Advanced BFS

Understanding BFS

Breadth First Search (BFS) is an algorithm for traversing or searching layer wise in tree, graph or grid data structures where we start traversing from a selected source node layer wise by exploring the neighboring nodes.

The data structure used in BFS is a **queue** and a graph. The algorithm makes sure that every node is visited not more than once.

BFS follows the following steps:

1. Begin the search algorithm, by knowing the key which is to be searched. Once the key/element to be searched is decided the searching begins with the root (source) first.
2. Visit the contiguous unvisited vertex. Mark it as visited. Display it (if needed). If this is the required key, stop. Else, add it in a queue.
3. On the off chance that no neighboring vertex is discovered, expel the first vertex from the Queue.
4. Repeat step 2 and 3 until the queue is empty.

The above algorithm is a search algorithm that identifies whether a node exists in the graph. We can convert the algorithm to traversal algorithm to find all the reachable nodes from a given node.

The time complexity of BFS is **O(V + E)** if the graph is represented as **adjacency list**; **O(V*V)** if the graph is represented as **adjacency matrix**.

Pseudocode of BFS

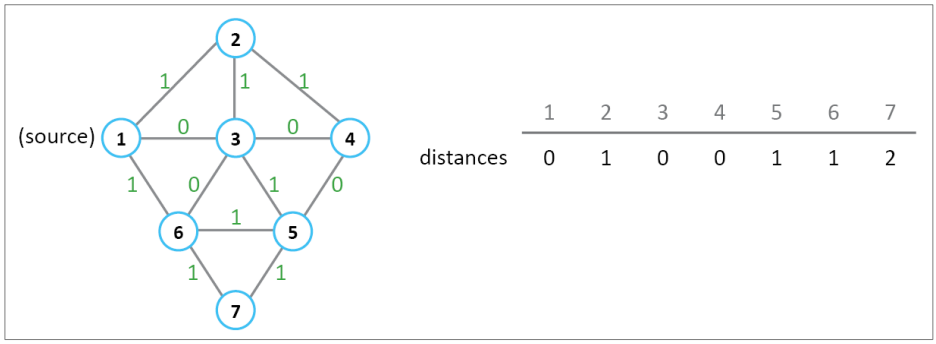
```
// Iterative BFS. Return true if target is reachable from the source.
void (Graph graph, int source,  int target)
{
      Queue q = Queue();     // create a queue needed for BFS

      // declare a visited array and initialize it to false
      boolean[] visited = { false };
      // mark source node as discovered
      visited[source] = true;

      // push source node into the queue
      q.add(source);

      // while queue is not empty
      while (!q.isEmpty())
      {
            // pop front node from queue
            cur = q.poll();
            if (cur == key) return "Found"; //Key has been found.

            // for every neighboring node of cur
            for (int v : graph.get(cur)) {
                  if (!visited[v]) {
                        // mark it visited and enqueue to queue
                        visited[v] = true;
                        q.add(v);
                  }
            }
      }
      return "Not Found"; //If key has not been found
}
```

0-1 BFS

This type of BFS is used to find shortest distance or path from a source node to a destination node in a graph with edge values **0** or **1**.

When the weights of edges are 0 or 1, the normal BFS techniques provide erroneous results because in normal BFS technique, it is assumed that the weight of edges would be equal in the graph.

In this technique, we will check for the optimal distance condition instead of using bool array to mark visited nodes. We always follow edges with value 0 if possible.



