LeetCode Training Day 7 Graph Travel

For graph, we have many topics, today we discuss the simples one, use DFS or BFS to traverse the graph.

**Data structure for Graph**

First, let’s discuss how a graph can be represented. A graph can be specified in two forms. One is called grid. It is a 2D matrix, each cell with a special mark such as ‘0’, ‘1’, space, ‘X’ or other sign which indicate a special state for that cell. Another is edge, which is given in the form of (u, v, d), which indicates a path between nodes u and v with distance d.

If the graph is in grid, we normally put a cursor at a position (r,c),as the current position, and we construct four directions {{-1, 0}, {1, 0}, {0,-1}, {0, 1}} to tell the where is the next step to go. When we move the next\_pos to the with the directions to 4 new position, first we need to check if this is not out of boundary, so if (r < 0 || r >= row\_size || c < 0 || c >= col\_size) we should skip that position. Second, we need to check if the cell we already visited, to do so we clone a visited matrix the same size as grid, set it all zero first, when we visit the cell, we set it as 1. By checking visited matrix, we can avoid visiting duplicate position again, otherwise we will enter an endless loop.

If the graph is in edge, we need to first convert to neighbors, which is on each node (either as index in array or key in hash table), we add a list to put all its neighbors. Such form will allow us to reach the next node from current node efficiently. When we process the neighbors, we also need a array as visited to check if the node is already visited or not.

**DFS or BFS**

For DFS traverse, we use recursive function call, as long as there is a neighbor not visited, we enter one level deeper, if all neigbors are visited, we terminate the recursive call.

For BFS, we use a queue, put the neighbors from the current nodes in the queue as the next batch to process. If we need to track the distance of the nodes, then every time we process only the current size in the queue as the batch size, regardless the queue may still grow during the process.

Now the question is when to use DFS and when to use BFS. Generally, if tha path has a loop, we must use DFS, because BFS can not detect loop. But when we need to track the distance from the origin place when we start, we should use BFS to keep the shortest distance.

## 733. Flood Fill

Easy

An image is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, "flood fill" the image.

To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the newColor.

At the end, return the modified image.

**Example 1:**

**Input:**

image = [[1,1,1],[1,1,0],[1,0,1]]

sr = 1, sc = 1, newColor = 2

**Output:** [[2,2,2],[2,2,0],[2,0,1]]

**Explanation:**

From the center of the image (with position (sr, sc) = (1, 1)), all pixels connected

by a path of the same color as the starting pixel are colored with the new color.

Note the bottom corner is not colored 2, because it is not 4-directionally connected

to the starting pixel.

**Note:**

 The length of image and image[0] will be in the range [1, 50].

 The given starting pixel will satisfy 0 <= sr < image.length and 0 <= sc < image[0].length.

 The value of each color in image[i][j] and newColor will be an integer in [0, 65535].

### Analysis:

The flood fill is that we pick up a pixel with old color and do BFS with the same old color and make it as new color, until the end. The visited vector will allow us to remember which pixel we have visited, which are not.

Please notice in this problem we can not depend on the old color to determine if we have visited this position, because the old color and new color can be same, which will lead us endless loop. Please watch the techniques we used for 4 direction search and check when it is out of boundary.

For this problem we can use either DFS or BFS

/// <summary>

/// Leet code #733. Flood Fill

///

/// An image is represented by a 2-D array of integers, each integer

/// representing the pixel value of the image (from 0 to 65535).

///

/// Given a coordinate (sr, sc) representing the starting pixel

/// (row and column) of the flood fill, and a pixel value newColor,

/// "flood fill" the image.

///

/// To perform a "flood fill", consider the starting pixel, plus any

/// pixels connected 4-directionally to the starting pixel of the same

/// color as the starting pixel, plus any pixels connected 4-directionally

/// to those pixels (also with the same color as the starting pixel), and

/// so on. Replace the color of all of the aforementioned pixels with the

/// newColor.

///

/// At the end, return the modified image.

///

/// Example 1:

/// Input:

/// image = [[1,1,1],[1,1,0],[1,0,1]]

/// sr = 1, sc = 1, newColor = 2

/// Output: [[2,2,2],[2,2,0],[2,0,1]]

/// Explanation:

/// From the center of the image (with position (sr, sc) = (1, 1)), all

/// pixels connected by a path of the same color as the starting pixel are

/// colored with the new color.

/// Note the bottom corner is not colored 2, because it is not

/// 4-directionally connected to the starting pixel.

/// Note:

///

/// The length of image and image[0] will be in the range [1, 50].

/// The given starting pixel will satisfy 0 <= sr < image.length and

/// 0 <= sc < image[0].length.

/// The value of each color in image[i][j] and newColor will be an

/// integer in [0, 65535].

/// </summary>

vector<vector<int>> LeetCode::floodFill(vector<vector<int>>& image,

int sr, int sc, int newColor)

{

queue<pair<int, int>> search\_queue;

vector<vector<int>> result = image;

vector<vector<int>> visited(image.size(), vector<int>(image[0].size()));

vector<vector<int>> directions = { { -1, 0 },{ 1, 0 },{ 0, -1 },{ 0, 1 } };

int oldColor = image[sr][sc];

visited[sr][sc] = 1;

search\_queue.push(make\_pair(sr, sc));

while (!search\_queue.empty())

{

pair<int, int> pos = search\_queue.front();

search\_queue.pop();

result[pos.first][pos.second] = newColor;

for (size\_t i = 0; i < directions.size(); i++)

{

pair<int, int> new\_pos =

make\_pair(pos.first + directions[i][0],

pos.second + directions[i][1]);

// if new position out of boundary skip it

if ((new\_pos.first < 0) || (new\_pos.second < 0) ||

(new\_pos.first >= (int)image.size()) ||

(new\_pos.second >= (int)image[0].size()))

{

continue;

}

// if color not match skip

if (result[new\_pos.first][new\_pos.second] != oldColor)

{

continue;

}

// check visited flag, if already set skip

if (visited[new\_pos.first][new\_pos.second] == 1)

{

continue;

}

visited[new\_pos.first][new\_pos.second] = 1;

search\_queue.push(new\_pos);

}

}

return result;

}

## 200. Number of Islands

Medium

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

**Example 1:**

**Input:**

11110

11010

11000

00000

**Output:** 1

**Example 2:**

**Input:**

11000

11000

00100

00011

**Output:** 3

### Analysis:

We can do it in a typical floodfill, first we can copy the land map, and starting from any cell as land, search until no more land, we can reset the land as water during the search. After we process an area of lands (this means a DFS or BFS process finish), we add the count by 1. Please watch the techniques we used for 4 direction search and check when it is out of boundary.

