Anna Sun, Chen Bai, Shiyu Wang

Algorithm PS 3

1.

Proof :

The sufficiency:

First, construct the Algorithm for a closed trail

(1) Put $i = 1$.

(2) Put $W = \{ x_0, x_1\}$. (W is the initial set of vertices)

(3) Put $F = \{\alpha_1\}$. (F is the initial set of edges)

(4) While $x_i \neq x_0$, do the following:

a. Locate an edge $\alpha_{i+1} = \{x_i, x_{i+1}\}$ not in F.

b. Put x_{i+1} in W (x_{i+1} may already be in W).

c. Put α_{i+1} in F.

d. Increase i by 1.

Let $G = (V, E)$ be a general graph and assume that the degree of each vertex is even.

After the initialization in (1) - (3), suppose that an edge α_{i+1} satisfying (4)(a) exists while $x_i \neq x_0$. Let the terminal value of i be n, giving the set $W = \{ x_0, x_1, ..., x_n\}$ and the set $F = \{\alpha_1, ..., \alpha_n\}$. Thus, $\{\alpha_1, ..., \alpha_n\}$ is a closed trail containing the initial edge α_0.

At the end of each step (4)(d), each vertex of $G' = (W, F)$ has even degree, except for the vertex x_0 which starts out with degree 1 and the most recent new vertex x_i whose degree has just been increased by 1. x_0 and x_i would have even degree iff $x_0 = x_i$. Thus, if $x_i \neq x_0$, x_i has odd degree in G'. Because x_i has even degree in G, there must be an edge $\alpha_{i+1} = \{x_i, x_{i+1}\}$ not yet in F. Thus, when $x_n = x_0$, $\{\alpha_1, ..., \alpha_n\}$ would be a closed trail.

So each edge of G belongs to a closed trail and hence to a cycle.

It's known to us all that, if G has a closed Eulerian trail, then each vertex has even degree.

Let $G_1 = (V, E_1)$ be the graph G. We can choose an edge α_1 of G_1 and apply the algorithm for a closed trail, we would obtain a closed trail T_1 containing α_1.

Let $G_2 = (V, E_2)$ be the general graph obtained by removing the edges belong to T_1 from E_1. All vertices have even degree in G_2. Since G_1 is connected, if E_2 contains at least one edge, there must be an edge α_2 of G_2 that is incident with a vertex v_1 on the closed trail T_1. We apply the algorithm for a closed trail to G_2 and α_2. We would obtain a closed trail T_2 containing α_2. Now, we patch T_1 and T_2 together at the vertex v_1. Then, we obtain a closed trail $T_1 * T_2$ that includes all the edges of T_1 and T_2.

Let $G_3 = (V, E_3)$ be the general graph obtained by removing the edges belong to T_2 from E_2. If E_3 contains at least one edge, there must be an edge α_3 of G_3 that is incident with a vertex v_2 on the closed trail $T_1 * T_2$. We apply the algorithm for a closed trail to G_3 and α_3. We would obtain a closed trail T_3 containing α_3. Now, we patch $T_1 * T_2$ and T_3 together at the vertex v_2. Then, we obtain a closed trail $T_1 * T_2 * T_3$ that includes all the edges of T_1, T_2 and T_3.

We continue doing like this until all edges in the graph G have been included in a closed trail $T_1 * T_2 * T_3 * \cdots * T_n$. Finally, repeated calls to the algorithm for a closed trail would give an algorithm to construct a closed Eulerian trail in a connected general graph, each vertices of the graph has even degree.

Thus, if the graph G is connected and all vertices of G have even degree, there must be an Eulerian Path in G.

So if the graph is connected and there exists no vertices of odd degree, the drawing can be traced. And each edge is counted once, which takes O(E).

The necessity：

Let $G = (V, E)$ be a general graph and $V = \{V_1, V_2, \cdots, V_n\}$.

$\forall V_i$ $(i \in [1, n])$, $I_{(V_i)}$ represents the in-degree of V_i and $O_{(V_i)}$ represents the out-degree of V_i. At the begining, $I_{(V_i)} = O_{(V_i)} = 0$.