Sample Input

```
7 12
1 6 1
1 2 1
2 4 1
4 5 0
5 6 1
6 3 0
3 1 0
2 3 1
3 4 0
3 5 1
6 7 1
7 5 1
```

Sample Output

```
0 1 0 0 1 1 2
```

Implementation with a Deque

We use double ended queue (**deque**) to store the node details. While performing BFS, if we encounter an edge having **weight = 0**, the node is pushed at **front** of double ended queue and if an edge having **weight = 1** is found, the node is pushed at **back** of double ended queue.

```
// 0-1 BFS implementing with a deque
#include <fstream>
#include <queue>
#include <vector>
#include <algorithm>

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M;        //number of nodes and edges
int source;      //source node
vector<vector<pair<int, int>>> adjList;
vector<int> distances;
```

```
const int BIGNUM = 1000000;
//------------------------------
//read input and create adjacency list of the graph
void readInput()
{
	cin >> N >> M;				//number of nodes and edges

	adjList.resize(N + 1);		//1-based array

	//read every edge data
	for (int i = 0; i < M; i++)
	{
		int u, v, w;			// edge connecting u and v with the weight w.
		cin >> u >> v >> w;
		adjList[u].push_back({ v, w });
		adjList[v].push_back({ u, w });
	}
}
//------------------------------
//calculate all shortest paths from the source to all other nodes.
void bfs()
{
	distances.resize(N + 1, BIGNUM);
	deque<int> dq;

	//start from the source node.
	distances[source] = 0;
	dq.push_back(source);

	while (!dq.empty())
	{
		//get the front element of deque as the current node.
		int u = dq.front();
		dq.pop_front();

		//visit every unvisited neighbors of u
		for (auto p : adjList[u])
		{
			//neighbor node v and the distance between u and v.
			int v = p.first, w = p.second;
			if (distances[v] == BIGNUM)			//v is not visited yet
			{
				distances[v] = distances[u] + w;	//set the distance for v

				//depending of the w, push v at the back or front or back of the dq
				if (w == 0)
					dq.push_front(v);
				else
					dq.push_back(v);
			}
		}
	}
}
//------------------------------
//print distances, if not reachable "INF"
void printDistances()
{
	for (int i = 1; i <= N; i++)
		if (distances[i] == BIGNUM)
			cout << "INF ";
		else
			cout << distances[i] << " ";
}
//------------------------------
int main()
{
	readInput();
	source = 1;
	bfs();
	printDistances();
}
```

Implementation with Two Queue Structures

We use two separate queue data structures to store the node details in this implementation. One queue is used for the zero weighted connections (**zeroQueeu**) and the other queue is used for the one weighted connection (**oneQueue**). While performing BFS, if we encounter an edge having **weight = 0**, the node is pushed at back of the zeroQueue and if an edge having **weight = 1** is found, the node is pushed at back of the oneQueue. We process the nodes in the oneQueue only if the zeroQueue is empty. zeroQueeu has a higher priority to be processed.

```
// 0-1 BFS implementing with two queues
#include <fstream>
#include <queue>
#include <vector>
#include <algorithm>

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M;                    //number of nodes and edges
int source;                  //source node
vector<vector<pair<int, int>>> adjList;
vector<int> distances;
const int BIGNUM = 1000000;
//------------------------------
//read input and create adjacency list of the graph
void readInput()
{
      cin >> N >> M;         //number of nodes and edges

      adjList.resize(N + 1); //1-based array

      //read every edge data
      for (int i = 0; i < M; i++)
      {
            int u, v, w;     //edge connecting u and v with the weight w.
            cin >> u >> v >> w;
            adjList[u].push_back({ v, w });
            adjList[v].push_back({ u, w });
      }
}
//------------------------------
//Calculate all shortest paths from the source to all other nodes.
void bfs()
{
      distances.resize(N + 1, BIGNUM);
      queue<int> zeroQ, oneQ;

      //start from the source node
      distances[source] = 0;
      zeroQ.push(source);

      while (!zeroQ.empty() || !oneQ.empty())
      {
            //get the next element form zeroQ, if zeroQ is empty form oneQ.
            int u;
            if (!zeroQ.empty())
            {
                  u = zeroQ.front();
                  zeroQ.pop();
            }
            else if (!oneQ.empty())
            {
                  u = oneQ.front();
                  oneQ.pop();
            }
```

```
            //visit every unvisited neighbors of u
            for (auto p : adjList[u])
            {
                  //neighbor node v and the distance between u and v.
                  int v = p.first, w = p.second;
                  if (distances[v] == BIGNUM)            //v is not visited yet
                  {
                        distances[v] = distances[u] + w;   //set the distance for v

                        //depending of the w, push v at one of the queues
                        if (w == 0)
                              zeroQ.push(v);
                        else
                              oneQ.push(v);
                  }
            }
      }
}
//------------------------------
//print distances, if not reachable "INF"
void printDistances()
{
      for (int i = 1; i <= N; i++)
            if (distances[i] == BIGNUM)
                  cout << "INF ";
            else
                  cout << distances[i] << " ";
}
//------------------------------
int main()
{
      readInput();
      source = 1;
      bfs();
      printDistances();
}
```

