Laboratory work # 7

Student: *Zhanghao Chen*
Student ID: 21321205
Timus Name: *BennyChan*

Mail: 1824362950@qq.com

Problem # 1080 *Map Coloring*

Screenshot from Timus:

ID	Date	Author	Problem	Language	Judgement result	Test #	Execution time	Memory used
10297263	07:14:21 27 May 2023	BennyChan	1080. Map Coloring	G++ 9.2 x64	Accepted		0.015	364 KB

Explanation of algorithm:

I use the Breadth-First Search (BFS) algorithm to determine if a graph is bipartite. By traversing each node in turn, it is divided into two different sets with its adjacent nodes, and it is determined whether there are edges between nodes in the same set. If there is, then the graph is not bipartite; on the other hand, if all nodes can be divided into a bipartite graph, then the graph is bipartite.

Computational complexity of algorithm:

$$O(N^2)$$

Source code:

```
#include<iostream>
#include<queue>
#include<cstring>
using namespace std;

const int N = 110;
int c[N];
int map[N][N];
int pay[N];
queue<int> q;
int n;

int BFS(int x){
    pay[x] = 0;
    q.push(x);
    while(!q.empty()){
        int v = q.front();
        q.pop();
        for(int i = 1; i <= n; i++){
            if(map[v][i] == 1){
```

```
                if(pay[i] == -1){
                    pay[i] = 1 - pay[v];
                    q.push(i);
                }else{
                    if(pay[i] == pay[v]) {
                        return 0;
                    }
                }
            }
        }
    }
    return 1;
}

int main() {
    cin >> n;
    memset(pay, -1, sizeof(pay));
    for(int i = 1; i <= n; i++){
        int k;
        while(cin >> k){
            if(k == 0) break;
            map[i][k] = 1;
            map[k][i] = 1;
        }
    }

    while(!q.empty()){
        q.pop();
    }
    int k = BFS(1);
    if(k == 0){
        cout << "-1";
    }else{
        for(int i = 1; i <= n; i++){
            cout << pay[i];
        }
        cout << endl;
    }
    return 0;
}
```

Problem # 1806 *Mobile Telegraphs*

Screenshot from Timus:

ID	Date	Author	Problem	Language	Judgement result	Test #	Execution time	Memory used
10297285	07:58:27 27 May 2023	BennyChan	1806. Mobile Telegraphs	G++ 9.2 x64	Accepted		0.25	8 392 KB

Explanation of algorithm:

I use hash tables and Dijkstra's algorithm to find the shortest path from the first word to the last word.

Firstly, the words are stored in a hash table, using a hash function to map them to a bucket. The implementation of Dijkstra's algorithm uses an array to save the distance between nodes and uses a minimum heap to choose the next shortest path.

In each iteration, the algorithm uses a current word, and then considers all the words that can be transformed into other words by changing one letter. These words are used to update the distance array and predecessor array. The algorithm repeats this process until it reaches the last word or cannot reach it.

Computational complexity of algorithm:

$$O(N^2)$$

Source code:

```
#include<stdio.h>
#include<string.h>
#define INF 0x3f3f3f3f

int number_of_words, destination, letter_cost[15], head[1000005], next[50015];
char words[50010][15], current_word[15];
int distances[50010], predecessor[50015], tree[4 * 50015];

int hash(char* str)
{
    int i, h = 0, seed = 131;
    while (*str)
        h = h * seed + *(str++);
    return (h & 0x7fffffff) % 1000003;
}
void insert_into_hash_table(int s)
{
    int h = hash(words[s]);
    next[s] = head[h];
    head[h] = s;
}
int search_word_in_hash_table(char* str)
{
    int i, h = hash(str);
    for (i = head[h]; i != -1; i = next[i]) {
        if (strcmp(words[i], str) == 0) {
            break;
        }
    }
    return i;
}
void initialize()
{
    int i;
    for (i = 0; i < 10; i++) {
        scanf("%d", &letter_cost[i]);
    }
    memset(head, -1, sizeof(head));
    for (i = 1; i <= number_of_words; i++)
    {
        scanf("%s", words[i]);
        insert_into_hash_table(i);
    }
}
void update(int i)
{
    for (; i ^ 1; i >>= 1)
        tree[i >> 1] = distances[tree[i]] < distances[tree[i ^ 1]] ? tree[i] : tree[i ^
 1];
}
```

```
void swap_characters(char& x, char& y)
{
    char temp;
    temp = x, x = y, y = temp;
}
void depth_first_search(int current_word_index, int n)
{
    if (current_word_index == 1)
    {
        printf("%d\n%d", n, current_word_index);
        return;
    }
    depth_first_search(predecessor[current_word_index], n + 1);
    printf(" %d", current_word_index);
}
void find_shortest_path()
{
    int i, j, x, y;
    for (destination = 1; destination < number_of_words + 2; destination <<= 1);
    memset(tree, 0, sizeof(tree));
    memset(distances, 0x3f, sizeof(distances));
    distances[1] = 0, predecessor[1] = 0, tree[destination + 1] = 1, update(destination + 1);
    while (x = tree[1])
    {
        strcpy(current_word, words[x]);
        tree[destination + x] = 0, update(destination + x);
        for (i = 0; i < 10; i++)
            for (j = '0'; j <= '9'; j++) {
                if (j != current_word[i])
                {
                    current_word[i] = j;
                    y = search_word_in_hash_table(current_word);
                    if (y != -1 && distances[x] + letter_cost[i] < distances[y])
                        distances[y] = distances[x] + letter_cost[i], predecessor[y] = x, tree[destination + y] = y, update(destination + y);
                    current_word[i] = words[x][i];
                }
            }
        for (i = 0; i < 10; i++) {
            for (j = i + 1; j < 10; j++) {
                if (current_word[i] != current_word[j])
                {
                    swap_characters(current_word[i], current_word[j]);
                    y = search_word_in_hash_table(current_word);
                    if (y != -1 && distances[x] + letter_cost[i] < distances[y]) {
                        distances[y] = distances[x] + letter_cost[i], predecessor[y] = x, tree[destination + y] = y, update(destination + y);
                    }
                    swap_characters(current_word[i], current_word[j]);
                }
            }
        }
           }
           if (distances[number_of_words] == INF) {
               printf("-1\n");
           }
           else
           {
               printf("%d\n", distances[number_of_words]);
               depth_first_search(number_of_words, 1);
               printf("\n");
           }
       }
       int main()
       {
           scanf("%d", &number_of_words) == 1;
           initialize();
           find_shortest_path();
```

```
116.            return 0;
117.        }
```

