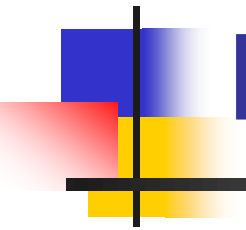


Iterative Improvement for Domain-Specific Problems

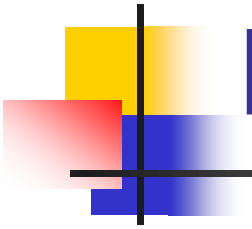
Lecturer: Jing Liu

Email: neouma@mail.xidian.edu.cn

Homepage: http://see.xidian.edu.cn/faculty/liujing

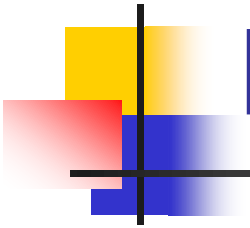
Iterative Improvement for Domain-Specific Problems

- In this part, we will study an algorithm design technique that we will refer to as iterative improvement.
- In its simplest form, this technique starts with a simple-minded (usually a greedy) solution and continues to improve on that solution in stages until an optimal solution is found.
  - *Finding a maximum flow in a network*
  - *Finding a maximum matching in undirected graphs*



Network Flow

- A network is a 4-tuple (G, s, t, c), where $G=(V, E)$ is a directed graph, s and t are two distinguished vertices called, respectively, the source and sink, and $c(u, v)$ is a capacity function defined on all pairs of vertices with $c(u, v)>0$ if $(u, v)\in E$ and $c(u, v)=0$ otherwise. $|V|=n$, $|E|=m$.



Network Flow

A ***flow*** in G is a real-valued function f on vertex pairs having the following four conditions:

(1) **Skew symmetry**. $\forall u, v \in V, f(u, v) = -f(v, u)$. We say there is a flow from u to v if $f(u, v) > 0$.

(2) **Capacity constraints**. $\forall u, v \in V, f(u, v) \leq c(u, v)$. We say edge (u, v) is saturated if $f(u, v) = c(u, v)$.

(3) **Flow conservation**. $\forall u \in V - \{s, t\}, \sum_{v \in V} f(u, v) = 0$. In other words, the *net flow* (total flow out minus total flow in) at any interior vertex is 0.

(4) $\forall v \in V, f(v, v) = 0$.

Network Flow

- A **cut** $\{S, T\}$ is a partition of the vertex set V into two subsets S and T such that $s \in S$ and $t \in T$. The capacity of the cut $\{S, T\}$, denoted by $c(S, T)$, is

$$c(S,T) = \sum_{u \in S, v \in T} c(u,v)$$

- The flow across the cut $\{S, T\}$, denoted by $f(S, T)$, is

$$f(S,T) = \sum_{u \in S, v \in T} f(u,v)$$

- Thus, the flow across the cut $\{S, T\}$ is the sum of the positive flow on edges from S to T minus the sum of the positive flow on edges from T to S.

Network Flow

- For any vertex u and any subset $A \subseteq V$, let $f(u, A)$ denote $f(\{u\}, A)$, and $f(A, u)$ denote $f(A, \{u\})$. For a capacity function c, $c(u, A)$ and $c(A, u)$ are defined similar.
- The ***value of a flow f***, denoted by $|f|$, is defined to be

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

- **Lemma:** For any cut $\{S, T\}$ and a flow f, $|f| = f(S, T)$.

- ***Max-Flow Problems****: Design a function f for the network* (G, s, t, c), so that $|f|$ is the maximum.

