CSS.201.1 Algorithms

Instructor: Umang Bhaskar

TIFR 2024, Aug-Dec

Scribe: Soham Chatterjee

sohamchatterjee999@gmail.com
Website: sohamch08.github.io

Contents

Chapter 1

Maximum Flow

1.1 Flow

Suppose we are given a directed graph $G = (V, E)$ with a source vertex s and a target vertex t. And additionally for every edge $e \in E$ we are given a number $c_e \in \mathbb{Z}_0$ which is called the capacity of the edge.

Definition 1.1.1: Flow

An $s-t$ flow is a function $f : E \to \mathbb{R}_0$ which satisfies the following:

① $\forall\, e \in E, f(e) \leq c_e$

② $\forall\, v \in V \setminus \{s,t\}, \sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e)$

Also the value of a flow f is denoted by $|f| := \sum_{e \in out(s)} f(e)$.

Before proceeding into the setup and the problem first we will assume some things

Assumption.

- $in(s) = \emptyset$ *i.e. there is no edge into* s.
- $out(t) = \emptyset$ *i.e. there is no edge out of* t.
- *There are no parallel edges*

Lemma 1.1.1

For any flow f, $|f| = \sum_{e \in in(t)} f(e)$

Proof: We have for every edge $e \in E$, $\exists\, v \in V$ such that $e \in in(v)$ and $\exists\, u \in V$ such that $e \in out(u)$. Hence we get

$$\sum_{e \in E} f(e) = \sum_{v \in V} \sum_{e \in in(v)} f(e) = \sum_{v \in V} \sum_{e \in out(v)} f(e) \implies \sum_{v \in V} \left[\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) \right] = 0$$

Now we know $\forall\, v \in V \setminus \{s,t\}$. $\sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e)$. Therefore we get

$$\sum_{v \in V} \left[\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) \right] = 0 \implies \sum_{v \in \{s,t\}} \left[\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) \right] = 0 \implies \sum_{e \in out(s)} f(e) - \sum_{e \in in(t)} f(e)$$

Hence we have $|f| = \sum_{e \in in(t)} f(e)$. ∎

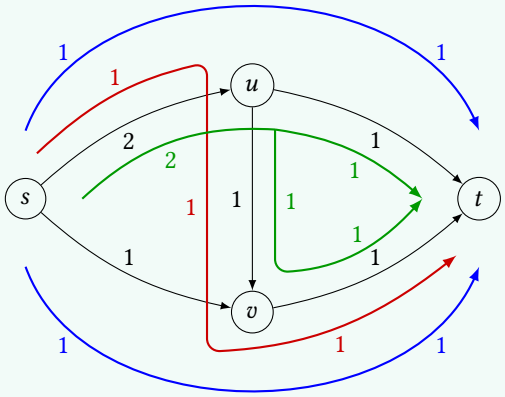
Max Flow

Input: A directed graph $G = (V, E)$ with source vertex s and target vertex t and for all edge $e \in E$ capacity of the edge $c_e \in \mathbb{Z}_+$

Question: Given such a graph and its capacities find an $s - t$ flow which has the maximum value

Example 1.1.1

Consider the following directed graph with capacities: $V = \{s, t, u, v\}$, $c_{s,u} = 2, c_{s,v} = c_{u,t} = c_{v,t} = c_{u,v} = 1$. Firstly the following function: $f' : f'(s,u) = 2 = f(u,t)$. It is not a flow since $f(u,t) = 2 > 1 = c_{u,t}$. Now we define three different flow functions:



- f: $f(s,u) = f(u,v) = f(v,t) = 1$ and otherwise 0. Therefore $|f| = 1$
- g: $g(s,u) = g(u,t) = 1$, $g(s,v) = g(v,t) = 1$ and otherwise 0. Therefore $|g| = 2$
- h: $h(s,u) = 2$, $h(u,t) = h(u,v) = h(v,t) = 1$ and otherwise 0. Therefore $|h| = 2$

Notice here g and h has the maximum flow value.

1.2 Ford-Fulkerson Algorithm

Definition 1.2.1: Residual Graph

Given a directed graph $G = (V, E)$ and capacities C_e for all $e \in E$ and an $s - t$ flow f the residual graph $G_f = (V, E_f)$ has the edges with the following properties:

① If $(u,v) \in E$ and $f(u,v) > 0$ then $(v,u) \in E_f$ and $c^f_{v,u} = f(u,v)$. Such an edge is called a *backward* edge.

② If $(u,v) \in E$ and $f(u,v) < c_{u,v}$ then $(u,v) \in E_f$ and $c^f_{u,v} = c_{u,v} - f(u,v)$. It is called *forward* edge.

Algorithm 1: Ford-Fulkerson

Input: Directed graph $G = (V, E)$, source s, target t and edge capacities C_e for all $e \in E$
Output: Flow f with maximum value

```
1 begin
2     for e ∈ E do
3         f(e) = 0
4     while ∃ s ⇝ t path P in G_f do
5         δ ⟵ min_{e∈P}{c_e^f} for e = (u,v) ∈ P do
6             if e is Forward Edge then
7                 f(u,v) ⟵ f(u,v) + δ
8             else
9                 f(u,v) ⟵ f(v,u) − δ
```

We call one iteration of the While loop at line 4 *Flow Augmentation*.

Lemma 1.2.1

At any iteration the f' obtained after the flow augmentation of the flow f is a valid flow

Proof: At any iteration let P be the path from $s \rightsquigarrow t$ and $\delta = \min_{e \in P} c_f(e)$. Let f' be the new function such that for each $(u,v) \in P$ if (u,v) is forward edge in G_f then $f'(u,v) = f(u,v) + \delta$ and if (u,v) is backward edge in G_f then $f'(v,u) = f(v,u) - \delta$ and for other edges $e \in E \setminus P$, $f'(e) = f(e)$.

Now since $\delta = \min_{e \in P} c_f(e)$, $c_f(e) \geq \delta$ for all $e \in P$. Hence if (u,v) is backward edge then $(v,u) \in E$ and $c_f(u,v) = f(u,v)$. Hence $f'(v,u) = f(v,u) - \delta \geq 0$. Therefore for all $e \in E$, $f'(e) \geq 0$.

Now first we will show $f'(e) \leq c_e$ for all $e \in E$. If $(u,v) \in P$ is a forward edge then $(u,v) \in E$ and $c_f(u,v) = c_{u,v} f(u,v)$. Therefore $f'(u,v) = f(u,v) + \delta \leq f(u,v) + c_{u,v} - f(u,v) = c_{u,v}$. Now if $(u,v) \in P$ is a backward edge then $(v,u) \in E$ and $c_f(u,v) = f(u,v)$. Therefore $f'(u,v) = f(u,v) - \delta \leq f(u,v) \leq c_{u,v}$. For other edges $e \in E \setminus P$, $f'(e) = f(u) \leq c_e$. Therefore $f'(e) \leq c_e$ for all $e \in E$

Now we will prove for all $v \in V \setminus \{s,t\}$, $\sum_{e \in in(v)} f'(e) = \sum_{e \in out(v)} f'(e)$. If v is not in the path P in G_f then, $f'(e) = f(e)$ for all edges $e \in in(v) \cup out(v)$. Hence the condition is satisfied for such vertices. Suppose v is in the path P. Then there are two edges e_1 and e_2 in P which are incident on e. If both are forward edges or both are backward edges then one of them is in $in(v)$ and other one is in $out(v)$. WLOG suppose $e_1 \in in(v)$ and $e_2 \in out(v)$ we have

$$\sum_{e \in in(v)} f'(e) = \sum_{e \in in(v) \setminus \{e_1\}} f(e) + f(e_1) \pm \delta = \sum_{e \in out(v) \setminus \{e_2\}} f(e) + f(e_2) \pm \delta = \sum_{e \in out(v)} f'(e)$$

If one of e_1, e_2 forward edge and other one is backward edge then either $e_1, e_2 \in in(v)$ (when e_1 is forward and e_2 is backward) or $e_1, e_2 \in out(v)$ (when e_1 is backward and e_2 is forward). Now if $e_1, e_2 \in in(v)$, $f'(e_1) + f'(e_2) = f(e_1) + \delta + f(e_2) - \delta = f(e_1) + f(e_2)$ and if $e_1, e_2 \in out(v)$ then $f'(e_1) + f'(e_2) = f(e_1) - \delta + f'(e_2) + \delta = f(e_1) + f(e_2)$. Hence

$$\sum_{e \in in(v)} f'(e) = \sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e) = \sum_{e \in out(v)} f'(e)$$

Hence f' is a valid flow. ■

Lemma 1.2.2

At any iteration Given G_f if the flow, f' obtained after flow augmentation of f by δ then

$$|f'| = |f| + \delta$$

Proof: Since we augment flow along an $s \rightsquigarrow t$ path, the first edge of the path is always in $out(s)$. Let the first edge is $e = (s,u)$. Now e has to be a forward edge because otherwise $(u,s) \in E$ and then there is an incoming edge in G which is not possible. Hence

$$|f'| = \sum_{e \in out(s)} f'(e) = \sum_{e \in out(s) \setminus \{e\}} f(e) + f'(e) = \sum_{e \in out(s) \setminus \{e\}} f(e) + f(e) + \delta = \sum_{e \in out(s)} f(e) + \delta = |f| + \delta$$

Hence we have the lemma. ■

Lemma 1.2.3

At every iteration of the Ford-Fulkerson Algorithm the flow values and the residual capacities of the residual graph are non-negative integers.