Now, assume we pick any vertex V_k. If there is an edge between V_k and V_j, we walk from V_k to V_j. After that, we set $O_{(V_k)} = 1$ and $I_{(V_j)} = 1$. We continue doing like this until we walk through all edges of G without repeating any edge twice and return to V_k. For each vertex, one in is incident with one out. Since both the starting and ending vertices are V_k. So when we finished the Eulerian Path, $I_{(V_i)} = O_{(V_i)}$ for any vertex V_i.

Thus, if there is an Eulerian Path in the graph G, G must be connected and all vertices of G have even degree.

So if the drawing can be traced, the graph is connected and there exist no vertices of odd degree.

2.

The data structure supports the following three operations:

• MAKESET(x) creates a new set {x} and adds it to the family. The element x must not be an element of any existing set in the family.

• UNION(x, y) changes the family by replacing two sets, the one containing x and the one containing y, by a single set that is the union of these two sets. It is a no-op if x and y are already in the same set.

• FIND(x) returns the label of the set containing x.

For Kruskal's algorithm, data structure supports can speed up the checking of cycle in the loop. We first perform a MAKESET for each vertex of the graph. Then for each edge (u, v) we do a FIND for both u and v, followed by UNION(u, v) if they are not already in the same set. And using "union by rank" to mark leading vertices in sets. This can be done in linear time. Path compression, is a way of flattening the structure of the tree whenever *Find* is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative. Therefore, we can use path compression to mark root vertices in order to alwasys bind the small set into big set to save time. However, all of these algorithm can only save time to check cycle. We still need O(e log e) time to sort the edges. So the complexity of Kruskal's algorithm cannot be improved.

3.

1. We label the vertices of P by 1 to n, starting from an arbitrary vertex and going clockwise.
2. For 1<= i <= j <= n, we denote a subprobem that A(i,j) which computes the minimum cost triangulation of the polygon spanned by vertices i to j.
3. Formulateing the subproblem this way we are enumerating all the possible diagonals such that diagonals do not cross each other. Since we are only considering in clockwise directions, diagonals generated by the subproblems can not cross each other.
4. Recursively define $A[i,j] = \min_{i<=k<=j}\{A[i,k] + A[k,j] * d_{i,k} * d_{k,j}\}$
5. For d,
    a. if j - i >= 2, $d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
    b. otherwise 0

Because We just look at all possible triangles and leftover smaller polygons and pick the configuration that minimizes the volume. And the number of triangles are n -2, it takes θ(n) time.

4.

1. Assume Lan[i][0] stands for this node is in the set of "minimum cover nodes". Lan[i][1] stands for this node is NOT in the set of "minimum cover nodes".
2. When Lan[i] is not a leaf, we do following:
    a. Lan[i][0] = Lan[j1][1] + Lan[j2][1] + …….. + Lan[jm][1] (j1 to jm are all hosts under i)
    b. Lan[i][1] = min(Lan[j1][0],Lan[j1][1] + min(Lan[j2][0], Lan[j2][1]..... + min(Lan[jm][0], Lan[jm][1](j1 to jm are all hosts under i)
3. When Lan[i] is a leaf

a. Lan[i][0] = 0
b. Lan[i][1] = 1

4. Finally we run min(Lan[root][0], Lan[root][1])) to get the result. which is the lowest cost deployment of software to cover the entire LAN. the time complextiry should be O(n).

5.

Hashing: hash functions maps keys "randomly" into slots of table.

Collision: When a record to be inserted maps to an already occupied slot, a collision occurs. In order to make all keys fit into slots, we need to resolve collision by chaining or open addressing.

number of collisions:

Because we choose the map h uniformlay at random. Each key k $\in$ 2^n is equally likely to be hashed to any slots in 2^m, independent of where other keys were hashed. So the probability of each collision is always $1/2^m$, the the number of collision is expected to be $2^n/2^m = 2^{n-m}$.

The worst-case is every key hashes to same slot, access takes O(n) time. Therefore, it can't be O(1) in the worst-case. However, in average case, because n < m. and number of collision is 2^{n-m}, which is smaller than 1. so the tme of acces is just O(1). And we can easily search (go over all slots only once) or insert (randomly insert) or delete (go over only once and delete it). They all take linear time.

6.

let $x = \langle x_o, x_1, \ldots x_k \rangle$

$y = \langle y_o, y_1, \ldots y_k \rangle$ be distinct keys.