Network Flow

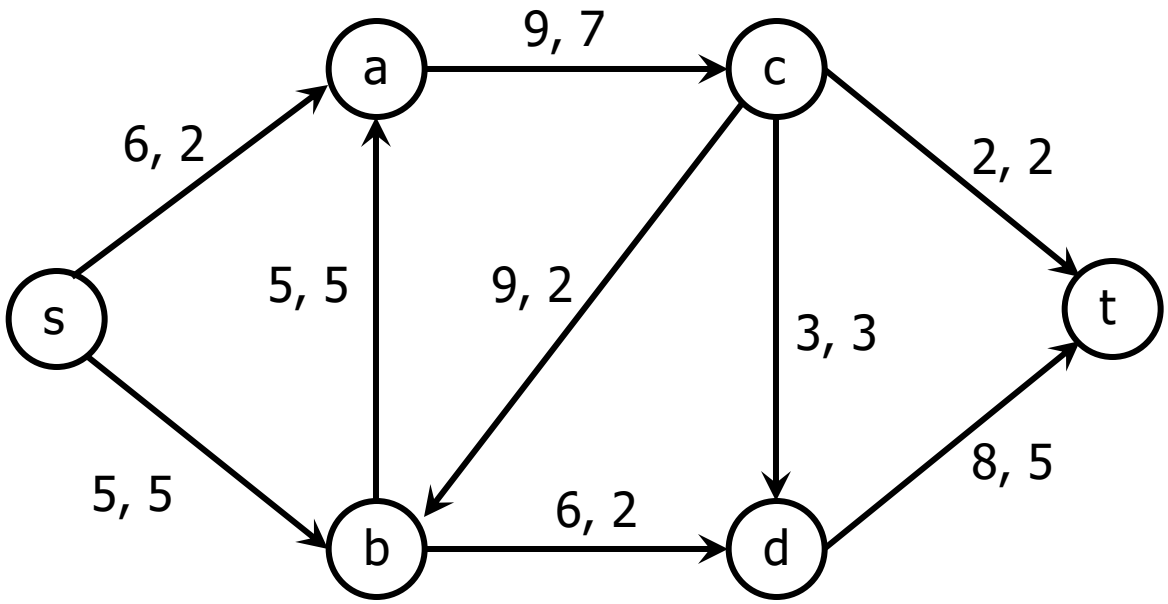
- Given a flow f on G with capacity function c, the ***residual capacity function*** for f on the set of pairs of vertices is defined as follows.
- For each pair of vertices, $u, v \in V$, the ***residual capacity*** $r(u, v)=c(u, v)-f(u, v)$. The ***residual graph*** for the flow f is the directed graph $R=(V, E_f)$, with capacities defined by r and

$$E_f=\{(u, v)|r(u, v)>0\}$$

- The residual capacity $r(u, v)$ represents the amount of additional flow that can be pushed along the edge (u, v) without violating the capacity constraints.
- If $f(u, v)<c(u, v)$, then both (u, v) and (v, u) are present in R. If there is no edge between u and v in G, then neither (u, v) nor (v, u) are in E_f. Thus, $|E_f|\leq 2|E|$.

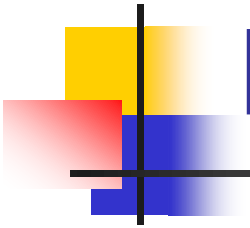
Network Flow

- **Example**: what's the residual graph of the following graph?



(x, y): x is the capacity, y is the flow

Network Flow



- Let f and f' be any two flows in a network G. Define the function $f+f'$ by $(f+f')(u, v)=f(u, v)+f'(u, v)$ for all pairs of vertices u and v. Similarly, define the function $f-f'$ by $(f-f')(u, v)=f(u, v)-f'(u, v)$.
- **Lemma**: Let f be a flow in G and f' the flow in the residual graph R for f. Then the function $f+f'$ is a flow in G of value $|f|+|f'|$.
- **Lemma**: Let f be any flow in G and f^* a maximum flow in G. If R is the residual graph for f, then the value of a maximum flow in R is $|f^*|-|f|$.

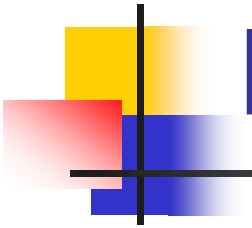
Network Flow

- Given a flow f in G, an ***augmenting path*** p is a directed path from s to t in the residual graph R. The ***bottleneck capacity*** of p is the minimum residual capacity along p. The number of edges in p will be denoted by $|p|$.
- **Theorem (max-flow min-cut theorem)**: Let (G, s, t, c) be a network and f a flow in G. The following three statements are equivalent:

(a) There is a cut $\{S, T\}$ with $c(S, T)=|f|$.

(b) f is a maximum flow in G.

(c) There is no augmenting path for f.



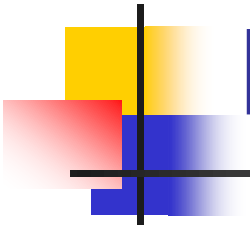
Network Flow

The Ford-Fulkerson Method:

- The previous theorem suggests a way to construct a maximum flow by iterative improvement
- One keeps finding an augmenting path arbitrarily and increases the flow by its bottleneck capacity

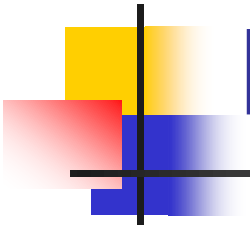
Network Flow

- **Input**: A network (G, s, t, c);
- **Output**: A flow in G;
- 1. Initialize the residual graph: $R \leftarrow G$;
- 2. for each edge $(u, v) \in E$
- 3. $f(u, v) \leftarrow 0$;
- 4. end for;
- 5. while there is an augmenting path $p = s, \ldots, t$ in R
- 6. Let Δ be the bottleneck capacity of p;
- 7. for each edge (u, v) in p
- 8. $f(u, v) \leftarrow f(u, v) + \Delta$;
- 9. end for;
- 10. Update the residual graph R;
- 11. end while;



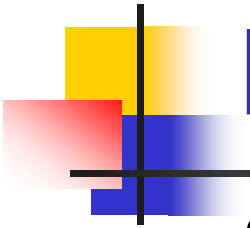
Network Flow

- What's the time complexity of the Ford-Fulkerson Method?



Network Flow

- The time complexity of the Ford-Fulkerson Method is $O(m \cdot |f^*|)$, where m is the number of edges and $|f^*|$ is the value of the maximum flow, which is an integer.



Network Flow

Shortest Path Augmentation:

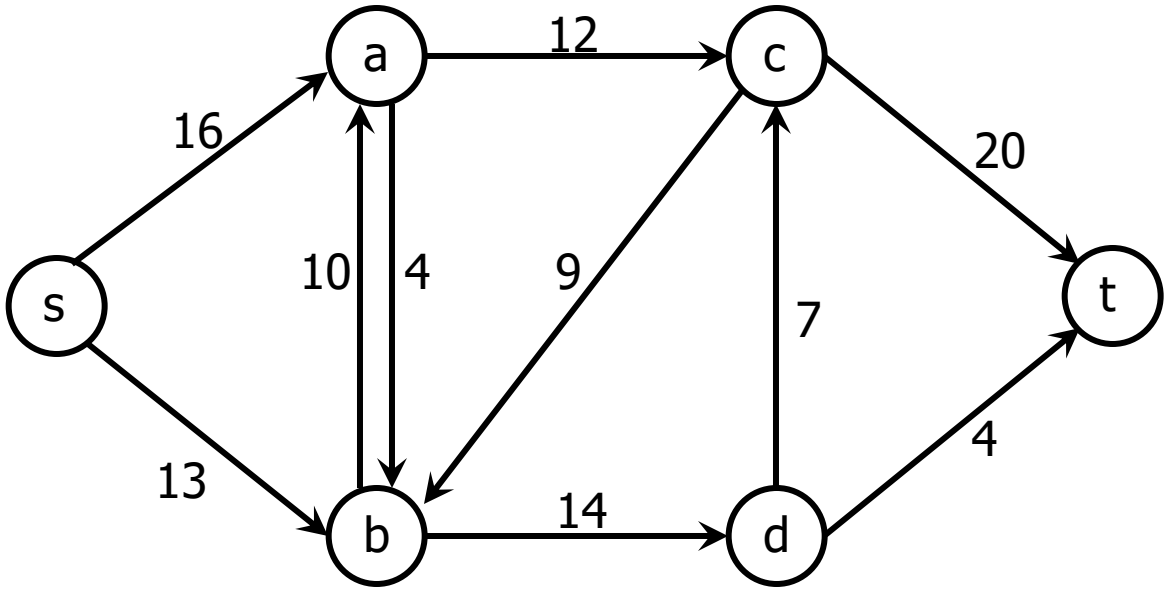
- Here we consider another heuristic that puts some order on the selection of augmenting paths.
- **Definition**: The ***level*** of a vertex v, denoted by $level(v)$, is the least number of edges in a path from s to v. Given a directed graph $G=(V, E)$, the level graph L is (V', E'), where V' is the set of vertices can be reached from s and

  $$E'=\{(u, v) \mid level(v)=level(u)+1 \text{ and } (u, v)\in E\}.$$

- Given a directed graph G and a source vertex s, its level graph L can easily be constructed using breadth-first search.

Network Flow

- **Example**: what's the level graph of the following graph?



Network Flow

Minimum path length augmentation (MPLA)

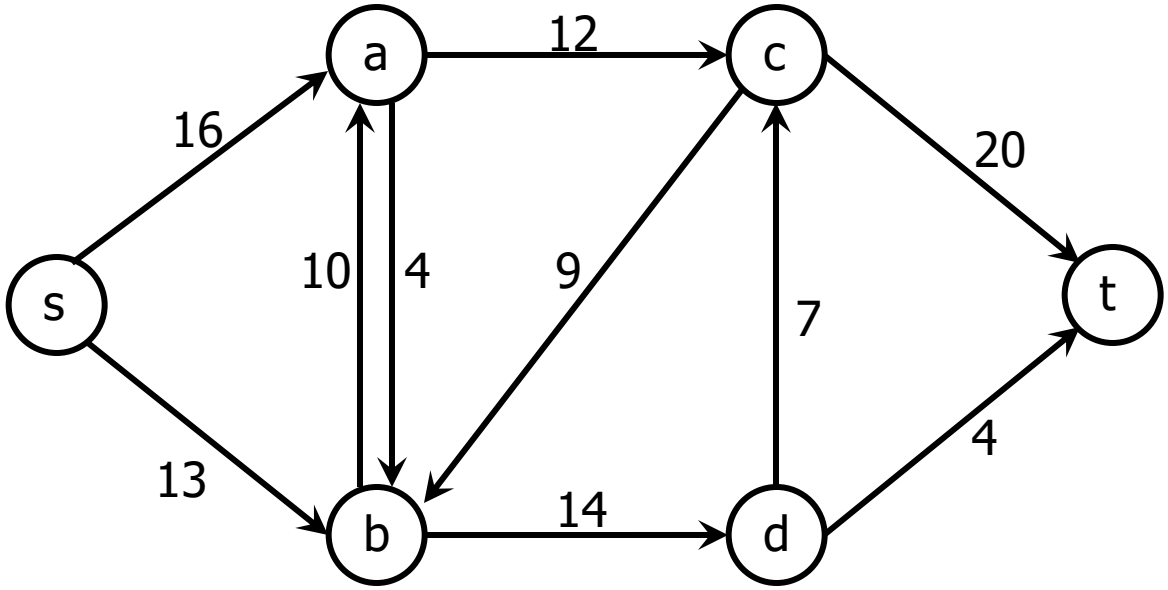
- MPLA selects an augmenting path of minimum length and increases the current flow by an amount equal to the bottleneck capacity of the path. The algorithm starts by initializing the flow to the zero flow and setting the residual graph *R* to the original network. It then proceeds in phases. Each phase consists of the following two steps:
- (1) Compute the level graph *L* from the residual graph *R*. If *t* is not in *L*, then halt; otherwise continue.
- (2) As long as there is a path *p* from *s* to *t* in *L*, augment the current flow by *p*, remove saturated edges from *L* and *R* and update them accordingly.

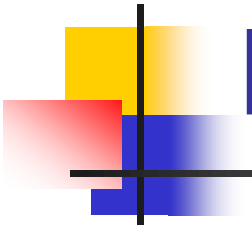
Network Flow

- **Input**: A network (G, s, t, c);
- **Output**: The maximum flow in G;
- 1. for each edge $(u, v) \in E$
- 2. $f(u, v) \leftarrow 0$;
- 3. end for;
- 4. Initialize the residual graph: $R \leftarrow G$;
- 5. Find the level graph L of R;
- 6. while t is a vertex in L
- 7. while t is reachable from s in L
- 8. Let p be a path from s to t in L;
- 9. Let Δ be the bottleneck capacity on p;
- 10. Augment the current flow f by Δ;
- 11. Update L and R along the path p;
- 12. end while;
- 13. Use the residual graph R to compute a new level graph L;
- 14. end while;

Network Flow

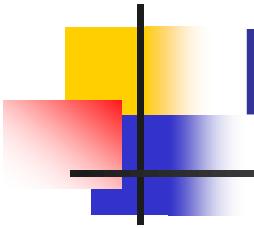
- **Example**: Use the minimum path length augmentation algorithm to calculate the maximum flow of the following graph?





Matching

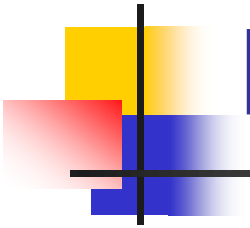
- Given an undirected graph $G=(V, E)$, $|V|=n$, and $|E|=m$. A **matching** in G is a subset $M \subseteq E$ such that no two edges in M have a vertex in common.
- An edge $e \in E$ is **matched** if it is in M, and **unmatched** or **free** otherwise.
- A vertex $v \in E$ is **matched** if it is incident to a matched edge, and **unmatched** or **free** otherwise.
- The size of a matching M, i.e. the number of matching edges in it, will be denoted by $|M|$.
- A **maximum matching** in a graph is a matching of maximum cardinality. A **perfect matching** is one in which every vertex in V is matched.



Matching

- The **maximum matching problem** asks for a subset $M \subseteq E$ with the maximum number of nonoverlapping edges; that is, no two edges in M have a vertex in common.

- This problem arises in many applications, particularly in the areas of communication and scheduling.



Matching

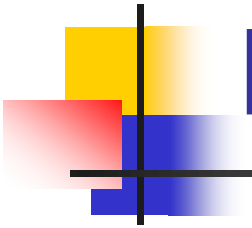
- Given a matching M in an undirected graph $G=(V, E)$, an **alternating path** p with respect to M is a simple path that consists of alternating matched and unmatched edges. The length of p is denoted by $|p|$.
- If the two endpoints of an alternating path coincide, then it is called an **alternating cycle**.
- An alternating path with respect to M is called an **augmenting path** with respect to M if all the matched edges in p are in M and its endpoints are free.

Matching

- Let M_1 and M_2 be two matchings in a graph G. Then

$$M_1 \oplus M_2 = (M_1 \cup M_2) - (M_1 \cap M_2)$$
$$= (M_1 - M_2) \cup (M_2 - M_1)$$

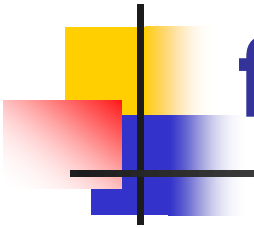
- That is, $M_1 \oplus M_2$ is the set of edges that are in M_1 or in M_2 but not in both.
- **Lemma**: Let M be a matching and p an augmenting path with respect to M, then $M \oplus p$ is a matching of size $|M \oplus p| = |M| + 1$.
- **Corollary**: A matching M in an undirected graph G is maximum if and only if G contains no augmenting paths with respect to M.



Matching

- **Theorem**: Let M_1 and M_2 be two matchings in an undirected G such that $|M_1|=r$, $|M_2|=s$ and $s>r$. Then, $M_1 \oplus M_2$ contains at least $k=s-r$ vertex-disjoint augmenting paths with respect to M_1.

The Hungarian Tree Method for Bipartite Graphs

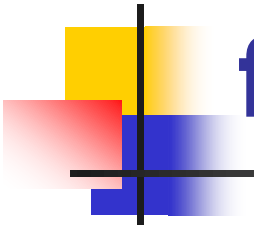


- The previous Lemma and Corollary suggest a procedure for finding a maximum matching in *G*:

  Starting from an arbitrary (e.g. empty) matching, we find an augmenting path p in G, invert the roles of the edges in p (matched to unmatched and vice-versa), and repeat the process until there are no more augmenting paths.

- However, to find an augmenting path efficiently in a general graph is not easy.

The Hungarian Tree Method for Bipartite Graphs

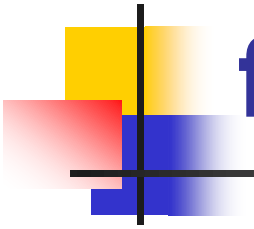


- Given an undirected graph $G=(V, E)$, if V can be divided into two disjoint subsets X and Y so that each edge in E has an end in X and an end in Y, then G is a **bipartite graph**.

- The most important feature of a bipartite graph is it contains no cycles of odd length.

- Finding an augmenting path in the case of bipartite graphs is much easier than in the case of general graphs.

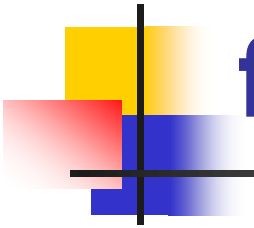
The Hungarian Tree Method for Bipartite Graphs



- Let $G=(X \cup Y, E)$ be a bipartite graph with $|X|+|Y|=n$ and $|E|=m$. Let M be a matching in G. We call a vertex in X an x-vertex. Similarly, a y-vertex denotes a vertex in Y.

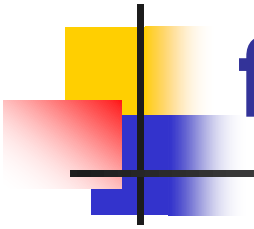
- First, we pick a free x-vertex, say r, and label it **outer**. From r, we grow an ***alternating path tree***, i.e., a tree in which each path from the root r to a leaf is an alternating path. This tree, call it T, is constructed as follows:

The Hungarian Tree Method for Bipartite Graphs



- Starting from r, add each unmatched edge (r, y) connecting r to the y-vertex y and label y inner.
- For each y-vertex y adjacent to r, add the matched edge (y, z) to T if such a matched edge exists, and label z outer.
- Repeat the above procedure and extend the tree until either a free y-vertex is encountered or the tree is blocked, i.e., cannot be extended any more (note that no vertex is added to the tree more than once).

