Lab09-Network Flow

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2020.

* If there is any problem, please contact TA Shuodian Yu.
* Name:Hongjie Fang Student ID:518030910150 Email: galaxies@sjtu.edu.cn

1. Given a weighted directed graph $G(V, E)$ and its corresponding weight matrix $W = (w_{ij})_{n\times n}$ and shortest path matrix $D = (d_{ij})_{n\times n}$, where w_{ij} is the weight of edge (v_i, v_j) and d_{ij} is the weight of a shortest path from pairwise vertex v_i to v_j. Now, assume the weight of a particular edge (v_a, v_b) is decreased from w_{ab} to w'_{ab}. Design an algorithm to update matrix D with respect to this change, whose time complexity should be no larger than $O(n^2)$. Describe your design first and write down your algorithm in the form of pseudo-code.

Solution. We assume that there is no negative cycle in both the original graph G and the new graph after updating. Here is my algorithm design.

For every pair of vertices (v_x, v_y), check if path $v_x \rightsquigarrow v_a \to v_b \rightsquigarrow v_y$ is shorter than the current shortest path $v_x \rightsquigarrow v_y$ in new graph. If so, then update the shortest path d_{xy} to $d_{xa} + w'_{ab} + d_{by}$.

(Correctness Explanation) First, there exists a shortest path from v_x to v_a that does not contain edge (v_a, v_b). If not, then there must be at least one cycle containing edge (v_a, v_b) in the shortest path from v_x to v_a because we visit v_a at least twice. We can eliminate all the zero cycle in the shortest path from v_x to v_a first without changing the total weight of the shortest path, and at least one non-zero cycle containing (v_a, v_b) must remain after the operation, or it will contradicts the premise that all the shortest path from v_x to v_a contains edge (v_a, v_b). After that, we make the following discussions.

- If it is a negative cycle, then it contradicts the assumption that there is no negative cycle;
- If it is a positive cycle, then we can eliminate it to form a shorter path from v_x to v_a, which contradicts the premise that it is a shortest path from x to a.

In summary, there exists a shortest path from v_x to v_a that does not contain edge (v_a, v_b). Similarly, there exists a shortest path from v_b to v_y that does not contain (v_a, v_b). Therefore, the weight change of edge (v_a, v_b) does not affect the shortest path from v_x to v_a and the shortest path from v_b to v_y, which means the shortest paths $d_{\cdot a}$ whose destinations are v_a and the shortest paths $d_{b\cdot}$ whose sources are v_b must remain the same after the weight change.

Since the weight of (v_a, v_b) is decreased, we can check if path $v_x \rightsquigarrow v_a \to v_b \rightsquigarrow v_y$ is shorter than the current shortest path. These two parts $v_x \rightsquigarrow v_a$ and $v_b \rightsquigarrow v_y$ are not influenced by the weight change of edge (v_a, v_b). Thus, for every original shortest path from v_x to v_y:

- If the weight change does not affect it, then d_{ij} will not change in our algorithm;
- If it has the edge (v_a, v_b), then d_{ij} is bound to be updated because the new weight of the shortest path $d_{xa} + w'_{ab} + d_{by}$ is less than the old one $d_{xy} = d_{xa} + w_{ab} + d_{by}$;
- If it does not have the edge (v_a, v_b) but after weight change the new shortest path should contain (v_a, v_b), then it will also be taken into consideration because we check the path $v_x \rightsquigarrow v_a \to v_b \rightsquigarrow v_y$. Since the weight change does not affect $v_x \rightsquigarrow v_a$ and $v_b \rightsquigarrow v_y$, then path $v_x \rightsquigarrow v_a \to v_b \rightsquigarrow v_y$ must be the shortest path from v_x to v_y passing edge (v_a, v_b). Therefore, we must update the shortest path from v_x to v_y in our algorithm.

In summary, our algorithm will update the shortest path matrix correctly.

(Time Complexity Analysis) We check every pair of vertices once, and the time complexity of the possible updating process is $O(1)$. There are n^2 pairs of vertices in total. Therefore, the total time complexity of the algorithm is $O(n^2)$.