Example: Matrix

Given a matrix of size N x M consisting of integers 1, 2, 3 and 4. Each value represents one of the four possible movements from that cell:

1: move up
2: move right
3: move down
4: move left

There is no diagonal movement.

What is the minimum number of possible changes required in the matrix so that there exists a path from top-left cell (1, 1) to the bottom-right cell (N, M).

Sample Input

```
3 4
3 2 3 3
2 1 4 3
1 3 2 1
```

Sample Output

```
1
```

Output Details

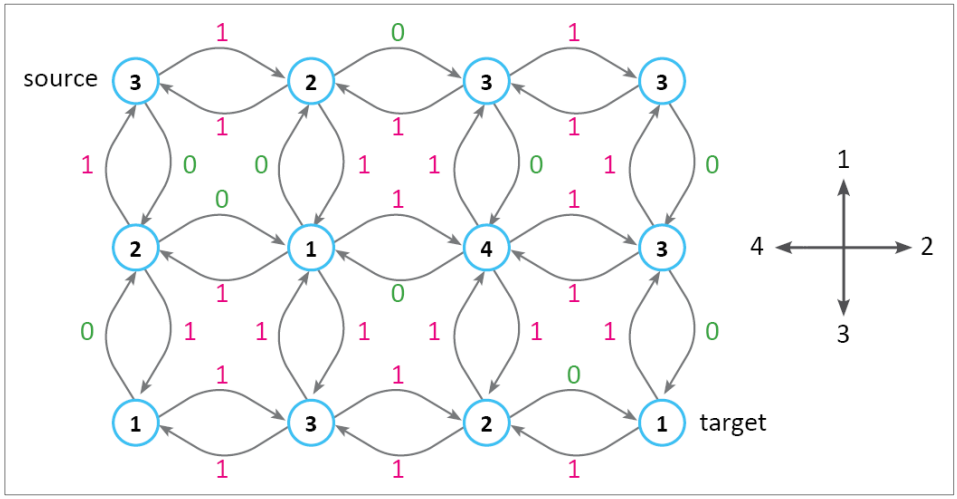
```
3 2 2 3
2 1 4 3
1 3 2 1
```

Solution Idea

Grids are a form of **implicit graph**. We can determine the neighbors of our current location by searching within the grid instead of representing the graph using adjacency list or adjacency matrix.

Consider each cell of the 2D matrix as a node of the weighted directed graph and each node can have at most four connected nodes (possible four directions). Each edge (*u*, *v*) is weighted as 0 if direction of movement of node *u* points to *v*, otherwise it is weighted as 1.

The shortest distance from the source node to the target node on the 0-1 graph is the minimum number of the changes on the matrix required to arrive the target from the source.



BFS with Splitting Edges

Any weighted graph can be converted into its equivalent unweighted graph by splitting the edges into smaller edges and adding new nodes so that all edges in the graph have the same weight. This method is useful when the difference of the largest edge weight and the smallest edge weight is a small number or the edge weights are multiples of each other.

Example: Shortest path on a 1-2 Graph

Calculate the distance of the shortest path from node 1 to node N in an undirected weighted graph where the edge costs are either 1 or 2.

