Depth First Search Source Code:

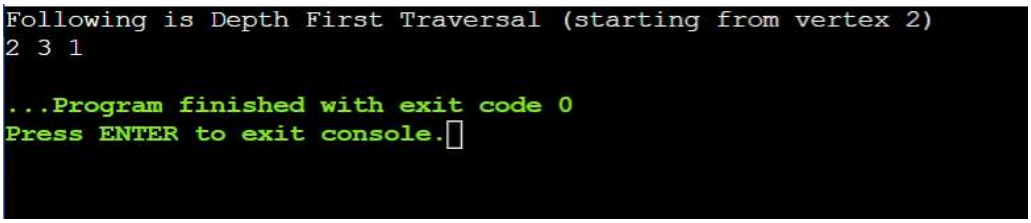
```
#include <bits/stdc++.h>
using namespace std;

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, 3);
g.addEdge(2, 1);

cout << "Following is Depth First Traversal"
" (starting from vertex 2) \n";
g.DFS(2);
return 0;
}
```

Output:



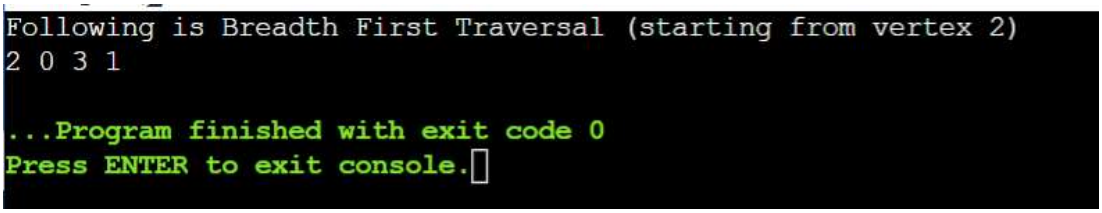
Breadth First Search Source Code:

```
#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:



A* Algorithm Source Code:

```
#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;
}
```

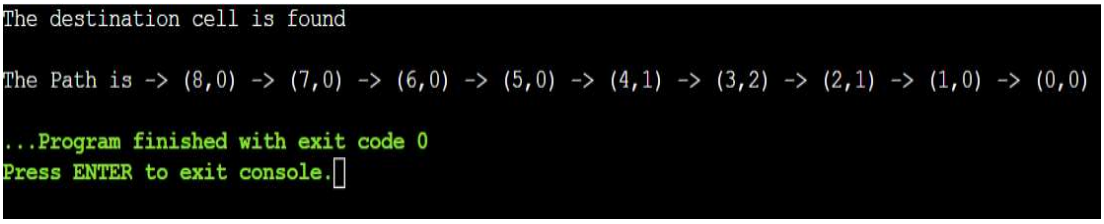
```
}
}
}
if (foundDest == false)
printf("Failed to find the Destination Cell\n");

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:


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.

Greedy Search Algorithm Source Code:

```
#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: