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 $O(m + n)$ space complexity and $O(n)$ time 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:

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

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:

```cpp
#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 $O(m^2 \cdot \alpha(n))$ space complexity and $O(m+n)$ time complexity.
The cpp code is attached as follow:

```
#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)
    {
```

```cpp
        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

LIS

Different from what we learn in class, I optimalize the algirithm with less memory and time cost. In this algorithm, we just creat one-demision array to save increasing sequence. In the i-th loop, if $nums[i] > back\ of\ dp$ put the nums[i] into the end of the array. Otherwise find the one which is larger than nums[i] from left, and replace it.

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

Problem 4 continued on next page...

```cpp
#include<iostream>
#include<vector>
using namespace std;
int main(){
    int n;
    cin>>n;
    vector<int> nums;
    int num;
    for(int i=0;i<n;i++) {
        cin>>num;
        nums.push_back(num);
    }
    if (n <= 1) return n;
    vector<int> dp;
    dp.push_back(nums[0]);
    for (int i = 1; i < n; ++i) {
        if (dp.back() < nums[i]) {
            dp.push_back(nums[i]);
        } else {
            /*      nums[i   ] */
            auto itr = lower_bound(dp.begin(), dp.end(), nums[i]);
            *itr = nums[i];
        }
    }
    cout<<(int)dp.size()<<endl;
}
```

Problem 5

Max M Sum Subsequences Problem

The state transition equation is as below:

$$DP[i,j] = \begin{cases} mk[i-1,j-1] + nums[j], & i == j \\ max(DP[i,j-1] + nums[j], mk[i-1,j-1] + nums[j]), & else. \end{cases} \tag{3}$$

In above equation, $DP[i,j]$ represents max i sum subsequnces of j preflix of sequence when the j-th number is in the i-th subsequence, and $mk[i,j]$ represents max i sum subsequnces of j preflix of the sequence. In order to compress states, we also adopt scrolling arrays to replace above two-demision arrays.

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

```cpp
#include<iostream>
#include<vector>
#include<cmath>
#include<cstring>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
```

Problem 5 continued on next page...

```
    int dp[n];
    int nums[n];
    // vector<int> nums(n,0);
    // vector<int> dp(n,0);
    for(int i=0;i<n;i++){
        int num;
        cin>>num;
        nums[i]=num;
    }
    int res;
    int max_sum;
    // vector<int> mk(n,0);
    int mk[n];
    memset(dp,0,sizeof(int)*n);
    memset(mk,0,sizeof(int)*n);
    for(int i=0;i<m;i++){
        max_sum=INT_MIN;
        for(int j=i;j<n;j++){
            if(j==i) {
                if(j==0) dp[j]=nums[j];
                else dp[j]=mk[j-1]+nums[j];
            }
            else {
                dp[j]=max(dp[j-1]+nums[j],mk[j-1]+nums[j]);
                mk[j-1]=max_sum;
            }
            max_sum=max(max_sum,dp[j]);
        }
    }
    cout<<max_sum<<endl;

}
```