DDA 6050: Homework #4

Due on Dec 7, 2021

Professor Yixiang Fang

Haoyu Kang

Problem 1

String Matching

For this question, we use KMP algorithm. In this algorithm, we should design *next* function in advance. Given $P[1, \cdots, m]$, let *next* be a function $\{1, 2, \cdots, m\} \to \{0, 1, \cdots, m-1\}$ such that

$$next(q) = max\{k : k < q \quad and \quad p[1 \cdots k] \text{ is a suffix of } p[1 \cdots q]\} \tag{1}$$

Hence the optimal method of KMP just takes $O(m + n)$ time complexity and $O(n)$ space complexity.
The cpp code is attached as follow:

```cpp
#include<iostream>
#include<string>
#include<vector>
using namespace std;
vector<int> compute_next(string P){
    int m=P.size();
    vector<int>next(m,-1);
    int k=-1;
    for(int q=1;q<m;q++){
        while(k>-1 and P[k+1]!=P[q]) k=next[k];
        if(P[k+1]==P[q]) k=k+1;
        next[q]=k;
    }
    return next;
}
int KMP_StringMatcher(string T,string P){
    int n=T.size();
    int m=P.size();
    vector<int> next=compute_next(P);
    int q=-1;
    for(int i=0;i<n;i++){
        while(q>-1 and P[q+1]!=T[i])q=next[q];
        if(P[q+1]==T[i]) q=q+1;
        if(q==m-1) return i-m+1;
    }
    return -1;
}
int main(){
    string T,P;
    cin>>T;
    cin>>P;
    int index=KMP_StringMatcher(T,P);
    cout<<index<<endl;
}
```

Problem 2

Edit Distance

Problem 2 continued on next page...

The state transition equation is as below:

$$D[m,n] = \begin{cases} min(min(D[m-1,n]+1, D[m][n]+1), D[m-1][n-1]), & \text{If } W_1[m] = W_2[n] \\ min(min(D[m-1,n]+1, D[m][n]+1), D[m-1][n-1]+1), & else. \end{cases} \tag{2}$$

We also adopt method of state compression instead of a traditional approach that takes $O(n^2)$ memory. In details, we use **scrolling array** which is a one-demision array to save each states.
Hence the optimal method just takes $O(n)$ space complexity and $O(n^2)$ time complexity.

The cpp code is attached as follow:

```
#include<iostream>
#include<string>
#include<vector>
using namespace std;
int minDistance(string word1, string word2) {
    int n=word2.size();
    int m=word1.size();
    vector<int> dp(n+1,0);
    // for(int i=0;i<word1.size()+1;i++) dp[i][0]=i;
    for(int j=0;j<n+1;j++) dp[j]=j;
    int pre=0;
    for(int i=1;i<=m;i++){
        for(int j=0;j<=n;j++){
            int temp=dp[j];
            if(j==0){
                dp[j]=i;
            }
            else{
                int a= dp[j]+1;
                int b= dp[j-1]+1;
                int c=pre;
                if(word1[i-1]!=word2[j-1]) c++;
                dp[j]=min(a,min(b,c));
            }
            pre=temp;
        }
    }
    return dp[n];
}
int main(){
    string A,B;
    cin>>A>>B;
    int distance=minDistance(A,B);
    cout<<distance<<endl;
```

Problem 3

Critical Edges of Minimum Spanning Tree

The method I use is: Enumeration + Kruscal
In the Kruscal algorithm, we apply union-find to generate mininmum spanning tree. I also optimize union-find. In order to reduce the time cost of stage of *find*, in the stage of *union*, I add the rank of each node(represents the height of it as the root).

