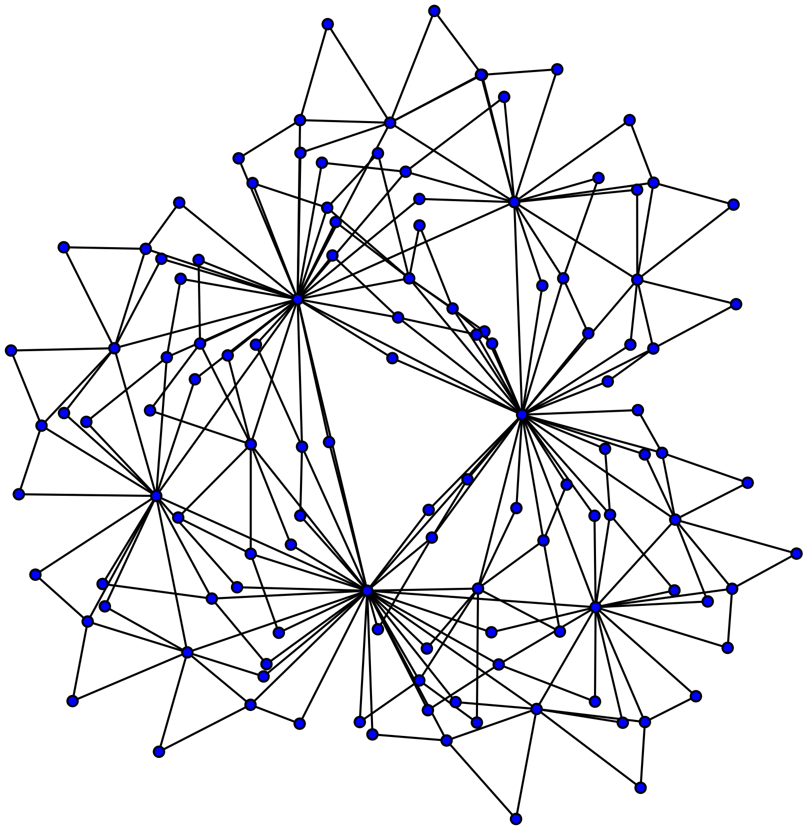
Algorithmics and Programming

Aleix Boné

May 16, 2016



Contents

1 Base Conversion

To convert a number n from a base 10 to base b, we use chained divisions. The algorithm is as follows:

1. We take the number and divide it by b and store the remainder r_i
2. While the result of the division is different from 0, we apply the previous procedure.
3. once we reach a quotient of 0, we take the remainders r_i and the result will be those remainders in the form: $r_i r_{i-1} ... r_1 r_0$

Following is a code in python to convert a number n from base 10 to base b (disregarding cases where $b > 26$):

```
def to_base_b(n,b):
    digits = "0123456789abcdefghijklmnopqrstuvwxyz"
    r = ""
    while (n!=0):
        r = digits[n% b] + r
        n//=b
    return r
```

To convert from a number n to a base b, we first convert it to base 10 and then apply the method above. To convert it to base 10, we use the following formulae:

$$(n)_b = a_0 a_1 a_2 ... a_{s-1} a_s$$
$$(n)_{10} = \sum_{i=0}^{s} a_i * b^i \tag{1}$$

Following is a code in python that demonstrates the concept:

```
def to_base_10(n,b):
    r = 0
    n = str(n)[::-1]
    for i in range(0,len(n)):
        d = n[i]
        if (d.isdigit()): d = int(d)
        else: d = ord(d)-ord('a')+10
        r+=b**i*d
    return r
```

2 Modular Arithmetic

Definition: If a number $n \exists \mathbb{Z}$ gives a remainder r when divided by a number $d \exists \mathbb{Z}$ then, we can express n as :

$$n = d * k + r \tag{2}$$

For some number $k \exists \mathbb{Z}$. We say that n is congruent to r in modulo d if and only if $d|(n-r)$ (d divides $n-r$):

$$n \equiv r \pmod{d} \tag{3}$$

2.1 Properties

- If $a \equiv b \pmod{m}$, then both $a \& b$ have the same remainder when divided by m.

2.1.1 Proof

$$\begin{aligned} a &= k_1 * m + r_1,\ b = k_2 * m + r_2 \\ a - b &= (k_1 - k_2) * m + r_1 - r_2 \\ &\downarrow \\ &m | r_1 - r_2 \qquad\qquad 0 \leq r_1 - r_2 < m \\ &\Downarrow \\ r_1 - r_2 &= 0 \rightarrow r_1 = r_2 \qquad \boxtimes \end{aligned}$$

- If $a \equiv b \pmod{m}$ & $c \equiv d \pmod{m}$, then : $a \pm c \equiv b \pm d \pmod{m}$.

2.1.2 Proof

$$\begin{aligned} a &= k_1 * m + b,\ c = k_2 * m + d \\ a \pm c &= (k_1 \pm k_2) * m + b \pm d \\ &\Downarrow \\ a \pm c &\equiv b \pm d \pmod{m} \qquad \boxtimes \end{aligned}$$

- If $a \equiv b \pmod{m}$, then : $a * k \equiv b * k \pmod{m}\ \ \forall k \exists \mathbb{Z}$.

2.1.3 Proof

$$\begin{aligned} a &= k_1 * m + b \\ a * k &= k_1 * k * m + b * k \\ &\Downarrow \\ a * k &\equiv b * k \pmod{m} \qquad \boxtimes \end{aligned}$$

- If $a \equiv b \pmod{m}$ & $c \equiv d \pmod{m}$, then : $a * c \equiv b * d \pmod{m}$.

2.1.4 Proof

$$
\begin{aligned}
a &= k_1 * m + b, \;\; c = k_2 * m + d \\
a * c &= (k_1 * m + b) * (k_2 * m + d) \qquad\qquad = k_1 k_2 m^2 + c k_1 m + b k_2 m + bd \\
&= m * (m k_1 k_2 + c k_1 + b k_2) + bd \\
&\Downarrow \\
a * c \;&\equiv\; b * d \pmod{m}
\end{aligned}
\qquad \boxtimes
$$

- If $a \;\equiv\; b \pmod{m}$, then : $a^n \;\equiv\; b^n \pmod{m} \;\; \forall n \ \exists \ \mathbb{N}$.

2.1.5 Proof

$$
\begin{aligned}
a &= k * m + b \\
a^n &= (km + b)^n \qquad\qquad = \sum_{i=0}^{n} \binom{n}{i} * k^i m^i * b^{n-i} \\
&= b^n + \sum_{i=1}^{n} \binom{n}{i} * k^i m^i * b^{n-i} \qquad m | \sum_{i=1}^{n} \binom{n}{i} * k^i m^i * b^{n-i} \\
&\Downarrow \\
a^n \;&\equiv\; b^n \pmod{m}
\end{aligned}
\qquad \boxtimes
$$

2.2 Inverse

The inverse of a must be a number a^{-1} such that $a * a^{-1} \equiv 1 \pmod{m}$. To find it, we can either try with all possible numbers from 0 to $m - 1$ or we can express our congruence as a diophantine equation and find its solution. To solve the general inverse $a * a^{-1} = 1 \pmod{m}$ we should solve the diophantine equation $a * a^{-1} + m * x = 1$

2.3 Division rules

- 1^{st} division rule:

  If $a \;\equiv\; b \pmod{m}$, $d|a$ & b and $gcd(d, m) = 1$, then: $\frac{a}{d} \;\equiv\; \frac{b}{d} \pmod{m}$

2.3.1 Proof

$$
\begin{aligned}
a &= k * m + b \\
\frac{a}{d} &= m * \frac{k}{d} + \frac{b}{d} \\
&\Downarrow \\
\frac{a}{d} \;&\equiv\; \frac{b}{d} \pmod{m}
\end{aligned}
\qquad \boxtimes
$$

- 2^{nd} division rule:

  If $a \;\equiv\; b \pmod{m}$, $d|a$ & b & m, then: $\frac{a}{d} \;\equiv\; \frac{b}{d} \pmod{\frac{m}{d}}$

2.3.2 Proof

$$
\begin{aligned}
a &= k * m + b \\
\frac{a}{d} &= k * \frac{m}{d} + \frac{b}{d} \\
&\Downarrow \\
\frac{a}{d} &\equiv \frac{b}{d} \left(mod \ \frac{m}{d} \right)
\end{aligned}
\qquad \boxtimes
$$