/// <summary>

/// Leet code #200. Number of Islands

/// Given a 2d grid map of '1's (land) and '0's (water), count the number of

/// islands.

/// An island is surrounded by water and is formed by connecting adjacent lands

/// horizontally or vertically. You may assume all four edges of the grid are

/// all surrounded by water.

/// Example 1:

/// 11110

/// 11010

/// 11000

/// 00000

/// Answer : 1

/// Example 2 :

/// 11000

/// 11000

/// 00100

/// 00011

/// Answer : 3

/// </summary>

int LeetCode::numIslands(vector<vector<char>>& grid)

{

size\_t result = 0;

// search from top to down, then from left to right

for (size\_t i = 0; i < grid.size(); i++)

{

for (size\_t j = 0; j < grid[i].size(); j++)

{

// if this is land, floodfill it

if (grid[i][j] == '1')

{

result++;

grid[i][j] = '0';

queue<pair<int, int>> search;

search.push({ i, j });

vector<vector<int>> directions =

{ { -1, 0 },{ 1, 0 },{ 0, -1 },{ 0, 1 } };

while (!search.empty())

{

pair<int, int> pos = search.front();

search.pop();

for (size\_t k = 0; k < 4; k++)

{

pair<int, int> next = pos;

next.first += directions[k][0];

next.second += directions[k][1];

if (next.first < 0 || next.first >= (int)grid.size() ||

next.second < 0 || next.second >= (int)grid[0].size())

{

continue;

}

if (grid[next.first][next.second] != '1')

{

continue;

}

grid[next.first][next.second] = '0';

search.push(next);

}

}

}

}

}

return result;

}

## 332. Reconstruct Itinerary

Medium

Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order. All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.

**Note:**

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets form at least one valid itinerary.

**Example 1:**

**Input:** [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]

**Output:** ["JFK", "MUC", "LHR", "SFO", "SJC"]

**Example 2:**

**Input:** [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]

**Output:** ["JFK","ATL","JFK","SFO","ATL","SFO"]

**Explanation:** Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"].

But it is larger in lexical order.

### Analysis:

This problem should also be done in DFS, not BFS. There are some hints here:

* The whole iteration is a single path. (BFS can lead to parallel path)
* The starting point is unknown.
* The final answer should bre returned in lexicon order, which means you have to find all the answers possible or always search in lexicon order.

/// <summary>

/// Leet code #332. Reconstruct Itinerary

/// </summary>

void LeetCode::findItinerary(vector<string> &path, unordered\_map<string, map<string, int>>& tickets,

unordered\_map<string, map<string, int>>& visited, int &count)

{

if (count == 0) return;

string city = path.back();

for (map<string, int>::iterator itr = tickets[city].begin();

itr != tickets[city].end(); itr++)

{

// The visited map is to count how many times, the city is visited

// which should not go beyond the count from the input

if (visited[city][itr->first] == itr->second) continue;

visited[city][itr->first]++;

path.push\_back(itr->first);

count--;

findItinerary(path, tickets, visited, count);

if (count != 0)

{

count++;

visited[city][itr->first]--;

path.pop\_back();

}

else

{

break;

}

}

return;

}

/// <summary>

/// Leet code #332. Reconstruct Itinerary

///

/// Given a list of airline tickets represented by pairs of departure and arrival

/// airports [from, to], reconstruct the itinerary in order. All of the tickets

/// belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.

///

/// Note:

/// 1.If there are multiple valid itineraries, you should return the itinerary

/// that has the smallest lexical order when read as a single string. For example,

/// the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].

/// 2.All airports are represented by three capital letters (IATA code).

/// 3.You may assume all tickets form at least one valid itinerary.

///

/// Example 1:

/// tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]

/// Return ["JFK", "MUC", "LHR", "SFO", "SJC"].

///

/// Example 2:

/// tickets = [["JFK","SFO"], ["JFK","ATL"], ["SFO","ATL"], ["ATL","JFK"], ["ATL","SFO"]]

/// Return ["JFK","ATL","JFK","SFO","ATL","SFO"].

/// Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"].

/// But it is larger in lexical order.

/// </summary>

vector<string> LeetCode::findItinerary(vector<pair<string, string>> tickets)

{

vector<string> result = { "JFK" };

unordered\_map<string, map<string, int>> route\_map, visited;

int count = 0;

for (size\_t i = 0; i < tickets.size(); i++)

{

route\_map[tickets[i].first][tickets[i].second]++;

count++;

}

findItinerary(result, route\_map, visited, count);

return result;

}

## 934. Shortest Bridge

Medium

In a given 2D binary array A, there are two islands.  (An island is a 4-directionally connected group of 1s not connected to any other 1s.)

Now, we may change 0s to 1s so as to connect the two islands together to form 1 island.

Return the smallest number of 0s that must be flipped.  (It is guaranteed that the answer is at least 1.)

**Example 1:**

**Input:** [[0,1],[1,0]]

**Output:** 1

**Example 2:**

**Input:** [[0,1,0],[0,0,0],[0,0,1]]

**Output:** 2

**Example 3:**

**Input:** [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]

**Output:** 1

**Note:**

1. 1 <= A.length = A[0].length <= 100
2. A[i][j] == 0 or A[i][j] == 1

### Analysis:

We can use either DFS or BFS to color one land from 1 to 2 and then build the bridge using BFS from land 2 to land 1. The path color is 2.