Sample Input

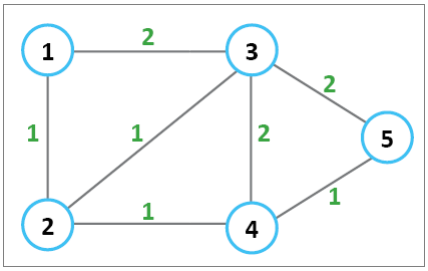
```
5 7
1 3 2
1 2 1
2 3 1
4 2 1
3 4 2
4 5 1
5 3 2
```

Sample Output

```
3
```

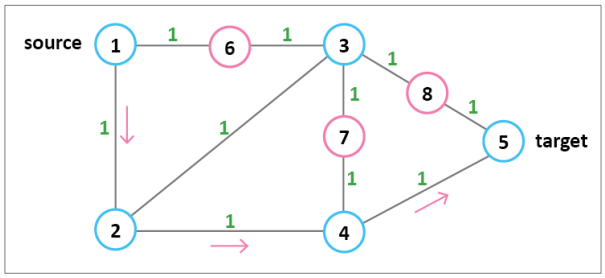
Output Details

```
1 -> 2 -> 4 -> 5
1 + 1 + 1 = 3
```



Solution Idea

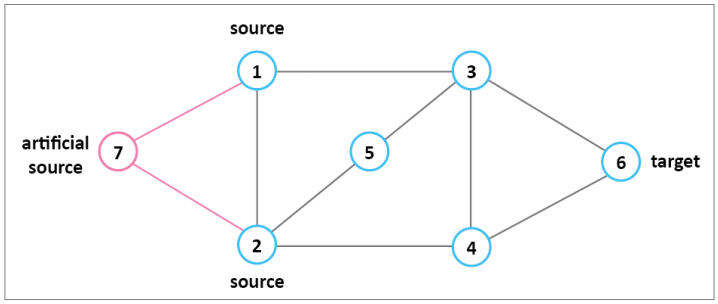
Split the edges with cost 2 by two edges with cost 1 and calculate the shortest distance from the source node to the target node using standard BFS algorithm.



Multisource BFS

In the standard BFS program, we start from one source point and calculate the shortest paths to all other points. If there are multiple source points, instead of doing a separate BFS for each source point, we initiate a single BFS search from all source points simultaneously. We can implement it in two ways:

1) Add all source nodes into the queue and loop for the remaining nodes.
2) Create an artificial node, connect it to all source nodes and initiate BFS search from the new artificial node. You should subtract 1 from the result after finishing BFS.



Flood Fill with BFS

Flood fill is a search that starts at a point and finds the areas connected to the start point. For example, the **Bucket Fill** in MS Paint uses flood fill to fill in the connecting areas of the same color.

Flood fill can be recursively implemented using a DFS or iteratively implemented with BFS.

Example: Bucket Fill

There is an N x M matrix; a start point on the matrix and a destination color *dstColor*. We want to replace all the cells in the matrix that are connected to the start point and having the same color as start point with the color *dstColor*. The coordinates of the top-left cell is (1, 1)

Sample Input

```
3 5 2 2 7       //3x5 matrix, start point is (2,2) and destination color is 7
1 2 3 4 5
3 3 3 3 4
2 2 2 2 2
```