Problem # 1450 *Russian Pipelines*

Screenshot from Timus:

10297271	07:36:09 27 May 2023	BennyChan	1450. Russian Pipelines	G++ 9.2 x64	Accepted	0.046	2 272 KB

Explanation of algorithm:

The code is an implementation of the **single-source shortest path** problem for **a directed graph with weighted edges**. The graph is represented using **an adjacency list**.

I use dynamic programming to avoid recalculating the same node, and to store the solution of each node in a dp[] array. It returns the longest path length from the current position to the destination through the 'solve(pos)' function. If the solution for the current position has been calculated before, it returns it. Otherwise, it explores all adjacent nodes of the current position and calculates the longest path to the destination for each node. It chooses the maximum value among these paths and adds the weight of the current edge to it. Finally, it stores this value in the dp[] array and returns it. If there is no solution, it outputs "No solution." Otherwise, it outputs the longest path length from the starting point to the destination. The input of this algorithm is the number of nodes '*n*', the number of edges '*m*', and '*m*' triples '(*u*, *v*, *w*)', which represent a weighted directed edge from node '*u*' to node '*v*' with weight '*w*'. It also requires the starting point and the destination as input.

There are other algorithms for solving the single-source shortest path problem such as "Dijkstra's algorithm" and "Bellman-Ford algorithm".

Dijkstra's algorithm is used for finding the shortest path between a source node and all other nodes in a graph with non-negative edge weights. It uses a priority queue to select the next node with the smallest distance from the source node.

Bellman-Ford algorithm can handle graphs with negative edge weights and can detect negative cycles. It works by relaxing the edges repeatedly to find the shortest path.

The choice of algorithm depends on the properties of the graph and the requirements of the problem. If the graph has non-negative edge weights, Dijkstra's algorithm is more efficient. If the graph has negative edge weights or the presence of negative cycles needs to be detected, Bellman-Ford algorithm is preferred.

Computational complexity of algorithm:

$$O(N \times M)$$

N is the number of nodes and M is the number of edges.

Source code:

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

using namespace std;

vector<int> adjList[501], weight[501];
int dp[501];
int startNode, endNode;

int solve(int pos){
    if(pos == endNode) return 0;

    int &ret = dp[pos];

    if(ret == -1){
        ret = -2;

        for(int i = adjList[pos].size() - 1;i >= 0;--i){
            int aux = solve(adjList[pos][i]);

            if(aux != -2){
                aux += weight[pos][i];
                ret = max(ret,aux);
            }
        }
    }

    return ret;
}

int main(){
    ios::sync_with_stdio(0);

    int n,m;

    cin >> n >> m;

    for(int i = 0,u,v,w;i < m;++i){
        cin >> u >> v >> w;

        adjList[u].push_back(v);
        weight[u].push_back(w);
    }

    memset(dp,-1,sizeof dp);

    cin >> startNode >> endNode;

    int ret = solve(startNode);
```

```
49.
50.     if(ret == -2) cout << "No solution\n";
51.     else cout << ret << '\n';
52.
53.     return 0;
54. }
```