- 3^{rd} division rule:
  If $a \equiv b \pmod{m}$, $d|a$ & b, then: $\frac{a}{d} \equiv \frac{b}{d} \left(mod \ \frac{m}{gcd(m,d)}\right)$

2.3.3 Proof

$$
\begin{aligned}
&a = k * m + b \\
&\frac{a}{d} = \frac{k*m}{d} + \frac{b}{d} = \frac{k*m}{\frac{gcd(d,m)}{gcd(d,m)} * d} + \frac{b}{d} = \frac{k * gcd(d,m)}{d} * \frac{m}{gcd(d,m)} + \frac{b}{d} \\
&\Downarrow \\
&\frac{a}{d} \equiv \frac{b}{d} \left(mod \ \frac{m}{gcd(m,d)} \right)
\end{aligned}
\qquad \boxtimes
$$

2.4 Fermat's little theorem

If p is prime, and $a \ni \mathbb{Z}$, then, we get that:

$$a^p \equiv a \pmod{p} \tag{4}$$
$$a^{p-1} \equiv 1 \pmod{p} \tag{5}$$

3 Linear congruences

> **Definition:** A linear congruence is a congruence in the form of an equation of degree 1. It can be expressed as:
>
> $$ax \equiv b \pmod{m}$$
>
> Were $a, b, m \ni \mathbb{Z}$ and x is a variable. Linear congruences can be solved using division rules and properties of congruences, but also, as seen in the special case of finding the inverse modulo m (2.2) which is a linear congruence where $b = 1$, a linear congruence can be solved using linear diophantine equations of the form:
>
> $$ax + my = b$$

Systems of simultaneous linear congruences can be solved using linear diophantine equations as many times as needed.

3.1 Chinese remainder theorem

If m_1, m_2 are co-prime $(gcd(m_1, m_2) = 1$, then the simultaneous linear congruences:

$$x \equiv a_1 \ (mod\ m_1), \ x \equiv a_2 \ (mod\ m_2) \ \ldots \ x \equiv a_{m_1} \ (mod\ m_{n-1}), \ x \equiv a_n \ (mod\ m_n)$$

Have a *unique* solution modulo $(\prod_{i=1}^{n} m_i)$.

4 Linear Diophantine equations

> **Definition:** A linear diophantine equation, is an equation of the form:
>
> $$ax + by = c$$
>
> Where $a, b, c \exists \mathbb{Z}$ and whose solution is of the form:
>
> $$x \equiv n_1 \quad (mod\ m_1), \ y \equiv \ n_2 \quad (mod\ m_2)$$

To solve it, we use the result of the euclidean algorithm.

4.1 Euclidean Algorithm

> **Definition:** The euclidean algorithm finds the gcd of two numbers a, b. To do so, we apply the following method:

- if $b \neq 0$, then, $a = b, \ b = a(modb)$
- if $b = 0$, then the gcd of a, b is a

Following is the one-line function that applies this method in c++:

```cpp
int gcd (int a, int b) { return (b)? gcd(b,a%b) : a; }
```

Similar function in python:

```python
def gcd (a,b) : return gcd(b,a%b) if b else a
```

Using the results from the euclidean algorithm, we can say that:

$$gcd = a_1 - b_1 * q_1$$

and we can extend this until we get an expression as:

$$gcd = k_1 * a + k_2 * b$$

And now we've got the solution for the linear diophantine equation.

5 Graph Theory

> **Definition:** A graph is formed by a set of vertices $\{V\}$ and a set of edges connecting those vertices $\{E\}$. Therefore:
> $$G = \{V, E\}$$

Vertex a point of a graph which can be connected with edges

Edge line in a graph that connects 2 edges and can have a direction

Adjacent two vertices are adjacent if they are connected by an edge

Faces area delimited by at least 3 edges.

Simple Graph an undirected graph with no loops and no double edges.

Directed Graph a graph i which edges have a direction

Loop a edge connecting two the same vertex

Multigraph a graph with multiple edges between the same vertices

Degree the degree of a vertex is the number of edges that are connected to it. For a simple graph, the following condition is satisfied:

$$0 \leq deg(V_i) \leq n - 1$$

Degree sequence a sequence of all the degrees of a graph

Connected Graph A graph such that given any two vertices, there is a path that connects them. And therefore, it cannot be divided into two separate graphs without removing an edge.

