Find the minimum number of moves needed to move from one cell of matrix to another

Given a N X N matrix (M) filled with 1 , 0 , 2 , 3 . Find the minimum numbers of moves needed to move from source to destination (sink) . while traversing through blank cells only. You can traverse up, down, right and left.
A value of cell **1** means Source.
A value of cell **2** means Destination.
A value of cell **3** means Blank cell.
A value of cell **0** means Blank Wall.
Note : there is only single source and single destination.they may be more than one path from source to destination(sink).each move in matrix we consider as '1'

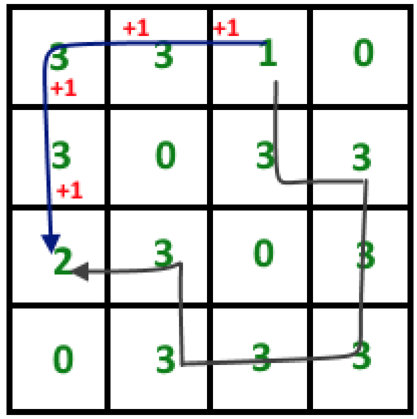
Examples:

```
Input : M[3][3] = {{ 0 , 3 , 2 },
                   { 3 , 3 , 0 },
                   { 1 , 3 , 0 }};
Output : 4

Input : M[4][4] = {{ 3 , 3 , 1 , 0 },
                   { 3 , 0 , 3 , 3 },
                   { 2 , 3 , 0 , 3 },
                   { 0 , 3 , 3 , 3 }};
Output : 4
```

Asked in: Adobe Interview

.



minimum 4 moves are required to reach sink

We strongly recommend that you click here and code it yourself first, before moving on to the solution.

The idea is to use Level graph (Breadth First Traversal). Consider each cell as a node and each boundary between any two adjacent cells be an edge . so total number of Node is N*N.

```
1. Create an empty Graph having N*N node ( Vertex ).
2. Push all node into graph.
3. Note down source and sink vertices.
4. Now Apply  level graph concept ( that we achieve using BFS  ) .
   In which we find level of every node from source vertex.
   After that we return  'Level[d]' ( d is destination ).
   (which is the minimum move from source to sink )
```

Below is C++ implementation of above idea.

```
// C++ program to find the minimum numbers
// of moves needed to move from source to
// destination .
#include<bits/stdc++.h>
using namespace std;
#define N 4

class Graph
{
    int V ;
    list < int > *adj;
public :
    Graph( int V )
    {
        this->V = V ;
        adj = new list<int>[V];
    }
    void addEdge( int s , int d ) ;
    int BFS ( int s , int d) ;
};

// add edge to graph
void Graph :: addEdge ( int s , int d )
{
```

```
    adj[s].push_back(d);
    adj[d].push_back(s);
}

// Level  BFS function to find minimum path
// from source to sink
int Graph :: BFS(int s, int d)
{
    // Base case
    if (s == d)
        return 0;

    // make initial distance of all vertex -1
    // from source
    int *level = new int[V];
    for (int i = 0; i < V; i++)
        level[i] = -1  ;

    // Create a queue for BFS
    list<int> queue;

    // Mark the source node level[s] = '0'
    level[s] = 0 ;
    queue.push_back(s);

    // it will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while (!queue.empty())
    {
        // Dequeue a vertex from queue
        s = queue.front();
        queue.pop_front();

        // Get all adjacent vertices of the
        // dequeued vertex s. If a adjacent has
        // not been visited ( level[i] < '0') ,
        // then update level[i] == parent_level[s] + 1
        // and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            // Else, continue to do BFS
            if (level[*i] < 0 || level[*i] > level[s] + 1 )
            {
                level[*i] = level[s] + 1 ;
                queue.push_back(*i);
            }
        }

    }

    // return minimum moves from source to sink
    return level[d] ;
}

bool isSafe(int i, int j, int M[][N])
{
    if ((i < 0 || i >= N) ||
            (j < 0 || j >= N ) || M[i][j] == 0)
        return false;
    return true;
}

// Returns minimum numbers of  moves  from a source (a
// cell with value 1) to a destination (a cell with
// value 2)
int MinimumPath(int M[][N])
{
    int s , d ; // source and destination
```

```
    int V = N*N+2;
    Graph g(V);

    // create graph with n*n node
    // each cell consider as node
    int k = 1 ; // Number of current vertex
    for (int i =0 ; i < N ; i++)
    {
        for (int j = 0 ; j < N; j++)
        {
            if (M[i][j] != 0)
            {
                // connect all 4 adjacent cell to
                // current cell
                if ( isSafe ( i , j+1 , M ) )
                    g.addEdge ( k , k+1 );
                if ( isSafe ( i , j-1 , M ) )
                    g.addEdge ( k , k-1 );
                if (j< N-1 && isSafe ( i+1 , j , M ) )
                    g.addEdge ( k , k+N );
                if ( i > 0 && isSafe ( i-1 , j , M ) )
                    g.addEdge ( k , k-N );
            }

            // source index
            if( M[i][j] == 1 )
                s = k ;

            // destination index
            if (M[i][j] == 2)
                d = k;
            k++;
        }
    }

    // find minimum moves
    return g.BFS (s, d) ;
}

// driver program to check above function
int main()
{
    int M[N][N] = {{ 3 , 3 , 1 , 0 },
        { 3 , 0 , 3 , 3 },
        { 2 , 3 , 0 , 3 },
        { 0 , 3 , 3 , 3 }
    };

    cout << MinimumPath(M) << endl;

    return 0;
}
```

Run on IDE

Output:

```
4
```

This article is contributed by **Nishant Singh** . If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner **Company Wise Coding Practice**

Graph BFS shortest path

(Login to Rate and Mark)

0 Average Difficulty : **0/5.0**
No votes yet.

Add to TODO List

Mark as DONE

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Load Comments Share this post!