/// <summary>

/// Leet code #934. Shortest Bridge

///

/// In a given 2D binary array A, there are two islands. (An island is a

/// 4-directionally connected group of 1s not connected to any other 1s.)

///

/// Now, we may change 0s to 1s so as to connect the two islands together

/// to form 1 island.

///

/// Return the smallest number of 0s that must be flipped. (It is guaranteed

/// that the answer is at least 1.)

///

/// Example 1:

///

/// Input: [[0,1],[1,0]]

/// Output: 1

/// Example 2:

///

/// Input: [[0,1,0],[0,0,0],[0,0,1]]

/// Output: 2

///

/// Example 3:

///

/// Input: [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]

/// Output: 1

///

///

/// Note:

///

/// 1 <= A.length = A[0].length <= 100

/// A[i][j] == 0 or A[i][j] == 1

/// </summary>

int LeetCode::shortestBridge(vector<vector<int>>& A)

{

queue<vector<int>> first\_land;

bool hit = false;

for (size\_t i = 0; i < A.size(); i++)

{

for (size\_t j = 0; j < A[i].size(); j++)

{

if (A[i][j] == 1)

{

shortestBridgeFill(A, i, j);

hit = true;

break;

}

}

if (hit) break;

}

return shortestBridgePath(A);

}

/// <summary>

/// Leet code #934. Shortest Bridge

/// </summary>

int LeetCode::shortestBridgePath(vector<vector<int>>& A)

{

queue<pair<int, int>> search;

for (size\_t i = 0; i < A.size(); i++)

{

for (size\_t j = 0; j < A[i].size(); j++)

{

if (A[i][j] == 2)

{

search.push(make\_pair(i, j));

}

}

}

int result = 0;

vector<vector<int>> direction = { { -1, 0 },{ 1, 0 },{ 0, -1 },{ 0, 1 } };

while (!search.empty())

{

size\_t size = search.size();

for (size\_t i = 0; i < size; i++)

{

pair<int, int> pos = search.front();

search.pop();

for (size\_t i = 0; i < direction.size(); i++)

{

pair<int, int> next = make\_pair(pos.first + direction[i][0], pos.second + direction[i][1]);

if (next.first < 0 || next.first >= (int)A.size() ||

next.second < 0 || next.second >= (int)A[0].size())

{

continue;

}

if (A[next.first][next.second] == 1)

{

return result;

}

else if (A[next.first][next.second] == 0)

{

A[next.first][next.second] = 2;

search.push(next);

}

}

}

result++;

}

return result;

}

/// <summary>

/// Leet code #934. Shortest Bridge

/// </summary>

void LeetCode::shortestBridgeFill(vector<vector<int>>& A, int row, int col)

{

queue<pair<int, int>> search;

search.push({ row, col });

A[row][col] = 2;

vector<vector<int>> direction = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

while (!search.empty())

{

pair<int, int> pos = search.front();

search.pop();

for (size\_t i = 0; i < direction.size(); i++)

{

pair<int, int> next = make\_pair(pos.first + direction[i][0], pos.second + direction[i][1]);

if (next.first < 0 || next.first >= (int)A.size() ||

next.second < 0 || next.second >= (int)A[row].size())

{

continue;

}

if (A[next.first][next.second] == 1)

{

A[next.first][next.second] = 2;

search.push(next);

}

}

}

}

## 1162. As Far from Land as Possible

Medium

Given an N x N grid containing only values 0 and 1, where 0 represents water and 1 represents land, find a water cell such that its distance to the nearest land cell is maximized and return the distance.

The distance used in this problem is the *Manhattan distance*: the distance between two cells (x0, y0) and (x1, y1) is |x0 - x1| + |y0 - y1|.

If no land or water exists in the grid, return -1.

**Example 1:**

**![A picture containing furniture, wardrobe, cabinet

Description automatically generated]()**

**Input:** [[1,0,1],[0,0,0],[1,0,1]]

**Output:** 2

**Explanation:**

The cell (1, 1) is as far as possible from all the land with distance 2.

**Example 2:**

**![Table

Description automatically generated]()**

**Input:** [[1,0,0],[0,0,0],[0,0,0]]

**Output:** 4

**Explanation:**

The cell (2, 2) is as far as possible from all the land with distance 4.

**Note:**

1. 1 <= grid.length == grid[0].length <= 100
2. grid[i][j] is 0 or 1

### Analysis:

For this problem you can do DP as well, but we will do it in a classic way, BFS.

/// <summary>

/// Leet code #1162. As Far from Land as Possible

///

/// Given an N x N grid containing only values 0 and 1, where 0 represents

/// water and 1 represents land, find a water cell such that its distance

/// to the nearest land cell is maximized and return the distance.

/// The distance used in this problem is the Manhattan distance: the distance

/// between two cells (x0, y0) and (x1, y1) is |x0 - x1| + |y0 - y1|.

/// If no land or water exists in the grid, return -1.

///

/// Example 1:

///

/// Input: [[1,0,1],[0,0,0],[1,0,1]]

/// Output: 2

/// Explanation:

/// The cell (1, 1) is as far as possible from all the land with distance 2.

///

/// Example 2:

///

/// Input: [[1,0,0],[0,0,0],[0,0,0]]

/// Output: 4

/// Explanation:

/// The cell (2, 2) is as far as possible from all the land with distance 4.

///

/// Note:

/// 1. 1 <= grid.length == grid[0].length <= 100

/// 2. grid[i][j] is 0 or 1

/// </summary>

int LeetCode::maxLandDistance(vector<vector<int>>& grid)

{

int distance = 0;

int result = -1;

queue<pair<int, int>> search;

for (size\_t i = 0; i < grid.size(); i++)

{

for (size\_t j = 0; j < grid[i].size(); j++)

{

if (grid[i][j] == 1) search.push(make\_pair(i, j));

}

}

while (!search.empty())

{

size\_t size = search.size();

distance++;

for (size\_t i = 0; i < size; i++)

{

pair<int, int> pos = search.front();

search.pop();

vector<pair<int, int>> directions =

{ {0, -1}, {0, 1}, {-1, 0}, {1, 0} };

for (size\_t d = 0; d < directions.size(); d++)

{

pair<int, int> next = pos;

next.first += directions[d].first;

next.second += directions[d].second;

if (next.first < 0 || next.first >= (int)grid.size() ||

next.second < 0 || next.second >= (int)grid[0].size())

{

continue;

}

if (grid[next.first][next.second] == 0)

{

grid[next.first][next.second] = distance;

result = distance;

search.push(next);

}

}

}

}

return result;

}

## 1059. All Paths from Source Lead to Destination

Medium

Given the edges of a directed graph, and two nodes source and destination of this graph, determine whether or not all paths starting from source eventually end at destination, that is:

* At least one path exists from the source node to the destinationnode
* If a path exists from the source node to a node with no outgoing edges, then that node is equal to destination.
* The number of possible paths from source to destination is a finite number.

Return true if and only if all roads from source lead to destination.

**Example 1:**

Diagram, schematic

Description automatically generated

**Input:** n = 3, edges = [[0,1],[0,2]], source = 0, destination = 2

**Output:** false

**Explanation:** It is possible to reach and get stuck on both node 1 and node 2.

**Example 2:**

Diagram

Description automatically generated

**Input:** n = 4, edges = [[0,1],[0,3],[1,2],[2,1]], source = 0, destination = 3

**Output:** false

**Explanation:** We have two possibilities: to end at node 3, or to loop over node 1 and node 2 indefinitely.

**Example 3:**

Diagram

Description automatically generated

**Input:** n = 4, edges = [[0,1],[0,2],[1,3],[2,3]], source = 0, destination = 3

**Output:** true

**Example 4:**

Diagram

Description automatically generated

**Input:** n = 3, edges = [[0,1],[1,1],[1,2]], source = 0, destination = 2

**Output:** false

**Explanation:** All paths from the source node end at the destination node, but there are an infinite number of paths, such as 0-1-2, 0-1-1-2, 0-1-1-1-2, 0-1-1-1-1-2, and so on.

**Example 5:**

Diagram

Description automatically generated

**Input:** n = 2, edges = [[0,1],[1,1]], source = 0, destination = 1

**Output:** false

**Explanation:** There is infinite self-loop at destination node.

**Note:**

1. The given graph may have self loops and parallel edges.
2. The number of nodes n in the graph is between 1 and 10000
3. The number of edges in the graph is between 0 and 10000
4. 0 <= edges.length <= 10000
5. edges[i].length == 2
6. 0 <= source <= n - 1
7. 0 <= destination <= n - 1

### Analysis:

There is a key pattern in this problem, we care about the loop in the path which traverse the graph, so we can not use BFS, we can only use DFS, because in BFS, if we found a node is visited, we do not know the node belonging to which path.

/// <summary>

/// Leet code 1059. All Paths from Source Lead to Destination

/// </summary>

bool LeetCode::leadsToDestination(vector<vector<int>>& edges, vector<int> &visited, int source, int destination)

{

if (source == destination)

{

if (edges[destination].empty()) return true;

else return false;

}

else if (edges[source].empty())

{

return false;

}

else

{

for (auto next : edges[source])

{

if (visited[next] == 1) return false;

visited[next] = 1;

if (leadsToDestination(edges, visited, next, destination) == false) return false;

visited[next] = 0;

}

edges[source].clear();

edges[source].push\_back(destination);

return true;

}

}

/// <summary>

/// Leet code 1059. All Paths from Source Lead to Destination

///

/// Given the edges of a directed graph, and two nodes source and destination

/// of this graph, determine whether or not all paths starting from source

/// eventually end at destination, that is:

///

/// At least one path exists from the source node to the destination node

/// If a path exists from the source node to a node with no outgoing edges,

/// then that node is equal to destination.

/// The number of possible paths from source to destination is a finite number.

/// Return true if and only if all roads from source lead to destination.

///

/// Example 1:

/// Input: n = 3, edges = [[0,1],[0,2]], source = 0, destination = 2

/// Output: false

/// Explanation: It is possible to reach and get stuck on both node 1 and

/// node 2.

///

/// Example 2:

/// Input: n = 4, edges = [[0,1],[0,3],[1,2],[2,1]], source = 0,

/// destination = 3

/// Output: false

/// Explanation: We have two possibilities: to end at node 3, or to loop

/// over node 1 and node 2 indefinitely.

///

/// Example 3:

/// Input: n = 4, edges = [[0,1],[0,2],[1,3],[2,3]], source = 0,

/// destination = 3

/// Output: true

///

/// Example 4:

/// Input: n = 3, edges = [[0,1],[1,1],[1,2]], source = 0, destination = 2

/// Output: false

/// Explanation: All paths from the source node end at the destination node,

/// but there are an infinite number of paths, such as 0-1-2, 0-1-1-2,

/// 0-1-1-1-2, 0-1-1-1-1-2, and so on.

///

/// Example 5:

/// Input: n = 2, edges = [[0,1],[1,1]], source = 0, destination = 1

/// Output: false

/// Explanation: There is infinite self-loop at destination node.

///

/// Note:

///

/// 1. The given graph may have self loops and parallel edges.

/// 2. The number of nodes n in the graph is between 1 and 10000

/// 3. The number of edges in the graph is between 0 and 10000

/// 4. 0 <= edges.length <= 10000

/// 5. edges[i].length == 2

/// 6. 0 <= source <= n - 1

/// 7. 0 <= destination <= n - 1

/// </summary>

bool LeetCode::leadsToDestination(int n, vector<vector<int>>& edges, int source, int destination)

{

vector<vector<int>> new\_edges(n, vector<int>());

vector<int> visited(n);

for (size\_t i = 0; i < edges.size(); i++)

{

new\_edges[edges[i][0]].push\_back(edges[i][1]);

}

return leadsToDestination(new\_edges, visited, source, destination);

}

## 1730. Shortest Path to Get Food

Medium

You are starving and you want to eat food as quickly as possible. You want to find the shortest path to arrive at any food cell.

You are given an m x n character matrix, grid, of these different types of cells:

* '\*' is your location. There is **exactly one**'\*' cell.
* '#' is a food cell. There may be **multiple** food cells.
* 'O' is free space, and you can travel through these cells.
* 'X' is an obstacle, and you cannot travel through these cells.

You can travel to any adjacent cell north, east, south, or west of your current location if there is not an obstacle.

Return *the****length****of the shortest path for you to reach****any****food cell*. If there is no path for you to reach food, return -1.

**Example 1:**

A clock on a brick wall

Description automatically generated with medium confidence

**Input:** grid = [["X","X","X","X","X","X"],["X","\*","O","O","O","X"],["X","O","O","#","O","X"],["X","X","X","X","X","X"]]

**Output:** 3

**Explanation:** It takes 3 steps to reach the food.

**Example 2:**

A clock on a brick wall

Description automatically generated with medium confidence

**Input:** grid = [["X","X","X","X","X"],["X","\*","X","O","X"],["X","O","X","#","X"],["X","X","X","X","X"]]

**Output:** -1

**Explanation:** It is not possible to reach the food.

**Example 3:**

Clocks on a brick building

Description automatically generated with medium confidence

**Input:** grid = [["X","X","X","X","X","X","X","X"],["X","\*","O","X","O","#","O","X"],["X","O","O","X","O","O","X","X"],["X","O","O","O","O","#","O","X"],["X","X","X","X","X","X","X","X"]]

**Output:** 6

**Explanation:** There can be multiple food cells. It only takes 6 steps to reach the bottom food.

**Example 4:**

**Input:** grid = [["O","\*"],["#","O"]]

**Output:** 2

**Example 5:**

**Input:** grid = [["X","\*"],["#","X"]]

**Output:** -1

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 200
* grid[row][col] is '\*', 'X', 'O', or '#'.
* The grid contains **exactly one** '\*'.

### Analysis:

First you need to find the location of the dog, then because we need shortest path, we will use BFS. Please notice that we are walking in grid, so it is not a Dijkstra problem.

/// <summary>

/// Leet code 1730. Shortest Path to Get Food

///

/// Medium

///

/// You are starving and you want to eat food as quickly as possible.

/// You want to find the shortest path to arrive at any food cell.

///

/// You are given an m x n character matrix, grid, of these different

/// types of cells:

///

/// '\*' is your location. There is exactly one '\*' cell.

/// '#' is a food cell. There may be multiple food cells.

/// 'O' is free space, and you can travel through these cells.

/// 'X' is an obstacle, and you cannot travel through these cells.

/// You can travel to any adjacent cell north, east, south, or west of

/// your current location if there is not an obstacle.

///

/// Return the length of the shortest path for you to reach any food cell.

/// If there is no path for you to reach food, return -1.

/// Example 1:

/// Input: grid = [["X","X","X","X","X","X"],["X","\*","O","O","O","X"]

/// ,["X","O","O","#","O","X"],["X","X","X","X","X","X"]]

/// Output: 3

/// Explanation: It takes 3 steps to reach the food.

///

/// Example 2:

/// Input: grid = [["X","X","X","X","X"],["X","\*","X","O","X"],

/// ["X","O","X","#","X"],["X","X","X","X","X"]]

/// Output: -1

/// Explanation: It is not possible to reach the food.

///

/// Example 3:

/// Input: grid =

/// [["X","X","X","X","X","X","X","X"],["X","\*","O","X","O","#","O","X"],

/// ["X","O","O","X","O","O","X","X"],["X","O","O","O","O","#","O","X"],

/// ["X","X","X","X","X","X","X","X"]]

/// Output: 6

/// Explanation: There can be multiple food cells. It only takes 6

/// steps to reach the bottom food.

///

/// Example 4:

/// Input: grid = [["O","\*"],["#","O"]]

/// Output: 2

///

/// Example 5:

/// Input: grid = [["X","\*"],["#","X"]]

/// Output: -1

///

/// Constraints:

/// 1. m == grid.length

/// 2. n == grid[i].length

/// 3. 1 <= m, n <= 200

/// 4. grid[row][col] is '\*', 'X', 'O', or '#'.

/// 5. The grid contains exactly one '\*'.

/// </summary>

int LeetCodeGraph::getFood(vector<vector<char>>& grid)

{

vector<int> pos = { -1, -1 };

int m = grid.size();

int n = grid[0].size();

vector<vector<int>> visited(m, vector<int>(n));

for (int i = 0; i < m; i++)

{

for (int j = 0; j < n; j++)

{

if (grid[i][j] == '\*')

{

pos = { i, j };

break;

}

}

if (pos[0] != -1) break;

}

vector<vector<int>> directions = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

queue<vector<int>> queue;

visited[pos[0]][pos[1]] = 1;

queue.push(pos);

int result = 0;

while (!queue.empty())

{

result++;

int size = queue.size();

for (int i = 0; i < size; i++)

{

pos = queue.front();

queue.pop();

for (size\_t d = 0; d < directions.size(); d++)

{

vector<int> next = pos;

next[0] += directions[d][0];

next[1] += directions[d][1];

if (next[0] < 0 || next[0] >= m ||

next[1] < 0 || next[1] >= n) continue;

if (grid[next[0]][next[1]] == 'X') continue;

if (visited[next[0]][next[1]] == 1) continue;

if (grid[next[0]][next[1]] == '#')

{

return result;

}

visited[next[0]][next[1]] = 1;

queue.push(next);

}

}

}

return -1;

}

# Advanced Problem

## 752. Open the Lock

Medium

You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels can rotate freely and wrap around: for example we can turn '9' to be '0', or '0' to be '9'. Each move consists of turning one wheel one slot.

The lock initially starts at '0000', a string representing the state of the 4 wheels.

You are given a list of deadends dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.

Given a target representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

**Example 1:**

**Input:** deadends = ["0201","0101","0102","1212","2002"], target = "0202"

**Output:** 6

**Explanation:**

A sequence of valid moves would be "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202".

Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" -> "0202" would be invalid,

because the wheels of the lock become stuck after the display becomes the dead end "0102".