Sample Input

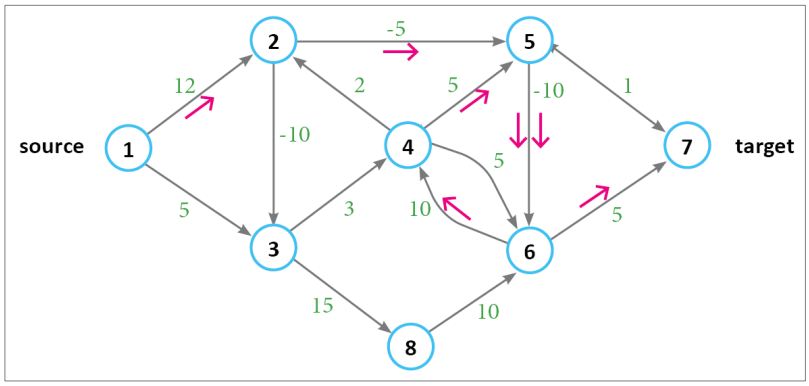
```
1 2 7 4 5
7 7 7 7 4
2 2 5 2 2
```

```
// Flood Fill with BFS
#include <fstream>
#include <queue>
#include <vector>

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M, x, y, dstColor; //dimensions of the grid, start point and the target color
vector<vector<int>> grid;
//------------------------------
void readInput()
{
	cin >> N >> M >> x >> y >> dstColor;

	grid.resize(N + 1, vector<int>(M + 1)); //allocate space for the grid

	for (int i = 1; i <= N; i++)
		for (int j = 1; j <= M; j++)
			cin >> grid[i][j];
}
//------------------------------
void bfs()
{
	//to test four neighbors. 0:up, 1:right, 2:down, 3:left
	vector<int> xDir = { -1, 0, 1, 0 }, yDir = { 0, 1, 0, -1 };

	int srcColor = grid[x][y];   //color of the source cell
	queue<pair<int, int>> q;
	q.push({ x, y });

	while (!q.empty())
	{
		int curX = q.front().first;
		int curY = q.front().second;
		q.pop();
```

```
            for (int i = 0; i < 4; i++)
            {
                  //One neighbor
                  int xx = curX + xDir[i];
                  int yy = curY + yDir[i];

                  //test if the neighbor is a valid cell
                  if (xx < 1 || xx >N || yy < 0 || yy > M || grid[xx][yy] != srcColor)
                        continue;

                  //neighbor is a valid cell, color it and push into the queue
                  grid[xx][yy] = dstColor;
                  q.push({ xx, yy });
            }
      }
}
//------------------------------
void printResult()
{
      for (int i = 1; i <= N; i++)
      {
            for (int j = 1; j <= M; j++)
                  cout << grid[i][j] << " ";
            cout << endl;
      }
}
//------------------------------
int main()
{
      readInput();
      bfs();
      printResult();
}
```

Minimum Cost Path with K Edges on a Weighted Graph

Given a weighted, directed graph G(V, E), find the minimum cost path from a given source to a target node with exactly k edges on the path.



Sample Input

```
8 14 7 1 7      //8 nodes, 14 edges, path length is 7, source is 1, target is 7
1 2 12
1 3 5
2 3 -10
3 4 3
3 8 15
8 6 10
```

```
4 6 5
6 4 10
2 4 2
2 5 -5
5 6 -10
5 7 1
6 7 5
4 5 5
```

Sample Output

```
7
```

Output Details

```
1 -> 2 -> 5 -> 6 -> 4 -> 5 -> 6 -> 7
12 - 5 - 10 + 10 + 5 -10 + 5 = 7
```

Normally the shortest paths problems on a weighted graph are solved by shortest paths algorithms like Dijkstra, Bellman-Ford or Floyd-Warshall. However, there is constrain (path must have exactly K edges) in this problem that prevents us to use standard shortest paths algorithms.

We can modify the standard BFS algorithm and solve this problem in O(K(V+E)) time. We start BFS traversal from the source node. Usually, BFS does not explore already visited nodes again, but here we do the opposite. In order to cover all possible K-length paths from source to target, we remove this check from BFS. We prevent the infinitive loop by eliminating the paths longer than K edges.

```
//K-edge min cost path in a weighted graph. BFS O(K(N+M)).
#include <fstream>
#include <queue>
#include <vector>

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M, K, source, target;
vector<vector<pair<int,int>>> adjList; //first: v, second
//------------------------------
//read input and create adjacency list of the graph
void readInput()
{
	cin >> N >> M >> K >> source >> target;

	adjList.resize(N + 1);

	for (int i = 0; i < M; i++)
	{
		int u, v, w;		//an edge from u to v with the cost w
		cin >> u >> v >> w;
		adjList[u].push_back({ v, w });
	}
}
//------------------------------
//print the distance of the min cost path with K edges form source to target.
void bfs()
{
	const int BIGNUM = 1000000000;

	//2d array for memoization
	vector<vector<int>> distances(N + 1, vector<int>(K + 1, BIGNUM));
```

```
	queue<pair<int,int>> q; //first:v, second: len

	distances[source][0] = 0;
	q.push({ source, 0 });

	while (!q.empty())
	{
		int u = q.front().first;
		int len = q.front().second;
		q.pop();

		if (len >= K)	//no need to continue
			continue;

		for (auto p : adjList[u])
		{
			int v = p.first;
			int w = p.second;

			//there is a shorter path to v with len+1 edges.
			if (distances[v][len + 1] > distances[u][len] + w)
			{
				distances[v][len + 1] = distances[u][len] + w;
				q.push({ v, len + 1 });
			}
		}
	}

	//print result
	if (distances[target][K] == BIGNUM)
		cout << "Impossible" << endl;
	else
		cout << distances[target][K] << endl;
}
//------------------------------
int main()
{
	readInput();
	bfs();
}
```

