Matchings II

Lecture 17

October 22, 2015

17.1: Maximum weight matchings in a bipartite graph

17.1.1: On the structure of the problem

Weight of path/cycle

1. For alternating path/cycle π:
2. **weight** (for matching M):

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \cap M} w(e). \qquad (1)$$

3. $=$ total weight of the free edges in π minus weight of matched edges.
4. Useful lemma: $\gamma(\pi, M) > 0 \implies w(M') > w(M)$.

Weight of path/cycle

1. For alternating path/cycle π:
2. **weight** (for matching M):

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \cap M} w(e). \qquad (1)$$

3. $=$ total weight of the free edges in π minus weight of matched edges.
4. Useful lemma: $\gamma(\pi, M) > 0 \implies w(M') > w(M)$.

Weight of path/cycle

1. For alternating path/cycle π:
2. **weight** (for matching M):

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \cap M} w(e). \tag{1}$$

3. $=$ total weight of the free edges in π minus weight of matched edges.
4. Useful lemma: $\gamma(\pi, M) > 0 \implies w(M') > w(M)$.

Weight of path/cycle

1. For alternating path/cycle π:
2. **weight** (for matching M):

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \cap M} w(e). \qquad (1)$$

3. $=$ total weight of the free edges in π minus weight of matched edges.
4. Useful lemma: $\gamma(\pi, M) > 0 \implies w(M') > w(M)$.

Lemma

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \cap M} w(e). \qquad (2)$$

Lemma

M: a matching. π: alternating path/cycle with positive weight relative to M.
$\gamma(\pi, M) > 0$. Furthermore, assume that

$$M' = M \oplus \pi = (M \setminus \pi) \cup (\pi \setminus M)$$

is a matching. Then $w(M')$ is bigger; namely, $w(M') > w(M)$.

Proof

Proof.

$$\begin{aligned} w(M') - w(M) &= \sum_{e \in M'} w(e) - \sum_{e \in M} w(e) \\ &= \sum_{e \in M' \setminus M} w(e) - \sum_{e \in M \setminus M'} w(e) \\ &= \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in M \setminus \pi} w(e) \\ &= \gamma(\pi, M). \end{aligned}$$

Just observe that $w(M') = w(M) + \gamma(\pi, M)$. □

Augmenting...

1. Augmenting path in the weighted case:

Definition

An alternating path is **augmenting** if it starts and ends in a free vertex.

2. Observation:
   If M has an augmenting path π then M is not of maximum size matching (this is for the unweighted case), since $M \oplus \pi$ is a larger matching.

Augmenting...

1. Augmenting path in the weighted case:

Definition

An alternating path is **augmenting** if it starts and ends in a free vertex.

2. Observation:
   If M has an augmenting path π then M is not of maximum size matching (this is for the unweighted case), since $M \oplus \pi$ is a larger matching.

Augmenting by heaviest augmenting path is good...

Theorem

Let M be a matching of maximum weight among matchings of size $|M|$. Let π be an augmenting path for M of maximum weight, and let T be the matching formed by augmenting M using π. Then T is of maximum weight among matchings of size $|M| + 1$.

Proof

1. S: matching of maximum weight among all matchings with $|M| + 1$ edges.
2. $H = (V, M \oplus S)$.
3. Cycle or even length path σ in H.
   1. Must be $\gamma(\sigma, M) = 0$.
   2. If $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ matching of same size as M but heavier. Contradiction.
   3. if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.
4. Same arg: If σ is even path in H then $\gamma(\sigma, M) = 0$.
5. U_S: All odd length paths in H that have one edge more in S than in M.
6. U_M: All odd length paths in H that have one edge more of M than an edge of S.

Proof

1. S: matching of maximum weight among all matchings with $|M| + 1$ edges.
2. $H = (V, M \oplus S)$.
3. Cycle or even length path σ in H.
   1. Must be $\gamma(\sigma, M) = 0$.
   2. If $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ matching of same size as M but heavier. Contradiction.
   3. if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.
4. Same arg: If σ is even path in H then $\gamma(\sigma, M) = 0$.
5. U_S: All odd length paths in H that have one edge more in S than in M.
6. U_M: All odd length paths in H that have one edge more of M than an edge of S.

Proof

1. S: matching of maximum weight among all matchings with $|M| + 1$ edges.
2. $H = (V, M \oplus S)$.
3. Cycle or even length path σ in H.
   1. Must be $\gamma(\sigma, M) = 0$.
   2. If $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ matching of same size as M but heavier. Contradiction.
   3. if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.
4. Same arg: If σ is even path in H then $\gamma(\sigma, M) = 0$.
5. U_S: All odd length paths in H that have one edge more in S than in M.
6. U_M: All odd length paths in H that have one edge more of M than an edge of S.

Proof

1. S: matching of maximum weight among all matchings with $|M| + 1$ edges.
2. $H = (V, M \oplus S)$.
3. Cycle or even length path σ in H.
   1. Must be $\gamma(\sigma, M) = 0$.
   2. If $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ matching of same size as M but heavier. Contradiction.
   3. if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.
4. Same arg: If σ is even path in H then $\gamma(\sigma, M) = 0$.
5. U_S: All odd length paths in H that have one edge more in S than in M.
6. U_M: All odd length paths in H that have one edge more of M than an edge of S.

Proof

1. S: matching of maximum weight among all matchings with $|M| + 1$ edges.
2. $H = (V, M \oplus S)$.
3. Cycle or even length path σ in H.
   1. Must be $\gamma(\sigma, M) = 0$.
   2. If $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ matching of same size as M but heavier. Contradiction.
   3. if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.
4. Same arg: If σ is even path in H then $\gamma(\sigma, M) = 0$.
5. U_S: All odd length paths in H that have one edge more in S than in M.
6. U_M: All odd length paths in H that have one edge more of M than an edge of S.

Proof

1. S: matching of maximum weight among all matchings with $|M|+1$ edges.
2. $H = (V, M \oplus S)$.
3. Cycle or even length path σ in H.
   1. Must be $\gamma(\sigma, M) = 0$.
   2. If $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ matching of same size as M but heavier. Contradiction.
   3. if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.
4. Same arg: If σ is even path in H then $\gamma(\sigma, M) = 0$.
5. U_S: All odd length paths in H that have one edge more in S than in M.
6. U_M: All odd length paths in H that have one edge more of M than an edge of S.

Proof

1. S: matching of maximum weight among all matchings with $|M| + 1$ edges.
2. $H = (V, M \oplus S)$.
3. Cycle or even length path σ in H.
   1. Must be $\gamma(\sigma, M) = 0$.
   2. If $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ matching of same size as M but heavier. Contradiction.
   3. if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.
4. Same arg: If σ is even path in H then $\gamma(\sigma, M) = 0$.
5. U_S: All odd length paths in H that have one edge more in S than in M.
6. U_M: All odd length paths in H that have one edge more of M than an edge of S.

Proof

1. S: matching of maximum weight among all matchings with $|M| + 1$ edges.
2. $H = (V, M \oplus S)$.
3. Cycle or even length path σ in H.
   1. Must be $\gamma(\sigma, M) = 0$.
   2. If $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ matching of same size as M but heavier. Contradiction.
   3. if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.
4. Same arg: If σ is even path in H then $\gamma(\sigma, M) = 0$.
5. U_S: All odd length paths in H that have one edge more in S than in M.
6. U_M: All odd length paths in H that have one edge more of M than an edge of S.

Proof

1. S: matching of maximum weight among all matchings with $|M| + 1$ edges.
2. $H = (V, M \oplus S)$.
3. Cycle or even length path σ in H.
    1. Must be $\gamma(\sigma, M) = 0$.
    2. If $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ matching of same size as M but heavier. Contradiction.
    3. if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.
4. Same arg: If σ is even path in H then $\gamma(\sigma, M) = 0$.
5. U_S: All odd length paths in H that have one edge more in S than in M.
6. U_M: All odd length paths in H that have one edge more of M than an edge of S.

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
   1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
      (With same number of edges.)
   2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction. ∎

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
   1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
      (With same number of edges.)
   2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction. ∎

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
    1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
       (With same number of edges.)
    2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction. ■

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
    1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
       (With same number of edges.)
    2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction.

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
   1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
      (With same number of edges.)
   2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction.

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
   1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
      (With same number of edges.)
   2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction.

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
    1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
       (With same number of edges.)
    2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction.

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
    1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
       (With same number of edges.)
    2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction.

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
    1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
       (With same number of edges.)
    2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction.

Proof continued...

1. Know: $|U_S| - |U_M| = 1$ since $|S| = |M| + 1$.
2. For $\pi \in U_S$ and $\pi' \in U_M$...
3. Must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$.
    1. If $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ bigger weight than M.
       (With same number of edges.)
    2. If $\gamma(\pi, M) + \gamma(\pi', M) < 0$ then $S \oplus \pi \oplus \pi'$ same number of edges as S but heavier matching. A contradiction.
4. Pair up the paths in U_S to paths in U_M.
5. Total weight of such a pair is zero.
6. Only one path μ in U_S which not paired.
7. $\gamma(\mu, M) = w(S) - w(M)$ (everything else has balance 0).
8. Path must be the heaviest augmenting path for M... Otherwise, $\exists$ heavier augmenting path σ' for M s.t. $w(M \oplus \sigma') > w(S)$. A contradiction. ■

Conclusion...

The above theorem imply that if we always augment along the maximum weight augmenting path, than we would get the maximum weight matching in the end.

17.2: Maximum weight matchings in a bipartite Graph

To be given a more exciting title...