They differ k at least one digit. Without loss of generality, we assume it is position 0. The number $h_r \square H$ that x, y collide must have $h_r(x) = h_r(y)$.

$$\Rightarrow \sum_{i=0}^{k} r_i(x_i - y_i) \equiv 0 \ (mod\ p)$$

$$\Rightarrow r_0(x_0 - y_0) + \sum_{i=1}^{k} r_i(x_i - y_i) \equiv 0 \ (mod\ p)$$

$$\Rightarrow r_0(x_0 - y_0) \equiv -\sum_{i=1}^{k} r_i(x_i - y_i) \ (mod\ p)$$

Since $x_o \neq y_o$, $\exists$ $(x_o - y_o)^{-1}$ (Number theory fact)

$$\Rightarrow a_0 \equiv (-\sum_{i=1}^{k} r_i(x_i - y_i))\ (x_0 - y_0)^{-1} \ (\text{mod p})$$

Thus, for any choice of a_1, a_2…..a_r, exactly 1 of the p choices for a_o cause x , y to collide, and no collision for other m -1 choices for a_o.
number of h_a's that cause x, y to collide = m * m * m ……. * m * 1 = m^r = H*k/m

stands for:$(a_1)(a_2)(a_3)$..... $(a_k)(a_0)$

=> m^k/m^{k+1} = 1/m. Hence H is universal

---

7. For the problem, we need to find the minimum number of routers to install monitoring software to monitor all links between company routers and ISP routers. It is a minimum vertex cover problem.

According to König's theorem, there is an equivalence between the maximum matching problem and the minimum vertex cover problem in bipartite. Thus, we can solve this problem by converting it into a flow network.

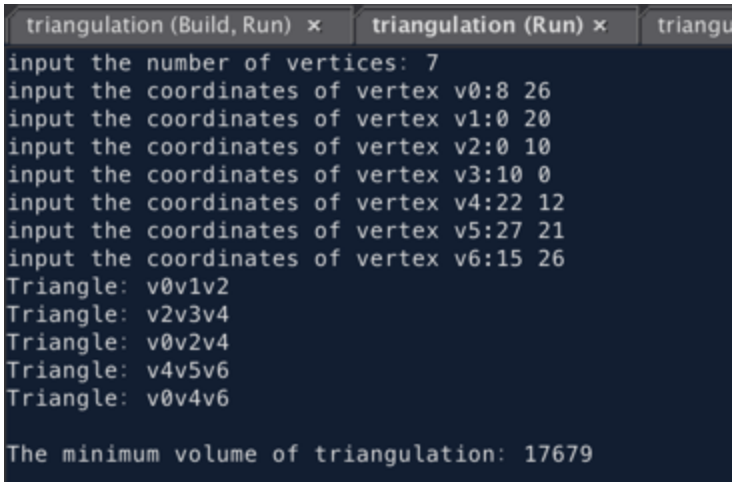
Following are the steps.

a. Construct a bipartite graph using the company's routers and the ISP's routers.
b. Add a source and add edges from source to all company's routers. Similarly, add edges from all ISP's routers to sink. The capacity of every edge is marked as 1 unit.
c. Use Ford-Fulkerson algorithm to find the maximum flow in the flow network built in step b. The maximum flow is actually the maximum bipartite matching we are looking for.

The minimum number of routers to install monitoring software is twice as much the number of matching obtained in step c. We can install the monitoring software on all routers of those matching links so that every link will be monitored.

We use Ford-Fulkerson algorithm to find the maximum flow in step c, the time complexity of this algorithm is O(Ef) where E is the number of edges in the graph and f is the maximum flow in the graph.

