Algorithm Homework 2

Jingwei Zhang 201528013229095

2015-10-13

1 Problem 1

1.1 Algorithm

Non-decreasingly sort all vertex according to its degree. For every vertex i link it to d_i vertices following. If this forms a graph, then these vertices can form a graph, if not, they can not form a graph.

Pseudo-code: D is the array storing the degree of every vertex. SORT(D) is a sorting procedure which sorts all integers in array D non-decreasingly.

```
WHETHER-FORM-GRAPH(D)
 1  n = D.length
 2  SORT(D)
 3  for i = 1 to n
 4       j = i + 1
 5       while D[i] > 0 and j ≤ n
 6            if D[j] > 0
 7                 D[j] − −
 8                 D[i] − −
 9                 j + +
10       if D[i] > 0
11            return Can-Not-Form-Graph
12  return Can-Form-Graph
```

1.2 Correctness

Proof. We will prove our strategy can get an undirected graph if and only if there exits a graph whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$.

If degrees in D can form a graph, non-decreasingly sort all vertex according to its degree. For each vertex v_i with degreed_i in non-decreasingly degree order, if it links to a vertex $v_j \notin v[i+1, i+d'](d'$ denotes next d' vertices that have exactly d_i vertices linked to $v[i+1, n])$, then there must exits a vertex $v_k \in v[i+1, i+d']$ that links to $v_p, p > i$. Then rearrange the linkages from :$v_i - v_j, v_k - v_p$ to $v_i - v_k, v_j - v_p$. By such procedure, a new graph that following our strategy can be formed. Thus, our strategy can get an undirected graph.

If our strategy can get an undirected graph, it is obvious that there exits a graph whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. ■

1.3 Complexity

The program WHETHER-FORM-GRAPH(D) contains one sorting functions. If we use quick sort for sorting function, the time complexity of sort is $O(n \log n)$. The **for** loop in line 3 will run n times, the **for** loop in line 5 will run at most $n - i$ times. Totally $\sum_{i=1}^{n} \sum_{j=1}^{n-i} O(1) = O(n^2)$.

Thus, total time complexity is $O(n^2)$. This algorithm's space complexity is $O(n)$ for storage of array F.

2 Problem 4

2.1 Algorithm

We can prove that setting $a_i = a_{(i)}$ and $b_i = b_{(i)}$ will maximize the payoff. Thus, the algorithm is simply sort array A and B in non-decreasing order and summing all $a_i^{b_i}$.

Pseudo-code: A, B are sets each containing n integers. SORT(A) is a sorting procedure which sorts all integers in array A non-decreasingly.

```
MAXIMUM-PAYOFF(A, B)
1  n = A.length
2  SORT(A)
3  SORT(B)
4  payoff = 0
5  for i = 1 to n
6        payoff = payoff + A[i]^B[i]
7  return payoff
```

2.2 Correctness

Proof. Firstly we rearrange the elements in array A and B in non-decreasing order, then the problem is converted to equivalent one: all elements in A is fixed, find a permutation of B that maximize $p = \sum_{i=1}^{n} a_i^{b_i}$.

Then we will prove that for any two elements b_i and b_j in B, the non-decreasing order of them will maximize the sum $a_i^{b_i} + a_j^{b_j}$. The mathematical representation is: $\forall a_i, a_j \in Z^+$, $\forall b_i, b_j \in Z^+$, if $a_i < a_j$ and $b_i < b_j$, then $a_i^{b_i} + a_j^{b_j} > a_i^{b_j} + a_j^{b_i}$.

According to $a_i < a_j$ and $b_i < b_j$, we will get:

$$
\begin{aligned}
& a_j^{b_i} > a_i^{b_i} \\
& a_j^{b_j - b_i} > a_j^{b_j - b_i} \\
\Longrightarrow\ & a_j^{b_i}(a_j^{b_j - b_i} - 1) > a_i^{b_i}(a_j^{b_j - b_i} - 1) \\
\Longrightarrow\ & a_j^{b_j} - a_j^{b_i} > a_j^{b_j} - a_i^{b_i} \\
\Longrightarrow\ & a_i^{b_i} + a_j^{b_j} > a_i^{b_j} + a_j^{b_i}
\end{aligned}
$$

Finally, we prove that the payoff p^* generated by B^*, the non-decreasing order of B, is larger than payoff p' generated by any other permutation B' of array B. For any permutation $B' \neq B^*$, perform a bubble sort on B' to change B' to B_s with $O(n^2)$ times of swapping adjacent increasing elements. Suppose this process performs altogether m times of swap. According to the two elements situation above, each swap will generates a new sequence $B'_k (k = 1, \ldots, m)$ with a greater payoff $p'_k (k = 1, \ldots, m)$. After m times of swap, we get $B'_m = B^*$ with payoff $p'_m = p^* > p'$. ■

2.3 Complexity

The program MAXIUM-PAYOFF(A, b) contains two sorting functions and a scan from 1 to n. If we use quick sort for sorting function, then its time complexity is $O(n \log n)$.