**Example 2:**

**Input:** deadends = ["8888"], target = "0009"

**Output:** 1

**Explanation:**

We can turn the last wheel in reverse to move from "0000" -> "0009".

**Example 3:**

**Input:** deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], target = "8888"

**Output:** -1

**Explanation:**

We can't reach the target without getting stuck.

**Example 4:**

**Input:** deadends = ["0000"], target = "8888"

**Output:** -1

**Note:**

1. The length of deadends will be in the range [1, 500].
2. target will not be in the list deadends.
3. Every string in deadends and the string target will be a string of 4 digits from the 10,000 possibilities '0000' to '9999'.

### Analysis:

You may be tempted to use DFS, but in fact BFS is more suitable, because you are asked the minimum turns. From any digit with one move you will get the next state.

/// <summary>

/// Leet code #752. Open the Lock

///

/// You have a lock in front of you with 4 circular wheels. Each wheel has

/// 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels

/// can rotate freely and wrap around: for example we can turn '9' to be

/// '0', or '0' to be '9'. Each move consists of turning one wheel one

/// slot.

///

/// The lock initially starts at '0000', a string representing the state

/// of the 4 wheels.

///

/// You are given a list of deadends dead ends, meaning if the lock

/// displays any of these codes, the wheels of the lock will stop turning

/// and you will be unable to open it.

///

/// Given a target representing the value of the wheels that will unlock

/// the lock, return the minimum total number of turns required to open

/// the lock, or -1 if it is impossible.

///

/// Example 1:

/// Input: deadends = ["0201","0101","0102","1212","2002"], target = "0202"

/// Output: 6

/// Explanation:

/// A sequence of valid moves would be "0000" -> "1000" -> "1100" ->

/// "1200" -> "1201" -> "1202" -> "0202".

/// Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" ->

/// "0202" would be invalid,

/// because the wheels of the lock become stuck after the display

/// becomes the dead end "0102".

/// Example 2:

/// Input: deadends = ["8888"], target = "0009"

/// Output: 1

/// Explanation:

/// We can turn the last wheel in reverse to move from "0000" -> "0009".

/// Example 3:

/// Input: deadends = ["8887","8889","8878","8898","8788","8988","7888",

/// "9888"], target = "8888"

/// Output: -1

/// Explanation:

/// We can't reach the target without getting stuck.

/// Example 4:

/// Input: deadends = ["0000"], target = "8888"

/// Output: -1

/// Note:

/// The length of deadends will be in the range [1, 500].

/// target will not be in the list deadends.

/// Every string in deadends and the string target will be a string of 4

/// digits from the 10,000 possibilities '0000' to '9999'.

/// </summary>

int LeetCode::openLock(vector<string>& deadends, string target)

{

queue<string> bfs;

unordered\_set<string> visited;

unordered\_set<string> deadset;

for (string str : deadends) deadset.insert(str);

int result = 0;

string lock = "0000";

if (deadset.count(lock) == 0)

{

visited.insert(lock);

bfs.push(lock);

}

while (!bfs.empty())

{

size\_t size = bfs.size();

for (size\_t i = 0; i < size; i++)

{

string lock = bfs.front();

bfs.pop();

if (deadset.count(lock) > 0) continue;

if (lock == target) return result;

for (size\_t j = 0; j < 4; j++)

{

for (int k = -1; k <= 1; k += 2)

{

string next = lock;

next[j] = '0' + ((next[j] - '0') + k + 10) % 10;

if ((deadset.count(lock) == 0) && (visited.count(next) == 0))

{

visited.insert(next);

bfs.push(next);

}

}

}

}

result++;

}

return -1;

}

## 529. Minesweeper

Medium

Let's play the minesweeper game ([Wikipedia](https://en.wikipedia.org/wiki/Minesweeper_(video_game)), [online game](http://minesweeperonline.com/))!

You are given a 2D char matrix representing the game board. **'M'** represents an **unrevealed** mine, **'E'** represents an **unrevealed** empty square, **'B'** represents a **revealed** blank square that has no adjacent (above, below, left, right, and all 4 diagonals) mines, **digit** ('1' to '8') represents how many mines are adjacent to this **revealed** square, and finally **'X'** represents a **revealed** mine.

Now given the next click position (row and column indices) among all the **unrevealed** squares ('M' or 'E'), return the board after revealing this position according to the following rules:

1. If a mine ('M') is revealed, then the game is over - change it to **'X'**.
2. If an empty square ('E') with **no adjacent mines** is revealed, then change it to revealed blank ('B') and all of its adjacent **unrevealed** squares should be revealed recursively.
3. If an empty square ('E') with **at least one adjacent mine** is revealed, then change it to a digit ('1' to '8') representing the number of adjacent mines.
4. Return the board when no more squares will be revealed.

**Example 1:**

**Input:**

[['E', 'E', 'E', 'E', 'E'],

['E', 'E', 'M', 'E', 'E'],

['E', 'E', 'E', 'E', 'E'],

['E', 'E', 'E', 'E', 'E']]

Click : [3,0]

**Output:**

[['B', '1', 'E', '1', 'B'],

['B', '1', 'M', '1', 'B'],

['B', '1', '1', '1', 'B'],

['B', 'B', 'B', 'B', 'B']]

**Explanation:**

Graphical user interface, application, Teams

Description automatically generated

**Example 2:**

**Input:**

[['B', '1', 'E', '1', 'B'],

['B', '1', 'M', '1', 'B'],

['B', '1', '1', '1', 'B'],

['B', 'B', 'B', 'B', 'B']]

Click : [1,2]

**Output:**

[['B', '1', 'E', '1', 'B'],

['B', '1', 'X', '1', 'B'],

['B', '1', '1', '1', 'B'],

['B', 'B', 'B', 'B', 'B']]

**Explanation:**

Graphical user interface, application, Teams

Description automatically generated

**Note:**

1. The range of the input matrix's height and width is [1,50].
2. The click position will only be an unrevealed square ('M' or 'E'), which also means the input board contains at least one clickable square.
3. The input board won't be a stage when game is over (some mines have been revealed).
4. For simplicity, not mentioned rules should be ignored in this problem. For example, you **don't** need to reveal all the unrevealed mines when the game is over, consider any cases that you will win the game or flag any squares.