8.
Following is the running result:



Triangle.h:

```
#ifndef TRIANGLE_H
#define TRIANGLE_H
typedef struct
{
   int x;
   int y;
}point;

class CTriangle
{
   public:
      bool Run();
      CTriangle();
      virtual ~CTriangle();

   private:
      void Traceback(int i,int j,int **s); //outout each triangle
      bool minVolumeTriangulation(); //min volume triangulation
      bool Input();              //handle the input and determine whether or not form a polygon
      int volume(point X,point Y,point Z);   //compute the volume of a triangle
      int distance(point X,point Y);    //compute the length from a vertex to another
      int **s;                  //record all triangles of the min volume triangulation
      int **t;                   //record the volumes of those triangles
      point *v;                  //record the coordinates of the polygon's vertices
      int *total;                  //record the coordinates value in the equation
      int M;
};
#endif  /* TRIANGLE_H */
```

Triangle.cpp:

```
#include <iostream>
#include <cmath>
#include <stdlib.h>
#include "Triangle.h"
using namespace std;

#define N 50

CTriangle::CTriangle()
{
   M = 0;
   t = new int *[N];
   s = new int *[N];
```

```
    for(int i=0 ; i<N ; i++)
    {
        t[i] = new int[N];
        s[i] = new int[N];
    }
    v = new point[N];
    total = new int[N];
}

CTriangle::~CTriangle()
{
    for(int i=0 ; i<N ; i++)
    {
        delete []t[i];
        delete []s[i];
    }
    delete []t;
    delete []s;
    delete []v;
    delete []total;
}

int CTriangle::distance(point X, point Y)
{
    int dis = (Y.x-X.x)*(Y.x-X.x) + (Y.y-X.y)*(Y.y-X.y);
    return (int)sqrt(dis);
}

int CTriangle::volume(point X, point Y, point Z)
{
    return distance(X,Y) * distance(Y,Z) * distance(Z,X);
}

bool CTriangle::Input()
{
    int m;
    int a,b,c;
    cout<<"input the number of vertices : ";
    cin>>m;
    M = m-1;
    for(int i=0 ; i<m ; i++)
    {
        cout<<"input the coordinates of vertex v"<<i<<":";
```

```
    cin>>v[i].x>>v[i].y;
}

//determine whether or not form a polygon according to the coordinates
for(int j=0 ; j<m ; j++)
{
    int p = 0;
    int q = 0;
    if(m-1 == j)
    {
        a = v[m-1].y - v[0].y;
        b = v[m-1].x - v[0].y;
        c = b * v[m-1].y - a * v[m-1].x;
    }
    else
    {
        a = v[j].y - v[j+1].y;
        b = v[j].x - v[j+1].x;
        c = b * v[j].y - a * v[j].x;
    }

    for(int k=0 ; k<m ; k++)
    {
        total[k] = a * v[k].x - b * v[k].y + c;
        if(total[k] > 0)
        {
            p = p+1;
        }
        else
            if(total[k] < 0)
            {
                q = q+1;
            }
    }
    if((p>0 && q>0) || (p==0 && q==0))
    {
        cout<<"Cannot form a polygon ! "<<endl;
        exit(1);
    }
}

if(NULL != v)
    return true;
```

```
    else
        return false;
}

bool CTriangle::minVolumeTriangulation()
{
    if(NULL == v)
        return false;

    for(int i=1 ; i<=M ; i++)
        t[i][i] = 0;
    for(int r=2 ; r<=M ; r++)
        for(int i=1 ; i<=M-r+1 ; i++)
        {
            int j = i+r-1;
            t[i][j] = t[i+1][j] + volume(v[i-1],v[i],v[j]);
            s[i][j] = i;
            for(int k=i+1 ; k<i+r-1 ; k++)
            {
                int u = t[i][k] + t[k+1][j] + volume(v[i-1],v[k],v[j]);
                if(u < t[i][j])
                {
                    t[i][j] = u;
                    s[i][j] = k;
                }
            }
        }
    return true;
}

void CTriangle::Traceback(int i, int j, int **s)
{
    if(i == j)
        return;
    Traceback(i,s[i][j],s);
    Traceback(s[i][j]+1,j,s);
    cout<<"Triangle : v"<<i-1<<"v"<<s[i][j]<<"v"<<j<<endl;
}

bool CTriangle::Run()
{
    if(Input())
    {
```

```
        if(CTriangle::minVolumeTriangulation())
        {
            CTriangle::Traceback(1,M,s);
            cout<<endl;
            cout<<"The minimum volume of triangulation : "<<t[1][M]<<endl;
            return true;
        }
        else
            return false;
    }
    else
        return false;
}
```

main.cpp:

```
#include "Triangle.h"

int main()
{
    CTriangle triangle;
    triangle.Run();
}
```