Proof: Initial flow and the residual capacities are non-negative integers. Let till i^{th} iteration the flow values and the residual capacities were non-negative integers. Let the flow after i^{th} iteration was f. Hence $\forall\, e \in E, f(e) \in \mathbb{Z}_0$. Therefore in the G_f for all $e \in E_f$, $c_f(e) \in \mathbb{Z}_0$. Hence $\delta \in \mathbb{Z}_0$. Therefore $\forall\, e \in E, f'(e) \in \mathbb{Z}_0$. And therefore for all $e \in E_{f'}$ where $G_{f'}$ is the residual graph of the flow f', $c_{f'}(e) \in \mathbb{Z}_0$. Hence by mathematical induction the lemma follows. ■

At any iteration let P be the path from $s \rightsquigarrow t$. Then for all $e \in P$, $c_f(e) > 0$. Therefore $\delta = \min_{e \in P} c_f(e) \geq 1$. Therefore the algorithm must stop in at most $\sum_{e \in out(s)} c_e$ since we can have the value of a flow to be at max the value of the sum of capacities of edges in $out(s)$ and therefore we can increase the flow at max that many times.

Lemma 1.2.4

If f is a max flow then there is no $s \rightsquigarrow t$ path in G_f.

Proof: Suppose there is an $s \rightsquigarrow t$ path P in G_f. We will show that then f is not a max flow following the algorithm. Then $\forall\ e \in P$, $c_f(e) > 0$. Hence $\delta = \min_{e \in P} c_f(e) \geq 1$. Now after the flow augmentation process of f by δ we get a new valid flow f' by Lemma 1.2.1 and by Lemma 1.2.2 we have $|f'| = |f| + \delta > ||f|$. Hence f is not a maximum flow. Hence contradiction. Therefore there is no $s \rightsquigarrow t$ path in G_f. ∎

1.2.1 Max Flow Min Cut

Definition 1.2.2: Cut Set

For a graph $G = (V, E)$ and a subset $A \subseteq V$, the cut $(A, V \setminus A)$ is a bipartition of V where the edges E_A of the graph $G_A = (A, V \setminus A, E_A)$ is the set $E_A = E \cap (A \times (V \setminus A))$.

Now if s, t are two vertices of G then an $s - t$ *Cut* $(A, V \setminus A)$ is a cut such that $s \in A$ and $t \in V \setminus A$.

Now we define for a cut $(A, V \setminus A)$ the *Capacity of the Cut* $(A, V \setminus A) = \sum_{e \in E_A} c_e$. For an $s - t$ cut $(A, V \setminus A)$ we denote the capacity of the cut by $cap(A)$ A *Min* $s - t$ *Cut* is a $s - t$ cut of minimum capacity. Then we have the following relation between cut and flow.

Lemma 1.2.5

Given a graph $G = (V, E)$, s, t, $c_e \in \mathbb{Z}_0$ for all $e \in E$ for any flow f and a $s - t$ cut $(A, V \setminus A)$

$$|f| \leq cap(A)$$

Proof: Given f and the $s - t$ cut $(A, V \setminus A)$ we have

$$\begin{aligned}
|f| &= \sum_{e \in out(s)} f(e) \\
&= \sum_{v \in A} \left[\sum_{e \in out(v)} f(e) - \sum_{e \in in(v)} f(e) \right] \\
&= \sum_{\substack{e=(u,v),\\ u \in A, v \notin A}} f(e) - \sum_{\substack{e=(u,v),\\ u \notin A, v \in A}} f(e) && \text{[Edges for both endpoints in } A \text{ are canceled out]} \\
&= \sum_{e \in out(A)} f(e) - \sum_{e \in in(A)} f(e) \\
&\leq \sum_{e \in out(A)} f(e) \leq \sum_{e \in out(A)} c_e = cap(A)
\end{aligned}$$

Hence we have the lemma. ∎

Having this lemma we have have for any flow f and $s - t$ cut $(A, V \setminus A)$ we have

$$|f| \leq cap(A) \implies \max_f |f| \leq \min_{s-t \text{ cut } (A, V \setminus A)} cap(A)$$

So we have the following theorem that the value of maximum flow is equal to the capacity of minimum cut.

Theorem 1.2.6 Max Flow Min Cut

Given a graph $G = (V, E)$, $s, t, c_e \in \mathbb{Z}_0$ for all $e \in E$. Then the following are equivalent:

(1) f is a maximum flow.

(2) There is no $s \rightsquigarrow t$ path in G_f

(3) There exists an $s - t$ cut of capacity $|f|$

Proof:

(1) $\implies$ (2): This is by Lemma 1.2.4.

(2) $\implies$ (3): We are given a flow f such that there is no $s \rightsquigarrow t$ path in G_f. We will construct a $s - t$ cut which has the capacity $|f|$. Now take A to be all the vertices reachable from s in G_f. This is a valid $s - t$ cut since $s \in A$ and as there is no $s \rightsquigarrow t$ path in G_f, $t \notin A$. Now

$$|f| = \sum_{e \in out(A)} f(e) - \sum_{e \in in(A)} f(e)$$

Now $\forall\ e = (u, v) \in E$ where $u \in A$ and $v \notin A$ we have $c_{u,v} = f(u, v) \implies c_{u,v} - f(u, v) = 0$ since otherwise $c_{u,v} - f(u, v) \neq 0 \implies c_{u,v} > f(u, v) \implies (u, v) \in E_f$ and therefore v is reachable from s but $v \notin A$, contradiction. Therefore (u, v) is a backward edge and hence . Now $\forall\ e = (u, v) \in E$ where $u \notin A$ and $v \in A$ we have $f(u, v) = 0$ since otherwise $f(u, v) > 0 \implies (v, u) \in E_f$ and therefore u is reachable from s but $u \notin A$, contradiction. Hence we have

$$|f| = \sum_{e \in out(A)} f(e) - \sum_{e \in in(A)} f(e) = \sum_{e \in out(A)} c_e = cap(A)$$

(3) $\implies$ (1): Now by Lemma 1.2.5 we have for any flow f and $s - t$ cut

$$|f| \leq cap(A) \implies \max_f |f| \leq \min_{s-t\ cut\ (A, V \setminus A)} cap(A)$$

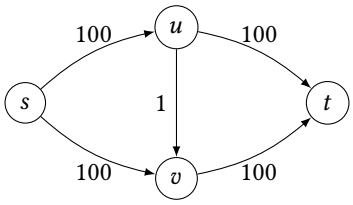
Now given f there exists an $s - t$ cut of capacity $|f|$. Hence f is a max flow.

■

Hence at the end of the Ford-Fulkerson Algorithm let the flow returned by the algorithm is f. The algorithm terminates when there is no $s \rightsquigarrow t$ path in G_f. Hence by Max Flow Min Cut Theorem we have f is a maximum flow. This completes the analysis of the Ford-Fulkerson Algorithm.

Since the capacities of the edges can be very large we want an algorithm return the maximum flow with running time $poly(n, m, \log c_e)$ where n is the number of vertices and m is number of edges and $\log c_e$ basically means number of bits at most needed to represent the capacities.

But Ford-Fulkerson algorithm takes does not run in $poly(n, m, \log c_e)$ instead $poly(n, m, c_e)$ as the while loop in the algorithm takes $poly(c_e)$ many iterations. For example in the following graph: it takes around 100 steps



and in general Ford-Fulkerson takes $O(|f_{\max}|)$ time. For this reason we will now discuss a modification of the Ford-Fulkerson Algorithm which takes $poly(n, m, \log c_e)$ time, Edmonds-Karp Algorithm.

1.2.2 Edmonds-Karp Algorithm

To get a $poly(n, m, \log c_e)$ time algorithm we will always pick the shortest $s \rightsquigarrow t$ path in the residual graph. This algorithm is known as the Edmonds-Karp Algorithm

Suppose f_i be the total flow after i^{th} iteration. And G_{f_i} be the residual graph with respect f_i. Then $f_0(e) = 0$ for all $e \in E$ and $G_{f_0} = G$. Also suppose $dist_i(v)$ = Shortest $s \rightsquigarrow v$ path distance in the residual graph G_{f_i}. Hence $dist_i(s) = 0$ for all i and $dist_i(t) = \infty$ at the end of the algorithm.

Note:-

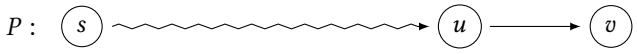
In i^{th} iteration of the Ford-Fulkerson Algorithm or Edmonds-Karp Algorithm if P is the $s \rightsquigarrow t$ in the residual graph G_{f_i} where $e = (u, v) \in P$ and $c_{f_i}(u, v) = \delta = \min_{e \in P} c_{f_i}(e)$ then the edge (u, v) is not present in the next residual graph $G_{f_{i+1}}$. Thus at least one edge disappears in each iteration of Ford-Fulkerson or Edmonds-Karp Algorithm.

Now we will prove following two lemmas which will help us to prove that the Edmond-Karp algorithm takes $O(mn)$ iterations.

Lemma 1.2.7

At any iteration i, $\forall\, v \in V$, $dist_i(v) \leq dist_{i+1}(v)$

Proof: Suppose this is not true. Then let i be the first iteration in which there exists a vertex $v \in V$ such that $dist_i(v) > dist_{i+1}(v)$. We pick such v which minimizes $dist_{i+1}(v)$. Consider the shortest path P from $s \rightsquigarrow v$ in $G_{f_{i+1}}$. Hence length of P, $|P| = dist_{i+1}(v)$. Let (u, v) be the last edge of P.



Then

$$dist_{i+1}(v) = dist_{i+1}(u) + 1 \geq dist_i(u) + 1$$

Here the last inequality follows because v is the vertex which has the minimum $dist_{i+1}(v)$ among all the vertices $w \in V$ which follows $dist_i(w) > dist_{i+1}(w)$. Now we will analyze case wise.

- **Case 1:** $(u, v) \in E_{f_i}$. Then
$$dist_i(v) \leq dist_i(u) + 1 \leq dist_{i+1}(v)$$
But this is not possible since $dist_i(v) > dist_{i+1}(v)$.
- **Case 2:** $(u, v) \notin E_{f_i}$. Then $(v, u) \in E_{f_i}$. Since $(u, v) \in E_{f_{i+1}}$ then we must have sent flow along (v, u). Since we take the shortest $s \rightsquigarrow t$ path in G_{f_i} in the algorithm we have $dist_i(u) = dist_i(v) + 1$. But then
$$dist_i(u) \leq dist_{i+1}(v) - 1 \implies dist_{i+1}(v) \geq dist_i(v) + 2$$
But this is not possible.

Hence contradiction ↯ Therefore for all iterations i, for all vertices $v \in V$, $dist_i(v) \leq dist_{i+1}(v)$. ■

Lemma 1.2.8

For any edge $e = (u, v) \in E$ the number of iterations where either (u, v) appears or (v, u) appears is at most $O(n)$ i.e.

$$\left|\left\{i \colon (u, v) \notin G_{f_i}, (u, v) \in G_{f_{i+1}}\right\}\right| + \left|\left\{i \colon (v, u) \notin G_{f_i}, (v, u) \in G_{f_{i+1}}\right\}\right| = O(n)$$

Proof: Following the proof of Lemma 1.2.7 in the second case we showed if $(u, v) \notin G_{f_i}$ but $(u, v) \in G_{f_{i+1}}$ then $dist_{i+1}(v) \geq dist_i(v) + 2$. Hence the distance increases by at least 2. Now this can happen at most $O(n)$ many times since $\forall\, i$, $dist_i(v) \leq n - 1$. Hence the number of iterations where either (u, v) appears or (v, u) appears is at most $O(n)$. ■

With this this lemma we will prove that the Edmonds-Karp Algorithm takes $O(mn)$ iterations.

Theorem 1.2.9

Edmonds-Karp Algorithm terminates in $O(mn)$ many iterations.

Proof: For k iterations at least k edges must disappear. Since each edge can reappear $O(n)$ times by Lemma 1.2.8, it can disappear at most $O(n)$ many times. In each iteration at least one edge disappears. Now after $O(mn)$ iterations number of disappearances is at most $O(mn)$. But after $O(mn)$ many disappearances there are no edge remaining and therefore there is no $s \rightsquigarrow t$ path. Hence the algorithm terminates. Therefore the Algorithm terminates in $O(mn)$ iterations. ■

Hence Edmond-Karp Algorithm takes $O(m^2n)poly(\log c_e) = O\left(m^2 n \log^{O(1)}(c_e)\right)$ time since it takes $O(mn)$ iterations and in each iteration it finds the shortest $s \rightsquigarrow t$ path in G_{f_i} in $O(m)$ time and in each iteration it does addition and subtraction and finds minimum of the capacities which takes polynomial of the bits needed to represent them time.

1.3 Preflow-Push/Push-Relabel Algorithm

In this algorithm we will maintain something called "Preflow" which is not a valid flow. Unlike Ford-Fulkerson, Edmonds-Karp it does not maintain a $s \rightsquigarrow t$ path in the residual graph and the algorithm stops when the preflow is actually a valid flow.

Definition 1.3.1: Preflow

Given a graph $G = (V, E)$ and the edge capacities c_e, a function $f : E \to \mathbb{R}_0$ is a preflow if is satisfies:

① $\forall\, e \in E, f(e) \leq c_e$.

② $\forall\, v \in V \setminus \{s\}, \sum\limits_{e in(v)} f(e) \geq \sum\limits_{e \in out(v)} f(e)$

Notice here unlike the definition of Flow here in the second criteria we need $\sum\limits_{e in(v)} f(e) \geq \sum\limits_{e \in out(v)} f(e)$ instead of $\sum\limits_{e in(v)} f(e) = \sum\limits_{e \in out(v)} f(e)$.

Now define for all $v \in V$ and for all preflow f, $excess_f(v) = \sum\limits_{e \in in(v)} f(e) - \sum\limits_{e \in out(v)} f(e)$. If f is a preflow then $excess_f(s) \leq 0$ and $\forall\, v \in V \setminus \{s\}$, $excess_f(v) \geq 0$

Lemma 1.3.1

For all preflow f

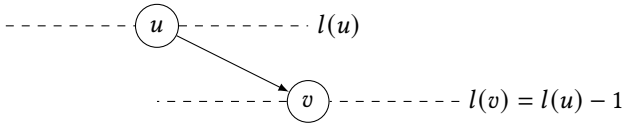
$$\sum_{v \in V} excess_f(v) = 0$$

Proof:

$$\begin{aligned}\sum_{v \in V} excess_f(v) &= \sum_{v \in V} \left[\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) \right] \\ &= \sum_{v \in V} \sum_{e \in in(v)} f(e) - \sum_{v \in V} \sum_{e \in out(v)} f(e) \\ &= \sum_{e \in E} f(e) - \sum_{e \in E} f(e) = 0\end{aligned}$$

■

Now for each $v \in V$ we assign a label $l(v) \in \mathbb{Z}_0$. The algorithm then sends flow from $u \to v$ if $l(v) = l(u) - 1$.



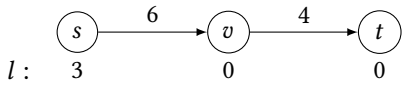
Algorithm 2: PREFLOW-PUSH

Input: Directed graph $G = (V, E)$, source s, target t and edge capacities C_e for all $e \in E$
Output: Flow f with maximum value

```
1  begin
2      Initially ∀ e = (s,u) ∈ E, f(e) = c_e and f(e) for all other edges.
3      l(s) ⟵ n
4      for v ∈ V \ {s} do
5          l(v) ⟵ 0
6      while ∃ v ≠ t, excess_f(v) > 0 do
7          if ∃ u, such that (v,u) ∈ E_f and l(u) = l(v) − 1 then
8              δ ⟵ min{excess_f(v), c_f(v,u)}
9              if (v,u) is Forward Edge then
10                 f(v,u) ⟵ f(v,u) + δ
11             else
12                 f(u,v) ⟵ f(u,v) − δ
13         else
14             l(v) ⟵ l(v) + 1                                  //Relabeling
```

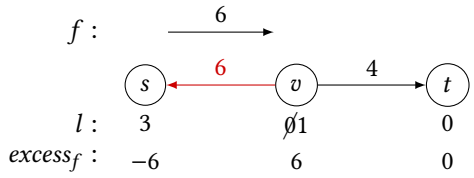
In the algorithm in line 8 if $\delta = c_f(v, u)$ then we call it *saturating push* and if $\delta = excess_f(v)$ then we call it *non-saturating push*.

Now we will show an example of how the algorithm on a graph. We will start the algorithm with the following graph:



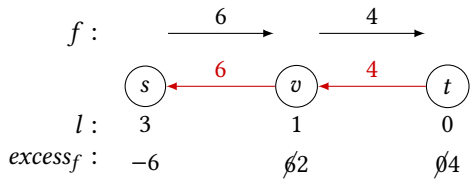
Below we will show change of the residual graph and preflow in each iteration of the WHILE loop:

- Step 1:



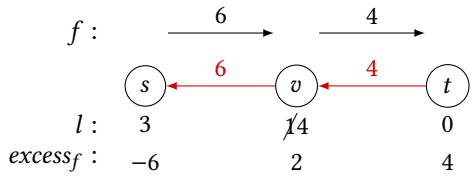
Since $excess_f(v) = 6 > 0$. So in first iteration v is taken. Since there is no edge (v, u) with $l(u) = l(v) - 1$, label of v got increased

- Step 2:



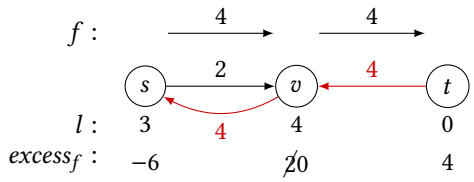
Since $excess_f f(v) = 2 > 0$, in second iteration again v is selected. There is an edge (v, t) with $l(t) = 0 = l(v) - 1 = 1 - 1$. Now $\delta = c_f(v, t) = 4$. Hence saturating push. The preflow gets updated, $f(s, v) = 6$, $f(v, t) = 4$.