1. $G = (L \cup R, E)$: given bipartite graph.
2. $w : E \to \mathbb{R}$: non-negative weights on edges.
3. M: matching.
4. $\mathbf{G}_M$: directed graph (like unweighted graph):
    1. $rl \in M$, $l \in L$ and $r \in R$: add (r, l) to $\mathbf{E}(\mathbf{G}_M)$. Weight $\alpha((r, l)) = w(rl)$.
    2. $rl \in E \setminus M$: add edge $(l \to r) \in \mathbf{E}(\mathbf{G}_M)$. With weight $\alpha((l, r)) = -w(rl)$.
5. π: augmenting path in $\mathbf{G} = \pi$ path from free vertex in L to free vertex in R in $\mathbf{G}_M$.
6. path π in $\mathbf{G}_M$ has weight $\alpha(\pi) = -\gamma(\pi, M)$.
7. U_L: free vertices in L. U_R free vertices in R.
8. Looking for: path π in $\mathbf{G}_M$ starting U_L going to U_R with maximum weight $\gamma(\pi)$. Min weight $\alpha(\pi)$.

To be given a more exciting title...

1. $G = (L \cup R, E)$: given bipartite graph.
2. $w : E \rightarrow \mathbb{R}$: non-negative weights on edges.
3. M: matching.
4. $\mathbf{G}_M$: directed graph (like unweighted graph):
    1. $rl \in M$, $l \in L$ and $r \in R$: add (r, l) to $\mathbf{E}(\mathbf{G}_M)$. Weight $\alpha((r, l)) = w(rl)$.
    2. $rl \in E \setminus M$: add edge $(l \rightarrow r) \in \mathbf{E}(\mathbf{G}_M)$. With weight $\alpha((l, r)) = -w(rl)$.
5. π: augmenting path in $\mathbf{G}$ = π path from free vertex in L to free vertex in R in $\mathbf{G}_M$.
6. path π in $\mathbf{G}_M$ has weight $\alpha(\pi) = -\gamma(\pi, M)$.
7. U_L: free vertices in L. U_R free vertices in R.
8. Looking for: path π in $\mathbf{G}_M$ starting U_L going to U_R with maximum weight $\gamma(\pi)$. Min weight $\alpha(\pi)$.

To be given a more exciting title...

1. $G = (L \cup R, E)$: given bipartite graph.
2. $w : E \to \mathbb{R}$: non-negative weights on edges.
3. M: matching.
4. G_M: directed graph (like unweighted graph):
   1. $rl \in M$, $l \in L$ and $r \in R$: add (r, l) to $\mathsf{E}(\mathsf{G}_M)$. Weight $\alpha((r, l)) = w(rl)$.
   2. $rl \in E \setminus M$: add edge $(l \to r) \in \mathsf{E}(\mathsf{G}_M)$. With weight $\alpha((l, r)) = -w(rl)$.
5. π: augmenting path in $\mathsf{G} = \pi$ path from free vertex in L to free vertex in R in G_M.
6. path π in G_M has weight $\alpha(\pi) = -\gamma(\pi, M)$.
7. U_L: free vertices in L. U_R free vertices in R.
8. Looking for: path π in G_M starting U_L going to U_R with maximum weight $\gamma(\pi)$. Min weight $\alpha(\pi)$.

To be given a more exciting title...

1. $G = (L \cup R, E)$: given bipartite graph.
2. $w : E \rightarrow \mathbb{R}$: non-negative weights on edges.
3. M: matching.
4. $\mathbf{G}_M$: directed graph (like unweighted graph):
    1. $rl \in M$, $l \in L$ and $r \in R$: add (r, l) to $\mathbf{E}(\mathbf{G}_M)$. Weight $\alpha((r, l)) = w(rl)$.
    2. $rl \in E \setminus M$: add edge $(l \rightarrow r) \in \mathbf{E}(\mathbf{G}_M)$. With weight $\alpha((l, r)) = -w(rl)$.
5. π: augmenting path in $\mathbf{G}$ = π path from free vertex in L to free vertex in R in $\mathbf{G}_M$.
6. path π in $\mathbf{G}_M$ has weight $\alpha(\pi) = -\gamma(\pi, M)$.
7. U_L: free vertices in L. U_R free vertices in R.
8. Looking for: path π in $\mathbf{G}_M$ starting U_L going to U_R with maximum weight $\gamma(\pi)$. Min weight $\alpha(\pi)$.

To be given a more exciting title...

1. $G = (L \cup R, E)$: given bipartite graph.
2. $w : E \to \mathbb{R}$: non-negative weights on edges.
3. M: matching.
4. $\mathbf{G}_M$: directed graph (like unweighted graph):
   1. $rl \in M$, $l \in L$ and $r \in R$: add (r, l) to $\mathbf{E}(\mathbf{G}_M)$. Weight $\alpha((r, l)) = w(rl)$.
   2. $rl \in E \setminus M$: add edge $(l \to r) \in \mathbf{E}(\mathbf{G}_M)$. With weight $\alpha((l, r)) = -w(rl)$.
5. π: augmenting path in $\mathbf{G}$ = π path from free vertex in L to free vertex in R in $\mathbf{G}_M$.
6. path π in $\mathbf{G}_M$ has weight $\alpha(\pi) = -\gamma(\pi, M)$.
7. U_L: free vertices in L. U_R free vertices in R.
8. Looking for: path π in $\mathbf{G}_M$ starting U_L going to U_R with maximum weight $\gamma(\pi)$. Min weight $\alpha(\pi)$.

To be given a more exciting title...

1. $G = (L \cup R, E)$: given bipartite graph.
2. $w : E \to \mathbb{R}$: non-negative weights on edges.
3. M: matching.
4. $\mathbf{G}_M$: directed graph (like unweighted graph):
   1. $rl \in M$, $l \in L$ and $r \in R$: add (r, l) to $\mathbf{E}(\mathbf{G}_M)$. Weight $\alpha((r, l)) = w(rl)$.
   2. $rl \in E \setminus M$: add edge $(l \to r) \in \mathbf{E}(\mathbf{G}_M)$. With weight $\alpha((l, r)) = -w(rl)$.
5. π: augmenting path in $\mathbf{G}$ = π path from free vertex in L to free vertex in R in $\mathbf{G}_M$.
6. path π in $\mathbf{G}_M$ has weight $\alpha(\pi) = -\gamma(\pi, M)$.
7. U_L: free vertices in L. U_R free vertices in R.
8. Looking for: path π in $\mathbf{G}_M$ starting U_L going to U_R with maximum weight $\gamma(\pi)$. Min weight $\alpha(\pi)$.

To be given a more exciting title...

1. $G = (L \cup R, E)$: given bipartite graph.
2. $w : E \rightarrow \mathbb{R}$: non-negative weights on edges.
3. M: matching.
4. $\mathbf{G}_M$: directed graph (like unweighted graph):
    1. $rl \in M$, $l \in L$ and $r \in R$: add (r, l) to $\mathbf{E}(\mathbf{G}_M)$. Weight $\alpha((r, l)) = w(rl)$.
    2. $rl \in E \setminus M$: add edge $(l \rightarrow r) \in \mathbf{E}(\mathbf{G}_M)$. With weight $\alpha((l, r)) = -w(rl)$.
5. π: augmenting path in $\mathbf{G}$ = π path from free vertex in L to free vertex in R in $\mathbf{G}_M$.
6. path π in $\mathbf{G}_M$ has weight $\alpha(\pi) = -\gamma(\pi, M)$.
7. U_L: free vertices in L. U_R free vertices in R.
8. Looking for: path π in $\mathbf{G}_M$ starting U_L going to U_R with maximum weight $\gamma(\pi)$. Min weight $\alpha(\pi)$.

To be given a more exciting title...

1. $G = (L \cup R, E)$: given bipartite graph.
2. $w : E \rightarrow \mathbb{R}$: non-negative weights on edges.
3. M: matching.
4. $\mathbf{G}_M$: directed graph (like unweighted graph):
    1. $rl \in M$, $l \in L$ and $r \in R$: add (r, l) to $\mathbf{E}(\mathbf{G}_M)$. Weight $\alpha((r, l)) = w(rl)$.
    2. $rl \in E \setminus M$: add edge $(l \rightarrow r) \in \mathbf{E}(\mathbf{G}_M)$. With weight $\alpha((l, r)) = -w(rl)$.
5. π: augmenting path in $\mathbf{G}$ = π path from free vertex in L to free vertex in R in $\mathbf{G}_M$.
6. path π in $\mathbf{G}_M$ has weight $\alpha(\pi) = -\gamma(\pi, M)$.
7. U_L: free vertices in L. U_R free vertices in R.
8. Looking for: path π in $\mathbf{G}_M$ starting U_L going to U_R with maximum weight $\gamma(\pi)$. Min weight $\alpha(\pi)$.

No negative cycles for max weight matching

Lemma

If M is a maximum weight matching with k edges in $\mathbf{G}$, than there is no negative cycle in $\mathbf{G}_M$ where $\alpha(\cdot)$ is the associated weight function.

Proof.

Assume for the sake of contradiction that there is a cycle C, and observe that $\gamma(C) = -\alpha(C) > 0$. Namely, $M \oplus C$ is a new matching with bigger weight and the same number of edges. A contradiction to the maximality of M. □

17.2.1: The algorithm.

The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
    1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
    1. Collapse all free vertices of U_L into a single vertex.
    2. Collapse all free vertices of U_R into a single vertex.
    3. H_M: resulting graph.
    4. Compute shortest path from U_L to U_R in H_M.
    5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
   1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
   1. Collapse all free vertices of U_L into a single vertex.
   2. Collapse all free vertices of U_R into a single vertex.
   3. H_M: resulting graph.
   4. Compute shortest path from U_L to U_R in H_M.
   5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
   1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
   1. Collapse all free vertices of U_L into a single vertex.
   2. Collapse all free vertices of U_R into a single vertex.
   3. H_M: resulting graph.
   4. Compute shortest path from U_L to U_R in H_M.
   5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
    1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
    1. Collapse all free vertices of U_L into a single vertex.
    2. Collapse all free vertices of U_R into a single vertex.
    3. H_M: resulting graph.
    4. Compute shortest path from U_L to U_R in H_M.
    5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
    1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
    1. Collapse all free vertices of U_L into a single vertex.
    2. Collapse all free vertices of U_R into a single vertex.
    3. H_M: resulting graph.
    4. Compute shortest path from U_L to U_R in H_M.
    5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
   1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
   1. Collapse all free vertices of U_L into a single vertex.
   2. Collapse all free vertices of U_R into a single vertex.
   3. H_M: resulting graph.
   4. Compute shortest path from U_L to U_R in H_M.
   5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
    1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
    1. Collapse all free vertices of U_L into a single vertex.
    2. Collapse all free vertices of U_R into a single vertex.
    3. H_M: resulting graph.
    4. Compute shortest path from U_L to U_R in H_M.
    5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
   1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
   1. Collapse all free vertices of U_L into a single vertex.
   2. Collapse all free vertices of U_R into a single vertex.
   3. H_M: resulting graph.
   4. Compute shortest path from U_L to U_R in H_M.
   5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

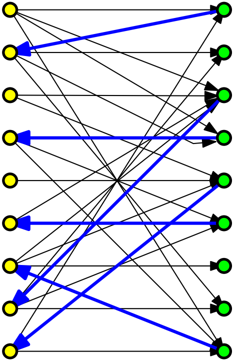
The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
   1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
   1. Collapse all free vertices of U_L into a single vertex.
   2. Collapse all free vertices of U_R into a single vertex.
   3. H_M: resulting graph.
   4. Compute shortest path from U_L to U_R in H_M.
   5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

The algorithm...

1. Compute a maximum weight in the bipartite graph **G** as follows:
   1. Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.
2. Compute a minimum weight path in $\mathbf{G}_M$ between U_L and U_R.
3. Shortest path in $\mathbf{G}_M$ with no negative cycles (but negative weights on edges).
4. Use **Bellman-Ford** algorithm.
   1. Collapse all free vertices of U_L into a single vertex.
   2. Collapse all free vertices of U_R into a single vertex.
   3. H_M: resulting graph.
   4. Compute shortest path from U_L to U_R in H_M.
   5. since no negative cycles. **Bellman-Ford** algorithm works in $O(nm)$ time.

A figure...



Result

1. Result:

Lemma

Given a bipartite graph $\mathbf{G}$ *and a maximum weight matching* M *of size* k *one can find a maximum weight augmenting path for* $\mathbf{G}$ *in* $O(nm)$ *time, where* n *is the number of vertices of* $\mathbf{G}$ *and* m *is the number of edges.*

2. Applying this algorithm $n/2$ times at most:

Theorem

Given a weight bipartite graph $\mathbf{G}$*, with* n *vertices and* m *edges, one can compute a maximum weight matching in* $\mathbf{G}$ *in* $O(n^2 m)$ *time.*

Result

1. Result:

Lemma

Given a bipartite graph **G** *and a maximum weight matching* M *of size* k *one can find a maximum weight augmenting path for* **G** *in* $O(nm)$ *time, where* n *is the number of vertices of* **G** *and* m *is the number of edges.*

2. Applying this algorithm $n/2$ times at most:

Theorem

Given a weight bipartite graph **G**, *with* n *vertices and* m *edges, one can compute a maximum weight matching in* **G** *in* $O(n^2 m)$ *time.*

Faster algorithm...

Working harder, one can get a faster algorithm. We state the result without proof:

Theorem

Given a weight bipartite graph **G**, *with* n *vertices and* m *edges, one can compute a maximum weight matching in* **G** *in* $O(n(n \log n + m))$ *time.*

17.2.1.1: The Bellman-Ford Algorithm - A Quick Reminder

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
    1. scan all the edges.
    2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
    3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
    4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
   1. scan all the edges.
   2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
   3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
   4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
   1. scan all the edges.
   2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
   3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
   4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
    1. scan all the edges.
    2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
    3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
    4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
   1. scan all the edges.
   2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
   3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
   4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
    1. scan all the edges.
    2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
    3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
    4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
   1. scan all the edges.
   2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
   3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
   4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
    1. scan all the edges.
    2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
    3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
    4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
   1. scan all the edges.
   2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
   3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
   4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
    1. scan all the edges.
    2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
    3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
    4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph **G**.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
   1. scan all the edges.
   2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
   3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
   4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

Bellman-Ford

1. **Bellman-Ford** computes shortest path from a single source s in a graph $\mathbf{G}$.
2. Assumption: no negative cycles (but weights can be negative).
3. Init: $\forall u \in \mathbf{V}(\mathbf{G})$: $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$.
4. Repeat n times:
   1. scan all the edges.
   2. $\forall (u, v) \in \mathbf{E}(\mathbf{G})$ it performs a **Relax**(u, v) operation.
   3. **relax**(u, v): if $x = d[u] + w((u, v)) < d[v]$, set $d[v]$ to x
   4. $d[u]$: current distance from s to u.
5. Overall running time is $O(mn)$.
6. Claim: in end of exec- shortest path length from s to u is $d[u]$.
7. By induction: All vertices with shortest path to s with i edges, are being set to their shortest path length in the ith iteration
8. Can modify to detect negative cycles.

17.3: Maximum Size Matching in a Non-Bipartite Graph

Non-bipartite matching...

1. Graph not bipartite. No weights on edges.
2. Start from an empty matching M
3. repeatedly find an augmenting path from an unmatched vertex to an unmatched vertex.

Non-bipartite matching...

1. Graph not bipartite. No weights on edges.
2. Start from an empty matching M
3. repeatedly find an augmenting path from an unmatched vertex to an unmatched vertex.

Non-bipartite matching...

1. Graph not bipartite. No weights on edges.
2. Start from an empty matching M
3. repeatedly find an augmenting path from an unmatched vertex to an unmatched vertex.

Notations

1. $\mathcal{T}$: a given tree.
2. For two vertices $x, y \in V(\mathcal{T})$: τ_{xy} denote the path in $\mathcal{T}$ between x and y.
3. For two paths π and π' that share an endpoint.
4. $\pi \mid\mid \pi'$ concatenated path
5. $|\pi|$ denote the number of edges in π.

Notations

1. $\mathcal{T}$: a given tree.
2. For two vertices $x, y \in V(\mathcal{T})$: τ_{xy} denote the path in $\mathcal{T}$ between x and y.
3. For two paths π and π' that share an endpoint.
4. $\pi \mid\mid \pi'$ concatenated path
5. $|\pi|$ denote the number of edges in π.

Notations

1. $\mathcal{T}$: a given tree.
2. For two vertices $x, y \in V(\mathcal{T})$: τ_{xy} denote the path in $\mathcal{T}$ between x and y.
3. For two paths π and π' that share an endpoint.
4. $\pi \mid\mid \pi'$ concatenated path
5. $|\pi|$ denote the number of edges in π.

Notations

1. $\mathcal{T}$: a given tree.
2. For two vertices $x, y \in V(\mathcal{T})$: τ_{xy} denote the path in $\mathcal{T}$ between x and y.
3. For two paths π and π' that share an endpoint.
4. $\pi \mid\mid \pi'$ concatenated path
5. $|\pi|$ denote the number of edges in π.

Notations

1. $\mathcal{T}$: a given tree.
2. For two vertices $x, y \in V(\mathcal{T})$: τ_{xy} denote the path in $\mathcal{T}$ between x and y.
3. For two paths π and π' that share an endpoint.
4. $\pi \mid\mid \pi'$ concatenated path
5. $|\pi|$ denote the number of edges in π.

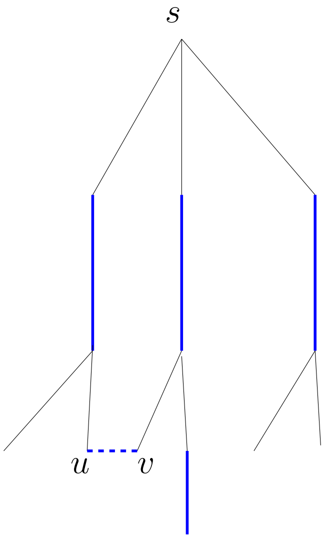
Notations

1. $\mathcal{T}$: a given tree.
2. For two vertices $x, y \in V(\mathcal{T})$: τ_{xy} denote the path in $\mathcal{T}$ between x and y.
3. For two paths π and π' that share an endpoint.
4. $\pi \mid\mid \pi'$ concatenated path
5. $|\pi|$ denote the number of edges in π.

17.3.1: Finding an augmenting path

A figure

A cycle in the alternating BFS tree.



Algorithm: First try

1. **G**: graph. M: matching.
2. Task: compute bigger matching in **G**.
3. Compute an augmenting path for M.
4. Add edges that are both endpoints free to matching.
5. Assume $\forall$ edges at least one of their endpoint adjacent to matching edge.
6. Collapse unmatched vertices to single vertex s.
7. H : resulting graph.
8. compute an **alternating BFS** of H starting from s.
9. **BFS** on H from s.
    1. even levels of **BFS** tree use only matching edges.
    2. odd levels **BFS** tree: only free edges.
    3. Let $\mathcal{T}$ denote the resulting tree.
10. Augmenting path in **G** corresponds to an odd cycle in H passing through s.