(Pseudo-Code) The pseudo-code of the algorithm is as follows.

Algorithm 1: Shortest Path Matrix Update Algorithm

Input: A directed graph $G(V, E)$; the weight matrix $W = (w_{ij})_{n \times n}$ and the shortest path matrix $D = (d_{ij})_{n \times n}$ of graph G; the particular edge (v_a, v_b) and its new weight w'_{ab}.

Output: The new shortest path matrix $D = (d_{ij})_{n \times n}$ after updating with respect to the weight change of (v_a, v_b).

```
1 foreach $v_x \in V$ do
2     foreach $v_y \in V$ do
3         if $d_{xa} + w'_{ab} + d_{by} < d_{xy}$ then
4             $d_{xy} \leftarrow d_{xa} + w'_{ab} + d_{by}$;
5 return $(d_{ij})_{n \times n}$;
```

□

2. Given a directed graph G, whose vertices and edges information are introduced in data file "SCC.in". Please find its number of Strongly Connected Components with respect to the following subquestions.

   (a) Read the code and explanations of the provided C/C++ source code "SCC.cpp", and try to complete this implementation.

   (b) Visualize the above selected Strongly Connected Components for this graph G. Use the *Gephi* or other software you preferred to draw the graph. (If you feel that the data provided in "SCC.in" is not beautiful, you can also generate your own data with more vertices and edges than G and draw an additional graph. Notice that results of your visualization will be taken into the consideration of Best Lab.)

Solution. Here is the answers to each sub-question.

(a) I use two different algorithms to compute the number of the strongly connected components in the graph G, and both of them output 666 for the given input graph "SCC.in", which means there are 666 strongly connected components in the graph G.

The first algorithm is **the Kosaraju algorithm**, which we have learnt in the course. The main source code of this algorithm is as follows. You can also refer to code/SCC.cpp for full implementation. You may need to open the C++11 switch using `-std=c++11`.

```
void dfs(const vector < vector <int> > &E, int x, vector <int> &dfn,
   vector <int> &out, int &idx) {
  dfn[x] = ++idx;
  for (auto &dest: E[x])
    if (! dfn[dest]) dfs(E, dest, dfn, out, idx);
  out[x] = ++idx;
}

int SCC (int n, vector < pair <int, int> > &edge) {
  vector < vector <int> > E, Er;
  vector <int> dfn, out;
  vector <int> order;
  int idx, scc;

  E.resize(n);
  Er.resize(n);
  dfn.resize(n);
  out.resize(n);
  order.resize(n << 1);

  for (auto &edge_list: E) edge_list.clear();
  for (auto &edge_list: Er) edge_list.clear();

  for (auto &e: edge) {
    E[e.first].push_back(e.second);
    Er[e.second].push_back(e.first);
  }

  idx = 0;
  for (auto &d: dfn) d = 0;
  for (int i = 0; i < n; ++ i)
    if(! dfn[i]) dfs(Er, i, dfn, out, idx);

  for (auto &o: order) o = -1;
  for (int i = 0; i < n; ++ i) order[out[i] - 1] = i;

  scc = 0;
  idx = 0;
  for (auto &d: dfn) d = 0;
  for (int i = (n << 1) - 1; ~i; -- i)
    if(order[i] != -1 && ! dfn[order[i]]) {
      ++ scc;
      dfs(E, order[i], dfn, out, idx);
    }
  return scc;
}
```

The second algorithm is **the Tarjan algorithm**. The main source code of this algorithm is as follows. You can also refer to code/SCC-tarjan.cpp for full implementation. You may need to open the C++11 switch using `-std=c++11`.

```
void tarjan(vector < vector <int> > &E, int x, vector <int> &dfn, vector
     <int> &low, vector <int> &instack, stack <int> &sta, int &idx, int &
   scc) {
  dfn[x] = low[x] = ++idx;
  sta.push(x);
  instack[x] = 1;

  for (auto &dest: E[x]) {
    if(! dfn[dest]) {
      tarjan(E, dest, dfn, low, instack, sta, idx, scc);
      low[x] = min(low[dest], low[x]);
    } else if (instack[dest]) low[x] = min(low[x], dfn[dest]);
  }

  if(dfn[x] == low[x]) {
    ++ scc;
    while(true) {
      int y = sta.top();
      sta.pop(); instack[y] = 0;
      if (y == x) break;
    }
  }
}

int SCC (int n, vector < pair <int, int> > &edge) {
  vector < vector <int> > E;
  vector <int> dfn, low;
  vector <int> instack;
  stack <int> sta;
  int idx, scc;

  E.resize(n);
  dfn.resize(n);
  low.resize(n);
  instack.resize(n);
  while(!sta.empty()) sta.pop();

  for (auto &edge_list: E) edge_list.clear();

  for (auto &e: edge)
    E[e.first].push_back(e.second);

  scc = 0;
  idx = 0;
  for (auto &d: dfn) d = 0;
  for (auto &in: instack) in = 0;
  for (int i = 0; i < n; ++ i)
    if(! dfn[i]) tarjan(E, i, dfn, low, instack, sta, idx, scc);

  return scc;
}
```

(b) If we regard each strongly connected component as a meta-node, we can use Gephi to draw the following picture of the graph (Fig. 1).

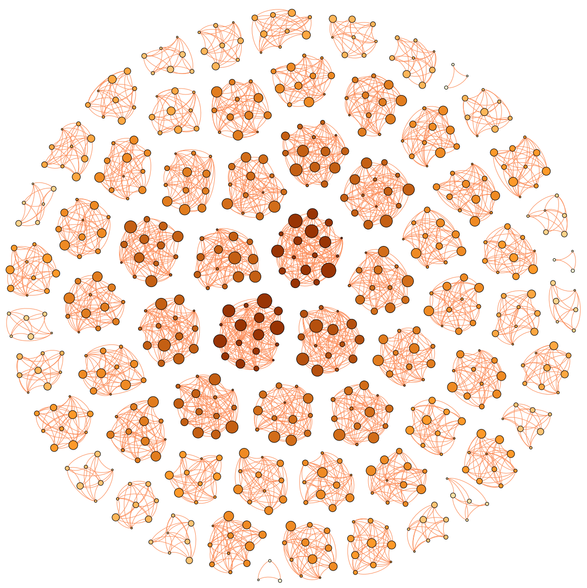


Figure 1: The visualization of the graph

I generate a graph with 100000 vertices and 601999 edges myself, and the data is in the code/newSCC.in. The visualization of the newly generated graph is as follows. (Fig. 2).

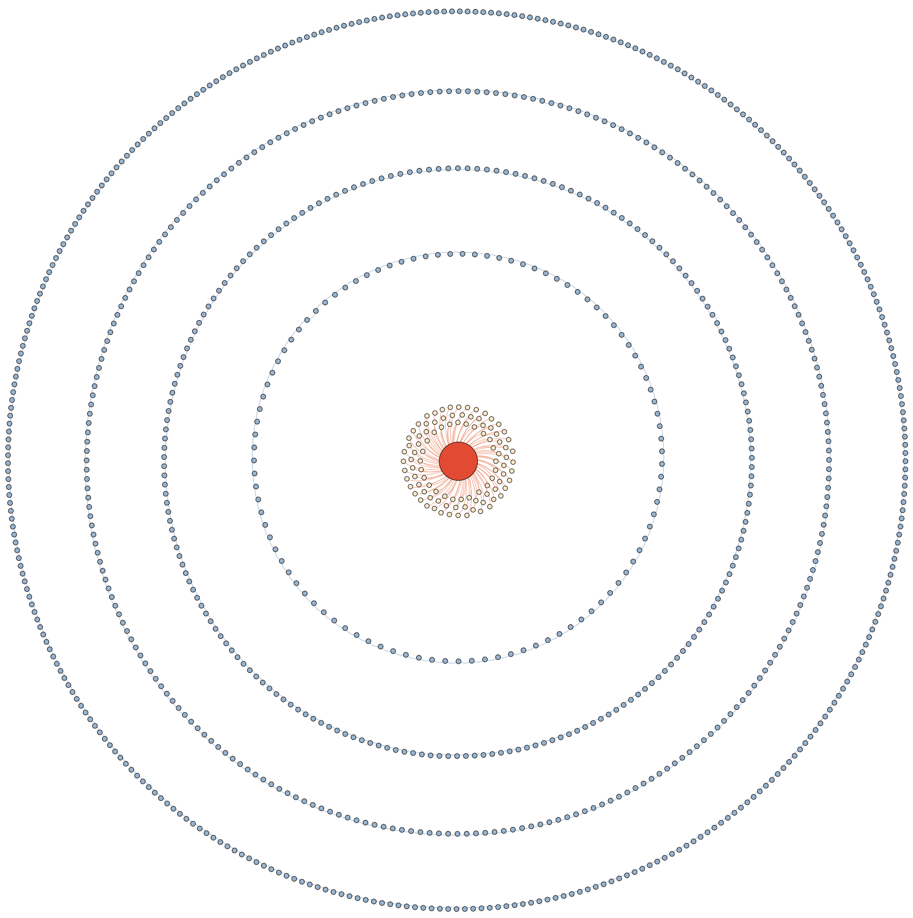


Figure 2: The visualization of the newly generated graph

□

3. The **Minimum Cost Maximum Flow** problem (MCMF) is an optimization problem to find the cheapest possible way of sending the maximum amount of flow through a flow network. That is, in a flow network $G = (V, E)$ with a source $s \in V$ and a sink $t \in V$, where each edge $(u, v) \in E$ has a capacity $c(u, v) > 0$ and a cost $a(u, v) \geq 0$, find a maximum s-t flow f over all edges ($f(u, v) \geq 0$), such that the total cost of $\sum_{(u,v)\in E} a(u, v) \cdot f(u, v)$ is minimized.

   A common greedy approach to solve the MCMF problem can be described as follows: We can modify Ford-Fulkerson algorithm, where each time we choose the least cost path from s to t. To do this correctly, when we add a back-edge to some edge e into the residual graph, we give it a cost of $-a(e)$, representing that we get our money back if we undo the flow on it.

   Note that such procedure may create a residual graph with negative-weight edges, which is not suitable for Dijkstra's Algorithm. However, motivated by Johnson's Algorithm, we can reweight the edge cost with vertex labels and convert the weight non-negative again.

   Please prove the correctness of such greedy approach and implement this algorithm in C/C++. The file *MCMF.in* is a test case, where the first line contains four graph parameters n, m, s, t, and the rest m lines exhibit the information of m edges. Each line contains four integers: u_i, v_i, c_i, a_i, denoting that there is an edge from u_i to v_i with capacity c_i and cost a_i. (Your source code should be named as *MCMF.cpp* and output the maximum flow and minimum cost of this test case.)

Sample Input:	**Sample Output:**
4 5 4 3	50 280
4 2 30 2	
4 3 20 3	
2 3 20 1	
2 1 30 9	
1 3 40 5	

Definition 1. *Let $a \rightarrow b$ denote an edge from a to b; let $a \xrightarrow{F} b$ denote an edge from a to b in flow F; let $a \xrightarrow{-F} b$ denote a back edge from a to b in flow F; let $a \xrightarrow{r(F)} b$ denote an edge from a to b in the residual graph of flow F. A simple property is the edge $a \xrightarrow{-F} b$ can also be represented as $a \xrightarrow{r(F)} b$.*

Definition 2. *Let $a \rightsquigarrow b$ denote a path from a to b; let $a \overset{F}{\rightsquigarrow} b$ denote a path from a to b in flow F; let $a \overset{-F}{\rightsquigarrow} b$ denote a path from a to b constructed by back edges in flow F. let $a \overset{r(F)}{\rightsquigarrow} b$ denote a path from a to b in the residual graph of flow F. A simple property is the path $a \overset{-F}{\rightsquigarrow} b$ can also be represented as $a \overset{r(F)}{\rightsquigarrow} b$.*

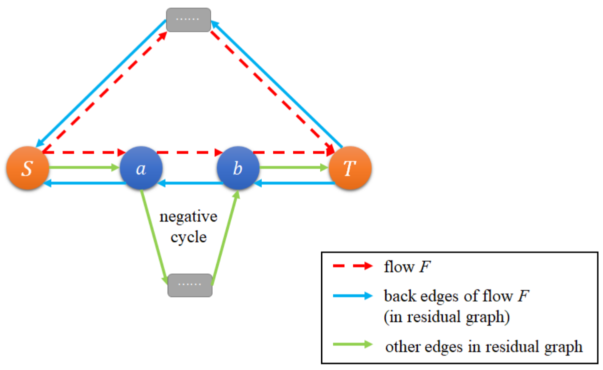
Lemma 1. *Given a value v, flow F is the minimum cost flow among all the flows with value v **if and only if** there is no negative cycle in the residual graph of flow F.*

Proof. We prove two directions of this lemma respectively as follows.

- ($\Longrightarrow$) **(Contradiction)** If there exists at least one negative cycle in the residual graph of flow F (Fig. 3), then the negative cycle must contain at least one back edge, say $b \xrightarrow{-F} a$, since there is no negative weight edge in the initial graph. The negative cycle in the residual graph can be represented as $a \overset{r(F)}{\rightsquigarrow} b \xrightarrow{-F} a$. Then we can reduce the flow in edge $a \xrightarrow{F} b$ by f ($f > 0$) and add the same amount of flow in path $a \overset{r(F)}{\rightsquigarrow} b$ to form a new flow F'. Since $a \overset{r(F)}{\rightsquigarrow} b \xrightarrow{-F} a$ is a negative cycle in the residual graph, we can make the following derivations.

$$\begin{aligned} cost(F') &= cost(F) - f \cdot cost(a \xrightarrow{F} b) + f \cdot cost(a \overset{r(F)}{\rightsquigarrow} b) \\ &= cost(F) + f \cdot cost(b \xrightarrow{-F} a) + f \cdot cost(a \overset{r(F)}{\rightsquigarrow} b) \\ &= cost(F) + f \cdot cost(a \overset{r(F)}{\rightsquigarrow} b \xrightarrow{-F} a) \\ &< cost(F) \end{aligned}$$

Hence, flow F' has the same value of flow F but it costs less than flow F, which contradicts the premise that flow F is the minimum cost flow of value v.



......
S
a
b
T
negative cycle
......
flow F
back edges of flow F (in residual graph)
other edges in residual graph

Figure 3: A negative cycle in the residual graph

- $(\Longleftarrow)$ **(Contradiction)** Suppose there is no negative cycle in the residual graph of flow F but flow F is not the minimum cost flow of value v. Then there exists another flow F' with the same value, whose cost is less than flow F. According to the Flow Conservation Property and the Flow Value Lemma, the graph $F' - F$ is actually constructed by several cycles. At least one of the cycles has a negative weight because the cost of flow F' is less than the cost of flow F (Fig. 4). And the negative cycle in the graph $F' - F$ can be represented as $a \overset{F'}{\rightsquigarrow} b \overset{-F}{\rightsquigarrow} a$. Notice that the path $a \overset{F'}{\rightsquigarrow} b$ is also in the residual graph of flow F since it is not in flow F. Therefore, there exists a negative cycle $a \overset{F'}{\rightsquigarrow} b \overset{-F}{\rightsquigarrow} a$ in the residual graph of flow F, which contradicts the premise.