- Step 5:



Since $excess_f f(v) = 2 > 0$, in next 3 iterations again v is selected. Since there is no edge (v, u) with $l(u) = l(v) - 1$, label of v gets increased every time. Which becomes 4 after 3 iterations.

- Step 6:



Since $excess_f f(v) = 2 > 0$, in this iteration again v is selected. There is an edge (v, s) with $l(t) = 3 = l(v) - 1 = 4 - 1$. Now $\delta = excess_f(v, s) = 2$. Hence it's non-saturating push. So the preflow gets updated $f(s, v) = 6 - 2 = 4$, $f(v, t) = 4$. Now it's a valid flow. Now there is no vertex with postive *excess*. Hence the algorithm stops.

Observation 1.1. *Labels are monotone non-decreasing.*

Observation 1.2. *For every iteration f is always a preflow. The proof is similar to Lemma 1.2.1 but use inequalities.*

Observation 1.3. $\sum_{v \in V} excess_f(v) = 0$ *and* $\forall\, v \in V \setminus \{s\}$, $excess_f(v) \geq 0$. *Hence* $excess_f(s) \leq 0 \implies l(s)$ *is unchanged.*

Now suppose f^i denote the preflow after the i^{th} iteration of the algorithm. Then

$$f^0(e) = \begin{cases} c_e & \text{when } e = (s, u) \\ 0 & \text{otherwise} \end{cases}$$

Now we will show the correctness of the algorithm.

Lemma 1.3.2

$\forall\, v \in V, \forall\, i, excess_{f^i}(v) > 0 \implies \exists\, v \rightsquigarrow s$ in G_{f^i}

Proof: First we fix v and i such that $excess_{f^i} > 0$. Let X be the set of vertices reachable from v in G_{f^i}. Now

$$\sum_{u \in X} excess_{f^i}(u) = \sum_{u \in X} \left[\sum_{e \in in(v)} f^i(e) - \sum_{e \in out(v)} f^i(e) \right] = \sum_{e \in in(X)} f^i(e) - \sum_{e \in out(X)} f^i(e)$$

Now if $\sum_{e \in in(X)} f^i(e) > 0$ then $\exists\ e = (u', u) \in E$ such that $u' \notin X$ and $u \in X$ and $f^i(e) > 0$. Then the backward edge $(u, u') \in E_{f^i}$. Then u' is reachable from v in G_{f^i}. But $u' \notin X$. Contradiction ↯ Therefore $\sum_{e \in in(X)} f^i(e) = 0$. Hence

$$\sum_{u \in X} excess_{f^i}(u) = \cancel{\sum_{e \in in(X)} f^i(e)} - \sum_{e \in out(X)} f^i(e) \leq 0$$

But from Observation 1.3 we have $\forall\ w \in V \setminus \{s\}$, $excess_{f^i}(w) \geq 0$. But at the same time $\sum_{u \in X} excess_{f^i}(u) \leq 0$ and $excess_{f^i}(v) > 0$. Hence $\exists$ a vertex $u \in X$ such that $excess_{f^i}(u) < 0$. But we know only vertex with negative *excess* is s. Therefore $s \in X$. Hence s is reachable from v. ∎

Lemma 1.3.3

$\forall\, i$, if $(u, v) \in G_{f^i}$ then $l(v) \geq l(u) - 1$.

Proof: We will prove this using induction on i. Initially $l(s) = n$ and $l(v) = 0$ for all $v \in V \setminus \{s\}$. Hence for all edges (u, v) where $u, v \neq s$ this is satisfied. All the other edges incident on s are in $in(s)$ in the residual graph. And $l(s) = n \geq l(u) = 0$. Therefore the base case is followed.

Now suppose the condition is true for f^{i-1}. Now in the i^{th} iteration suppose the selected vertex is $v \in V \setminus \{t\}$ with $excess_{f^{i-1}} > 0$. Now there are two possible cases.

- **Case 1:** If the step is relabeling then $f^{i-1} = f^i$, $G_{f^{i-1}} = G_{f^i}$ but v is relabeled by $l(v) + 1$. Now for any edge $e = (u, v) \in in(v)$ by Inductive Hypothesis $l(v) \geq l(u) - 1 \implies l(v) + 1 \geq l(u) - 1$. Now consider any edge $e = (v, w) \in out(v)$. By Inductive Hypothesis we have $l(w) \geq l(v) - 1$. Now if $l(w) = l(v) - 1$ then we would have pushed flow along the edge (v, w). Since that is not the case we have $l(w) > l(v) - 1$. Therefore $l(w) \geq (l(v) + 1) - 1$. Hence the condition is satisfied.

- **Case 2:** If the step is pushing flow then suppose we push flow along the edge $(v, w) \in E_{f^{i-1}}$ and $l(w) = l(v) - 1$. Now if we push flow along the edge (v, w) we might introduce the reverse edge (w, v) in G_{f^i}. In that case $l(v) = l(w) + 1 \geq l(w) - 1$. Hence the condition is satisfied.

Therefore by mathematical induction $\forall\ i, \forall\ (u, v) \in E_{f^i}, l(v) \geq l(u) - 1$. ■

Corollary 1.3.4

There is no $s \rightsquigarrow t$ path in G_{f^i} in any iteration i. Thus when the algorithm terminates f is a max flow.

Proof: Now $l(s) = n$ and $l(t) = 0$. We fix v and i. If there is a $s \rightsquigarrow v$ path in G_{f^i} then length of the path is at most $n - 1$. For each edge in the path the label decreases by at most 1 by Lemma 1.3.3. Hence $l(v) \geq 1$. Therefore for every vertex $v \in V$, reachable from s we have $l(v) \geq 1$. But $l(t) = 0$. Hence t is not reachable from s. Hence if the algorithm terminates, band if f is a valid flow then y Max Flow Min Cut Theorem it is a max flow. ■

Corollary 1.3.5

$\forall\ v \in V, \forall\ i, l(v) \leq 2n$.

Proof: Suppose $\exists\ v, i$ such that $l(v) = 2n$ and $excess_{f^i}(v) > 0$. By Lemma 1.3.2 there exists an $v \rightsquigarrow s$ path in G_{f^i}. Now by Lemma 1.3.3 for each edge in the path the label decreases by at most 1 and the length of the path is at most $n - 1$. Since $l(v) = 2n$, $l(s) \geq n + 1$. But we know $l(s)$ for all i by Observation 1.3. Hence contradiction ↯ Therefore for all $v \in V$ and $\forall\ i, l(v) \leq 2n$. ■

Corollary 1.3.6

Total number relabeling operations is $\leq 2n^2$

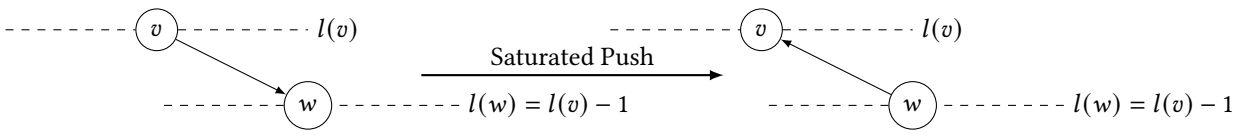
Proof: By Corollary 1.3.5 each vertex label can be at most $2n$. So total number of relabeling operations done in the algorithm is at most $2n^2$ ■

Now we need a bound on the number of push operations. We will count separately the number of Saturating Pushes and number of Non-Saturating Pushes.

Lemma 1.3.7

Total number of saturating pushes is $\leq 2mn$

Proof: We first fix an edge (v, w). Now we will count the number of saturating pushes along (v, w). Then $\delta = c_f(v, w)$. Now consider the scenario of two consecutive saturating pushes along (v, w). When the first saturating push along (v, w) occurred we have $l(w) = l(v) - 1$. Now if (v, w) is forward edge then $\delta = c_f(v, w) = c_{v,w} - f(v, w)$. Then new flow along (v, w) is $f(v, w) + c_f(v, w) = c_{v,w}$. Hence the edge (v, w) vanishes and the flow along (w, v) is $c_{v,w}$. If (v, w) is a backward edge then $\delta = c_f(w, v) = f(w, v)$. Hence then new flow along (w, v) is $f(w, v) - \delta = 0$. Hence again the (w, v) edge vanishes and the flow along (w, v) is $f(w, v)$.



Therefore after a saturated push along (v, w) the edge vanishes and the (w, v) edge is there. Hence in order for another push along (v, w) the algorithm must push flow along (w, v). And this happens when we have the new labels of

v, w follow the condition $l'(w) = l'(v) + 1$. Since by Observation 1.1 the labels never decreases in order for $l(w) = l(v) + 1$ the label of v must increase by at least 2.