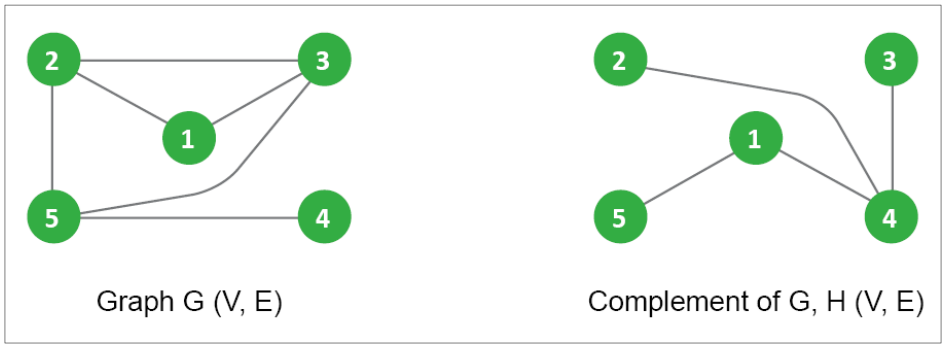
Exercise: K-length Min Cost Path

Implement the K-length minimum cost path problem with tabulation method without using BFS traversal.

BFS on a Complementary Graph

In graph theory, the complement or inverse of a graph G is a graph H on the same vertices (only the edges are complemented) such that two distinct vertices of H are adjacent if and only if they are not adjacent in G. That is, to generate the complement of a graph, one fills in all the missing edges required to form a complete graph, and removes all the edges that were previously there.

A combination of two complementary graphs gives a complete graph.



We can slightly modify the standard BFS implementation and solve the shortest path problem on an complementary unweighted graph in O((V + E) time in the original graph if we maintain the graph edges and complementary edges in Hash Set data structures. Instead of Hash Set, if we use Set data structures, time complexity will be O((V+E)lgV).

The following program calculates all shortest distances from the source node 1 to all other nodes on the complementary graph in O(V+E) time.