Complete graph A simple graph where all vertices are joined by an edge, they are commonly referred by the letter K and they have $\frac{1}{2}v(v-1)$ edges.

Complementary graph The graph G' complementary to another graph G is the same graph but with the edges swapped, meaning that if an edge existed in the complementary it doesn't and vice-versa. Thus, it will have $\frac{1}{2}v(v-1) - e$ edges.

Bipartite graph A graph that can be separated in two sets of vertices in which there are no connections between members of the same set.

Tree A simple connected acyclic graph

Complete Bipartite graph a bipartite graph with all the edges between the two sets of vertices, its usually refered as $K_{a,b}$ where a, b are the number of vertices in each set.

Planar Graph a graph that can be represented in 2 dimensions without overlapping edges. It satisfies:

$$e \leq 3v - 6$$

Sub-graph A graph that is contained in another, meaning that if you remove some vertices from the original, you get the sub-graph.

Minimum spanning tree A subgraph that is a tree and connects all the vertices with the minimum weight possible

Figure 1: Complete graphs K_n from $n = 3$ to $n = 12$

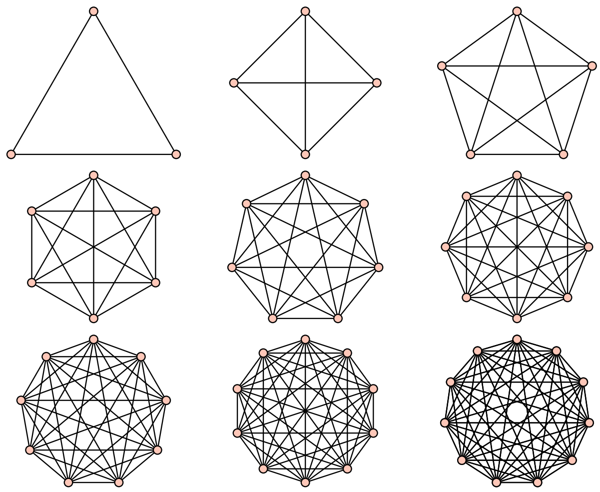


Figure 2: Trees of 8 vertices

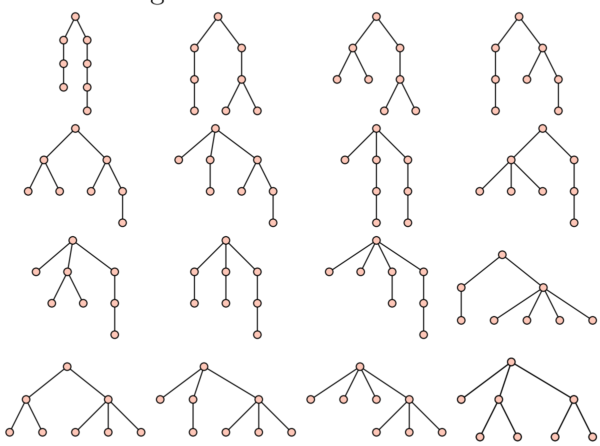
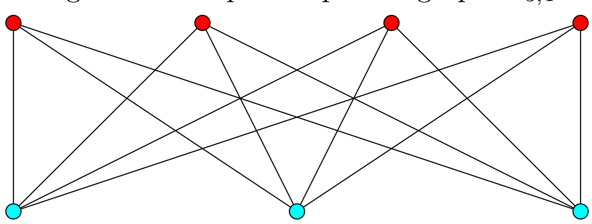


Figure 3: Complete bipartite graph $K_{3,4}$



Walk Sequence of adjacent edges.

Path A walk with no repeated *vertices*.

Trail A walk with no repeated *edges*.

Cycle A closed path. (No repeated *vertices*)

Circuit A closed path. (No repeated *edges*)

Hamiltonian cycle A closed walk that uses each *vertex* exactly once.

Isomorphic graph A graph that is equivalent to another graph even if its representation is similar, an isomorphic graph has same degrees and connections.

Eulerian circuit A closed walk that uses each *edge* exactly once. So as to accomplish that, each vertex has to have even degree.

Semi eulerian graph A non-closed walk that uses each *edge* exactly once. All vertices have to have even degree except 2 (the starting and ending vertices).