Now starting from $l(v) = 0$ we have by Lemma 1.3.5 $l(v) \leq 2n$ and for each saturating push along (v, w) the $l(v)$ increase by 2. Hence at most n many saturating pushes occurred along (v, w). Now in the original graph since there are m edges the total number of saturating pushes is $\leq 2mn$. ■

Now we will count the number of non-saturating pushes. For such pushes along any edge (v, u) the $excess_f(v)$ goes to 0. We define the potential function for a preflow f,

$$\Phi(f) = \sum_{v:\ excess_f(v)>0} l(v)$$

Now $\Phi(f) \geq 0$ for all preflow f and initially at the start of the algorithm $\Phi(f^0) = 0$.

Lemma 1.3.8

For each non-saturating push $\Phi(f)$ decreases by at least 1.

Proof: Suppose at any iteration i a non-saturating push occur along an edge (v, w). Therefore $l(w) = l(v) - 1$. We will show that $\Phi(f^i) \leq \Phi(f^{i-1}) - 1$. We have $\delta = excess_{f^{i-1}}(v)$. Now if (v, w) is a forward edge then new flow along (v, w) is $f^i(v, w) = f^{i-1}(v, w) + excess_{f^{i-1}}(v)$. Since $(v, w) \in out(v)$

$$excess_{f^i}(v) = \sum_{e\in in(v)} f^i(e) - \sum_{e\in out(v)} f^i(e) = \sum_{e\in in(v)} f^{i-1}(e) - \sum_{e\in out(v)\setminus\{(v,w)\}} f^{i-1}(e) - f^i(v, w) = excess_{f^{i-1}}(v) - \delta = 0$$

Otherwise if (v, w) is a backward edge. Then ew flow along (w, v) is $f^i(w, v) = f^{i-1}(w, v) - excess_{f^{i-1}}(v)$. Since $(w, v) \in in(v)$

$$excess_{f^i}(v) = \sum_{e\in in(v)} f^i(e) - \sum_{e\in out(v)} f^i(e) = f^i(w, v) + \sum_{e\in in(v)\setminus\{(w,v)\}} f^{i-1}(e) - \sum_{e\in out(v)} f^{i-1}(e) = -\delta + excess_{f^{i-1}}(v) = 0$$

In both cases $excess_{f^i}(v) = 0$. Therefore v goes out of the summation. Now there are two cases depending on the value of $excess_{f^{i-1}}(w)$

- **Case 1:** If $excess_{f^{i-1}}(w) > 0$ i.e. w had excess flow before push operation then $\Phi(f^{i-1})$ decreases by $l(v)$ i.e. $\Phi(f^i) = \Phi(f^{i-1}) - l(v)$. Since $l(w) = l(v) - 1$ and by Observation 1.1 $l(v) \geq 1$. Therefore $\Phi(f^i) = \Phi(f^{i-1}) - l(v) \leq \Phi(f^{i-1}) - 1$.
- **Case 2:** If $excess_{f^{i-1}}(w) = 0$, then $excess_{f^i}(w) = excess_{f^{i-1}}(w) + \delta > 0$ since $\delta = excess_{f^{i-1}}(v) > 0$ and therefore $\Phi(f^i) = \Phi(f^{i-1}) - l(v) + l(w) = \Phi(f^{i-1}) - 1$

Hence for both the cases $\Phi(f^i) \leq \Phi(f^{i-1}) - 1$. Therefore $\Phi(f^{i-1})$ decreases by at least 1. ■

Observation 1.4. *For relabeling operation $\Phi(f)$ increases by* 1.

Since there are at most $2n^2$ relabeling operations by Corollary 1.3.6, $\Phi(f)$ increases by at most $2n^2$ with relabeling operations.

Observation 1.5. *For each saturating push $excess_f(v, w)$ might not go to 0 and therefore Φ might increase.*

Now by Lemma 1.3.7 total number of saturated pushes is at most $2mn$. And by Corollary 1.3.5 each vertex has label at most $2n$. Hence in total $\Phi(f)$ can increase at most $2mn \times 2n = 4mn^2$ by saturated pushes. Hence $\Phi(f)$ increases at most $2n^2 + 2mn \times 2m = O(mn^2)$.

Now

$$\#\text{Non-saturating Pushes} \leq \text{Total decrease in } \Phi \leq \text{Total increase in } \Phi \leq 2n^2 + 4mn^2 = O(mn^2)$$

Therefore total number of iterations of the WHILE loop is #Relabeling+#Saturated Push+#Non-saturated Push= $2n^2 + 2mn + O(mn^2) = O(mn^2)$. There fore the algorithm takes $O(mn^2)$ iteration. In each iteration it takes $O(m + n)$ time. Therefore the runtime of the algorithm is $O(mn^2)O(n + m) = O(m^2n^2)$.

Chapter 2

Matching

In **??** we saw how to find a maximal matching in a graph using matroids. Here we will try to find maximum matching.

Maximum Matching
Input: Graph $G = (V, E)$
Question: Find a maximum matching $M \subseteq E$ of G

First we will solve finding maximum matching in bipartite graphs first. Then we will extend the algorithm to general graphs.

2.1 Bipartite Matching

So in this section we will study the following problem:

Bipartite Maximum Matching
Input: Graph $G = (L \cup R, E)$
Question: Find a maximum matching $M \subseteq E$ of G

2.1.1 Using Max Flow

One approach to find a maximum matching is by using max-flow algorithm. For this we introduce 2 new vertices s and t where there is an edge from s to every vertex in L and there is an edge from every vertex in R to t and all edges have capacity 1. Then the max-flow for this directed graph is the maximum matching of the bipartite graph. So we have the algorithm:

Algorithm 3: BP-Max-Matching-Flow

Input: $G = (L \cup R, E)$ bipartite graph
Output: Find a maximum matching

```
1  begin
2      V ⟵ A ∪ B ∪ {s, t}
3      E′ ⟵ E
4      for v ∈ L do
5          E′ ⟵ E′ ∪ {(s, v)}
6      for v ∈ R do
7          E′ ⟵ E′ ∪ {(v, t)}
8      for e ∈ E′ do
9          c_e ⟵ 1
10     f ⟵ Edmonds-Karp(G′ = (V, E′), {c_e : e ∈ E′})
11     return {e : f(e) > 0, e ∈ E}
```

Lemma 2.1.1

There exists a max-flow of value k in the modified graph $G' = (V, E')$ if and only there is a matching of size k

Proof: Suppose G' has a matching M of size k. Let $M = \{(u_i, v_i) : i \in [k]\}$ where $u_i \in L$ and $v_i \in R$ for all $i \in [k]$. Then we have the flow f, $f(s, u_i) = f(u_i, v_i) = f(v_i, t) = 1$ for all $i \in [k]$. This flow has value k.

Now suppose there is a flow f of value k. Since each edge has capacity 1 then either an edge has flow 1 or it has 0 flow. Since value of flow is k there are exactly k edges outgoing from s with positive flow. Let the edges are (s, u_i) for $i \in [k]$. Now from each u_i there is exactly one edge going out which has positive flow. Now if $\exists\ i \neq j \in [k]$ such that $\exists\ v \in R$, $f(u_i, v) = (u_j, v) = 1$ then $f(v, t) = 2$ but $c_{v,t} = 1$. So this is not possible. Therefore the edges going out from each u_i goes to distinct vertices. These edges now form a matching of size k. ■

Therefore the algorithm successfully returns a maximum matching of the bipertite graph. But we don't know any algorithm for finding maximum matching in general graphs using max-flow. In the next algorithm we will use something called Augmenting paths to find a maximum matching which we will extend to general graphs.

2.1.2 Using Augmenting Paths

Definition 2.1.1: M-Alternating Path and Augmenting Path

In a graph $G = (V, E)$ and M be a matching in G. Then an M-alternating path is where the edges from M and $E \setminus M$ appear alternatively.

An M-alternating path between two unmatched (also called exposed) vertices is called an augmenting path.

Given a matching M and if there exists an augmenting path p then we can obtain a larger matching M' just by taking the edges in p not in M. Now suppose we are given a bipartite graph $G = (L \cup R, E)$. Let M is a matching in G. Suppse M is a maximum matching. If there exists an augmenting path p then we can obtain a larger matching just by taking the edges in p not in M. This contradicts with M is maximum matching. Hence there are no augmenting paths.