The Hungarian Tree Method for Bipartite Graphs

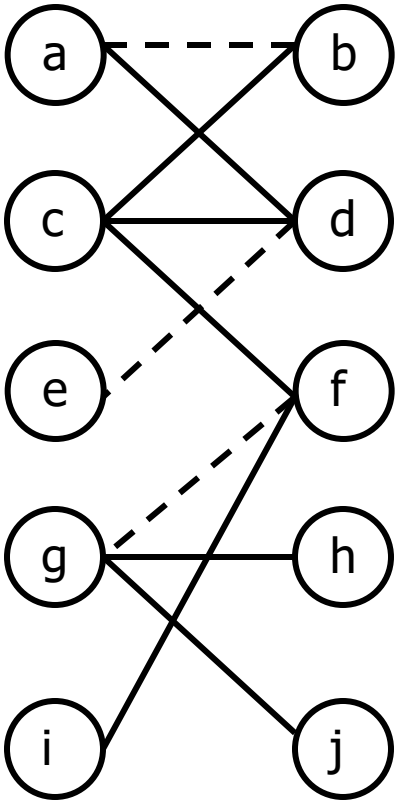


- If a free *y*-vertex is found, say *v*, then the alternating path from the root *r* to *v* is an augmenting path. On the other hand, if the tree is blocked, then in this case the tree is called a **Hungarian tree**.

- Next, we start from another free *x*-vertex, if any, and repeat the above procedure.

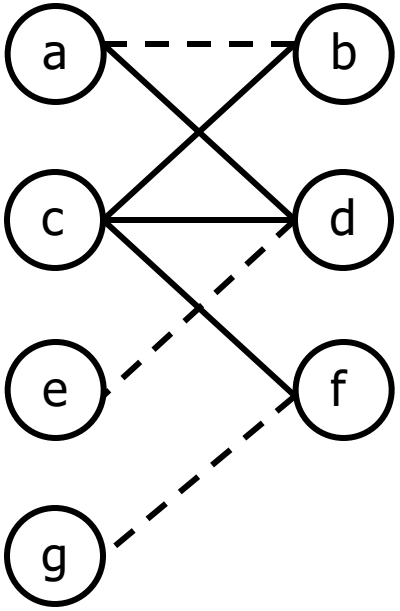
The Hungarian Tree Method for Bipartite Graphs

- **Example**: The dashed lines represent a matching.



The Hungarian Tree Method for Bipartite Graphs

- **Example**: The dashed lines represent a matching.



The Hungarian Tree Method for Bipartite Graphs

Observation

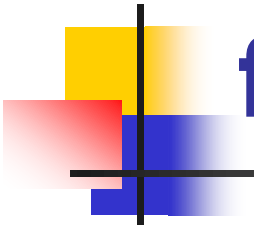
- If T is a Hungarian tree, then it cannot be extended; each alternating path traced from the root is stopped at some outer vertex.
  - The only free vertex in T is its root. Notice that if (x, y) is an edge such that x is in T and y is not in T, then x must be labeled inner. Otherwise, x must be connected to a free vertex or T is extendable through x.

The Hungarian Tree Method for Bipartite Graphs

Observation

- It follows that no vertex in a Hungarian tree can occur in an augmenting path.
  - Suppose that p is an alternating path that shares at least one vertex with T. If p "enters" T, then it must be through a vertex labeled inner. If it "leaves" T, then it must also be through a vertex labeled as inner. But, then, p is not an alternating path; a contradiction.

The Hungarian Tree Method for Bipartite Graphs



- Therefore, if, in the process of searching for an augmenting path, a Hungarian tree is found, then it can be removed permanently without affecting the search.

The Hungarian Tree Method for Bipartite Graphs

Input: A bipartite graph $G=(X \cup Y, E)$;
Output: A maximum matching M in G;

1. Initialize M to any arbitrary (possibly empty) matching;
2. ***while*** (there exists a free x-vertex and a free y vertex)
3. Let r be a free x-vertex, and using breadth-first search, grow an alternating path tree T rooted at r;
4. if (T is a Hungarian tree) then let $G \leftarrow G-T$;
5. else (find an augmenting path p in T and let $M=M \oplus p$);
6. ***end while.***

The Hungarian Tree Method for Bipartite Graphs

- **Example**: