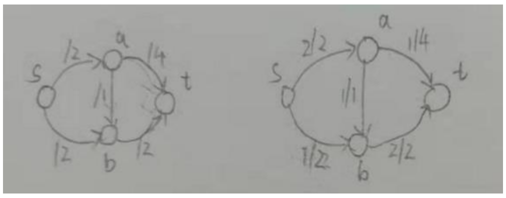
Part 6 Max Flow

(**Example 1**) Consider the following directed graph G=(V, E) with two special vertexes, i.e., source s and sink t. Each edge $e \in E$ is associated with a **capacity** $c(e)$ value and a **flow** $f(e)$, denoted as '$f(e)/c(e)$'. We want to send as much flow as possible from s to t.

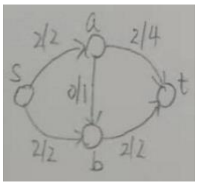
For each **vertex** $v \in V - \{s,t\}$, the **flow** is the **conserved** (守恒), where the sum of flow **coming into** v is **equal to** the sum of flow **coming out of** v. For each **edge** $e \in E$, there should be $f(e) \le c(e)$. Especially, the **flow** is **saturated** (饱和) if $f(e) = c(e)$.

A **feasible flow** is as follow



In the aforementioned flow, the paths $s \to a \to t$, $s \to b \to t$, and $s \to a \to b \to t$ cannot send more flow, since the flows w.r.t. $s \to a$ and $b \to t$ are saturated.

More flow can be sent from s to t based on the following assignment of flow:



Notes: The aforementioned example can model the **flows of 'traffic'**, e.g., water, cars, etc.

Notes: It can also be used to model the problems that on surface don't seem to be related to flow, e.g., the **bipartite matching** problem.

(**Definition 1**) (**Network Flow Maximization**) Given a directed graph G=(V, E) with two special vertexes s and t, called the *source* and *sink* respectively, each edge $e \in E$ is associated with a *capacity* $c(e) \ge 0$. Such a network is defined as a **flow network**.

A **flow** is an **assignment of value to edges** E such that:

(1) (**Capacity Constraint**) for each edge $e \in E$, $f(e) \le c(e)$ (it will be convenient to assume that $f(e)$=0, if there is no such edge e; especially, there are **no edges** coming into s and leaving t);

(2) (**Flow Conservation**) for each vertex $v \in V - \{s,t\}$, we have

$$\sum_{u \in V} f(u,v) = \sum_{w \in V} f(v,w).$$

Given a flow f, the **value of the flow**, denoted as $|f|$ is defined as

$$|f| = \sum_{v \in V} f(s,v) = \sum_{v \in V} f(v,t).$$

Given a **flow network**, **source** s, and **sink** t, the **goal** of **Network Flow Maximization** is to find the **maximum value of flow** $|f|^*$ from s to t.

(**Definition 2**) A **cut** (or more precisely an s-t **cut**) is a **partition** (A, V-A) is to partition the vertex set A into **2 groups** such that $s \in A$ and $t \in V - A$. The **capacity of a cut** is the quantity

$$c(A)=\sum_{u\in A, v\in V-A} c(u,v),$$

which is **not symmetric**, i.e., $\sum_{u\in A, v\in V-A} c(u,v) \neq \sum_{u\in A, v\in V-A} c(v,u)$.

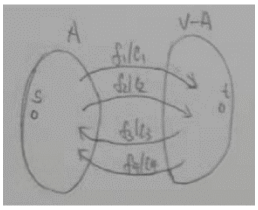
(**Definition 3**) Given a flow f and a cut (A, V-A), we define the **network flow out of** A (denoted as $f(A)$) as follow

$$f(A)=\sum_{a\in A, b\notin A} f(a,b)-\sum_{a\in A, b\notin A} f(b,a).$$

(**Example 2**) In the following example, we have

$$f(A)=f_1+f_2-f_3-f_4,$$

$$c(A)=c_1+c_2.$$



(**Lemma 1**) Consider **any flow** f and **any cut** (A, V-A). The **value of flow** f (i.e., $|f|$) is equal to the **network flow out of** A, i.e.,

$$|f|=f(A).$$

Proof of Lemma 1. For all the vertexes in A, consider the following quantity:

$$L=\sum_{u\in A}[\underbrace{\sum_{v\in V} f(u,v)}_{\text{flow leaving } u}-\underbrace{\sum_{v\in V} f(v,u)}_{\text{flow entering } u}].$$