Now we will show that given G and M which is not maximum then we can find an augmenting path with an algorithm. Since M is not maximum there is a vertex v which is not matched

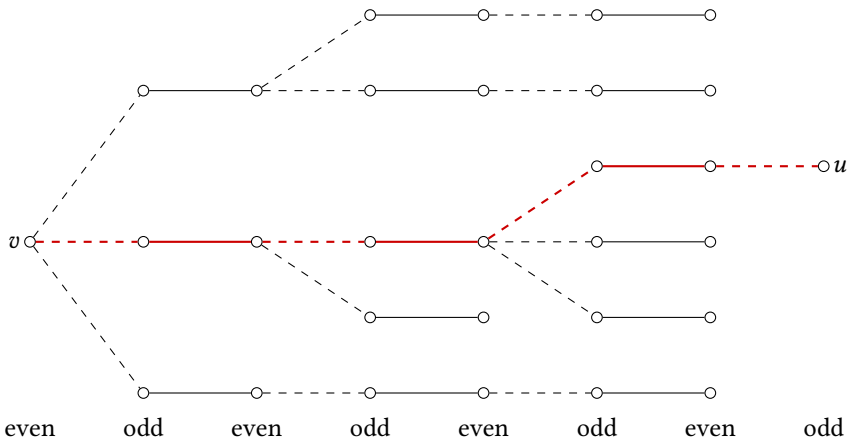
Algorithm 4: FIND-AUGMENTING-PATH(G, v)

Input: $G = (L \cup R, E)$ bipartite graph, matching M (not maximum) and an exposed vertex v
Output: Find an augmenting path starting from v

```
1 begin
2     v.mark ⟵ even
3     for u ∈ L ∪ R \ {v} do
4         u.mark ⟵ Null
5     Queue Q                          // For BFS
6     Enqueue(Q, v)
7     while Q not empty do
8         AUTree(Q)
9     return Fail
```

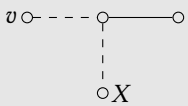
Algorithm 5: AUTree(Q)

```
1  u ⟵ Dequeue(Q)
2  if u.mark == even then
3      for (u, w) ∈ E \ M do
4          if w.mark == Null then
5              Enqueue(Q, w)
6              w.mark ⟵ odd
7              w.p ⟵ u
8  if u.mark == odd then
9      if ∃ (u, w) ∈ M and w.mark == Null then
10         w.mark ⟵ even
11         w.p ⟵ u
12         Enqueue(Q, w)
13     else
14         Print "v ⇝ u augmenting path found"
```



The above algorithm in each iteration checks if the new vertex has mark Null before adding to the queue. Because of this we are not adding same vertex more than one into the queue and if we follow the parent and child pointers, this forms a tree. We call this tree to be a M-alternating tree. Denote the tree by T.

Note:-



The algorithm may not visit all the vertices in $L \cup R$ in the tree. For example in case of the graph at left the algorithm will not find the vertex

Since the algorithm runs a BFS if there was an edge between two vertices at levels separated by 2 we would have explored that vertex earlier. So our first observation is:

Observation 2.1. *In the tree T there are no edges between vertices at levels separated by 2.*

Observation 2.2. *All even vertices except v are matched in T.*

Observation 2.3. *There are no edges between two odd levels or even levels.*

Lemma 2.1.2

If leaf u is odd there is a $v \rightsquigarrow u$ augmenting path.

Proof: If the odd vertex u is unmatched then clearly there is a $v \rightsquigarrow u$ augmenting path. So let's assume u is matched. Say $(u, w) \in M$. If w is not in T then u can not be a leaf as the algorithm will take the edge $(u, w) \in M$ for next iteration.

So suppose w is in T. Then $w.mark = even$ since otherwise we would have taken then (w, u) edge in T before. But by Observation 2.2 all the even vertices except v are matched in the tree already. So u can not be matched with w ■

Now from the tree T we partition the vertices of T into the even marked vertices and odd marked vertices. So let $L_T = L \cap T$ and $R_T = R \cap T$. Therefore L_T is the set of even marked vertices and R_T is the set of odd marked vertices.

Lemma 2.1.3

$N(L_T) = R_T$

Proof: Vertices in L_T are even vertices from which we explore all the edges not in M. Also all the even vertices except v are matched. So except v for all the vertices in L_T their parent is the matched vertex. Hence for all even vertices except v all the neighbors are in R_T. Since v is exposed v has no matched neighbor. So all the neighbors of v are also in R_T. Therefore $N(L_T) = R_T$. ■

Lemma 2.1.4

Suppose we start the algorithm from an exposed vertex v. Suppose there is no augmenting path from v and let the tree formed by the algorithm is T. Then $|L_T| = |R_T| + 1$.

Proof: Since there is no augmenting path the graph all the leaves of T are even vertices. Otherwise the leaves are odd vertices and then all of them have to be matched. If not then there will exists an augmenting path. Therefore all the leaves of T are even vertices. Now since the vertices in L_T are even vertices and all even vertices except v are matched to unique odd vertex in R_T we have $|L_T| = |R_T| + 1$. ■

Now suppose M is a matching. Let $L' = \{v_1, \ldots, v_k\} \subseteq L$ are unmatched vertices. Therefore $|M| = |L| - k$. Then consider the following algorithm:

- Let T_1 be M-alternating tree from v_1 by FIND-AUGMENTING-PATH(G, v_1). L_{T_1}, R_{T_1} are vertices of T_1.
- Let T_2 be M-alternating tree from v_2 by FIND-AUGMENTING-PATH$(G \setminus T_1, v_2)$. L_{T_2}, R_{T_2} are vertices of T_2.
- Let T_3 be M-alternating tree from v_3 by FIND-AUGMENTING-PATH$(G \setminus (T_1 \cup T_2), v_3)$. L_{T_3}, R_{T_3} are vertices of T_3. $\cdots$
- Let T_k be M-alternating tree from v_k by FIND-AUGMENTING-PATH$\left(G \setminus \left(\bigcup_{i=1}^{k-1} T_i\right), v_k\right)$. L_{T_k}, R_{T_k} are vertices of T_k.

Observation 2.4. *v_i is not in T_j for any $j < i$ because otherwise we would have found an augmenting path in T_j.*

Now L_{T_i} for all $i \in [k]$ are disjoint and R_{T_i} for all $i \in [k]$ are disjoint. If G had no augmenting path from v_i for all $i \in [k]$ then there are no augmenting paths in $G \setminus \left(\bigcup_{i=1}^{j} T_i\right)$ for all $j \in [k-1]$ from v_{j+1}. Therefore by Lemma 2.1.4 we have $|L_{T_i}| = |R_{T_i}| + 1 \; \forall \; i \in [k]$. Hence we have

$$\sum_{i=1}^{k} |L_{T_i}| = \sum_{i=1}^{K} (|R_{T_i}| + 1) \implies \left|\bigcup_{i=1}^{k} L_{T_i}\right| = \left|\bigcup_{i=1}^{k} R_{T_i}\right| + k$$

Now by Lemma 2.1.3, $N(L_{T_{j+1}}) = R_{T_{j+1}}$ for all $j \in [k-1]$ in $G \setminus \left(\bigcup_{i=1}^{j} T_i\right)$. Hence

$$N(L_{T_j}) \subseteq \bigcup_{i=1}^{j} R_{T_i} \implies N\left(\bigcup_{i=1}^{k} L_{T_i}\right) = \bigcup_{i=1}^{k} R_{T_i}$$

But $\left|\bigcup_{i=1}^{k} L_{T_i}\right| = \left|\bigcup_{i=1}^{k} R_{T_i}\right| + k$. Therefore any matching of $\bigcup_{i=1}^{k} L_{T_i}$ must leave at least k vertices unmatched. Now all the vertices in $L \setminus \left(\bigcup_{i=1}^{k} L_{T_i}\right)$ with $R \setminus \left(\bigcup_{i=1}^{k} R_{T_i}\right)$ and vice versa. Therefore any matching of L must leave at least k vertices unmatched. Since M is a matching such that exactly k vertices are unmatched. M is a maximum matching. Therefore if there is no augmenting path in G then M is a maximum matching.

We also showed before that if M is a maximum matching then there is no augmenting path in G. Therefore we have the following theorem:

Theorem 2.1.5 Berge's Theorem

A matching M is maximum if and only if there are no augmenting path in G.

Therefore if we start with any matching and each time we find a augmenting path we update the matching by taking the odd edges in the augmenting path and obtain a larger matching. After continuously doing this once when there is no augmenting path we can conclude that we obtained a maximum matching.

Since every time the size of the maximal matching is increased by at least 1. The total number of iterations the algorithm takes to output the maximal matching is $O(n)$ where n is the number of vertices in G. In each iteration it calls the FIND-AUGMENTING-PATH algorithm which takes the time same as time taken in BFS. Hence FIND-AUGMENTING-PATH takes $O(m+n)$ time. Therefore the BP-MAXIMUM-MATCHING algorithm takes $O(n(n+m))$ time.

Algorithm 6: BP-Maximum-Matching(G)

```
Input: G = (L ∪ R, E) bipartite graph
Output: Find a maximum matching
1  begin
2      M ⟵ ∅
3      while True do
4          v ⟵ unmatched vertex
5          p ⟵ Find-Augmenting-Path
6          if p == Fail then
7              return M
8          for e ∈ p do
9              if e ∈ M then
10                 M ⟵ M \ {e}
11             else
12                 M ⟵ M ∪ {e}
```

2.1.3 Using Matrix Scaling

Here we will show a new algorithm for deciding if a bipartite graph has a perfect matching using matrix scaling. The paper which we will follow is [LSW98]

Bipartite Perfect Matching
Input: Graph $G = (L \cup R, E)$
Question: Decide if G has a perfect matching or not.