Algorithm: First try

1. **G**: graph. M: matching.
2. Task: compute bigger matching in **G**.
3. Compute an augmenting path for M.
4. Add edges that are both endpoints free to matching.
5. Assume ∀ edges at least one of their endpoint adjacent to matching edge.
6. Collapse unmatched vertices to single vertex s.
7. H : resulting graph.
8. compute an **alternating BFS** of H starting from s.
9. **BFS** on H from s.
    1. even levels of **BFS** tree use only matching edges.
    2. odd levels **BFS** tree: only free edges.
    3. Let $\mathcal{T}$ denote the resulting tree.
10. Augmenting path in **G** corresponds to an odd cycle in H passing through s.

Algorithm: First try

1. **G**: graph. M: matching.
2. Task: compute bigger matching in **G**.
3. Compute an augmenting path for M.
4. Add edges that are both endpoints free to matching.
5. Assume ∀ edges at least one of their endpoint adjacent to matching edge.
6. Collapse unmatched vertices to single vertex s.
7. H : resulting graph.
8. compute an **alternating BFS** of H starting from s.
9. **BFS** on H from s.
    1. even levels of **BFS** tree use only matching edges.
    2. odd levels **BFS** tree: only free edges.
    3. Let $\mathcal{T}$ denote the resulting tree.
10. Augmenting path in **G** corresponds to an odd cycle in H passing through s.

Algorithm: First try

1. **G**: graph. M: matching.
2. Task: compute bigger matching in **G**.
3. Compute an augmenting path for M.
4. Add edges that are both endpoints free to matching.
5. Assume ∀ edges at least one of their endpoint adjacent to matching edge.
6. Collapse unmatched vertices to single vertex s.
7. H : resulting graph.
8. compute an **alternating BFS** of H starting from s.
9. **BFS** on H from s.
   1. even levels of **BFS** tree use only matching edges.
   2. odd levels **BFS** tree: only free edges.
   3. Let $\mathcal{T}$ denote the resulting tree.
10. Augmenting path in **G** corresponds to an odd cycle in H passing through s.

Algorithm: First try

1. **G**: graph. M: matching.
2. Task: compute bigger matching in **G**.
3. Compute an augmenting path for M.
4. Add edges that are both endpoints free to matching.
5. Assume $\forall$ edges at least one of their endpoint adjacent to matching edge.
6. Collapse unmatched vertices to single vertex s.
7. H : resulting graph.
8. compute an **alternating BFS** of H starting from s.
9. **BFS** on H from s.
   1. even levels of **BFS** tree use only matching edges.
   2. odd levels **BFS** tree: only free edges.
   3. Let $\mathcal{T}$ denote the resulting tree.
10. Augmenting path in **G** corresponds to an odd cycle in H passing through s.

Algorithm: First try

1. **G**: graph. M: matching.
2. Task: compute bigger matching in **G**.
3. Compute an augmenting path for M.
4. Add edges that are both endpoints free to matching.
5. Assume ∀ edges at least one of their endpoint adjacent to matching edge.
6. Collapse unmatched vertices to single vertex s.
7. H : resulting graph.
8. compute an **alternating BFS** of H starting from s.
9. **BFS** on H from s.
    1. even levels of **BFS** tree use only matching edges.
    2. odd levels **BFS** tree: only free edges.
    3. Let $\mathcal{T}$ denote the resulting tree.
10. Augmenting path in **G** corresponds to an odd cycle in H passing through s.

Algorithm: First try

1. **G**: graph. M: matching.
2. Task: compute bigger matching in **G**.
3. Compute an augmenting path for M.
4. Add edges that are both endpoints free to matching.
5. Assume ∀ edges at least one of their endpoint adjacent to matching edge.
6. Collapse unmatched vertices to single vertex s.
7. H : resulting graph.
8. compute an **alternating BFS** of H starting from s.
9. **BFS** on H from s.
    1. even levels of **BFS** tree use only matching edges.
    2. odd levels **BFS** tree: only free edges.
    3. Let $\mathcal{T}$ denote the resulting tree.
10. Augmenting path in **G** corresponds to an odd cycle in H passing through s.

Algorithm: First try

1. **G**: graph. M: matching.
2. Task: compute bigger matching in **G**.
3. Compute an augmenting path for M.
4. Add edges that are both endpoints free to matching.
5. Assume ∀ edges at least one of their endpoint adjacent to matching edge.
6. Collapse unmatched vertices to single vertex s.
7. H : resulting graph.
8. compute an **alternating BFS** of H starting from s.
9. **BFS** on H from s.
    1. even levels of **BFS** tree use only matching edges.
    2. odd levels **BFS** tree: only free edges.
    3. Let $\mathcal{T}$ denote the resulting tree.
10. Augmenting path in **G** corresponds to an odd cycle in H passing through s.

Algorithm: First try

1. **G**: graph. M: matching.
2. Task: compute bigger matching in **G**.
3. Compute an augmenting path for M.
4. Add edges that are both endpoints free to matching.
5. Assume $\forall$ edges at least one of their endpoint adjacent to matching edge.
6. Collapse unmatched vertices to single vertex s.
7. H : resulting graph.
8. compute an **alternating BFS** of H starting from s.
9. **BFS** on H from s.
   1. even levels of **BFS** tree use only matching edges.
   2. odd levels **BFS** tree: only free edges.
   3. Let $\mathcal{T}$ denote the resulting tree.
10. Augmenting path in **G** corresponds to an odd cycle in H passing through s.

Like a bridge over troubled matching...

Definition

An edge $uv \in E(G)$ is a **bridge** if the following conditions are met:

(i) u and v have the same depth in $\mathcal{T}$,

(ii) if the depth of u in $\mathcal{T}$ is even then uv is free (i.e., $uv \notin M$), and

(iii) if the depth of u in $\mathcal{T}$ is odd then $uv \in M$.

Finding odd cycles...

1. given an edge uv... can check if it is a bridge in constant time.
2. We need the following:

Lemma

Let v be a vertex of $\mathbf{G}$, M a matching in $\mathbf{G}$, and let π be the shortest alternating path between s and v in $\mathbf{G}$. Furthermore, assume that for any vertex w of π the shortest alternating path between w and s is the path along π.
Then, the depth $d_{\mathcal{T}}(v)$ of v in $\mathcal{T}$ is $|\pi|$.

Finding odd cycles...

1. given an edge uv... can check if it is a bridge in constant time.
2. We need the following:

Lemma

Let v be a vertex of $\mathbf{G}$, M a matching in $\mathbf{G}$, and let π be the shortest alternating path between s and v in $\mathbf{G}$. Furthermore, assume that for any vertex w of π the shortest alternating path between w and s is the path along π.
Then, the depth $d_{\mathcal{T}}(v)$ of v in $\mathcal{T}$ is $|\pi|$.

Finding odd cycles...

1. given an edge uv... can check if it is a bridge in constant time.
2. We need the following:

Lemma

Let v be a vertex of $\mathbf{G}$, M a matching in $\mathbf{G}$, and let π be the shortest alternating path between s and v in $\mathbf{G}$. Furthermore, assume that for any vertex w of π the shortest alternating path between w and s is the path along π.
Then, the depth $d_{\mathcal{T}}(v)$ of v in $\mathcal{T}$ is $|\pi|$.

Proof

Proof.

1. Induction on $|\pi|$. For $|\pi| = 1$: easy... v is a neighbor of s in **G**... v child of s in **BFS** tree $\mathcal{T}$.
2. $|\pi| = k$. u: vertex just before v on π.
3. By induction, depth of u in $\mathcal{T}$ is $k - 1$.
4. When alternating **BFS** algorithm visited u: tried hang v from u...
5. failure only if v already in $\mathcal{T}$.
6. $\implies$ exists a shorter alternating path from s to v
7. A contradiction.

□

Proof

Proof.

1. Induction on $|\pi|$. For $|\pi| = 1$: easy... v is a neighbor of s in **G**... v child of s in **BFS** tree $\mathcal{T}$.
2. $|\pi| = k$. u: vertex just before v on π.
3. By induction, depth of u in $\mathcal{T}$ is $k - 1$.
4. When alternating **BFS** algorithm visited u: tried hang v from u...
5. failure only if v already in $\mathcal{T}$.
6. $\implies$ exists a shorter alternating path from s to v
7. A contradiction.

□

Proof

Proof.

1. Induction on $|\pi|$. For $|\pi| = 1$: easy... v is a neighbor of s in **G**... v child of s in **BFS** tree $\mathcal{T}$.
2. $|\pi| = k$. u: vertex just before v on π.
3. By induction, depth of u in $\mathcal{T}$ is $k - 1$.
4. When alternating **BFS** algorithm visited u: tried hang v from u...
5. failure only if v already in $\mathcal{T}$.
6. $\implies$ exists a shorter alternating path from s to v
7. A contradiction.

Proof

Proof.

1. Induction on $|\pi|$. For $|\pi| = 1$: easy... v is a neighbor of s in **G**... v child of s in **BFS** tree $\mathcal{T}$.
2. $|\pi| = k$. u: vertex just before v on π.
3. By induction, depth of u in $\mathcal{T}$ is $k - 1$.
4. When alternating **BFS** algorithm visited u: tried hang v from u...
5. failure only if v already in $\mathcal{T}$.
6. $\implies$ exists a shorter alternating path from s to v
7. A contradiction.

□

Proof

Proof.