Due to the **conservation constraint**, for an arbitrary vertex $u\in A-\{s\}$, the sum of flow **leaving** u should be equal to the sum of flow **entering** u, where we have

$$\sum_{v\in V} f(u,v)-\sum_{v\in V} f(v,u)=0.$$

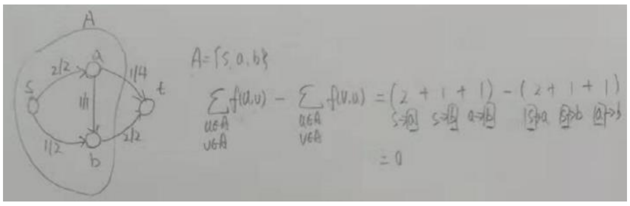
Hence, we have

$$L=\sum_{v\in V} f(s,v)+\sum_{u\in A-\{s\}}[\sum_{v\in V} f(u,v)-\sum_{v\in V} f(v,u)]=\sum_{v\in V} f(s,v)=|f|.$$

Namely, the considered quantity L is equal to the **value of flow** $|f|$.

Look at L from **an edge perspective**. Consider the following three cases.

(1) For any edge (u, v) with $u,v\in A$, edge (u, v) contributes $f(u, v)$ to vertex u and also contributes $-f(u, v)$ to vertex v, which together contribute 0 to the value of L.



(2) For any edge (u, v) with $u \in A$ but $v \notin A$, edge (u, v) contributes $f(u, v)$ to vertex u, which only contributes $f(u, v)$ to the value of L.

(3) For any edge (v, u) with $u \in A$ but $v \notin A$, edge (v, u) contributes $-f(u, v)$ to vertex u, which only contributes $-f(u, v)$ to the value of L.

Hence, we have

$$\begin{aligned} L &= [\sum_{u\in A, v\in A} f(u,v) - \sum_{u\in A, v\in A} f(v,u)] + \sum_{u\in A, v\notin A} f(u,v) - \sum_{u\in A, v\notin A} f(u,v) \\ &= 0 + \sum_{u\in A, v\notin A} f(u,v) - \sum_{u\in A, v\notin A} f(u,v) \\ &= f(A) \end{aligned}.$$

Namely, the considered quantity L is also equal to **network flow out of** A, i.e., $f(A)$.

In summary, we have

$$|f| = f(A).$$

(**Lemma 2**) For **any flow** f and **any** corresponding s-t **cut** (A, V-A), the **value of flow** $|f|$ is **at most** the **capacity of the cut** $c(A)$, i.e.,

$$|f| \le c(A).$$

Proof of Lemma 2. By **Lemma 1**, we have

$$|f| = f(A) = \sum_{u\in A, v\notin A} f(u,v) - \sum_{u\in A, v\notin A} f(v,u) \le \sum_{u\in A, v\notin A} c(u,v) - \sum_{u\in A, v\notin A} 0 = c(A),$$

which finish the proof.

(**Corollary 1**) Suppose we're able to find a flow f and an s-t cut (A, V-A), such that the value of the flow $|f|$ is equal to the capacity of the cut $c(A)$. Then, **the flow is optimal** and **the cut has the minimum capacity**.

Notes: By **Lemma 2**, for any flow f and any s-t cut (A, V-A), we have

$$|f| \le c(A).$$

Namely, the maximum value of $|f|$ is $c(A)$ w.r.t. any s-t cut (A, V-A) on f. The minimum value of $c(A)$ is $|f|$ w.r.t. the corresponding flow f.

(**Definition 4**) (**Residual Network**) Given a flow f, define a **residual network** (w.r.t. f) as another flow network with the **same vertex set** V, **same source** s, and **same sink** t. The **edge set** of the residual network is constructed in the following way.

Suppose there's an **edge** (u, v) with **capacity** $c(u, v)$ and **flow** $f(u, v)$ in the **original network**.

(1) If $f(u, v)<c(u, v)$, then introduce an **edge** (u, v) with **capacity** $c(u, v)-f(u, v)$. Edge (u, v) is defined as a **forward edge**.