Hence the optimal method just takes $O(m^2 \cdot \alpha(n))$ time complexity and $O(m + n)$ space complexity.
The cpp code is attached as follow:

```cpp
#include<iostream>
#include<string>
#include<vector>
#include<set>
using namespace std;
int Find(int x, vector<int> uf){
    if(uf[x]!=x){
        uf[x]=Find(uf[x],uf);
    }
    return uf[x];
}
bool Union(int x, int y, vector<int> &uf, vector<int>&rank){
    int px=Find(x,uf);
    int py=Find(y,uf);
    if(px==py) return false;
    else if(rank[px]<rank[py]) uf[px]=py;
    else if(rank[px]>rank[py]) uf[py]=px;
    else
        {
            uf[py] = px;
            ++rank[px];
        }
    return true;
}
int main(){
    int n,m;
    cin>>n>>m;
    vector<vector<int>> edges;
    for(int i=0;i<m;i++){
        int s,t,w;
        cin>>s>>t>>w;
        vector<int> temp={s,t,w,i};
        edges.push_back(temp);
    }
    vector<int>uf(n+1);
    vector<int>rank(n+1);
    for(int i=1;i<n+1;i++) {
        uf[i]=i;
        rank[i]=1;
    }
    sort(edges.begin(), edges.end(), [](const vector<int>& a, const vector<int>& b)
    {
```

Problem 3 continued on next page...

```
            return a[2] < b[2];
    });
    int weights=0;
    vector<int> set;
    for(int i=0;i<m;i++){
        if(Union(edges[i][0],edges[i][1],uf,rank)) {
            weights+=edges[i][2];
            set.push_back(i);
        }
    }
    vector<int> res;
    for (int i = 0; i < m; ++i)
    {
        vector<int>uf1(n+1);
        vector<int>rank1(n+1);
        for(int i=1;i<n+1;i++) {
            uf1[i]=i;
            rank1[i]=1;
        }
        int w1=0;
        int n1=0;
        for (int j = 0; j < m; ++j)
        {
            if(i!=j && Union(edges[j][0],edges[j][1],uf1,rank1)) {
                w1+=edges[j][2];
                n1++;
            }
        }
        //
        if (n1 != n-1 ||  (n1==n-1 && w1 > weights))
        {
            res.push_back(edges[i][3]);
        }
    }
    sort(res.begin(),res.end());
    for(int i=0;i<res.size();i++) cout<<res[i]<<endl;
}
```

Problem 4

Minimum Cost to Connect Two Groups of Points

We convert this question to **Binary Graph problem** and use **KM algorithm**:
If all edge weights are non-negative, then the minimum weight set of edges that covers all the nodes automatically has the property that it has no three-edge paths, because the middle edge of any such path would be redundant. If we assign each vertex to an edge that covers it, some edges will cover both of their endpoints (forming a matching M) and others will cover only one of their endpoints (and must be the minimum weight edge adjacent to the covered endpoint). If we let c_v be the cost of the minimum weight edge incident to vertex v and w_e be the weight of e , then the cost of a solution:

Problem 4 continued on next page...

$$\sum_{v\in G} C_v + \sum_{u,v\in M} (w_{(u,v)} - C_u - C_v) \tag{3}$$

The first sum doesn't depend on the choice of the cover, so the problem becomes one of finding a matching that maximizes the total weight, for edge weights $C_u + C_v - W_{(u,v)}$. If you really want this to be a minimum weight perfect matching problem, then instead use weights $W_{(u,v)} - C_u - C_v$ and add enough dummy edges with weight zero to guarantee that any matching with the real edges can be extended to a perfect matching by adding dummy edges.
Hence the optimal method just takes $O(n \cdot m)$ space complexity and $O(n^2)$ time complexity.
The cpp code is attached as follow:

```cpp
#include <iostream>
#include <vector>
using namespace std;

#define MAXN 505

int link[MAXN], visx[MAXN], visy[MAXN],lx[MAXN],ly[MAXN];
int w[MAXN][MAXN];
int cost[MAXN][MAXN];
int n, m;
int MAX = 0xffffff;
int can(int t){
    visx[t] = 1;
    for(int i = 1; i <= m; i++){
        // "lx[t]+ly[i]==w[t][i"          ]
        if(!visy[i] && lx[t] + ly[i] == w[t][i]){
            visy[i] = 1;
            if(link[i] == -1 || can(link[i])){
                link[i] = t;
                return 1;
            }
        }
    }
    return 0;
}

int km(){
    int sum = 0;
    for(int i=0; i<=n; i++)
        ly[i] = 0;
    for(int i = 1; i <= n; i++){//      lxw[i][j   ]
        lx[i] = -MAX;
        for(int j = 1; j <= n; j++){
            if(lx[i] < w[i][j])
                lx[i] = w[i][j];
        }
    }
    for(int i=1; i<=n; i++)
```

Problem 4 continued on next page...

```
        link[i] = -1;
    for(int i = 1; i <= n; i++){
        while(1){
            for(int k=0; k<=n; k++){
                visx[k] = 0;
                visy[k] = 0;
            }
            if(can(i))//        break
                break;
            int d = MAX;//          , d
            //                    XYXSYTSxiTyj
            for(int j = 1; j <= n; j++)
                if(visx[j]){
                    for(int k = 1; k <= m; k++)
                        if(!visy[k])
                            d = min(d, lx[j] + ly[k] - w[j][k]);
                }
            if(d == MAX)
                return -1;//
            for (int j = 1; j <= n; j++){
                if (visx[j])
                    lx[j] -= d;
            }
            for(int j = 1; j <= m; j++){
                if(visy[j])
                    ly[j] += d;
            }
        }
    }
    for(int i = 1; i <= m; i++)
        if(link[i] > -1)
            sum += w[link[i]][i];
    return sum;
}

int connectTwoGroups() {
    int tn = n, tm = m;
    n = max(n, m);
   m = max(m, n);  //
    vector<int> lmin(tn + 1, MAX), rmin(tm + 1, MAX);
    for (int i = 1; i <= tn; ++i) {
        for (int j = 1; j <= tm; ++j) {
            lmin[i] = min(lmin[i], cost[i - 1][j - 1]);
            rmin[j] = min(rmin[j], cost[i - 1][j - 1]);
        }
    }
    int ans = 0;
    for(int i=1; i<=tn; i++)
        ans += lmin[i];
```

Problem 4 continued on next page...

```
    for(int i=1; i<=tm; i++)
        ans += rmin[i];
    for (int i = 1; i <= tn; ++i) {
        for (int j = 1; j <= tm; ++j) {
            w[i][j] = max(0 , lmin[i] + rmin[j] - cost[i - 1][j - 1]);
        }
    }
    return ans - km();
}
int main()
{
    cin >> n >> m;
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            cin >> cost[i][j];
        }
    }
    cout << connectTwoGroups() << endl;;
    return 0;
}
```

Problem 5

Find Strongly Connected Components Problem

For this question, we use **Kosaraju-sharir algorithm:** the transpose graph (the same graph with the direction of every edge reversed) has exactly the same strongly connected components as the original graph. The detailed step is as follow:
1. Run DFS and compute the finishing time (reverse post-order) of each node
2. Reverse the edge directions
3. Run DFS and consider vertices in the decreasing post-order
Hence the optimal method just takes $O(V+E)$ time complexity and $O(VE)$ space complexity.
The cpp code is attached as follow:

```
#include<iostream>
#include<string>
#include<vector>
#include<set>
#include<queue>
using namespace std;
void dfs(int node, vector<vector<int>> &M, vector<int> &V, int &ft,
priority_queue<pair<int,int>> &finish_time){
    V[node]=1;
    ft++;
    for(int i=0;i<M[node].size();i++){
        if(V[M[node][i]]) continue;
        dfs(M[node][i],M,V,ft,finish_time);
    }
    ft++;
    finish_time.push({ft,node});
}
void dfs_2(int node, vector<vector<int>> &M1, vector<int> &V1){
    V1[node]=1;
    for(int i=0;i<M1[node].size();i++){
        if(V1[M1[node][i]]) continue;
        dfs_2(M1[node][i],M1,V1);
    }
}
int main(){
    int n,m;
    cin>>n>>m;
    vector<vector<int>> M(n);
    vector<vector<int>> M1(n);
    for(int i=0;i<m;i++){
        int s,e;
        cin>>s>>e;
        M[s].push_back(e);
        M1[e].push_back(s);

    }
    vector<int>V(n,0);
    int ft=0;
    priority_queue<pair<int,int>> finish_time;
    for(int i=0;i<n;i++){
        if(V[i]) continue;
        dfs(i,M,V,ft,finish_time);
    }
    vector<int>V1(n,0);
    int num=0;
    while(!(finish_time.empty())){
        int node=finish_time.top().second;
        // cout<<finish_time.top().second<<" "<<finish_time.top().first<<endl;
        finish_time.pop();
        if(V1[node]) continue;
        num++;
        dfs_2(node,M1,V1);
    }
    cout<<num<<endl;

}
```