1. Induction on $|\pi|$. For $|\pi| = 1$: easy... v is a neighbor of s in **G**... v child of s in **BFS** tree $\mathcal{T}$.
2. $|\pi| = k$. u: vertex just before v on π.
3. By induction, depth of u in $\mathcal{T}$ is $k - 1$.
4. When alternating **BFS** algorithm visited u: tried hang v from u...
5. failure only if v already in $\mathcal{T}$.
6. $\implies$ exists a shorter alternating path from s to v
7. A contradiction.

Proof

Proof.

1. Induction on $|\pi|$. For $|\pi| = 1$: easy... v is a neighbor of s in **G**... v child of s in **BFS** tree $\mathcal{T}$.
2. $|\pi| = k$. u: vertex just before v on π.
3. By induction, depth of u in $\mathcal{T}$ is $k - 1$.
4. When alternating **BFS** algorithm visited u: tried hang v from u...
5. failure only if v already in $\mathcal{T}$.
6. $\implies$ exists a shorter alternating path from s to v
7. A contradiction.

Proof

Proof.

1. Induction on $|\pi|$. For $|\pi| = 1$: easy... v is a neighbor of s in **G**... v child of s in **BFS** tree $\mathcal{T}$.
2. $|\pi| = k$. u: vertex just before v on π.
3. By induction, depth of u in $\mathcal{T}$ is $k - 1$.
4. When alternating **BFS** algorithm visited u: tried hang v from u...
5. failure only if v already in $\mathcal{T}$.
6. $\implies$ exists a shorter alternating path from s to v
7. A contradiction.

Proof

Proof.

1. Induction on $|\pi|$. For $|\pi| = 1$: easy... v is a neighbor of s in **G**... v child of s in **BFS** tree $\mathcal{T}$.
2. $|\pi| = k$. u: vertex just before v on π.
3. By induction, depth of u in $\mathcal{T}$ is $k - 1$.
4. When alternating **BFS** algorithm visited u: tried hang v from u...
5. failure only if v already in $\mathcal{T}$.
6. $\implies$ exists a shorter alternating path from s to v
7. A contradiction.

Proof

Proof.

1. Induction on $|\pi|$. For $|\pi| = 1$: easy... v is a neighbor of s in **G**... v child of s in **BFS** tree $\mathcal{T}$.
2. $|\pi| = k$. u: vertex just before v on π.
3. By induction, depth of u in $\mathcal{T}$ is $k - 1$.
4. When alternating **BFS** algorithm visited u: tried hang v from u...
5. failure only if v already in $\mathcal{T}$.
6. $\implies$ exists a shorter alternating path from s to v
7. A contradiction.

□

If there is an augmenting path...

.., then there is a bridge

Lemma

If there is an augmenting path in **G** *for a matching* M*, then there exists an edge* $uv \in E(G)$ *which is a bridge in* $\mathfrak{T}$*.*

Proof

1. π: an augmenting path in $\mathbf{G}$.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in $\mathbf{G}$ going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathsf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path.

Proof

1. π: an augmenting path in **G**.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in **G** going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathsf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ∎

Proof

1. π: an augmenting path in **G**.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in **G** going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathsf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ■

Proof

1. π: an augmenting path in **G**.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in **G** going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathsf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ■

Proof

1. π: an augmenting path in **G**.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in **G** going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathsf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ■

Proof

1. π: an augmenting path in $\mathbf{G}$.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in $\mathbf{G}$ going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathbf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ■

Proof

1. π: an augmenting path in **G**.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in **G** going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathsf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path.

Proof

1. π: an augmenting path in **G**.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in **G** going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathbf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ■

Proof

1. π: an augmenting path in **G**.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in **G** going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathsf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ■

Proof

1. π: an augmenting path in **G**.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in **G** going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathsf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \le d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ■

Proof

1. π: an augmenting path in **G**.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in **G** going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathbf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ■

Proof

1. π: an augmenting path in $\mathbf{G}$.
2. π: odd length alternating cycle in H.
3. σ: shortest odd length alternating cycle in $\mathbf{G}$ going through s.
4. both edges in σ adjacent to s are unmatched.
5. $x \in \mathsf{V}(\sigma)$: $d(x)$ length of shortest alternating path between x and s in H.
6. $d'(x)$ len shortest alternating path between s and x along σ.
7. Clearly: $d(x) \leq d'(x)$.
8. Claim: $d(x) = d'(x)$, for all $x \in \sigma$. (See next slide for proof.)
9. Take two vertices of σ furthest away from s.
10. Both have same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$.
11. By previous lemma: $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Found bridge!
12. Observe: σ is created from an alternating path. ■

Proof of subclaim

Claim: $d(x) = d'(x)$, for all $x \in \sigma$.

1. assume for contradiction: $d(x) < d'(x)$.
2. π_1, π_2: paths from x to s formed by σ.
3. η: shortest alternating path between s and x.
4. Know: $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$.
5. Easy to verify: $\pi_1 \, || \, \eta$ or $\pi_2 \, || \, \eta$ is an alternating cycle shorter than σ involving s.
6. A contradiction.

Proof of subclaim

Claim: $d(x) = d'(x)$, for all $x \in \sigma$.

1. assume for contradiction: $d(x) < d'(x)$.
2. π_1, π_2: paths from x to s formed by σ.
3. η: shortest alternating path between s and x.
4. Know: $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$.
5. Easy to verify: $\pi_1 \mid\mid \eta$ or $\pi_2 \mid\mid \eta$ is an alternating cycle shorter than σ involving s.
6. A contradiction.

Proof of subclaim

Claim: $d(x) = d'(x)$, for all $x \in \sigma$.

1. assume for contradiction: $d(x) < d'(x)$.
2. π_1, π_2: paths from x to s formed by σ.
3. η: shortest alternating path between s and x.
4. Know: $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$.
5. Easy to verify: $\pi_1 \mid\mid \eta$ or $\pi_2 \mid\mid \eta$ is an alternating cycle shorter than σ involving s.
6. A contradiction.

Proof of subclaim

Claim: $d(x) = d'(x)$, for all $x \in \sigma$.

1. assume for contradiction: $d(x) < d'(x)$.
2. π_1, π_2: paths from x to s formed by σ.
3. η: shortest alternating path between s and x.
4. Know: $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$.
5. Easy to verify: $\pi_1 \,||\, \eta$ or $\pi_2 \,||\, \eta$ is an alternating cycle shorter than σ involving s.
6. A contradiction.

Proof of subclaim

Claim: $d(x) = d'(x)$, for all $x \in \sigma$.

1. assume for contradiction: $d(x) < d'(x)$.
2. π_1, π_2: paths from x to s formed by σ.
3. η: shortest alternating path between s and x.
4. Know: $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$.
5. Easy to verify: $\pi_1 \,||\, \eta$ or $\pi_2 \,||\, \eta$ is an alternating cycle shorter than σ involving s.
6. A contradiction.

Proof of subclaim

Claim: $d(x) = d'(x)$, for all $x \in \sigma$.

1. assume for contradiction: $d(x) < d'(x)$.
2. π_1, π_2: paths from x to s formed by σ.
3. η: shortest alternating path between s and x.
4. Know: $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$.
5. Easy to verify: $\pi_1 \mid\mid \eta$ or $\pi_2 \mid\mid \eta$ is an alternating cycle shorter than σ involving s.
6. A contradiction.

Proof of subclaim

Claim: $d(x) = d'(x)$, for all $x \in \sigma$.

1. assume for contradiction: $d(x) < d'(x)$.
2. π_1, π_2: paths from x to s formed by σ.
3. η: shortest alternating path between s and x.
4. Know: $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$.
5. Easy to verify: $\pi_1 \mid\mid \eta$ or $\pi_2 \mid\mid \eta$ is an alternating cycle shorter than σ involving s.
6. A contradiction.

Algorithm: First try

1. Compute alternating **BFS** $\mathcal{T}$ for H, and find a bridge uv in it.
2. If M is not a maximal matching, then there exists an augmenting path for **G**.
3. By lemma $\exists$ bridge.
4. Computing the bridge uv takes $O(m)$ time.
5. Extract paths from s to u and from s to v in $\mathcal{T}$.
6. Glue path together with uv to form an odd cycle μ in H.
7. namely, $\mu = \tau_{su} \mid\mid uv \mid\mid \tau_{vs}$.
8. If μ corresponds to an alternating path in **G** then done.

Algorithm: First try

1. Compute alternating **BFS** $\mathcal{T}$ for H, and find a bridge uv in it.
2. If M is not a maximal matching, then there exists an augmenting path for **G**.
3. By lemma $\exists$ bridge.
4. Computing the bridge uv takes $O(m)$ time.
5. Extract paths from s to u and from s to v in $\mathcal{T}$.
6. Glue path together with uv to form an odd cycle μ in H.
7. namely, $\mu = \tau_{su} \mid\mid uv \mid\mid \tau_{vs}$.
8. If μ corresponds to an alternating path in **G** then done.

Algorithm: First try

1. Compute alternating **BFS** $\mathcal{T}$ for H, and find a bridge uv in it.
2. If M is not a maximal matching, then there exists an augmenting path for **G**.
3. By lemma $\exists$ bridge.
4. Computing the bridge uv takes $O(m)$ time.
5. Extract paths from s to u and from s to v in $\mathcal{T}$.
6. Glue path together with uv to form an odd cycle μ in H.
7. namely, $\mu = \tau_{su} \mid\mid uv \mid\mid \tau_{vs}$.
8. If μ corresponds to an alternating path in **G** then done.

Algorithm: First try

1. Compute alternating **BFS** $\mathcal{T}$ for H, and find a bridge uv in it.
2. If M is not a maximal matching, then there exists an augmenting path for **G**.
3. By lemma $\exists$ bridge.
4. Computing the bridge uv takes $O(m)$ time.
5. Extract paths from s to u and from s to v in $\mathcal{T}$.
6. Glue path together with uv to form an odd cycle μ in H.
7. namely, $\mu = \tau_{su} \mid\mid uv \mid\mid \tau_{vs}$.
8. If μ corresponds to an alternating path in **G** then done.

Algorithm: First try

1. Compute alternating **BFS** $\mathcal{T}$ for H, and find a bridge uv in it.
2. If M is not a maximal matching, then there exists an augmenting path for **G**.
3. By lemma $\exists$ bridge.
4. Computing the bridge uv takes $O(m)$ time.
5. Extract paths from s to u and from s to v in $\mathcal{T}$.
6. Glue path together with uv to form an odd cycle μ in H.
7. namely, $\mu = \tau_{su} \mid\mid uv \mid\mid \tau_{vs}$.
8. If μ corresponds to an alternating path in **G** then done.

Algorithm: First try

1. Compute alternating **BFS** $\mathcal{T}$ for H, and find a bridge uv in it.
2. If M is not a maximal matching, then there exists an augmenting path for **G**.
3. By lemma $\exists$ bridge.
4. Computing the bridge uv takes $O(m)$ time.
5. Extract paths from s to u and from s to v in $\mathcal{T}$.
6. Glue path together with uv to form an odd cycle μ in H.
7. namely, $\mu = \tau_{su} \,||\, uv \,||\, \tau_{vs}$.
8. If μ corresponds to an alternating path in **G** then done.

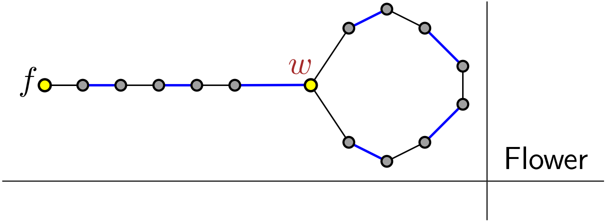
Algorithm: First try

1. Compute alternating **BFS** $\mathcal{T}$ for H, and find a bridge uv in it.
2. If M is not a maximal matching, then there exists an augmenting path for **G**.
3. By lemma $\exists$ bridge.
4. Computing the bridge uv takes $O(m)$ time.
5. Extract paths from s to u and from s to v in $\mathcal{T}$.
6. Glue path together with uv to form an odd cycle μ in H.
7. namely, $\mu = \tau_{su} \;||\; uv \;||\; \tau_{vs}$.
8. If μ corresponds to an alternating path in **G** then done.

Algorithm: First try

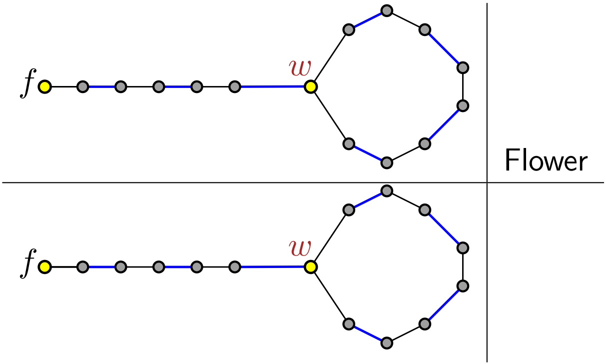
1. Compute alternating **BFS** $\mathcal{T}$ for H, and find a bridge uv in it.
2. If M is not a maximal matching, then there exists an augmenting path for **G**.
3. By lemma $\exists$ bridge.
4. Computing the bridge uv takes $O(m)$ time.
5. Extract paths from s to u and from s to v in $\mathcal{T}$.
6. Glue path together with uv to form an odd cycle μ in H.
7. namely, $\mu = \tau_{su} \mid\mid uv \mid\mid \tau_{vs}$.
8. If μ corresponds to an alternating path in **G** then done.

Flowers, stem, blossom, inverting stem



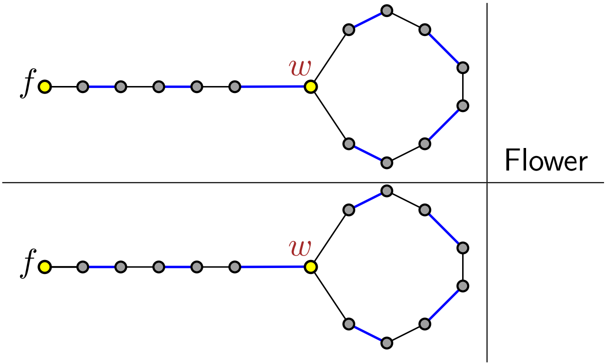
1. Flower is made out of a **stem** (the path fw), and an odd length cycle which is the blossom.
2. Stem: Even length alternating path starting with a free vertex

Flowers, stem, blossom, inverting stem



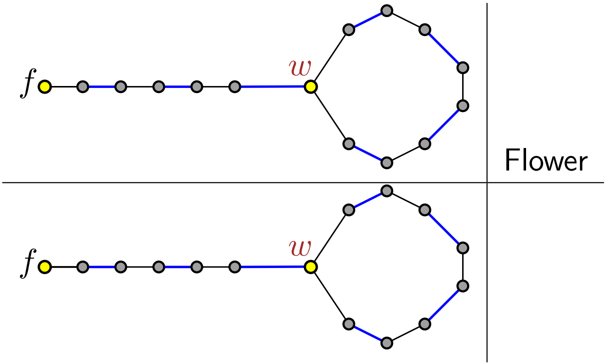
1. Flower is made out of a **stem** (the path fw), and an odd length cycle which is the blossom.
2. Stem: Even length alternating path starting with a free vertex

Flowers, stem, blossom, inverting stem



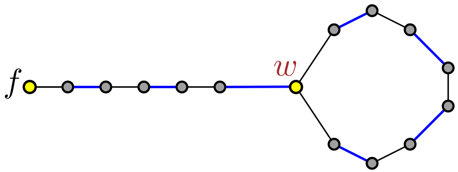
1. Flower is made out of a **stem** (the path fw), and an odd length cycle which is the blossom.
2. Stem: Even length alternating path starting with a free vertex

Flowers, stem, blossom, inverting stem

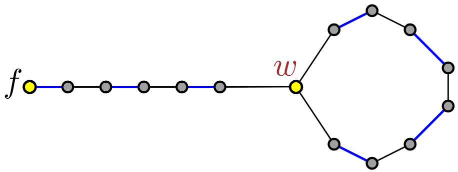


1. Flower is made out of a **stem** (the path fw), and an odd length cycle which is the blossom.
2. Stem: Even length alternating path starting with a free vertex

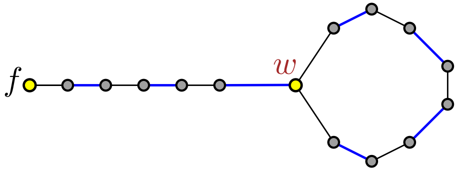
Inverting stem, rotating flower...



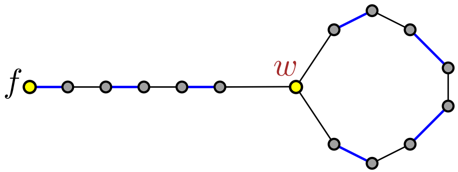
Inverting stem, rotating flower...



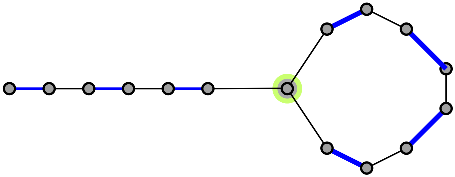
Inverting stem, rotating flower...



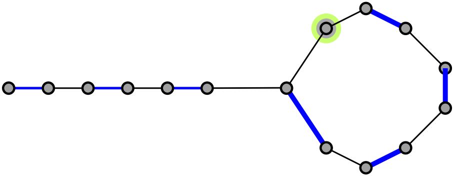
Inverting stem, rotating flower...



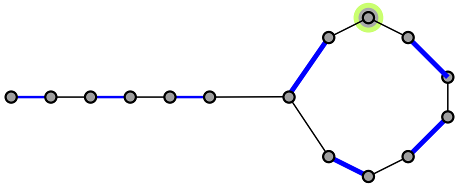
Inverting stem, rotating flower...



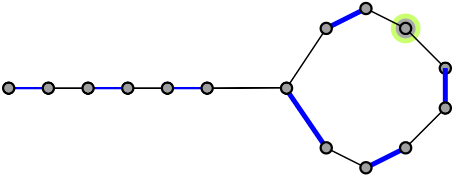
Inverting stem, rotating flower...



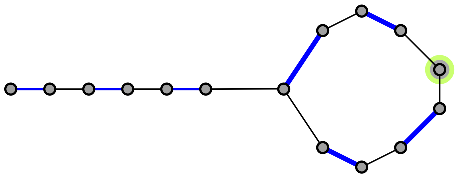
Inverting stem, rotating flower...



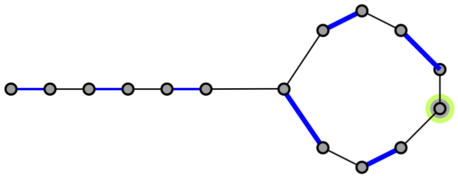
Inverting stem, rotating flower...



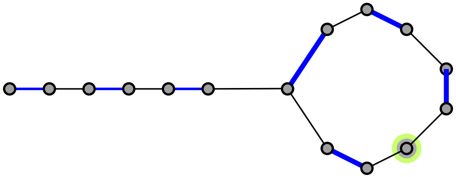
Inverting stem, rotating flower...