5.1 Properties

5.1.1 Planar Graph

$$e \leq 3v - 6 \tag{6}$$

$$f + v - e = 2 \tag{7}$$

5.1.2 Simple Graph

$$e \leq \frac{3f}{2} \tag{8}$$

5.1.3 Tree

$$e = v - 1 \tag{9}$$

5.2 Didac's theorem

If a simple connected graph G has v vertices and all vertices have degree $\frac{v}{2}$, then G is Hamiltonian.

5.3 Graph Algorithms

5.3.1 Kruskal algorithm

> **Definition:** Kruskal's algorithm is a greedy algorithm that finds the minimum spanning tree of a simple graph G

The algorithm is as follows:

- Take the edge with minimum weight from the graph G that is not already on our graph T.
- If the edge doesn't form a cycle when connected, add it to T otherwise discard it.
- Repeat the previous steps until the graph T contains $v - 1$ edges.

Following is a simple implementation of the above algorithm extracted from a complete code that can be found in the annex (7.1):

```
    adj kruskal () const {
        priority_queue <edge, vector <edge>, comp > q;
        for (size_t i = 0; i < v.size(); i++) for (int j = i+1; j < v.size(); j++)
             q.push(edge(v[i][j],i,j));
        adj <T> mat (v.size());
        size_t c = 0;
        while (!q.empty()) {
            edge ed = q.top(); q.pop();
            if (c == v.size()-1) return mat;
            if (ed.w != 0)
                if (!mat.dfs(ed.a,ed.b)) {
                    mat[ed.a][ed.b] = ed.w;
                    if (simple) mat[ed.b][ed.a] = ed.w;
                    c++;
                }
        }
        throw error::NOT_CONNECTED;
    }
```

(To check is the new vertex forms a cycle, the best way is to use union sets although the above code uses a DFS for simplicity)

5.3.2 Check if bipartite

To check if a graph is bipartite, we start at an arbitrary point and color it with the first colour a, we then colour all its adjacent vertices in the second colour b and apply the same procedure on them, if we find one vertex that has two colour, the graph is *not* bipartite otherwise it is bipartite.

5.3.3 Prim algorithm

> **Definition:** Prim's algorithm is a greedy algorithm that finds the minimum spanning tree of a simple graph G

The algorithm is as follows:

- Take any arbitrary vertex of the graph
- From all the vertices that we have visited, had to T the edge with the minimum weight that connects to an unvisited vertex.
- Repeat the previous step until the graph T contains $v - 1$ edges.

Following is a simple implementation of the above algorithm extracted from a complete code that can be found in the annex (7.1):

```
    adj prim () const {
        priority_queue <edge, vector <edge>, comp> q;
        for (size_t i = 1; i < v.size(); i++) q.push(edge(v[0][i],0,i));
        adj <T> mat (v.size());
        vector <bool> fets (v.size(),false);
        size_t c = 0;
        fets[0]=true;
        while (!q.empty()) {
            edge ed = q.top(); q.pop();
            if (c == v.size()-1) return mat;
            if (!fets[ed.b] and ed.w != 0) {
                mat[ed.a][ed.b] = ed.w;
                if (simple) mat[ed.b][ed.a] = ed.w;
                c++;
                fets[ed.b] = true;
                for (size_t i = 0; i < v.size(); i++) q.push(edge(v[ed.b][i],ed.b,
                    i));
            }
        }
        throw error::NOT_CONNECTED;
    }
```

5.3.4 Dijkstra

> **Definition:** Dijkstra's algorithm is a greedy algorithm that finds the shortest path between any two vertices of a graph G with no negative edges.

The algorithm is as follows:

- Start at vertex o, label it with distance 0 and previous vertex $NULL$.
- Add all the edges with the current distance plus the edges weight.
- take the edge with the shortest distance that we have procesed and apply the above steps if the vertex we have reached has not yet been visited.
- Repeat the previous steps until we reach vertex b or we have checked all the vertices and found no path, so the graph is not connected and there is no possible path between the vertices.

Following is a simple implementation of the above algorithm extracted from a complete code that can be found in the annex (section 7.1) (an implementation of dijkstra with pointers can be found on annex section 7.2):

```cpp
T dijkstra (size_t b, const size_t& e) const {
    vector <T> d (v.size(), INF);
    d[b]=0;
    priority_queue <pair <T, size_t >, vector <pair <T, size_t> >, greater<pair
        <T,size_t> > > q;
    q.push(pair <T, size_t> (0,b));
    while (!q.empty()) {
        size_t pos = q.top().second;
        if (q.top().first > d[pos]) {
            q.pop();
            continue;
        }
        if (pos == e) return d[pos];
        q.pop();
        for (size_t i = 0; i < v.size(); i++)
            if (v[pos][i]!=0 and d[i] > d[pos] + v[pos][i]) {
                d[i] = d[pos] + v[pos][i];
                q.push(pair <T, size_t> (d[i],i));
            }
    }
    throw error::NOT_CONNECTED;
}
```

5.3.5 Chinese Postman problem

> **Definition:** Find the shortest circuit that goes through all edges (unlike an eulerian circuit, edges can be repeated).

We can do it easily if all the degrees are even (it has an eulerian circuit). If it has 2 vertices with odd degree, we must find the shortest path between those 2 (using Dijkstra or similar) and repeat the Dijkstra path. Similar approach can be applied if there are 4 vertices with odd degree (brute forcing which pairs to connect).

5.3.6 Travelling salesman problem

> **Definition:** Find the shortest cycle that goes through all vertices (unlike a Hamiltonian circuit, vertices can be repeated).

Only method is buy brute force (costs $1/2*(v-1)!$). Since the cost of the brute force is enormous, the best approach is to determine some lower and upper bounds ant take the best cycle found in some trials.

The simplest approach to find the upper and lower bounds consist of taking the v smaller and bigger edges.

A better method to find the upper bound is to use the nearest neighbourhood algorithm (taking the nearest vertex that doesn't form a cycle until we have $v-1$ edges and the form the cycle).

A better method to find a lower bound is to remove a vertex and compute the minimum spanning tree of the resulting graph and finally add the two smaller edges of the removed vertex.

6 Recurrences

> **Definition:** A recurrence is a set of numbers a_n defined in terms of other numbers of the recurrence or in terms of n

A well known recurrence is the famous Fibonacci sequence defined as follows:

$$a_n = a_{n-1} + a_{n-2},\ a_0 = 1,\ a_1 = 1$$

6.1 Homogeneous recurrences

> **Definition:** Linear Homogeneous recurrences with constant coefficients are recurrences which are defined only in terms of its previous elements (not in terms of n or with a constant). They have the form:
>
> $$a_n = c_1 * a_{n-1} + c_2 * a_{n-2} \dots c_i * a_{n-i}$$

To find the general solution of an homogeneous linear recurrence, we use the characteristic polynomial of the recurrence. To do so, we express the recurrence in terms of $a_n * t^n$ and factor out t^{n-m}.

For an homogeneous recurrence:

$$a_n = \sum_{i=0}^{m} a_{n-i} * c_i$$

We define its characteristic polynomial as:

$$t^n = \sum_{i=0}^{m} t^{n-i} * c_i$$

And if we factor t^{n-m}, we get:

$$t^m = \sum_{i=0}^{m} t^i * c_i$$

Which is a polynomial which solutions $r_0, r_1 \dots r_{m-1}, r_m$ can be used to determine a_n in the form:

$$a_n = \sum_{i=0}^{m} k_i * r_i^n$$

The factors k_i can be determined using the base terms of the recurrence.

If some of the roots (r_i) have multiplicity bigger than one, then we have to multiply the root by the sum of$n^{\alpha_i - 1 - j}$ where α is the multiplicity of r_i and j ranges from 0 to $\alpha - 1$. The solution then is as follows:

$$a_n = \sum_{i=0}^{m} \left(r_i^n * \sum_{j=0}^{\alpha_i - 1} k_{i,j} * n^j \right)$$

6.2 Non homogeneous recurrences

Definition: Non homogeneous linear recurrences with constant coefficients are recurrences of the form:

$$a_n = c_1 * a_{n-1} + c_2 * a_{n-2} \dots c_i * a_{n-i} + f(n)$$

We have two different methods to find the general solution of a non homogeneous recurrence, the first consists on subtracting recurrences to eliminate the non homogeneous part at the cost of increasing the degree of the characteristic polynomial resulting.