### Analysis:

This is a simulation problem, you do BFS search from the neighbors from the click point, the search will end by the cell with mines nearby.

/// <summary>

/// Leet code #529. Minesweeper

///

/// Let's play the minesweeper game (Wikipedia, online game)!

/// You are given a 2D char matrix representing the game board. 'M'

/// represents an unrevealed mine, 'E' represents an unrevealed empty square,

/// 'B' represents a revealed blank square that has no adjacent (above, below,

/// left, right, and all 4 diagonals) mines, digit ('1' to '8') represents how

/// many mines are adjacent to this revealed square, and finally 'X' represents

/// a revealed mine.

///

/// Now given the next click position (row and column indices) among all the

/// unrevealed squares ('M' or 'E'), return the board after revealing this

/// position according to the following rules:

///

/// If a mine ('M') is revealed, then the game is over - change it to 'X'.

/// If an empty square ('E') with no adjacent mines is revealed, then change it

/// to revealed blank ('B') and all of its adjacent unrevealed squares should be

/// revealed recursively.

/// If an empty square ('E') with at least one adjacent mine is revealed, then

/// change it to a digit ('1' to '8') representing the number of adjacent mines.

/// Return the board when no more squares will be revealed.

/// Example 1:

/// Input:

///

/// [['E', 'E', 'E', 'E', 'E'],

/// ['E', 'E', 'M', 'E', 'E'],

/// ['E', 'E', 'E', 'E', 'E'],

/// ['E', 'E', 'E', 'E', 'E']]

/// Click : [3,0]

/// Output:

///

/// [['B', '1', 'E', '1', 'B'],

/// ['B', '1', 'M', '1', 'B'],

/// ['B', '1', '1', '1', 'B'],

/// ['B', 'B', 'B', 'B', 'B']]

///

/// Explanation:

///

/// Example 2:

/// Input:

///

/// [['B', '1', 'E', '1', 'B'],

/// ['B', '1', 'M', '1', 'B'],

/// ['B', '1', '1', '1', 'B'],

/// ['B', 'B', 'B', 'B', 'B']]

/// Click : [1,2]

/// Output:

///

/// [['B', '1', 'E', '1', 'B'],

/// ['B', '1', 'X', '1', 'B'],

/// ['B', '1', '1', '1', 'B'],

/// ['B', 'B', 'B', 'B', 'B']]

///

/// Explanation:

/// Note:

/// The range of the input matrix's height and width is [1,50].

/// The click position will only be an unrevealed square ('M' or 'E'),

/// which also means the input board contains at least one clickable square.

/// The input board won't be a stage when game is over (some mines have been revealed).

/// For simplicity, not mentioned rules should be ignored in this problem. For example,

/// you don't need to reveal all the unrevealed mines when the game is over, consider

/// any cases that you will win the game or flag any squares.

/// </summary>

vector<vector<char>> LeetCodeArray::updateBoard(vector<vector<char>>& board, vector<int>& click)

{

vector<vector<int>> directions =

{

{-1, -1}, {-1, 0}, {-1, 1},

{0, -1}, {0, 1},

{1, -1}, {1, 0}, {1, 1}

};

vector<int> curr = click;

if (board.empty() || board[0].empty() || curr.size() < 2) return board;

queue<vector<int>> process\_queue;

if (board[curr[0]][curr[1]] == 'M')

{

board[curr[0]][curr[1]] = 'X';

return board;

}

process\_queue.push(curr);

while (!process\_queue.empty())

{

curr = process\_queue.front();

process\_queue.pop();

int count = 0;

vector<vector<int>> result;

if (board[curr[0]][curr[1]] == 'E')

{

for (size\_t d = 0; d < directions.size(); d++)

{

vector<int> next = curr;

next[0] = curr[0] + directions[d][0];

next[1] = curr[1] + directions[d][1];

if (next[0] < 0 || next[0] >= (int)board.size() ||

next[1] < 0 || next[1] >= (int)board[0].size())

{

continue;

}

if (board[next[0]][next[1]] == 'M') count++;

else result.push\_back(next);

}

if (count == 0)

{

board[curr[0]][curr[1]] = 'B';

for (size\_t i = 0; i < result.size(); i++)

{

process\_queue.push(result[i]);

}

}

else board[curr[0]][curr[1]] = '0' + count;

}

}

return board;

}

## 773. Sliding Puzzle

Hard

On a 2x3 board, there are 5 tiles represented by the integers 1 through 5, and an empty square represented by 0.

A move consists of choosing 0 and a 4-directionally adjacent number and swapping it.

The state of the board is *solved* if and only if the board is [[1,2,3],[4,5,0]].

Given a puzzle board, return the least number of moves required so that the state of the board is solved. If it is impossible for the state of the board to be solved, return -1.

**Examples:**

**Input:** board = [[1,2,3],[4,0,5]]

**Output:** 1

**Explanation:** Swap the 0 and the 5 in one move.

**Input:** board = [[1,2,3],[5,4,0]]

**Output:** -1

**Explanation:** No number of moves will make the board solved.

**Input:** board = [[4,1,2],[5,0,3]]

**Output:** 5

**Explanation:** 5 is the smallest number of moves that solves the board.

An example path:

After move 0: [[4,1,2],[5,0,3]]

After move 1: [[4,1,2],[0,5,3]]

After move 2: [[0,1,2],[4,5,3]]

After move 3: [[1,0,2],[4,5,3]]

After move 4: [[1,2,0],[4,5,3]]

After move 5: [[1,2,3],[4,5,0]]

**Input:** board = [[3,2,4],[1,5,0]]

**Output:** 14

**Note:**

* board will be a 2 x 3 array as described above.
* board[i][j] will be a permutation of [0, 1, 2, 3, 4, 5].