(2) Further, if $f(u, v)>0$, then introduce an **edge** (v, u) with **capacity** $f(v, u)$. Edge (v, u) is defined as a **back edge**.

Especially, capacities of edges in the residual network are called as **residual capacities**. There's an easy observation that if we can push flow through this **residual network**, then we can push this additional flow in the **original network**.

(**Definition 5**) (**Ford-Fulkerson Method**) Suppose we have a **flow** f in the **original network**.

(1) First, we construct the **residual network** (w.r.t. f).

(2) Find **a simple path** from s to t in the residual network (defined as an **augmenting path**).

(3) Find the **minimum capacity** Δ of any edge on this path.

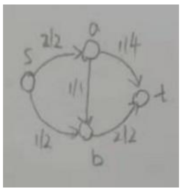
(4) Update the flow in the original network to push extra flow Δ.

(5) Repeat this entire procedure.

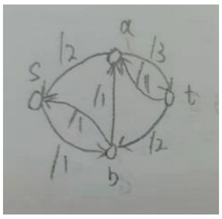
Initially, we start with **flow** 0. We terminate the procedure, when there's no s-t path in the **residual network**.

Notes: If all capacities are **integers**, the **Ford-Fulkerson method** will terminate in **finite steps** (but not in general if capacities are irrational (无理数)).

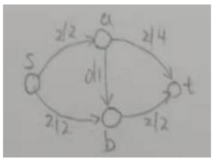
(**Example 3**) Consider the following **flow network**



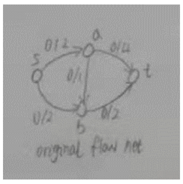
One can construct the corresponding **residual network**:

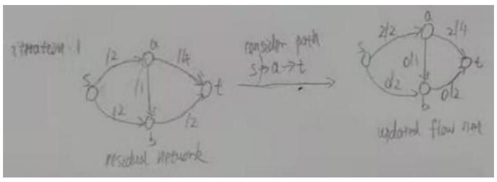


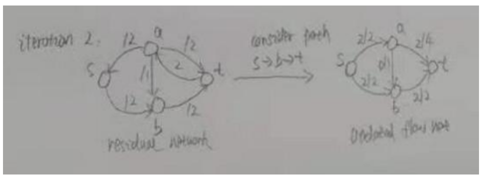
There's a path from s to t, i.e., $s \to b \to a \to t$, with minimum capacity of 1. Thus, the original flow network can be further improved:

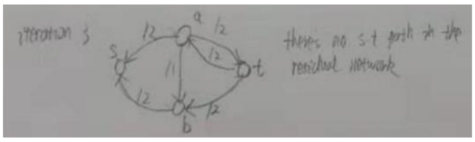


An example of the **Ford-Fulkerson Method** is as follow:









(**Theorem 1**) (**Correctness of Ford-Fulkerson** (F-F) **Method**) If there's **no augmenting path** for flow *f* (i.e., there's no path from *s* to *t* in the **residual network** w.r.t. *f*), then the flow *f* is **optimal**.

Proof of Theorem 1. Let *A* be the **set of vertexes** in the **residual network** that are **reachable** from *s*. Then, *V-A* is the set of vertexes that are not reachable from *s* in the residual network.

When this is an *s-t* cut (*A*, *V-A*) in the residual network, there is no path from *s* to *t* (i.e., no **augmenting path**). Namely, there is no edges from *A* to *V-A* in the **residual network**, which indicates that $f(u, v)=c(u, v)$ and $f(v, u)=0$ for $u \in A$ and $v \notin A$.

Further, for the network flow out of *A*, we have

$$f(A) = \sum_{u \in A, v \notin A} f(u,v) - \sum_{u \in A, v \notin A} f(v,u) = \sum_{u \in A, v \notin A} c(u,v) - 0 = c(A) \cdot$$

By **Lemma 1**, we have $|f| = f(A)$. By **Lemma 2**, we further have

$$|f| = f(A) = c(A) \cdot$$

Hence, the flow *f* is **optimal**.

(**Theorem 2**) (**Max-Flow Min-Cut**) The following 3 conditions are equivalent for a flow *f* in a network:

(1) There's a **cut** (*A, V-A*) whose capacity *c*(*A*) is equal to the value of flow, i.e., $c(A)=|f|$;

(2) The flow *f* is **optimal**;

(3) There's no **augmenting path** for flow *f*.