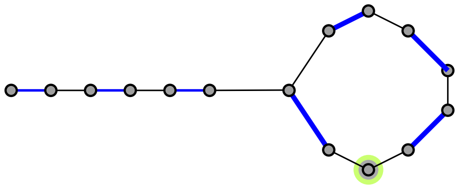
Inverting stem, rotating flower...



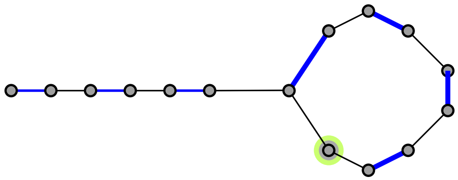
Inverting stem, rotating flower...



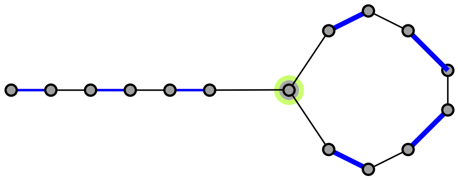
Inverting stem, rotating flower...



Inverting stem, rotating flower...



Inverting stem, rotating flower...



Flowers

1. π_{su} and π_{sv}: two paths from s to u and v.
2. w: lowest vertex in $\mathcal{T}$ common to π_{su} and π_{sv}.
3. Flower:

Definition

Given a matching M, a **flower** for M is formed by a **stem** and a **blossom**. The stem is an even length alternating path starting at a free vertex v ending at vertex w, and the blossom is an odd length (alternating) cycle based at w.

Flowers

1. π_{su} and π_{sv}: two paths from s to u and v.
2. w: lowest vertex in $\mathcal{T}$ common to π_{su} and π_{sv}.
3. Flower:

Definition

Given a matching M, a **flower** for M is formed by a **stem** and a **blossom**. The stem is an even length alternating path starting at a free vertex v ending at vertex w, and the blossom is an odd length (alternating) cycle based at w.

Flowers

1. π_{su} and π_{sv}: two paths from s to u and v.
2. w: lowest vertex in $\mathcal{T}$ common to π_{su} and π_{sv}.
3. Flower:

Definition

Given a matching M, a **flower** for M is formed by a **stem** and a **blossom**. The stem is an even length alternating path starting at a free vertex v ending at vertex w, and the blossom is an odd length (alternating) cycle based at w.

Lemma

Lemma

Consider a bridge edge $uv \in G$*, and let* w *be the least common ancestor (LCA) of* u *and* v *in* $\mathfrak{T}$*. Consider the path* π_{sw} *together with the cycle* $C = \pi_{wu} \mid\mid uv \mid\mid \pi_{vw}$*. Then* π_{sw} *and* C *together form a flower.*

Proof

Proof.

Since only the even depth nodes in $\mathcal{T}$ have more than one child, w must be of even depth, and as such π_{sw} is of even length. As for the second claim, observe that $\alpha = |\pi_{wu}| = |\pi_{wv}|$ since the two nodes have the same depth in $\mathcal{T}$. In particular, $|C| = |\pi_{wu}| + |\pi_{wv}| + 1 = 2\alpha + 1$, which is an odd number. □

Back to the future...

1. translate blossom of $H \Rightarrow$ original graph $\mathbf{G}$.
2. Path s to w corresponds to an alternating path starting at a free vertex f (of $\mathbf{G}$) and ending at w.
3. the last edge is in the stem is in the matching.
4. cycle $w \ldots u \ldots v \ldots w$ is an alternating odd length cycle in $\mathbf{G}$ where the two edges adjacent to w are unmatched.
5. Can not apply blossom to a matching to get better matching.
6. Yields an illegal matching!
7. But we discovered odd alternating cycle!

Back to the future...

1. translate blossom of $H \Rightarrow$ original graph $\mathbf{G}$.
2. Path s to w corresponds to an alternating path starting at a free vertex f (of $\mathbf{G}$) and ending at w.
3. the last edge is in the stem is in the matching.
4. cycle $w \ldots u \ldots v \ldots w$ is an alternating odd length cycle in $\mathbf{G}$ where the two edges adjacent to w are unmatched.
5. Can not apply blossom to a matching to get better matching.
6. Yields an illegal matching!
7. But we discovered odd alternating cycle!

Back to the future...

1. translate blossom of $H \Rightarrow$ original graph **G**.
2. Path s to w corresponds to an alternating path starting at a free vertex f (of **G**) and ending at w.
3. the last edge is in the stem is in the matching.
4. cycle $w \ldots u \ldots v \ldots w$ is an alternating odd length cycle in **G** where the two edges adjacent to w are unmatched.
5. Can not apply blossom to a matching to get better matching.
6. Yields an illegal matching!
7. But we discovered odd alternating cycle!

Back to the future...

1. translate blossom of $H \Rightarrow$ original graph $\mathbf{G}$.
2. Path s to w corresponds to an alternating path starting at a free vertex f (of $\mathbf{G}$) and ending at w.
3. the last edge is in the stem is in the matching.
4. cycle $w \ldots u \ldots v \ldots w$ is an alternating odd length cycle in $\mathbf{G}$ where the two edges adjacent to w are unmatched.
5. Can not apply blossom to a matching to get better matching.
6. Yields an illegal matching!
7. But we discovered odd alternating cycle!

Back to the future...

1. translate blossom of $H \Rightarrow$ original graph $\mathbf{G}$.
2. Path s to w corresponds to an alternating path starting at a free vertex f (of $\mathbf{G}$) and ending at w.
3. the last edge is in the stem is in the matching.
4. cycle $w \ldots u \ldots v \ldots w$ is an alternating odd length cycle in $\mathbf{G}$ where the two edges adjacent to w are unmatched.
5. Can not apply blossom to a matching to get better matching.
6. Yields an illegal matching!
7. But we discovered odd alternating cycle!

Back to the future...

1. translate blossom of $H \Rightarrow$ original graph **G**.
2. Path s to w corresponds to an alternating path starting at a free vertex f (of **G**) and ending at w.
3. the last edge is in the stem is in the matching.
4. cycle $w \ldots u \ldots v \ldots w$ is an alternating odd length cycle in **G** where the two edges adjacent to w are unmatched.
5. Can not apply blossom to a matching to get better matching.
6. Yields an illegal matching!
7. But we discovered odd alternating cycle!

Back to the future...

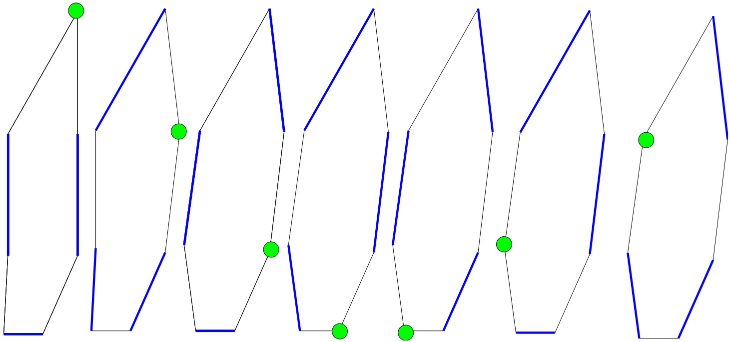
1. translate blossom of $H \Rightarrow$ original graph **G**.
2. Path s to w corresponds to an alternating path starting at a free vertex f (of **G**) and ending at w.
3. the last edge is in the stem is in the matching.
4. cycle $w \ldots u \ldots v \ldots w$ is an alternating odd length cycle in **G** where the two edges adjacent to w are unmatched.
5. Can not apply blossom to a matching to get better matching.
6. Yields an illegal matching!
7. But we discovered odd alternating cycle!

What we proved...

Lemma

Given a graph **G** *with* n *vertices and* m *edges, and a matching* M, *one can find in* $O(n + m)$ *time, either a blossom in* **G** *or an augmenting path in* **G**.

Odd alternating cycles are awesome!



Further to matching!

1. How matching in **G** interact with an odd length alternating cycle?
2. Assume free vertex in the cycle is unmatched.
3. Cycle with t vertices... Use at most $(t-1)/2$ edges in matching.
4. Rotate the matching edges in the cycle!
5. Any vertex on cycle can be free.

Further to matching!

1. How matching in **G** interact with an odd length alternating cycle?
2. Assume free vertex in the cycle is unmatched.
3. Cycle with t vertices... Use at most $(t-1)/2$ edges in matching.
4. Rotate the matching edges in the cycle!
5. Any vertex on cycle can be free.

Further to matching!

1. How matching in $\mathbf{G}$ interact with an odd length alternating cycle?
2. Assume free vertex in the cycle is unmatched.
3. Cycle with t vertices... Use at most $(t-1)/2$ edges in matching.
4. Rotate the matching edges in the cycle!
5. Any vertex on cycle can be free.

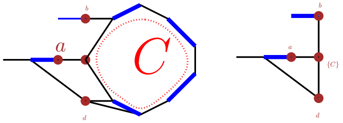
Further to matching!

1. How matching in $\mathbf{G}$ interact with an odd length alternating cycle?
2. Assume free vertex in the cycle is unmatched.
3. Cycle with t vertices... Use at most $(t-1)/2$ edges in matching.
4. Rotate the matching edges in the cycle!
5. Any vertex on cycle can be free.

Further to matching!

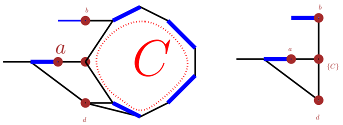
1. How matching in $\mathbf{G}$ interact with an odd length alternating cycle?
2. Assume free vertex in the cycle is unmatched.
3. Cycle with t vertices... Use at most $(t-1)/2$ edges in matching.
4. Rotate the matching edges in the cycle!
5. Any vertex on cycle can be free.