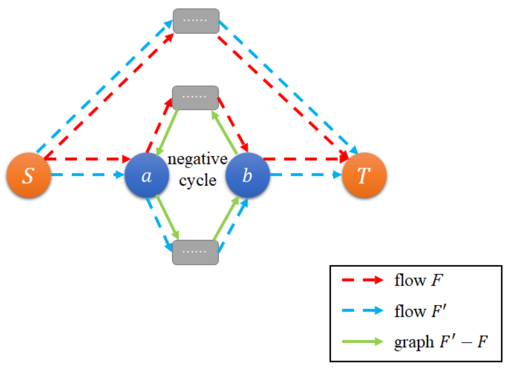


Figure 4: A negative cycle in graph $F' - F$

$\square$

Lemma 2. *Our algorithm can terminates after finding a finite number of augmenting paths, and the result flow F^* is the max flow of the flow network G.*

Proof. Our algorithm is actually a modified version of the Ford-Fulkerson Algorithm. Using exact the same method of proof of the Ford-Fulkerson Algorithm, we can derive that if the algorithm terminates then the result flow F^* is the max flow of the flow network G, and the algorithm terminates at most $val(F^*) \le n \times C$ augmenting paths, where C is the upper bound of the capacities of the edges. □

Lemma 3. *During the construction process of the max flow F^* in our algorithm, there is no negative cycle in the residual graph of the current flow F.*

Proof. (Induction) In the beginning, our current flow F is empty, so the residual graph of the current flow F is actually the initial network G. And there is no negative cycle in G according to the problem description.

Suppose there is no negative cycle in the current flow F after finding n augmenting paths in our algorithm. We are going to prove there is still no negative cycle in flow F' after adding an augmenting path based on flow F in our algorithm.

Flow F' can be represented as $F \cup \{s \overset{F'}{\rightsquigarrow} t\}$, where $s \overset{F'}{\rightsquigarrow} t$ is the new augmenting path. According to our algorithm, path $s \overset{F'}{\rightsquigarrow} t$ is the shortest path from s to t. Suppose the residual graph of flow F' contains a negative cycle (Fig. 5). Then the negative cycle must contain some of the back edges in the new augmenting path $s \overset{F'}{\rightsquigarrow} t$, say $b \overset{-F'}{\rightsquigarrow} a$. Hence, path $s \overset{F'}{\rightsquigarrow} t$ can also be represented as $s \overset{F'}{\rightsquigarrow} a \overset{F'}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t$, and the negative cycle can be represented as $a \overset{r(F')}{\rightsquigarrow} b \overset{-F'}{\rightsquigarrow} a$. Therefore, we can replace sub-path $a \overset{F'}{\rightsquigarrow} b$ with sub-path $a \overset{r(F')}{\rightsquigarrow} b$ to form a new path $s \overset{F'}{\rightsquigarrow} a \overset{r(F')}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t$ from s to t. Since $a \overset{r(F')}{\rightsquigarrow} b \overset{-F'}{\rightsquigarrow} a$ is a negative cycle, we can make the following derivations.

$$\begin{aligned}
cost(s \overset{F'}{\rightsquigarrow} a \overset{r(F')}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t) &= cost(s \overset{F'}{\rightsquigarrow} a \overset{F'}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t) - cost(a \overset{F'}{\rightsquigarrow} b) + cost(a \overset{r(F')}{\rightsquigarrow} b) \\
&= cost(s \overset{F'}{\rightsquigarrow} a \overset{F'}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t) + cost(b \overset{-F'}{\rightsquigarrow} a) + cost(a \overset{r(F')}{\rightsquigarrow} b) \\
&= cost(s \overset{F'}{\rightsquigarrow} a \overset{F'}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t) + cost(a \overset{r(F')}{\rightsquigarrow} b \overset{-F'}{\rightsquigarrow} a) \\
&< cost(s \overset{F'}{\rightsquigarrow} a \overset{F'}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t)
\end{aligned}$$

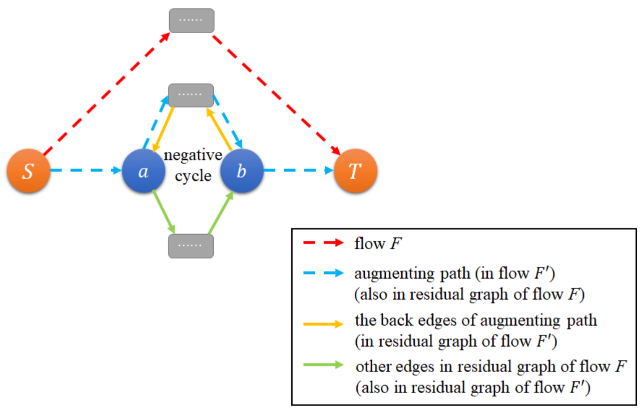

......
......
negative cycle
S
a
b
T
......
flow F
augmenting path (in flow F') (also in residual graph of flow F)
the back edges of augmenting path (in residual graph of flow F')
other edges in residual graph of flow F (also in residual graph of flow F')

Figure 5: A negative cycle in the residual graph of flow F'

Hence, the newpath $s \overset{F'}{\rightsquigarrow} a \overset{r(F')}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t$ is shorter than path $s \overset{F'}{\rightsquigarrow} a \overset{F'}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t$, which contradicts the premise that path $s \overset{F'}{\rightsquigarrow} a \overset{F'}{\rightsquigarrow} b \overset{F'}{\rightsquigarrow} t$ is the shortest path from s to t. Therefore, the residual graph of F' still does not contain a negative circle, which completes the induction step.

In summary, during the construction process of the max flow F^* in our algorithm, there is no negative cycle in the residual graph of the current flow F. □

Theorem 4. *The result flow F^* is actually a Minimum Cost Maximum Flow.*

Proof. According to Lemma 2, F^* is the max flow of network G.

According to Lemma 3, there is no cycle in the residual graph of F^*. Then according to Lemma 1, F^* is the minimum cost flow of the given value $v(F^*)$, which is the max flow value.

Therefore, F^* is actually a Minimum Cost Maximum Flow. □

Solution. Theorem 4 give the correctness proof of the algorithm, and Lemma 2 tells us the process can terminates after finding a finite number of augmenting paths. Therefore, the algorithm is correct. Since the residual graph contains negative edge, we must either use the method of Johnson's algorithm to re-weight the edges and use Dijkstra algorithm to calculate the shortest path, or use Bellman-Ford algorithm to calculate the shortest path.

We can use the distances $dis(u)$ calculated by the Dijkstra algorithm last time to update the potential values $\Phi(u)$ of each vertex. And the re-weight equations of each edge is as follows.

$$a'(u,v) = a(u,v) + \Phi(u) - \Phi(v)$$

Here are some discussions.

- For the new back edges (v,u), we must have $dis(u) + a(u,v) = dis(v)$ because it is in the shortest path. Therefore,
$$a'(v,u) = a(v,u) + \Phi(v) - \Phi(u) = -a(u,v) - dis(u) + dis(v) = 0$$
- For those edge (u,v) that are not affected, we must have $dis(u) + a(u,v) \geq dis(v)$ according to the Triangle Inequality. Therefore,
$$a'(u,v) = a(u,v) + \Phi(u) - \Phi(v) = a(u,v) + dis(u) - dis(v) \geq 0$$

Therefore, all the costs are non-negative and we can use Dijkstra algorithm to find the shortest path, instead of Bellman-Ford algorithm.

The implementation of the algorithm is in code/MCMF.cpp. We also provide another version of implementation, which uses Bellman-Ford algorithm instead of re-weighting and the Dijkstra algorithm, and it is in code/MCMF-BellmanFord.cpp. Both of them gives us the answer of the input data in "MCMF.in", which is displayed as follows.

```
Max Flow = 14098
Min Cost (under max flow) = 5290116
```

Therefore,

- Max flow = 14098;
- Minimum Cost of Max Flow = 5290116.

□

Remark: The source code *SCC.cpp*, and the input data *SCC.in* and *MCMF.in* are attached on the course webpage. Please include your .pdf, .tex, .cpp files for uploading with standard file names.