that to top of stack & visit it.




Visited : 0 1 2
Stack : 4 3

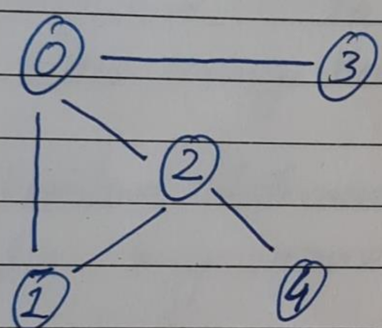
Vertex 2 has unvisite adjacent vertex i.e. 4. So we add that to top of stack and visit it.

Page No.
Date




Visited : 0 1 2 4
Stack : 3

After that we visit last element 3. It doesn't have any unvisited nodes, so we have completed the DFS of graph.




Visited : 0 1 2 4 3
Stack :

Applications of DFS:

1) For finding the path
2) To test if graph is bipartite.

★ Conclusion := We have implemented DFS and BFS using an undirected graph

Practical No. A1

Que :- Implement Depth First Search algorithm and Breadth First algorithm. Use an undirected graph and develop a recursive algorithm for searching all the vertices of the graph or tree data structure.

Solution :-

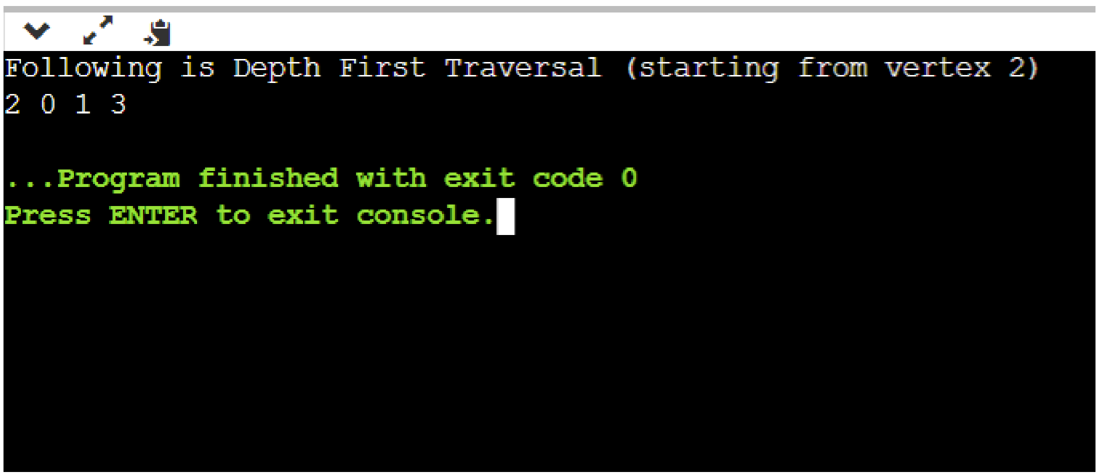
1. Depth First search :

Input =>

```
#include <iostream>
#include <map>
#include <list>

using namespace std;

class Graph {

	public:
		map<int, bool> visited;
		map<int, list<int> > adj;
		void addEdge(int v, int w);
		void DFS(int v);
};

void Graph::addEdge(int v, int w) {
	adj[v].push_back(w);
}

void Graph::DFS(int v) {
	visited[v] = true;
	cout << v << " ";
	list<int>::iterator i;
	for (i = adj[v].begin(); i != adj[v].end(); ++i)
		if (!visited[*i])
			DFS(*i);
}

int main() {
	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";
        g.DFS(2);
        return 0;
}
```