Proof of **Theorem 2**. $(1) \Rightarrow (2)$ We have already proved in **Lemma 1** and **Lemma 2**. It relies on the fact that the **capacity of any cut** (i.e., *c*(*A*)) is an **upper bound** on the value of any flow (i.e., $|f|$), namely $|f| \leqslant c(A)$.

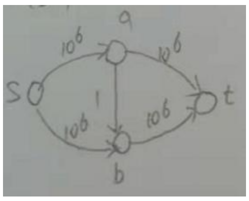
$(2) \Rightarrow (3)$ One can prove it by contradiction. For an optimal flow f^*, if thee exist an augmenting path (in the residual network), then the flow f^* can still be improved. It contradicts with the fact that f^* is an optimal flow.

$(3) \Rightarrow (1)$ We have already proved in **Theorem 1**. e.g., Let *A* be the set of vertices reachable from *s*. Then, (*V-A*) is the set of vertices that are not reachable from *s*.

Notes: If a flow is **optimal**, its optimality can always be established by exhibiting a **cut** (*A*, *V-A*) whose **capacity** is equal to the value of the flow $|f|$, i.e., $|f|=c(A)$.

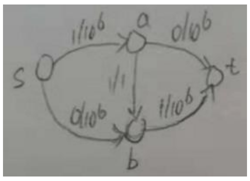
Notes: In any flow network, the **value of maximum flow** equals the **capacity of the minimum cut**, even if the capacities are non-integral.

(**Example 3**) Consider the following flow network, where all the capacities are integers.

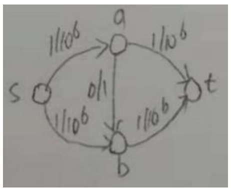


The maximum value of flow is $|f| = 2 \times 10^6$.

For the F-F Method, assume we pick the augmenting path *s*→*a*→*b*→*t* in the 1st iteration. Then, we have the following updated flow network:



In the 2nd iteration, assume we pick the augmenting path *s*→*b*→*a*→*t*. Then, we have the following updated flow network:



By using the similar strategy to select the augmenting path, we need 2×10^6 iterations to find the maximum flow.

However, if we pick the augmenting path *s*→*a*→*t* and *s*→*b*→*t* in the 1st and 2nd iteration, respectively, we can find the maximum flow in just 2 iterations.

The aforementioned example indicates that **F-F Method** may have **exponential running time** in the **input size**. Different strategy to select the augmenting path can result in different running time. Hence, one needs to carefully choose the augmenting path for a given network flow.

Notes: To speed up the convergence of **F-F method**, one needs to choose the augmenting path more carefully. A possible strategy is to choose the augmenting path along which the most flow can be sent (from *s* to *t*).

(**Definition 5**) Define the **fatness** of a path in a flow network to be the **minimum capacity** of the edges on the path.

(**Algorithm 1**) (**Fattest Path Algorithm**) At each iteration of **F-F method**, choose the **fattest augmenting path** in the **residual network**.

Notes: We assume that all the **capacities** of the flow network are **integers**.

(**Fact 1**) Given a **directed graph** G=(V, E) with **non-negative edge weights**, as well as two vertices *s* and *t*, we can find a **shortest path** in $O(|E|\log|V|)$ using the **Dijkstra's Algorithm** (via heap). In the **Dijkstra's Algorithm**, the **length of a path** is simply the sum of the weights of all the edges on the path.

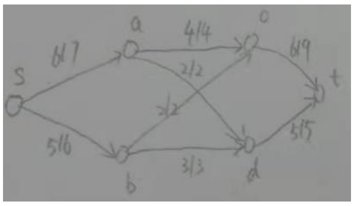
(1) Suppose we can modify the '**length of a path**' to refer to the **minimum weight** of an edge on the path, i.e., the **fatness**.

(2) And '**shortest path**' refers to the path with **maximum 'length'** (as defined in (1)), i.e., the **fattest path**.

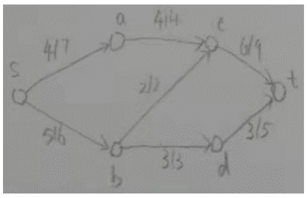
Thus, a straight-forward modification of the **Dijkstra's Algorithm** can solve our problem in the same time, i.e., $O(|E|\log|V|)$.

(**Example 4**) (**Flow Decomposition**) Consider the following flow network. The value of flow is

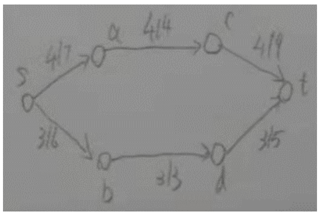
$|f|$=6+7=11. Let A={s, a, b}. The capacity of the cut (A, V-A) is $c(A)$=4+2+2+3=11. Since we have $|f|=c(A)$, the maximum value of flow is $|f^*|$=11.



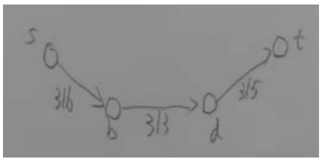
If we remove the path $s \rightarrow a \rightarrow d \rightarrow t$ with 2 units of flow, we have the following flow network with the value of flow $|f|$=9.



If we remove the path $s \rightarrow b \rightarrow c \rightarrow t$ with 2 units of flow, we have the following flow network with the value of flow $|f|$=7.



If we remove the path $s \rightarrow a \rightarrow c \rightarrow t$ with 4 units of flow, we have the following flow network with the value of flow $|f|$=3.



If we remove the path $s \rightarrow b \rightarrow d \rightarrow t$ with 3 units of flow, all the edges are deleted. Then, the value of flow is $|f|$=0.

Notes: In each iteration of the aforementioned flow decomposition procedure, at least one edge will be deleted from the flow network. It indicates that an **upper bound** of the **number of iterations** is $|E|$, i.e., **the number of edges**.

(**Lemma 3**) Consider a flow f in a flow network. Then, there's a collection of **feasible flows** $\{f_1, f_2, \cdots, f_k\}$ and a collection of *s*-*t* **path** $\{p_1, p_2, \cdots, p_k\}$, such that:

(1) the value of f (i.e., $|f|$) is equal to the sum of flows $\{f_1, f_2, \cdots, f_k\}$, i.e., $|f| = \sum_{i=1}^{k} f_i$;

(2) flow f_i sends **positive flow** only on the **edges of path** p_i;

(3) $k \leq |E|$, where E denotes the set of edges in the original flow network.

Proof of **Lemma 3**. Consider the following **Flow Decomposition** procedure.

In the i-th iteration, find a **path** p_i from s to t such that each **edge** on this path carries a **positive flow**. Let $f_{\min}$ denote the **minimum flow** associated with any **edge** of the **path**. Then, f_i will be the flow of value $f_{\min}$ alone the path p_i.

Finally, we reduce the flow on all the edges of p_i by removing $f_{\min}$ and go to the next iteration.

The procedure stops, when the value of flow is 0, i.e., $|f|=0$.

Note that in each iteration, the flow on **at least one of the edges** of path p_i goes to 0. Thus, the **number of paths** we can find is **at most** $|E|$, i.e., $k \le |E|$.

(**Corollary 2**) Let *OPT* denote the **value of maximum flow**. In a given flow network, there is a path from *s* to *t* (note that there're at most $|E|$ such paths), in which every edge has capacity at least $OPT/|E|$, i.e., $\ge OPT/|E|$. The **fattest path** must carry flow $OPT/|E|$.

Notes: **Corollary 2** indicates that in each iteration of the **Flow Decomposition**, the value of flow **decrease** at least the $OPT/|E|$. After the 1st iteration, the remaining value of flow is at most $OPT(1-1/|E|)$. After the 2nd iteration, the remaining value of flow is at most $OPT(1-1/|E|)^2$.

In particular, suppose $|E|\ge 2$. Then, after the 1st iteration, the remaining value of flow is about $OPT/2$. After the 2nd iteration, the remaining value of flow is about $OPT/4$. Similarly, in the *t*-th iteration, the remaining value of flow is about $OPT/2^t$.

Thus, the **number of iterations** is about $\log(OPT)$.

(**Theorem 3**) The time complexity of the **Fattest Path Algorithm** (i.e., **Algorithm 1**) is $O(|E|^2 \log|V| \cdot \log OPT)$, where *OPT* is the value of maximum flow.

Proof of **Theorem 3**. Let $m=|E|$. Let f_i denote the **flow value** after the *i*-th iteration. Let res_i denote the **optimal flow value** of the **residual network** after the *i*-th iteration. Then, we have

$$res_i=OPT-f_i.$$

By **Corollary 2**, in the $(i+1)$-th iteration, we find a flow with capacity $c_{i+1} \ge res_i/(2m)$ in the **residual network**. (Note that a residual network consists of both **forward edges** and **back edges**, with the total number of edges at most $2m$). Thus, we have the following relationship between res_i and res_{i+1}:

$$res_i=res_{i+1}+c_{i+1},$$

which indicates that

$$\begin{aligned} res_{i+1} &= res_i - c_{i+1} \\ &\le res_i(1-\frac{1}{2m}) \end{aligned}.$$

Note that we also have $res_0=OPT$, which further indicates that

$$res_t \le OPT(1-\frac{1}{2m})^t.$$

Let $t=2m\log(OPT)$. Then, we have

$$\begin{aligned} res_t &\le OPT\cdot(1-\frac{1}{2m})^{2m\log(OPT)} \\ &= OPT\cdot[(1-\frac{1}{2m})^{2m}]^{\log(OPT)} \\ &\le OPT\cdot\frac{1}{e^{\log(OPT)}} \\ &= \frac{OPT}{OPT}=1 \end{aligned},$$

which indicates that when $t>2m\log(OPT)$, we have $res_i<1$.

Note that we assume all the capacities of edges are integer. Thus, $res_i=0$ when $t>2m\log(OPT)$. It implies that **the number of iterations** of the **Fattest Path Algorithm** (**Algorithm 1**) is at least

$$2m\log(OPT)=O(|E|\log(OPT)).$$

In each iteration, we use the **modified Dijkstra's Algorithm** to find the fattest path, with the **complexity** $O(|E|\log|V|)$.

In summary, the total time complexity of **Algorithm 1** is $O(|E|^2\log|V|\log(OPT))$.

(**Definition 6**) (**Strongly & Weakly Polynomial Time**) An algorithm runs in **strongly polynomial time**, if we assume **unit-time arithmetic operations** (e.g., addition, substruction, etc.), then the running time is polynomial in **the number of numerical quantities** given in the **input**. Otherwise, it runs in **weakly polynomial time**.

Notes: The distinction between **strongly polynomial time** & **weakly polynomial** time is meaningful only when the **input** involves **integers**, e.g., capacities of a flow network are all integers.

Notes: Both the aforementioned two concepts are **polynomial** in the **input size**, but strongly polynomial time is nicer, e.g., in the sense that **the number of iterations** just depends on **the number of edges and vertices**.

Notes: For the **Maximum Flow** problem, the algorithm is **strongly polynomial** if it runs in time **polynomial** in **the number of vertices and edges** (assuming unit cost of arithmetic operation).

Notes: In particular, the **Fattest Path Algorithm** (**Algorithm 1**) is polynomial in input size, but not strongly polynomial, i.e., it's a **weakly polynomial** algorithm.

Notes: For **Linear Programming**, the **Ellipsoid Method** and **Interior Point Method** are both **weakly polynomial-time algorithms**. No strongly polynomial time algorithm is known so far. In fact, this is a big open problem.

(**Algorithm 2**) (**Edmonds-Karp Algorithm**) There exist a **strongly polynomial-time** algorithm for the **Maximum Flow** problem, which gives another specific implementation of the **F-F method**.

Concretely, in each iteration, we find an s-t path in the **residual network** with the **fewest number of edges**. We can do this in linear time using **BFS** with the **complexity** $O(|V|+|E|)=O(|E|)$.

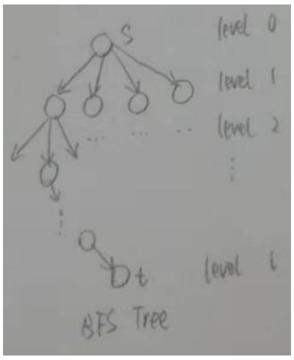
Notes: In the **residual network**, there're **no isolated vertices** and any vertices can connected to s and t, so we have $O(|V|+|E|)=O(|E|)$.

(**Theorem 4**) (i) If at a certain iteration, the **length** of the shortest s-t **path** is l, then at every **subsequent iteration**, it's **at least** l (i.e., $\geq l$).

(ii) Furthermore, after **at most $|E|$ iterations**, the **length** of the shortest s-t **path** becomes **at least** $(l+1)$ (i.e., $\geq(l+1)$).

(iii) Note that the **maximum length** of an s-t path should be $(|V|-1)$, so the **number iterations** of **Algorithm 2** is **at most** $(|V|-1)|E|=O(|V||E|)$. Since each iteration takes $O(|E|)$ time, the **total running time** of **Algorithm 2** is $O(|V||E|^2)$

Proof of **Theorem 4**. Consider the **residual network** after t iterations of **Algorithm 4**. Since we use BFS to find the s-t path with fewest number of edges, we can construct the following **BFS tree** with l levels:

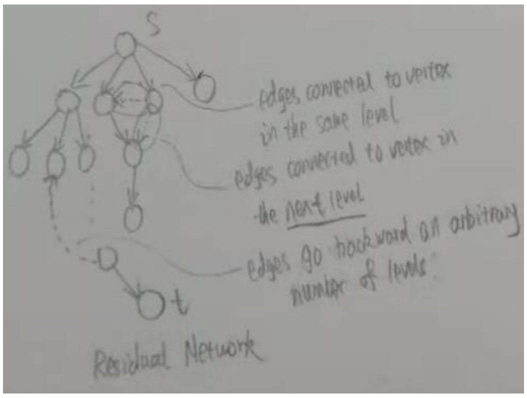


Define the **edges** go downwards in the **<u>residual network</u>** as **forward edges**. There 3 types of edges in the **residual network**:

(i) **Forward edges** can only go down one level in the BFS tree;

(ii) Edges can connect to vertices at the **same level**;

(iii) Edges can also go **backwards** an arbitrary number of levels.

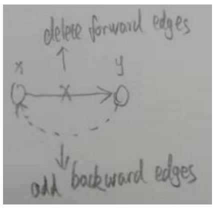


The **shortest path** goes one level each time until reaches t, i.e., with length l.

Subsequently, in iteration (t+1), we push flow on the selected shortest s-t path P, which makes at least one edge on P **saturated**. Accordingly, in the **residual network**, we have the following observation:

(i) All edges on the selected path P, which are already **saturated**, **disappear**;

(ii) We may introduce edges going upwards the BFS tree (i.e., **backward edges**) w.r.t. edges on the selected path P.



It implies that the shortest s-t path in the corresponding residual network will still have the **length** at least l (i.e., $\geq l$), which proves the first part of **Theorem 4**.

We further prove the second part of **Theorem 4**. Note that if the **length** of the shortest s-t path remains l (after t iterations), then the shortest s-t path should consist entirely of **forward edges** after t iterations. Since **at least 1 edge** in a shortest s-t path will **disappear** after each iteration, <u>the length can stay at l for at most $|E|$ iterations</u>, which proves the second part of **Theorem 4**.

Furthermore, since the **largest length** of the selected s-t path is **at most** ($|V|$-1)=$O(|V|)$, the total **number of iterations** of **Algorithm 2** is **at most** $O(|E||V|)$. In each iteration, the running time of

BFS (to find the shortest s-t path) is $O(|E|)$, so the overall running time of **Algorithm 2** is $O(|V||E|^2)$.

Notes: The complexity of the **Fattest Path Algorithm** (**Algorithm 1**) is $O(|E|^2\log|V|\log(OPT))$, while the complexity of the **E-K Algorithm** (**Algorithm 2**) is $O(|E|^2|V|)$. When we compare $O(|E|^2\log|V|\log(OPT))$ with $O(|E|^2|V|)$, we compare $O(\log|V|\log(OPT))$ with $O(|V|)$. Sometimes, $O(\log|V|\log(OPT))$ can be better than $O(|V|)$, which depends on how large are the flow capacities.

Notes: $O(|E|^2\log|V|\log(OPT))$ is **weakly polynomial time**, while $O(|E|^2|V|)$ is **strongly polynomial time**.