Collapse odd alternating cycles...



1. G/C: denote graph resulting from collapsing an odd cycle C into single vertex.
2. New vertex is marked by $\{C\}$.

Collapse odd alternating cycles...



1. G/C: denote graph resulting from collapsing an odd cycle C into single vertex.
2. New vertex is marked by $\{C\}$.

A lemma

Lemma

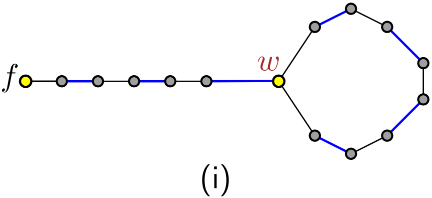
Given a graph $\mathbf{G}$, *a matching* M, *and a flower* B, *one can find a matching* M' *with the same cardinality, such that the blossom of* B *contains a free (i.e., unmatched) vertex in* M'.

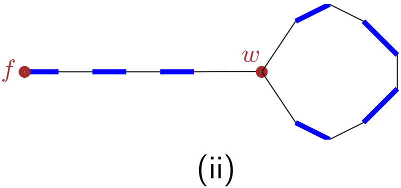
Proof

Proof.

If the stem of B is empty and B is just formed by a blossom, and then we are done. Otherwise, B was as stem π which is an even length alternating path starting from from a free vertex v. Observe that the matching $M' = M \oplus \pi$ is of the same cardinality, and the cycle in B now becomes an alternating odd cycle, with a free vertex. Intuitively, what we did is to apply the stem to the matching M. See Figure **??**. □

Proof by figure





(i) the flower, and (ii) the inverted stem.

Kill the flower, save the matching algorithm

Theorem

Let M be a matching, and let C be a blossom for M with an unmatched vertex v. Then, M is a maximum matching in **G** *if and only if $M/C = M \setminus C$ is a maximum matching in G/C.*

Proof

Proof.

Let G/C be the collapsed graph, with $\{C\}$ denoting the vertex that correspond to the cycle C.
Note, that the collapsed vertex $\{C\}$ in G/C is free. Thus, an augmenting path π in G/C either avoids the collapsed vertex $\{C\}$ altogether, or it starts or ends there. In any case, we can rotate the matching around C such that π would be an augmenting path in **G**.
Thus, if M/C is not a maximum matching in G/C then there exists an augmenting path in G/C, which in turn is an augmenting path in **G**, and as such M is not a maximum matching in **G**.
Similarly, if π is an augmenting path in **G** and it avoids C then it is also an augmenting path in G/C, and then M/C is not a maximum matching in G/C.
Otherwise, since π starts and ends in two different free vertices and C has only one free vertex, it follows that π has an endpoint outside

In other words...

Corollary

Let M be a matching, and let C be an alternating odd length cycle with the unmatched vertex being free. Then, there is an augmenting path in **G** *if and only if there is an augmenting path in G/C.*

17.3.2: The algorithm

The algorithm...

1. Start from empty matching M in graph **G**.
2. Now, repeatedly, try to enlarge the matching.
3. First, check if you can find an edge with both endpoints being free, and if so add it to the matching.
4. Compute the graph H (all free vertices collapsed into a single vertex).
5. Compute an alternating BFS tree in H.
6. Extract shortest alternating cycle based in the root (by finding the highest bridge).
7. If alternating cycle corresponds to an alternating path in **G** then apply and continue.

The algorithm...

1. Start from empty matching M in graph **G**.
2. Now, repeatedly, try to enlarge the matching.
3. First, check if you can find an edge with both endpoints being free, and if so add it to the matching.
4. Compute the graph H (all free vertices collapsed into a single vertex).
5. Compute an alternating BFS tree in H.
6. Extract shortest alternating cycle based in the root (by finding the highest bridge).
7. If alternating cycle corresponds to an alternating path in **G** then apply and continue.

The algorithm...

1. Start from empty matching M in graph **G**.
2. Now, repeatedly, try to enlarge the matching.
3. First, check if you can find an edge with both endpoints being free, and if so add it to the matching.
4. Compute the graph H (all free vertices collapsed into a single vertex).
5. Compute an alternating BFS tree in H.
6. Extract shortest alternating cycle based in the root (by finding the highest bridge).
7. If alternating cycle corresponds to an alternating path in **G** then apply and continue.

The algorithm...

1. Start from empty matching M in graph **G**.
2. Now, repeatedly, try to enlarge the matching.
3. First, check if you can find an edge with both endpoints being free, and if so add it to the matching.
4. Compute the graph H (all free vertices collapsed into a single vertex).
5. Compute an alternating BFS tree in H.
6. Extract shortest alternating cycle based in the root (by finding the highest bridge).
7. If alternating cycle corresponds to an alternating path in **G** then apply and continue.

The algorithm...

1. Start from empty matching M in graph **G**.
2. Now, repeatedly, try to enlarge the matching.
3. First, check if you can find an edge with both endpoints being free, and if so add it to the matching.
4. Compute the graph H (all free vertices collapsed into a single vertex).
5. Compute an alternating BFS tree in H.
6. Extract shortest alternating cycle based in the root (by finding the highest bridge).
7. If alternating cycle corresponds to an alternating path in **G** then apply and continue.

The algorithm...

1. Start from empty matching M in graph **G**.
2. Now, repeatedly, try to enlarge the matching.
3. First, check if you can find an edge with both endpoints being free, and if so add it to the matching.
4. Compute the graph H (all free vertices collapsed into a single vertex).
5. Compute an alternating BFS tree in H.
6. Extract shortest alternating cycle based in the root (by finding the highest bridge).
7. If alternating cycle corresponds to an alternating path in **G** then apply and continue.

The algorithm...

1. Start from empty matching M in graph **G**.
2. Now, repeatedly, try to enlarge the matching.
3. First, check if you can find an edge with both endpoints being free, and if so add it to the matching.
4. Compute the graph H (all free vertices collapsed into a single vertex).
5. Compute an alternating BFS tree in H.
6. Extract shortest alternating cycle based in the root (by finding the highest bridge).
7. If alternating cycle corresponds to an alternating path in **G** then apply and continue.

How to handle a flower...

1. If found a flower, with a stem ρ and a blossom C then:
   1. apply the stem to M (i.e., $M \oplus \rho$).
   2. C: odd cycle with the free vertex being unmatched.
   3. Compute recursively an augmenting path π in G/C.
   4. Transform this into an augmenting path in **G** – apply it.
2. succeeded computing a matching with one edge more in it. Continue till stuck.

How to handle a flower...

1. If found a flower, with a stem ρ and a blossom C then:
    1. apply the stem to M (i.e., $M \oplus \rho$).
    2. C: odd cycle with the free vertex being unmatched.
    3. Compute recursively an augmenting path π in G/C.
    4. Transform this into an augmenting path in **G** – apply it.
2. succeeded computing a matching with one edge more in it. Continue till stuck.

How to handle a flower...

1. If found a flower, with a stem ρ and a blossom C then:
   1. apply the stem to M (i.e., $M \oplus \rho$).
   2. C: odd cycle with the free vertex being unmatched.
   3. Compute recursively an augmenting path π in G/C.
   4. Transform this into an augmenting path in **G** – apply it.
2. succeeded computing a matching with one edge more in it. Continue till stuck.

How to handle a flower...

1. If found a flower, with a stem ρ and a blossom C then:
   1. apply the stem to M (i.e., $M \oplus \rho$).
   2. C: odd cycle with the free vertex being unmatched.
   3. Compute recursively an augmenting path π in G/C.
   4. Transform this into an augmenting path in **G** – apply it.
2. succeeded computing a matching with one edge more in it. Continue till stuck.

17.3.2.1: Running time analysis

Running time...

1. Every shrink cost us $O(m + n)$ time.
2. Need to perform $O(n)$ recursive shrink operations till find an augmenting path, if such a path exists.
3. Computing an augmenting path takes $O(n(m + n))$ time.
4. Have to repeat this $O(n)$ times.
5. Overall running time is $O(n^2(m + n)) = O(n^4)$.

Running time...

1. Every shrink cost us $O(m + n)$ time.
2. Need to perform $O(n)$ recursive shrink operations till find an augmenting path, if such a path exists.
3. Computing an augmenting path takes $O(n(m + n))$ time.
4. Have to repeat this $O(n)$ times.
5. Overall running time is $O(n^2(m + n)) = O(n^4)$.

Running time...

1. Every shrink cost us $O(m+n)$ time.
2. Need to perform $O(n)$ recursive shrink operations till find an augmenting path, if such a path exists.
3. Computing an augmenting path takes $O(n(m+n))$ time.
4. Have to repeat this $O(n)$ times.
5. Overall running time is $O(n^2(m+n)) = O(n^4)$.

Running time...

1. Every shrink cost us $O(m+n)$ time.
2. Need to perform $O(n)$ recursive shrink operations till find an augmenting path, if such a path exists.
3. Computing an augmenting path takes $O(n(m+n))$ time.
4. Have to repeat this $O(n)$ times.
5. Overall running time is $O(n^2(m+n)) = O(n^4)$.

The result

Theorem

Given a graph $\mathbf{G}$ *with* n *vertices and* m *edges, computing a maximum size matching in* $\mathbf{G}$ *can be done in* $O(n^2 m)$ *time.*

17.3.2.2: Maximum Weight Matching in A Non-Bipartite Graph

Maximum Weight Matching in A Non-Bipartite Graph

his the hardest case and it is non-trivial to handle. See internet/literature for details.

Notes

Notes

Notes

Notes

J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973.