### Analysis:

We need to get the shortest step, so we do BFS, the difficult part here is how to remember the visited pattern, given only 5 slides and 6 place, it is not difficult.

/// <summary>

/// Leetcode #773. Sliding Puzzle

///

/// On a 2x3 board, there are 5 tiles represented by the integers 1

/// through 5, and an empty square represented by 0.

///

/// A move consists of choosing 0 and a 4-directionally adjacent number

/// and swapping it.

///

/// The state of the board is solved if and only if the board is

/// [[1,2,3],[4,5,0]].

///

/// Given a puzzle board, return the least number of moves required so

/// that the state of the board is solved. If it is impossible for the

/// state of the board to be solved, return -1.

///

/// Examples:

///

/// Input: board = [[1,2,3],[4,0,5]]

/// Output: 1

/// Explanation: Swap the 0 and the 5 in one move.

///

/// Input: board = [[1,2,3],[5,4,0]]

/// Output: -1

/// Explanation: No number of moves will make the board solved.

///

/// Input: board = [[4,1,2],[5,0,3]]

/// Output: 5

/// Explanation: 5 is the smallest number of moves that solves the board.

/// An example path:

/// After move 0: [[4,1,2],[5,0,3]]

/// After move 1: [[4,1,2],[0,5,3]]

/// After move 2: [[0,1,2],[4,5,3]]

/// After move 3: [[1,0,2],[4,5,3]]

/// After move 4: [[1,2,0],[4,5,3]]

/// After move 5: [[1,2,3],[4,5,0]]

///

/// Input: board = [[3,2,4],[1,5,0]]

/// Output: 14

/// Note:

/// board will be a 2 x 3 array as described above.

/// board[i][j] will be a permutation of [0, 1, 2, 3, 4, 5].

/// </summary>

int LeetCode::slidingPuzzle(vector<vector<int>>& board)

{

set<string> visited;

queue<string> search\_queue;

string board\_str;

for (size\_t i = 0; i < board.size(); i++)

{

for (size\_t j = 0; j < board[i].size(); j++)

{

board\_str.push\_back('0' + board[i][j]);

}

}

search\_queue.push(board\_str);

visited.insert(board\_str);

int result = 0;

while (!search\_queue.empty())

{

size\_t size = search\_queue.size();

for (size\_t i = 0; i < size; i++)

{

string curr\_board = search\_queue.front();

search\_queue.pop();

if (curr\_board == "123450")

{

return result;

}

int pos = curr\_board.find('0');

vector<int> direction = { -1, 1, -3, 3 };

for (size\_t i = 0; i < direction.size(); i++)

{

int next = pos + direction[i];

if ((next < 0) || (next >= (int)curr\_board.size()))

{

continue;

}

// two edge cases

if (pos == 3 && direction[i] == -1) continue;

if (pos == 2 && direction[i] == 1) continue;

string next\_board = curr\_board;

swap(next\_board[pos], next\_board[next]);

if (visited.count(next\_board) == 0)

{

search\_queue.push(next\_board);

visited.insert(next\_board);

}

}

}

result++;

}

return -1;

}

## 815. Bus Routes

Hard

We have a list of bus routes. Each routes[i] is a bus route that the i-th bus repeats forever. For example if routes[0] = [1, 5, 7], this means that the first bus (0-th indexed) travels in the sequence 1->5->7->1->5->7->1->... forever.

We start at bus stop S (initially not on a bus), and we want to go to bus stop T. Travelling by buses only, what is the least number of buses we must take to reach our destination? Return -1 if it is not possible.

**Example:**

**Input:**

routes = [[1, 2, 7], [3, 6, 7]]

S = 1

T = 6

**Output:** 2

**Explanation:**

The best strategy is take the first bus to the bus stop 7, then take the second bus to the bus stop 6.

**Note:**

* 1 <= routes.length <= 500.
* 1 <= routes[i].length <= 500.
* 0 <= routes[i][j] < 10 ^ 6.

### Analysis:

Because we need to get the shortest step, we start the first bus and do BFS search, when we reach the destination, it is by default the shortest stops. During the search, we remember the visited stops.

/// <summary>

/// Leet code #815. Bus Routes

///

/// We have a list of bus routes. Each routes[i] is a bus route that the

/// i-th bus repeats forever. For example if routes[0] = [1, 5, 7], this

/// means that the first bus (0-th indexed) travels in the sequence

/// 1->5->7->1->5->7->1->... forever.

///

/// We start at bus stop S (initially not on a bus), and we want to go to

/// bus stop T. Travelling by buses only, what is the least number of

/// buses we must take to reach our destination? Return -1 if it is not

/// possible.

///

/// Example:

/// Input:

/// routes = [[1, 2, 7], [3, 6, 7]]

/// S = 1

/// T = 6

/// Output: 2

/// Explanation:

/// The best strategy is take the first bus to the bus stop 7, then take

/// the second bus to the bus stop 6.

/// Note:

///

/// 1. 1 <= routes.length <= 500.

/// 2. 1 <= routes[i].length <= 500.

/// 3. 0 <= routes[i][j] < 10 ^ 6.

/// </summary>

int LeetCode::numBusesToDestination(vector<vector<int>>& routes, int S, int T)

{

unordered\_map<int, set<int>> stop\_map;

unordered\_set<int> visited;

for (size\_t i = 0; i < routes.size(); i++)

{

for (size\_t j = 0; j < routes[i].size(); j++)

{

stop\_map[routes[i][j]].insert(i);

}

}

queue<int> stop\_queue;

stop\_queue.push(S);

visited.insert(S);

int result = 0;

while (!stop\_queue.empty())

{

size\_t size = stop\_queue.size();

for (size\_t i = 0; i < size; i++)

{

int stop = stop\_queue.front();

stop\_queue.pop();

if (stop == T) return result;

for (int bus : stop\_map[stop])

{

for (int next : routes[bus])

{

if (visited.count(next) > 0)

{

continue;

}

stop\_queue.push(next);

visited.insert(next);

stop\_map[next].erase(bus);

}

}

stop\_map[stop].clear();

}

result++;

}

return -1;

}