Suppose $G = (L \cup R, E)$ a bipartite graph. If bipartite adjacency matrix of the graph G is A then the permanent of the matrix A,

$$per(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} x_{i,\sigma(i)}$$

counts the number of perfect matchings in G. So we want to check if for a given bipartite graph $(L \cup R, E)$, $per(A) > 0$ or not where A is the bipartite adjacency matrix. Now there is a necessary and sufficient condition for existence of perfect matching in a bipartite graph which is called Hall's condition.

Theorem 2.1.6 Hall's Condition

A bipartite graph $G = (L \cup R, E)$ has an L-perfect matching if and only if $\forall\, S \subseteq L$, $|S| \leq |N(S)|$ where $N(S) = \{v \in R \colon \exists\, u \in L,\ (u, v) \in E\}$

Proof: Now if G has a L-perfect matching then for every $S \subseteq L$, S is matched with some $T \subseteq R$ such that $|S| = |T|$. Therefore $T \subseteq N(S) \implies |S| = |T| \leq |N(S)|$.

Now we will prove the opposite direction. Suppose for all $S \subseteq L$ we have $|S| \leq |N(S)|$. Assume there is no L-perfect matching in G. Let M be a maximum L-matching in G. Let $u \in L$ is unmatched. Now consider the following sets:

$$X = \{x \in L \colon \exists\, M\text{-alternating path from } u \text{ to } x\}, \qquad Y = \{y \in R \colon \exists\, M\text{-alternating path from } u \text{ to } y\}$$

Now notice that $N(X) \subseteq Y$. Since in a M-alternating path from u whenever the odd edges are not matching edges and the even edges are matching edges. So in the odd edges we can pick any neighbor except the one it is matched with and the immediate even edge before that connects that vertex with the vertex in R it is matched with. Hence we have $N(X) \subseteq Y$.

Now it suffices to prove that $|X| > |Y|$. Now let $y \in Y$. Suppose $u \rightsquigarrow x' \to y$ be the M-alternating path. If y is not matched then we could increase the matching by taking the odd edges of the path and thus obtain a matching with larger size than M. But M is maximum matching. Hence y is matched. Therefore we can extend the path by taking the matching edge incident on y and go the the vertex $x'' \in L$ i.e. the new M-alternating path becomes $u \rightsquigarrow x' \to y \to x''$ to have a M-alternating path $u \rightsquigarrow x''$. So $|X| > |Y|$.

Therefore we obtained a set of vertices $X \subseteq Y$ such that $|X| > |Y| \geq N(X)|$. This contradicts the assumption. Hence contradiction. Therefore G has a L-perfect matching. ∎

We will use hall's condition on the adjacency matrix to check if $per(A)$ is positive or not. Now multiplying a row or a column of a matrix by some constant c also multiplies the permanent of the matrix by c as well. In fact if $d_1, d_2 \in \mathbb{R}^n_+$ and $D_1 = diag(d_1)$ and $D_2 = diag(D_2)$ then $per(D_1AD_2) = \left(\prod_{i=1}^{n} d_{1_i}\right)\left(\prod_{i=1}^{n} d_{2_i}\right) per(A)$. So we can scale our original matrix A to obtain a different matrix B and from B we can approximate $per(A)$ by approximating $per(B)$. A natural strategy is to seek an efficient algorithm for scaling A to a doubly stochastic B.

Definition 2.1.2: Doubly Stochastic Matrix

A matrix $M \in \mathbb{R}^{m\times m}$ is doubly stochastic if entries are non-negative and each row and column sum to 1.

First we will show that Hall's Condition holds for doubly stochastic matrix. First let's see what it means for a matrix to satisfy hall's condition. A matrix with all entries non-negative holds Hall's Condition if for all $S \subseteq [n]$ if $T = \{i \in [n] : \exists\ j \in S,\ A(i,j) \neq 0\}$ then $|T| \geq |S|$. This also corresponds to the bipartite adjacency matrix satisfying the hall's condition since for any set of rows S the number of columns for which in the S rows at least one entry is non zero should be greater than or equal to $|S|$.

Lemma 2.1.7

Hall's Condition holds for doubly stochastic matrix.

Proof: Let M be the doubly stochastic matrix. Let $S \subseteq [n]$. So consider the $|S| \times n$ matrix which only consists of the rows in S. Call this matrix M^r_S. Now suppose T be the set of columns in M^r_S which has nonzero entries. Now consider the $n \times |T|$ matrix which only consists of the columns in T. Call this matrix M^c_T. Now since M is doubly stochastic we know sum of entries of M^r_S is $|S|$ and sum of entries of M^c_T is $|T|$. Our goal is to show $|S| \leq |T|$. Now since T is the only set of columns which have nonzero columns in M^r_S the elements which contributes to the sum of entries in M^r_S are in the T columns in M^r_S. Since these elements are also present in M^c_T we have $|T| \geq |S|$. ∎

Hence for doubly stochastic matrices the permanent is positive. Now not all matrices are doubly stochastic. And in fact matrices with permanent zero will not be doubly stochastic so no amount of scaling will make it doubly stochastic. So we will settle for approximately doubly stochastic matrix. In order to make a matrix doubly stochastic first for each row we will divide the row with their row some. Now it becomes row stochastic. Then if its not approximately doubly stochastic for each column we will divide the column entries with their column sum. But first what ϵ-approximate doubly stochastic matrix means.

Definition 2.1.3: ϵ−Approximate Doubly Stochastic Matrix

A matrix is ϵ−approximate doubly stochastic if for each column, the column sum is in $(1-\epsilon, 1+\epsilon)$ and for each row, the row sum is in $(1-\epsilon, 1+\epsilon)$

Now we will show that even for ϵ-approximate doubly stochastic matrix the hall's condition holds.

Lemma 2.1.8

Halls's Condition holds for ϵ-approximate doubly stochastic matrix for $\epsilon < \frac{1}{10n}$

Proof: Let M is ϵ−approximate doubly stochastic matrix. Let $S \subseteq [n]$. So consider the $|S| \times n$ matrix which only consists of the rows in S. Call this matrix M^r_S. Now suppose T be the set of columns in M^r_S which has nonzero entries. Now consider the $n \times |T|$ matrix which only consists of the columns in T. Call this matrix M^c_T. Now the sum of entries in M^r_S is $\geq |S|(1-\epsilon)$ and sum of entries in M^c_T is $\leq |T|(1-\epsilon)$. Now since T is the only set of columns which have nonzero columns in M^r_S the elements which contributes to the sum of entries in M^r_S are in the T columns in M^r_S. Since these elements are also present in M^c_T we have $||T|(1+\epsilon) \geq |S|(1-\epsilon)$. Therefore we have

$$|T| \geq |S|\frac{1-\epsilon}{1+\epsilon} = |S|\left(1 - \frac{2\epsilon}{1+\epsilon}\right) \geq |S|(1-2\epsilon) > |S|\left(1 - \frac{1}{5n}\right) \geq |S|\left(1 - \frac{1}{|S|}\right) > |S| - 1$$

Since T is an integer we have $|T| \geq |S|$. Hence the Hall's condition holds. ∎

Therefore permanent of ϵ-approximate doubly stochastic matrix is also positive. Hence our algorithm for bipartite perfect matching is:

Algorithm 7: BP-Matrix-Scaling

Input: Bipartite adjacency matrix A of $G = (L \cup R, E)$
Output: Decide if G has a perfect matching.

```
begin
    while True do
        A ⟵ Scale every rows of A to make it row stochastic.
        if All column-sums are in (1 − ϵ, 1 + ϵ) then
            return Yes
        A ⟵ Scale every column of A to make it column stochastic.
        if All row-sums are in (1 − ϵ, 1 + ϵ) then
            return Yes
```

In both if conditions we are checking if the matrix is $\epsilon-$approximate doubly stochastic matrix. The moment it becomes a $\epsilon-$approximate doubly stochastic matrix we are done.

Now if G doesn't have a perfect matching then we will never reach a ϵ-approximate doubly stochastic matrix since otherwise Hall's condition will hold and then we will have that the permanent is positive. So if G doesn't have a perfect matching the algorithm will run in an infinite loop. We only need to check if G has a perfect matching the algorithm returns Yes.

We will now define a potential function $\Phi\colon \mathbb{Z}_0 \to \mathbb{R}_+$. Let $\sigma \in S_n$ such that $a_{i,\sigma(i)} \neq 0$ for all $i \in [n]$.Now if an entry of the matrix is nonzero then it is always nonzero since all the entries are non-negative. Now since the scalings are symmetric we will define the potential function for i^{th} scaling (row/column) is $\Phi(i) = \prod_{i=1}^{n} a_{i,\sigma(i)}$. So we have $\Phi(0) = 1$ since at first all the entries of the matrix are from $\{0, 1\}$. Also we know $\Phi(t) \leq 1$ for all t since every time we are scaling the matrix. Now $\Phi(1) \geq \frac{1}{n^n}$ since every row-sum can be at most n so it will be divided by n and therefore $a_{i,\sigma(i)} \geq \frac{1}{n}$ for all $i \in [n]$. Now to show the while loop stops if G has a perfect matching it suffices to show that $\Phi(t)$ increases by a multiplicative factor. So we have the following lemma.