Sample Input

```
5 6
1 2
1 3
2 3
5 2
5 4
5 3
```

Sample Output

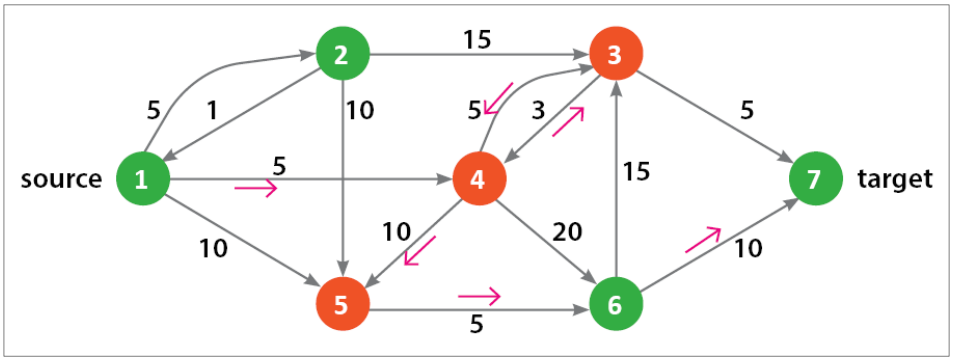
```
0 2 2 1 1
```

```
// BFS on complementary graph. O(V + E) implementation
#include <fstream>
#include <queue>
#include <vector>
#include <unordered_set>

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M ;
vector<vector<int>> adjList;
//------------------------------
//read input and create adjacency list of the graph
void readInput()
{
	cin >> N >> M;

	adjList.resize(N + 1);

	for (int i = 0; i < M; i++)
	{
		int u, v;		//an edge from u to v with the cost w
		cin >> u >> v;
		adjList[u].push_back(v);
		adjList[v].push_back(u);
	}
}
//------------------------------
//single-source all shortet pahts of the complementary graph.
```

```
void bfs()
{
	const int BIGNUM = 1000000;
	int source = 1;		//Node 1 is the source.

	vector<int> distances(N + 1, BIGNUM);
	queue<int> q;

	distances[source] = 0;
	q.push(source);

	//adjacent is a hash set for available edges
	//notAdjacent is another hash set for complementary edges
	unordered_set<int> adjacent, notAdjacent;

	//We consider all edges are on the complementary graph.
	for (int i = 1; i <= N; i++)
		notAdjacent.insert(i);

	//Since we only use the complementary edges, we stop if there is no such edge.
	while (!q.empty() && !notAdjacent.empty())
	{
		int u = q.front();
		q.pop();

		//Current node will not be used any more.
		notAdjacent.erase(u);

		//If any neighbor has not been discovered yet,
		//remove its edge from the complementary set.
		for(int v:adjList[u])
			if (distances[v] == BIGNUM)
			{
				adjacent.insert(v);
				notAdjacent.erase(v);
			}

		//Calculate distances for undiscovered nodes
		//using only complementary edges
		for (int v : notAdjacent)
		{
			if (distances[v] == BIGNUM)
			{
				distances[v] = distances[u] + 1;
				q.push(v);
			}
		}

		//Insert edges of u back into the complementary edge set
		//for the further calculations.
		notAdjacent.insert(adjacent.begin(), adjacent.end());
		adjacent.clear();
	}

	//Print shortest distances from the source.
	for (int i = 1; i <= N; i++)
		cout << distances[i] << " ";
}
//------------------------------
int main()
{
	readInput();
	bfs();
}
```

BFS with Bit Masking

Given a weighted, directed graph G (V, E), and a set X of vertices. Find the Minimum Cost Path passing through all the vertices of the set X, from a given source vertex S to a target vertex T. The size of X is K. Source and target nodes are not member of X.



Sample Input

```
7 14 1 7 3      // 7 nodes, 14 edges, 1 is source, 7 is target, 3 intermediate nodes
3 4 5           // 3, 4 and 5 are intermediate nodes.
1 2 5
2 1 1
2 3 15
2 5 10
5 6 5
6 7 10
3 7 5
6 3 15
4 3 5
3 4 3
1 4 5
4 5 10
4 6 20
1 5 10
```

Sample Output

```
38
```

Output Details

```
1 -> 4 -> 3 -> 4 -> 5 -> 6 -> 7
```

Solution Idea

This is a Minimum Cost Path problem in a directed graph via given set of intermediate nodes. This problem can be solved with 1) enumerating all possible paths with DFS, 2) Dijkstra and permutation of K nodes or 3) with a modified BFS. Here we discuss the BFS solution.

BFS is generally used to find all minimum distances of all nodes from the source node in an unweighted graph. Since there is a unique distance for any node, we can store all distances in a *distances[N]* array. We have to improve the method to store the minimum distances for this problem. We need to store 2^K

different distances for any node. Because, there are 2^K different subsets of the set X. We update the distance for any subset whenever we have an alternative shorter path.

The subsets for each node can be maintained with a map of sets *map<set<int>, int>* (min distance for every set) in $(O(V+E)Klg2^K)$ time, or with bit masking *map<int, int>* in $(O(V+E)lg2^K)$ time. Using bit masking technique saves us from comparing sets every time. Instead of sets, we only compare the integers.