The second method consists on solving the recurrence only for the homogeneous part and then solving it for the non-homogeneous part knowing its form. The general solution will be the sum of the non-homogeneous and the homogeneous parts.

7 Annex

7.1 graphs.h

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

#define INF numeric_limits<T>::max()

namespace error {
    enum graphs {INVALID_INPUT=0xff, ERR_READ_FILE, NOT_CONNECTED, NOT_SIMPLE_DIRECTED, NOT_SIMPLE_LOOP, INV_MEM, INV_COORD, OUT_OF_BOUNDS};
}

struct edge {
    size_t w, a, b;
    edge (size_t W, size_t A, size_t B): w(W), a(A), b(B) {}
};

struct comp { bool operator() (edge e1, edge e2) { return e1.w > e2.w; } };

template <class T>
class adj {

    mutable vector <bool> fets;
    mutable vector <size_t> vbfs;

    public:

    static bool simple;

    vector <vector <T> > v;

    adj (size_t N=0) { v = vector <vector <T> > (N, vector <T> (N,0)); }

    vector <T>& operator[] (const size_t& p) { if (p < v.size()) return v[p]; else throw error::INV_MEM; }
    T& operator[] (const edge& e) { if(e.a < v.size() and e.b < v.size()) return v[e.a][e.b]; else throw error::INV_MEM; }

    size_t size() { return v.size(); }

    friend ostream& operator<< (ostream& out, const adj& mat) {
        for (size_t i = 0; i < mat.v.size(); i++) {
            for (size_t j = 0; j < mat.v[i].size(); j++) {
                if (j!=0) cout << ' ';
                cout << mat.v[i][j];
            }
            cout << endl;
        }
        return out;
    }

    friend istream& operator>> (istream& in, adj& mat) {
        for (size_t i = 0; i < mat.v.size(); i++)
            for (size_t j = 0; j < mat.v[i].size(); j++) if (!(in >> mat.v[i][j])) throw error::INVALID_INPUT;
        if (in.fail()) throw error::INVALID_INPUT;
```

```
        if (adj::simple) mat.check();
        return in;
    }

    bool dfs (size_t b, const size_t& e, bool flag=1) const {
        if (flag) fets = vector <bool> (v.size(),false);
        if (b == e) return true;
        if (fets[b]) return false;
        fets[b]=true;
        for (size_t i = 0; i < v.size(); i++)
            if (v[b][i]!=0) if (dfs(i,e,0)) return true;
        return false;
    }

    int bfs (size_t b=0, const size_t& e=-1) const {
        vbfs.resize(fets.size(),-1);
        vbfs[b]=0;
        queue <T> q;
        q.push(b);
        do {
            b = q.top(); q.pop();
            if (b==e) return fets[b];
            for (size_t  i = 0; i < v.size(); i++) if (v[b][i] and vbfs[i]==-1) {
                q.push(v[b][i]);
                vbfs[i]=vbfs[b]+1;
            }
        } while (!q.empty());
        return -1;
    }

    adj kruskal () const {
        priority_queue <edge,vector <edge>, comp > q;
        for (size_t i = 0; i < v.size(); i++) for (int j = i+1; j < v.size(); j++)
             q.push(edge(v[i][j],i,j));
        adj <T> mat (v.size());
        size_t c = 0;
        while (!q.empty()) {
            edge ed = q.top(); q.pop();
            if (c == v.size()-1) return mat;
            if (ed.w != 0)
                if (!mat.dfs(ed.a,ed.b)) {
                    mat[ed.a][ed.b] = ed.w;
                    if (simple) mat[ed.b][ed.a] = ed.w;
                    c++;
                }
        }
        throw error::NOT_CONNECTED;
    }

    adj prim () const {
        priority_queue <edge,vector <edge>, comp> q;
        for (size_t i = 1; i < v.size(); i++) q.push(edge(v[0][i],0,i));
        adj <T> mat (v.size());
        vector <bool> fets (v.size(),false);
        size_t c = 0;
        fets[0]=true;
        while (!q.empty()) {
            edge ed = q.top(); q.pop();
            if (c == v.size()-1) return mat;
            if (!fets[ed.b] and ed.w != 0) {
                mat[ed.a][ed.b] = ed.w;
```