Output =>



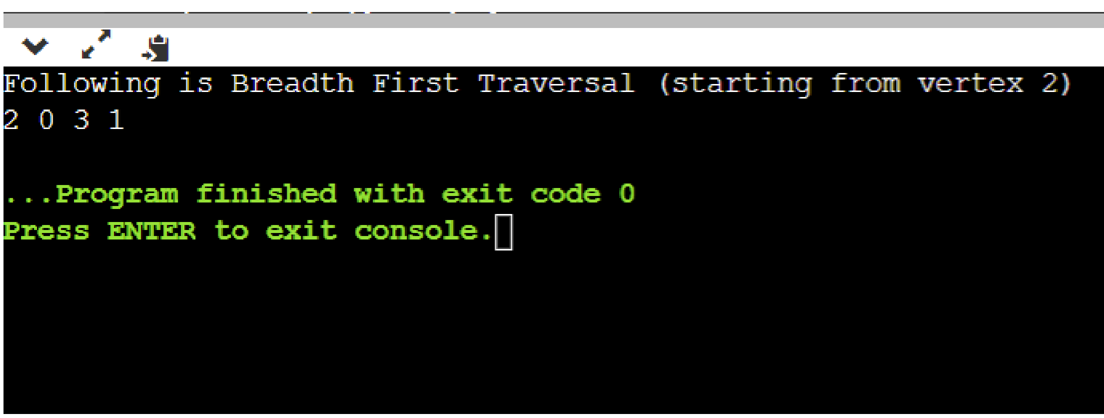
2. Breadth First Search :

Input =>

```
#include<iostream>
#include <list>
using namespace std;

class Graph {
    int V;
    list < int >*adj;

    public:
        Graph (int V);
        void addEdge (int v, int w);
        void BFS (int s);
};

Graph::Graph (int V) {
    this->V = V;
    adj = new list < int >[V];
}

void Graph::addEdge (int v, int w) {
    adj[v].push_back (w);
}


void Graph::BFS (int s) {
    bool * visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    list < int >queue;
    visited[s] = true;
    queue.push_back (s);
    list < int >::iterator i;
    while (!queue.empty ()) {
        s = queue.front ();
        cout << s << " ";
        queue.pop_front ();
        for (i = adj[s].begin (); i != adj[s].end (); ++i) {
            if (!visited[*i]) {
                visited[*i] = true;
                queue.push_back (*i);
            }
        }
    }
```

```
}

int main () {
    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 =>



# Group - A

## Practical No. 2

☆ Title :: Implement A star Algorithm for any game search problem.

☆ Date of Completion ::

☆ Objective :: To find shortest path through search space using Heuristic function.

☆ Problem Statement :: Implement A star Algorithm for any game search algorithm.

☆ Software and Hardware Requirements :: C++

☆ Theory :: A* search is the more commonly known as Best First Search. It uses heuristic function h(n), and cost to reach node n from start state g(n). It has combined features of UCS and greedy best first search, by which it solves problem efficiently. A* search algorithm finds the shortest path through search space using heuristic function. The search algorithm expands search tree and provides optimal result faster.

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

$f(n)$ → Estimated cost of cheapest solution

$g(n)$ → Cost to reach node n from start node

$h(n)$ → Cost to ~~search~~ reach from node n to ~~start~~ goal node

Algorithm :-

1) Place the starting node in OPEN list.
2) Check if OPEN list is empty or not, if the list is empty then return failure and Stop.
3) Select node from OPEN list which has smallest value of evaluation function (g+h). If node n is goal node Then return success and stop.
4) Otherwise, expand node n and generate all of its successors, and put n into closed list for each successor n, check whether n is already in OPEN or CLOSED list. If not, then compute evaluation function for n and place into OPEN list.
5) Else if, node n is already in OPEN and CLOSED then it should be attached to back pointer which rejects lowest g(n') value.
6) Return

Advantages:-