3 Problem 6

3.1 Result

The program is compiled using g++ with -O3 parameter. Blow is the running time of Dijkstra's shortest path algorithm using different data structures. It runs on graph.txt with 20000 nodes. Algorithm starts from node 6676 to node 18853 and returns shortest path length 29.

Table 1: Running Time of Dijkstra's Shortest Path using Different Data Structures

Data structure	Linked List	Binary Heap	Binomial Heap	Fibonacci Heap
Running time(second)	1.22607	0.000602	0.027132	0.018106

3.2 C++ Code

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdio>
#include <cstdlib>
#include <vector>
#include <queue>
#include <algorithm>
#include <ctime>
#include <climits>
#include <chrono>
#include <boost/timer.hpp>
#include <boost/heap/d_ary_heap.hpp>
#include <boost/heap/binomial_heap.hpp>
#include <boost/heap/fibonacci_heap.hpp>

using namespace std;
using namespace boost::heap;

const string file_name("graph.txt");
const int INF = INT_MAX;

struct Edge{
    int s; // start
    int t; // tail
    int w; // weight
    Edge(){}
    Edge(int ss,int tt,int ww):s(ss),t(tt),w(ww){}
};

vector< vector<Edge> > get_graph(){
    vector< vector<Edge> > graph;
    ifstream s_file;
    s_file.open(file_name);
    string str;
    int s,t,w;
    while(getline(s_file,str)){
        if(str.size()<=0 || str[0] == '#'){
            continue;
        } else {
            stringstream ss(str);
            ss>>s>>t>>w;
            while(graph.size() < s + 1 || graph.size() < t + 1){
                graph.push_back(vector<Edge>());
            }
            graph[s].push_back(Edge(s,t,w));
            graph[t].push_back(Edge(t,s,w));
        }
    }
```

```
    s_file.close();
    return graph;
}

struct Vertex{
    int v;
    int w;
    Vertex(){}
    Vertex(int vv,int ww):v(vv),w(ww){}
    bool operator<(Vertex const &a) const{
        return w>a.w;
    }
    bool operator>(Vertex const &a) const{
        return w<a.w;
    }
};

template<class Heap>
int Dijkstra_shortest_path(const vector< vector<Edge> > &graph,
                           int s, int t, Heap &heap,
                           bool store_pre, vector< vector<int> > &pre){
    int ans = 0;
    // initial part
    int n = graph.size();
    typedef typename Heap::handle_type handle_t;
    vector<handle_t> handles;
    handles.resize(n);
    vector<bool> vis;
    vis.resize(n);
    vector<int> dis; // distance from s
    dis.resize(n);
    heap.clear();

    // initial previous vertics
    if(store_pre){
        pre.clear();
        pre.resize(n);
        for(int i = 0;i < n;i++){
            pre[i] = vector<int>();
        }
    }

    // initial vertics weight
    // 0 for s and INF for others
    for(int i = 0;i<n;i++){
        const handle_t &h = heap.push(Vertex(i,i==s?0:INF));
        handles[i] = h;
        vis[i] = false;
        dis[i] = i==s?0:INF;
    }

    while(!heap.empty() && !vis[t]){
        const Vertex &v_top = heap.top();
        int u = v_top.v;
        vis[u] = true;
        heap.pop();
```

```
        // update all vertex v adjcent to u (u -> v)
        for(const Edge &e: graph[u]){
            int v = e.t;
            if(vis[v] == true){
                continue;
            }

            if(dis[v] - e.w > dis[u]){ // to prevent overflow
                // relax v
                handle_t &handle_v = handles[v];
                int new_dis = dis[u] + e.w;
                dis[v] = new_dis;
                heap.increase(handle_v, Vertex(v, new_dis));
                if(store_pre){
                    pre[v].clear();
                    pre[v].push_back(u);
                }
            } else if(store_pre && dis[v] - e.w == dis[u]) {
                // add pre
                pre[v].push_back(u);
            }

        }
    }
    ans = dis[t];
    return ans;
}

int Dijkstra_shortest_path_pq(const vector< vector<Edge> > &graph,
                              int s, int t){
    int ans = 0;
    // initial part
    int n = graph.size();
    vector<bool> vis;
    vis.resize(n);
    vector<int> dis; // distance from s
    dis.resize(n);
    std::priority_queue<Vertex> heap;

    // initial vertics weight
    // 0 for s and INF for others
    for(int i = 0;i<n;i++){
        heap.push(Vertex(i,i==s?0:INF));
        vis[i] = false;
        dis[i] = i==s?0:INF;
    }
    const Vertex &top = heap.top();

    while(!heap.empty() && !vis[t]){
        const Vertex &v_top = heap.top();
        int u = v_top.v;
        vis[u] = true;
        heap.pop();
        if(dis[u] < v_top.w){
            continue;
        }
```