```
                if (simple) mat[ed.b][ed.a] = ed.w;
                c++;
                fets[ed.b] = true;
                for (size_t i = 0; i < v.size(); i++) q.push(edge(v[ed.b][i],ed.b,i));
            }
        }
        throw error::NOT_CONNECTED;
    }

    T dijkstra (size_t b, const size_t& e) const {
        vector <T> d (v.size(), INF);
        d[b]=0;
        priority_queue <pair <T, size_t >, vector <pair <T, size_t> >, greater<pair <T,size_t> > > q;
        q.push(pair <T, size_t> (0,b));
        while (!q.empty()) {
            size_t pos = q.top().second;
            if (q.top().first > d[pos]) {
                q.pop();
                continue;
            }
            if (pos == e) return d[pos];
            q.pop();
            for (size_t i = 0; i < v.size(); i++)
                if (v[pos][i]!=0 and d[i] > d[pos] + v[pos][i]) {
                    d[i] = d[pos] + v[pos][i];
                    q.push(pair <T, size_t> (d[i],i));
                }
        }
        throw error::NOT_CONNECTED;
    }

    T weight () const {
        T c = 0;
        for (int i = 0; i < v.size(); i++)
            for (int j = 0; j < v[i].size(); j++) c += v[i][j];
        return (simple)? c/2 : c;
    }

    private:

    bool check () const {
        for (size_t i  = 0; i < v.size(); i++)
            for (size_t j = 0; j < v[i].size(); j++) if (v[i][j] != v[j][i]) throw error::NOT_SIMPLE_DIRECTED;
        for (size_t i = 0; i < v.size(); i++) if (v[i][i] != 0) throw error::NOT_SIMPLE_LOOP;
        return true;
    }
};

template <class T>
class adjlist {

    vector <bool> fets;

    public:

    vector <vector < pair <T,T> > > v;

```

```
    adjlist () {}

    adjlist (const adj<T>& a) {
        v.resize(a.v.size());
        for (int i=0; i < a.v.size(); i++)
            for (int j=0; j < a.v[i].size(); j++)
                if (a.v[i][j]!=0) v[i].push_back(j);
    }

};

template <class T>
bool adj<T>::simple = true;
```

7.2 Dijkstra with pointers

```
#include <iostream>
#include <queue>
#include <vector>
#include <new>
using namespace std;

enum err {BAD_INPUT=0xf, NOT_SIMPLE};

class vertex;

class node {
    public:
        int w;
        vertex* v, *prev;
        node (int _w=0, vertex* _v=NULL, vertex* _prev=NULL) : w(_w), v(_v),prev(_prev) {}
        inline friend bool operator> (const node& a, const node& b) { return a.w > b.w; }
        node operator+ (int n) {return node(w+n,v,prev);}
};

class vertex {
    vertex* prev;
    int d;
    int id;
    public:
    static int n;
    vector <node> v;
    vertex () : prev(NULL), d(-1), id(n++) {}

    friend int dijkstra (vertex *a, const vertex *b);
    friend void undopath(vertex *p);
};

int vertex::n=0;

int dijkstra (vertex* a, const vertex* b) {
    priority_queue <node, vector <node>, greater <node> > q;
    q.push(node(0,a));
    while (!q.empty()) {
        node n = q.top(); q.pop();
        if (n.v->d!=-1) continue;
        n.v->prev=n.prev;
        n.v->d=n.w;
        if (n.v==b) return n.w;
```

```cpp
        for (auto i : n.v->v) q.push(i+n.w);
    }
    return -1;
}

void undopath (vertex *p) {
    if (p->prev==NULL) cout << p->id;
    else {
        undopath(p->prev);
        cout << '-' << p->id;
    }
}

int main () {
    int n, e;
    cin >> n >> e;
    vertex *V = new vertex[n];
    while (e--) {
        int a,b,w;
        cin >> a >> b >> w;
        (V+a)->v.push_back(node(w,V+b,V+a));
        (V+b)->v.push_back(node(w,V+a,V+b));
    }
    int a,b;
    cin >> a >> b;
    cout << dijkstra(V+a,V+b) << endl;
    undopath(V+b);
}
```

7.3 More Algorithms

Figure 4: More Algorithms implemented in C++ can be found at: https://github.com/Leixb/Cpp_algorithms