1) A* search algorithm is better algorithm than other search algorithm.
2) It is optimal & complete.

Disadvantages :-

1) It does not always produce shortest part as it mostly based on heuristic and approximation.
2) A* search algorithm has some complexity issues.

* Conclusion := We have implemented A* search algorithm for game search problem.

Practical No. A2

Que :- Implement A star Algorithm for any game search problem.

Solution :-

Input =>

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

#define ROW 9
#define COL 10

typedef pair < int, int > Pair;
typedef pair < double, pair < int, int >> pPair;

struct cell {
   int parent_i, parent_j;
   double f, g, h;
};

bool isValid (int row, int col) {
   return (row >= 0) && (row < ROW) && (col >= 0) && (col < COL);
}

bool isUnBlocked (int grid[][COL], int row, int col) {
   if (grid[row][col] == 1)
      return (true);
   else
      return (false);
}


bool isDestination (int row, int col, Pair dest) {
   if (row == dest.first && col == dest.second)
      return (true);
   else
      return (false);
}


double calculateHValue (int row, int col, Pair dest) {
```

```
    return ((double) sqrt ((row - dest.first) * (row - dest.first)+(col - dest.second) * (col -
dest.second)));
}

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;
}


void aStarSearch (int grid[][COL], Pair src, Pair dest) {
    if (isValid (src.first, src.second) == false) {
        printf ("Source is invalid\n");
        return;
    }
    if (isValid (dest.first, dest.second) == false) {
        printf ("Destination is invalid\n");
        return;
    }
    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 (isDestination (src.first, src.second, dest) == true) {
        printf ("We are already at the destination\n");
```

```
    return;
}
bool closedList[ROW][COL];
memset (closedList, false, sizeof (closedList));
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;
    }
}
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;
set < pPair > openList;
openList.insert (make_pair (0.0, make_pair (i, j)));
bool foundDest = false;
while (!openList.empty ()) {
    pPair p = *openList.begin ();
    openList.erase (openList.begin ());
    i = p.second.first;
    j = p.second.second;
    closedList[i][j] = true;
    double gNew, hNew, fNew;
    if (isValid (i - 1, j) == true) {
        if (isDestination (i - 1, j, dest) == true) {
            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;
        }
            else if (closedList[i - 1][j] == false && isUnBlocked (grid, i - 1, j) == true) {
            gNew = cellDetails[i][j].g + 1.0;
            hNew = calculateHValue (i - 1, j, dest);
```

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

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

```
        if (isValid(i - 1, j + 1) == true)
        {
           if (isDestination(i - 1, j + 1, dest) == true)
           {
                    cellDetails[i - 1][j + 1].parent_i = i;
                    cellDetails[i - 1][j + 1].parent_j = j;
                    printf("The destination cell is found\n");
                    tracePath(cellDetails, dest);
                    foundDest = true;
                    return;
           }
           else if (closedList[i - 1][j + 1] == false && isUnBlocked(grid, i - 1, j + 1) == true)
           {
                    gNew = cellDetails[i][j].g + 1.414;
                    hNew = calculateHValue(i - 1, j + 1, dest);
                    fNew = gNew + hNew;
                    if (cellDetails[i - 1][j + 1].f == FLT_MAX || cellDetails[i - 1][j + 1].f >
fNew)
                    {
                             openList.insert(make_pair(fNew, make_pair(i - 1, j + 1)));
                             cellDetails[i - 1][j + 1].f = fNew;
                             cellDetails[i - 1][j + 1].g = gNew;
                             cellDetails[i - 1][j + 1].h = hNew;
                             cellDetails[i - 1][j + 1].parent_i = i;
                             cellDetails[i - 1][j + 1].parent_j = j;
                    }
           }
        }
        if (isValid(i - 1, j - 1) == true)
        {
           if (isDestination(i - 1, j - 1, dest) == true)
           {
                    cellDetails[i - 1][j - 1].parent_i = i;
                    cellDetails[i - 1][j - 1].parent_j = j;
                    printf("The destination cell is found\n");
                    tracePath(cellDetails, dest);
                    foundDest = true;
                    return;
           }
           else if (closedList[i - 1][j - 1] == false && isUnBlocked(grid, i - 1, j - 1) == true)
           {
                    gNew = cellDetails[i][j].g + 1.414;
                    hNew = calculateHValue(i - 1, j - 1, dest);
```

```
                fNew = gNew + hNew;
                if (cellDetails[i - 1][j - 1].f == FLT_MAX || cellDetails[i - 1][j - 1].f >
fNew)
                {
                        openList.insert(make_pair(fNew, make_pair(i - 1, j - 1)));
                        cellDetails[i - 1][j - 1].f = fNew;
                        cellDetails[i - 1][j - 1].g = gNew;
                        cellDetails[i - 1][j - 1].h = hNew;
                        cellDetails[i - 1][j - 1].parent_i = i;
                        cellDetails[i - 1][j - 1].parent_j = j;
                }
           }
      }
      if (isValid(i + 1, j + 1) == true)
      {
         if (isDestination(i + 1, j + 1, dest) == true)
         {
                cellDetails[i + 1][j + 1].parent_i = i;
                cellDetails[i + 1][j + 1].parent_j = j;
                printf("The destination cell is found\n");
                tracePath(cellDetails, dest);
                foundDest = true;
                return;
         }
         else if (closedList[i + 1][j + 1] == false && isUnBlocked(grid, i + 1, j + 1) == true)
         {
                gNew = cellDetails[i][j].g + 1.414;
                hNew = calculateHValue(i + 1, j + 1, dest);
                fNew = gNew + hNew;
                if (cellDetails[i + 1][j + 1].f == FLT_MAX || cellDetails[i + 1][j + 1].f >
fNew)
                {
                        openList.insert(make_pair(fNew, make_pair(i + 1, j + 1)));
                        cellDetails[i + 1][j + 1].f = fNew;
                        cellDetails[i + 1][j + 1].g = gNew;
                        cellDetails[i + 1][j + 1].h = hNew;
                        cellDetails[i + 1][j + 1].parent_i = i;
                        cellDetails[i + 1][j + 1].parent_j = j;
                }
         }
      }
      if (isValid(i + 1, j - 1) == true)
      {
```

```
            if (isDestination(i + 1, j - 1, dest) == true)
            {
                cellDetails[i + 1][j - 1].parent_i = i;
                cellDetails[i + 1][j - 1].parent_j = j;
                printf("The destination cell is found\n");
                tracePath(cellDetails, dest);
                foundDest = true;
                return;
            }
            else if (closedList[i + 1][j - 1] == false && isUnBlocked(grid, i + 1, j - 1) == true)
            {
                gNew = cellDetails[i][j].g + 1.414;

                hNew = calculateHValue(i + 1, j - 1, dest);
                fNew = gNew + hNew;
                if (cellDetails[i + 1][j - 1].f == FLT_MAX || cellDetails[i + 1][j - 1].f >
fNew)
                {
                    openList.insert(make_pair(fNew, make_pair(i + 1, j - 1)));
                    cellDetails[i + 1][j - 1].f = fNew;
                    cellDetails[i + 1][j - 1].g = gNew;
                    cellDetails[i + 1][j - 1].h = hNew;
                    cellDetails[i + 1][j - 1].parent_i = i;
                    cellDetails[i + 1][j - 1].parent_j = j;
                }
            }
        }
    }
    if (foundDest == false)
        printf("Failed to find the Destination Cell\n");
    return;
}

int main()
{
    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 }
    };
    Pair src = make_pair(8, 0);
    Pair dest = make_pair(0, 0);
    aStarSearch(grid, src, dest);
    return (0);
}
```