Lemma 2.1.9

For all t, $\Phi(t+1) \geq \Phi(t)(1+\alpha)$ for some $\alpha \in (0, 1)$.

Proof: Let A' denote the matrix at the t^{th} scaling where the $(t-1)^{th}$ scaling was column-scaling. Let A'' denote the matrix after row-scaling. Now since we went to the next iteration not all column-sums are in $(1-\epsilon, 1+\epsilon)$ after scaling the rows. Now the row sums of A'' are 1. Therefore we have

$$\frac{\Phi(t)}{\Phi(t+1)} = \prod_{i=1}^{n} \textit{Col-sum}_i(A'') \leq \left(\frac{\sum_{i=1}^{n} \textit{Col-sum}_i(A'')}{n}\right)^n = \left(\frac{\sum_{i=1}^{n} \textit{Row-sum}_i(A'')}{n}\right)^n = 1 \implies \Phi(t) \leq \Phi(t+1)$$

Similarly we can say the same if $(t-1)^{th}$ scaling was row-scaling. Since not all column-sums are in $(1-\epsilon, 1+\epsilon)$ we have $\sum_{i=1}^{n} (\textit{Col-sum}_i(A'') - 1)^2 \geq \epsilon^2$. Therefore using Lemma 2.1.10 we have

$$\frac{\Phi(t)}{\Phi(t+1)} \leq 1 - \frac{\epsilon^2}{2} \implies \Phi(t+1) \geq \left(1 + \frac{\epsilon^2}{2}\right)\Phi(t)$$

Therefore we have the lemma. ∎

We have $\epsilon < \frac{1}{10n}$. Therefore if $t \geq 200n^4$ then we have

$$1 \geq \Phi(t) \geq \frac{1}{n^n}\left(1 + \frac{1}{200n^2}\right)^t \geq \frac{1}{n^n} e^{n^2} > 1$$

Hence the while loop will iterate for at most $200n^4$ iterations. Hence this algorithm takes $O(n^4)$ time. Hence if G has a perfect matching the algorithm runs for at most $O(n^4)$ iterations. And if G doesn't have a perfect matching then the loop never stops. So we have the new modified algorithm to prevent infinite looping:

Algorithm 8: BP-MATRIX-SCALING

Input: Bipartite adjacency matrix A of $G = (L \cup R, E)$
Output: Decide if G has a perfect matching.

```
begin
    ϵ ⟵ 1/(20n)
    for i ∈ [200n^4] do
        A ⟵ Scale every rows of A to make it row stochastic.
        if All column-sums are in (1 − ϵ, 1 + ϵ) then
            return Yes
        A ⟵ Scale every column of A to make it column stochastic.
        if All row-sums are in (1 − ϵ, 1 + ϵ) then
            return Yes
```

We will prove the helping lemma needed to prove Lemma 2.1.9.

Lemma 2.1.10

Suppose $x_1, \ldots, x_n \geq 0$ and $\sum_{i=1}^{n} x_i = n$ and $\sum_{i=1}^{n} (1 - x_i)^2 \geq \delta$. Then $\prod_{i=1}^{n} x_i \leq 1 - \frac{\delta}{2} + o(\delta)$.

Proof: Denote $\rho_i = x_i - 1$. So $\sum_{i=1}^{n} \rho_i = 0$ and $\sum_{i=1}^{n} \rho^2 \geq \delta$. Now

$$\log(1+\rho_i) = \sum_{j=1}^{\infty} (-1)^{j-1} \frac{\rho_i^j}{j} \implies \log(1+\rho_i) \leq \rho_i - \frac{\rho_i^2}{3} + \frac{\rho_i^3}{3} \implies 1 + \rho_i \leq e^{\rho_i - \frac{\rho_i^2}{3} + \frac{\rho_i^3}{3}}$$

Therefore we have

$$\prod_{i=1}^{n} x_i \leq \exp\left[\sum_{i=1}^{n} \rho_i - \sum_{i=1}^{n} \frac{\rho_i^2}{3} + \sum_{i=1}^{n} \frac{\rho_i^3}{3}\right] \leq \exp\left[0 - \frac{\delta}{2} + \frac{\left(\sum_{i=1}^{n} \rho_i^2\right)^{\frac{3}{2}}}{3}\right] = \exp\left[-\frac{\delta}{2} + \frac{\delta^{\frac{3}{2}}}{3}\right] \leq 1 - \frac{\delta}{2} + o(\delta)$$

Therefore we have the lemma. ∎

There is also a survey, [Ide16] on use of matrix scaling in different results.

2.2 Matching in General Graphs

Here we give a similar algorithm for finding maximum matching in general graph as in the case of bipartite graphs in subsection 2.1.2. We will give a similar characterization for the maximum matching in general graphs. First we will show a extension of berge's lemma to general graphs.

Theorem 2.2.1

For any graph $G = (V, E)$, $M \subseteq E$ is a maximal matching if and only if there is no augmenting paths in G.

Proof: Suppose M is a maximal matching. Then if G has an augmenting path p. Then we can just take the odd edges in p and then replace the edges in $M \cap p$ with those edges i.e. $M \triangle p$ and this is a larger matching than M. But this contradicts the maximum property of M. Hence G has no augmenting paths.

Now we will show that if M is not a maximum matching then G has an augmenting path. So suppose M is not a maximum matching. Let M' is a maximum matching. Then consider the graph $G' = (V, E')$ where $E' = M \triangle M'$. Now every vertex in V has degree $\in \{0, 1, 2\}$ in G'. Hence the connected components of G' are isolated vertices, paths and cycles. In a path or cycle the edges of M and M' not in both appear alternatively. Therefore the cycles are even cycles. Since $|M'| > |M|$ there exists a path p such that number $|p \cap M'| > |p \cap M|$. Therefore the starting and ending edge of p are in M'. Hence p is an augmenting path in G. ■

Therefore like in the case of bipartite graphs we will search for augmenting paths in G for matching M and if we can find an augmenting path p we will update the matching by taking $M' = M \triangle p$ and obtain a larger matching. But unlike bipartite graphs we can not run the same algorithm for finding augmenting paths as there can be edges between two odd layers or two even layers. So in the M-alternating tree there can be odd cycles but these odd cycles have all vertices except one vertex are matched using edges of the cycle. So we look for these special structures in the M-alternating tree called *blossom*.

Definition 2.2.1: Blossom

A blossom is an odd cycle in which only one vertex is unmatched and the remaining vertices are matched using edges of the cycle. The exposed vertex is called the *base* of the blossom.

Note that given a matching M and blossom B we can modify M so that any $v \in B$ is the base, since no vertex B has a matching outside.

The algorithm for finding augmenting path in non-bipartite graphs works by detecting blossoms in M-alternating tree starting from some exposed vertex. The idea is to then contract the blossoms into single vertex and then this process is repeated in the modified graph.

Let B be a blossom in G. Then the new graph is $G' = (V', E')$ where

$$V' = (V \setminus B) \cup \{v_B\}, \qquad E' = \Big(E \setminus \{(u, v) : u \in B \text{ or } v \in B\}\Big) \cup \{(u, v_B) : u \notin B, v \in B, (u, v) \in E\}$$

Observation 2.5. *v_B is an exposed vertex in G'*

For this first time we have to show that finding augmenting path in the contracted graph gives an augmenting path in original graph and vice versa.

Lemma 2.2.2

$G = (V, E)$ is a graph with matching $M \subseteq E$ and a blossom B. Let $G' = (V', E')$ be the contacted graph after contracting B into a single vertex v_B. Then G with matching M has an augmenting path if and only G' with matching $M' = M \setminus \{(u, v) : u, v \in B\}$ has an augmenting path

Proof: Let p be an augmenting path in G. Let $p = (v_0, \ldots, v_k)$ where both v_0 and v_k are exposed vertices. Hence at least one of v_0 and v_k are not in B. WLOG $v_0 \notin B$. If none of the vertices in p are in B then p also exists in G' as well. Therefore G' has an augmenting path. So suppose $p \cap B \neq \emptyset$. Suppose v_r be the first vertex in p that is in B. Then $p' = (v_0, \ldots, v_{r-1}, v_B)$ is an augmenting path since v_B is an exposed vertex in G'.

Now let p' is an augmenting path in G'. If v_B is not in p' then p' also exists in G. Therefore p' is an augmenting path in G. Suppose v_B is in p'. ■

Chapter 3

Linear Programming

Chapter 4

Bibliography

[Ide16] Martin Idel. A review of matrix scaling and sinkhorn's normal form for matrices and positive maps. September 2016.

[LSW98] Nathan Linial, Alex Samorodnitsky, and Avi Wigderson. A deterministic strongly polynomial algorithm for matrix scaling and approximate permanents. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC '98*, STOC '98, pages 644–652. ACM Press, 1998.