```
        // update all vertex v adjcent to u (u -> v)
        for(const Edge &e: graph[u]){
            int v = e.t;
            if(vis[v] == true){
                continue;
            }
            if(dis[v] - e.w > dis[u]){ // to prevent overflow
                int new_dis = dis[u] + e.w;
                dis[v] = new_dis;
                heap.push(Vertex(v, new_dis));

            }
        }
    }
    ans = dis[t];
    return ans;
}

int Dijkstra_shortest_path_vector(const vector< vector<Edge> > &graph,
                                  int s, int t){
    int ans = 0;
    int n = graph.size();

    vector<int> v;
    vector<bool> vis;
    for(int i = 0;i < n;i++){
        v.push_back(INF);
        vis.push_back(false);
    }
    v[s] = 0;

    int left = n;
    while(left > 0){
        // Pick minium vertex(index) in array v
        int min_value = INF;
        int min_idx = -1;
        for(int i = 0;i<v.size();i++){
            if(!vis[i] && v[i] < min_value){
                min_value = v[i];
                min_idx = i;
            }
        }
        if(min_idx == t){ // t is already in SP
            break;
        }

        // relax all edges incident to this vertex
        int v_min = min_idx;
        vis[v_min] = true;
        left--;
        for(const Edge &e: graph[v_min]){
            if(!vis[t]){
                v[e.t] = min(v[e.t], v[e.s] + e.w);
            }
        }
    }
```

```
    ans = v[t];
    return ans;
}

// Problem 2
void counting_DFS(const vector<vector<int>> g,
                  int s,
                  vector<int> &before, vector<int> &after){
    // s -> v
    before[s]++;
    int ans = 0; // store the pathes after node s
    if(g[s].size() == 0) { // reaching the tail node
        ans = 1;
    }
    for(int v: g[s]){
        counting_DFS(g,v,before,after);
        ans += after[v];
    }
    after[s] = ans;
}

vector<int> counting_SP_on_nodes(const vector<vector<int>> &pre,
                                 int t){
    int n = pre.size();
    vector<int> ans(n, 0);
    vector<int> before(n, 0);
    vector<int> after(n, 0);

    counting_DFS(pre,t,before,after);

    for(int i = 0;i < n;i++){
        ans[i] = before[i] * after[i];
    }
    return ans;
}

int main()
{
    // Input, graph is a adjcency list
    const vector< vector<Edge> > &graph = get_graph();
    int V = graph.size();

    // Randomly choose starting vertex and ending vertex
    int s,t;
    srand(time(NULL));
    s = rand() % V;
    t = rand() % V;
    cout<<"starting from "<<s<<" to "<<t<<", all "<<V<<" vertics"<<endl;

    // Dijkstra's shortest path:
    vector< vector<int> > pre; // record the previous vertics
    // 1    linked list
    boost::timer timer_1;
    vector<double> time;
    time.push_back(timer_1.elapsed());
```

```
    int ans = Dijkstra_shortest_path_vector(graph,s,t);
    time.push_back(timer_1.elapsed());
    cout<<"array like: "<<ans<<"\t"
        <<time[time.size()-1] - time[time.size()-2]<<"seconds"<<endl;

    // 2) binary heap
    time.push_back(timer_1.elapsed());
    int ans_b_heap = Dijkstra_shortest_path_pq(graph,s,t);
    time.push_back(timer_1.elapsed());
    cout<<"binary heap: "<<ans<<"\t"
        <<time[time.size()-1] - time[time.size()-2]<<"seconds"<<endl;


    // 3) binomial heap
    binomial_heap<Vertex> bin_heap;
    time.push_back(timer_1.elapsed());
    int ans_bin_heap = Dijkstra_shortest_path(graph,s,t,bin_heap, false, pre);
    time.push_back(timer_1.elapsed());
    cout<<"binomial heap: "<<ans_bin_heap<<"\t"
        <<time[time.size()-1] - time[time.size()-2]<<"seconds"<<endl;

    // 4) fibonacci_heap
    fibonacci_heap<Vertex> fib_heap;
    time.push_back(timer_1.elapsed());
    int ans_fib_heap = Dijkstra_shortest_path(graph,s,t,fib_heap, false, pre);
    time.push_back(timer_1.elapsed());
    cout<<"fibonacci heap: "<<ans_fib_heap<<"\t"
        <<time[time.size()-1] - time[time.size()-2]<<"seconds"<<endl;

    // problem 2 counting pathes
    Dijkstra_shortest_path(graph,s,t,fib_heap, true, pre);
    vector<int> num = counting_SP_on_nodes(pre,t);
    for(int i = 0;i<num.size();i++){
        if(num[i] > 0 && i != s && i != t){
            cout<<"\tnode "<<i<<": "<<num[i]
                <<" path"<<(num[i]>1?"es":"")<<endl;
        }
    }
    return 0;
}
```