An integer is just a bunch of bits stringed together. In bit masking, the idea is to visualize a number in the form of its binary representation. Some bits are *set* and some are *unset*, *set* means its value is 1 and *unset* means its value is 0.

A *Bitmask* is simply a binary number that represents something. *K* is the number of elements in our set. Then, if we write the binary representation of all numbers from 0 to 2^K-1, we get all the possible combinations of selecting K items.

```
//BFS with Bitmasking
#include <fstream>
#include <vector>
#include <map>
#include <queue>
#include <algorithm>

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M, K, source, target;
vector < vector<pair<int, int>>> adjList;
vector<bool> xNodes;   //intermediate nodes to be visited.
//-----------------------------------------------
void readInput()
{
	cin >> N >> M >> source >> target >> K;
	xNodes.resize(N + 1, false);
	for (int i = 0; i < K; i++)
	{
		int u;
		cin >> u;
		xNodes[u] = true;
	}

	adjList.resize(N + 1);
	for (int i = 0; i < M; i++)
	{
		int u, v, w;
		cin >> u >> v >> w;

		adjList[u].push_back({ v, w });
	}
}
//-----------------------------------------------
int setABit(int n, int i)
{
	int mask = 1 << i;
	return (n | mask);
}
//-----------------------------------------------
int bfs()
{
	const int BIGNUM = 1000000;

	vector<map<int, int>>  mapVec(N + 1); //set of visited intermediate nodes and cost

	queue<pair<int,int>> q;  //current cow and set of visited intermediate nodes
	q.push({ source, 0 });
```

```
	while (!q.empty())
	{
		int u = q.front().first;
		int mask = q.front().second;
		q.pop();

		for (auto p : adjList[u])
		{
			int v = p.first, w = p.second;
			if (xNodes[v])   //v is an intermediate node
			{
				int newMask = setABit(mask, v); //add v into the visited nodes

				//first visit for the newMask
				if (mapVec[v].count(newMask) == 0)
				{
					mapVec[v][newMask] = mapVec[u][mask] + w;
					q.push({ v, newMask });
				}
				//already visited with the newMask
				else if (mapVec[v][newMask] > mapVec[u][mask] + w)
				{
					mapVec[v][newMask] = mapVec[u][mask] + w;
					q.push({ v, newMask });
				}
			}
			else //v is not an intermediate node
			{
				//first visit for this mask
				if (mapVec[v].count(mask) == 0)
				{
					mapVec[v][mask] = mapVec[u][mask] + w;
					q.push({ v, mask });
				}
				//already visited with this mask
				else if (mapVec[v][mask] > mapVec[u][mask] + w)
				{
					mapVec[v][mask] = mapVec[u][mask] + w;
					q.push({ v, mask });
				}
			}
		}
	}

	//mask for all intermediate nodes
	int mask = 0;
	for (int i = 1; i <= N; i++)
		if (xNodes[i])
			mask = setABit(mask, i);

	if (mapVec[target][mask] == BIGNUM)
		return -1;
	return mapVec[target][mask];
}
//-----------------------------------------------
int main()
{
	readInput();
	cout <<bfs()<<endl;
}
```

Useful Links and References

https://www.interviewbit.com/tutorial/breadth-first-search/
https://www.geeksforgeeks.org/minimum-possible-modifications-in-the-matrix-to-reach-destination/
https://www.thecshandbook.com/Flood_Fill
https://leetcode.com/problems/cheapest-flights-within-k-stops/
https://www.techiedelight.com/least-cost-path-digraph-source-destination-m-edges/
https://www.geeksforgeeks.org/shortest-path-exactly-k-edges-directed-weighted-graph/
https://en.wikipedia.org/wiki/Complement_graph
https://www.tutorialspoint.com/complement-of-graph
https://blog.bitsrc.io/the-art-of-bitmasking-ec58ab1b4c03
https://www.geeksforgeeks.org/minimum-cost-path-in-a-directed-graph-via-given-set-of-intermediate-nodes/