Output =>

```
input
The destination cell is found

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

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

Page No.
Date

## Group - A

## Practical No. 3

★ Title :- Greedy Search Algorithm

★ Date of Completion :-

★ Objective :- To find the overall optimal way to solve the entire problem.

★ Problem Statement :- Implement greedy search algorithm for any of the following application:-
1) Selection sort
2) Minimum Spanning Tree
3) Single Source Shortest Path Problem.
4) Job Scheduling Problem.
5) Prim's Minimum Spanning Tree Algorithm
6) Krushkal's Minimum Spanning Tree Algorithm
7) Djikstra's Minimum Spanning Tree Algorithm

★ Software and Hardware Requirements :- C++.

★ Theory :-

A) Minimum Spanning Tree :- A spanning tree is a subgraph of undirected connected graph where it includes all nodes of graph with minimum possible no. of edges. The subgraph should contain each & every

Teacher's Signature________

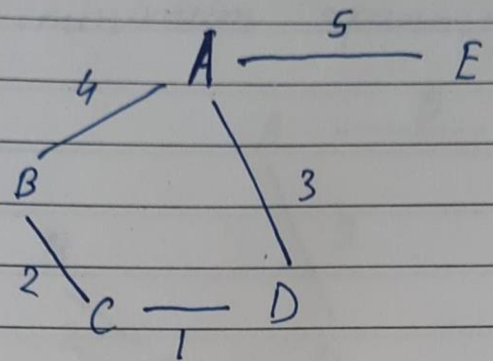
node of original graph. Spanning tree does not contain cycle. If graph has n number of nodes, total no. of spanning tree created from complete graph is $n^{n-2}$. The spanning tree in which the sum of edges is minimum as possible then it is called minimum spanning tree. There are 2 different ways to find out minimum spanning tree from complete graph:

1) Krushkal's algorithm
2) Prim's algorithm.

B) Prim's Algorithm :- It is minimum spanning tree algorithm which helps us to find out edges of graph to form tree including every node with sum of edges as minimum as possible, to form minimum spanning tree.
Prim's Algorithm basically follows the greedy algorithm approach to find optimal solution.
To find minimum spanning tree using Prim's algorithm, we will choose a source node keep adding the edges with lowest weight.

c) Algorithm :-
1) Initialise algorithm by choosing source vertex.
2) Find minimum weight edge connected to source node and another node and add it to tree.
3) Keep repeating this process until we find the minimum spanning tree.

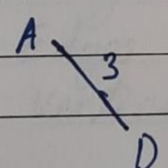
D) Example :-



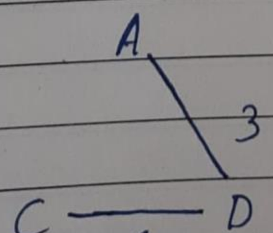

Let us consider, source vertex to initiate algorithm

A

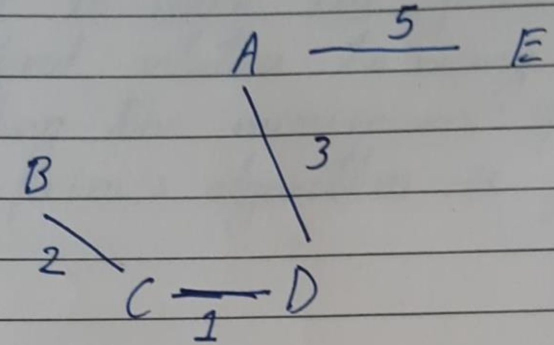
Now, we will choose shortest weight from source vertex and add it to spanning tree.




Then choose next nearest node connected with minimum edge & add it to solution. If there are multiple then choose anyone



Continue the steps until all node are included & find minimum spanning tree.




Weight of MST = $2+1+3+5 = 11$

E) Time Complexity :- The running time for Prim's algorithm is $O(V \log V + E \log V)$ which is equal to $O(E \log V)$ because every insertion of node in solution is taking logarithmic time.

$E \Rightarrow$ No. of Edges

$V \Rightarrow$ No. of Vertices.

F) Applications :=

1) It is used in network design
2) It is used in network cycle and rail tracks connecting all the cities.
3) It is used in laying cables of electrical wiring.

Page No.
Date

★ Conclusion :- As we studied, minimum spanning tree has its own importance in real world. It is importaint to learn the prim's algorithm which leads us to find solution to many problems. When it comes to finding the minimum spanning tree for dense graph, prim's algorithm is first choice.

Practical No. A3

Que :- Implement Greedy search algorithm for any of the following application:
Prim's Minimal Spanning Tree Algorithm

Solution :-

Input =>

```
#include <bits/stdc++.h>

using namespace std;

#define V 5

int minKey(int key[], bool mstSet[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (mstSet[v] == false && key[v] < min)
         min = key[v], min_index = v;
   return min_index;
}

void printMST(int parent[], int graph[V][V]) {
   cout<<"Edge \tWeight\n";
   for (int i = 1; i < V; i++)
      cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
}

void primMST(int graph[V][V]) {
   int parent[V];
   int key[V];
   bool mstSet[V];
   for (int i = 0; i < V; i++)
      key[i] = INT_MAX, mstSet[i] = false;
   key[0] = 0;
   parent[0] = -1;
   for (int count = 0; count < V - 1; count++){
      int u = minKey(key, mstSet);
      mstSet[u] = true;
      for (int v = 0; v < V; v++)
```

```
        if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
        parent[v] = u, key[v] = graph[u][v];
    }
    printMST(parent, graph);
}

int main() {
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };
    primMST(graph);
    return 0;
}
```

Output =>

```
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5


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