COMP202
Complexity of Algorithms

The Maximum Flow Problem, Bipartite Matchings

Reading material: Chapter 26 in CLRS.

Learning outcomes

By the end of this set of lecture notes, you should

1. Understand the *maximum flow problem*.
2. Comprehend and be able to utilize the Ford-Fulkerson augmenting path algorithm that can be used to find maximum flows in networks.
3. Know the *Max-Flow/Min-Cut Theorem*.
4. Know how to find *Maximum Matchings* in bipartite graphs.

Digraphs – reminder

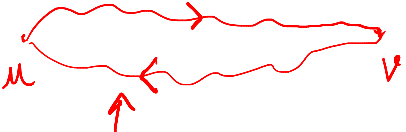
(u, v) : $u \longrightarrow v$

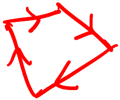
A *digraph* is a graph whose edges are all directed.

A fundamental issue with directed graphs is the notion of *reachability*, which deals with determining where we can get to in a directed graph.

Given two vertices u and v of a digraph G, we say that u reaches v (or v is reachable from u) if G has a *directed path* from u to v.

Digraphs (cont.)





A digraph G is *strongly connected* if, for any two distinct vertices u and v, we have that u reaches v, and v reaches u.

A *directed cycle* of G is a cycle where all the edges are traversed according to their respective directions.

A digraph is *acyclic* if it has no directed cycles.

Weighted Graphs

$u \xrightarrow{w_e} v$

A *weighted graph* is a graph that has a numerical label $w(e)$ associated with each edge e, called the *weight* of e.

Alternatively, we might sometimes consider graphs having weights on the *vertices*, or on both the vertices and edges.

X (FOR TODAY)

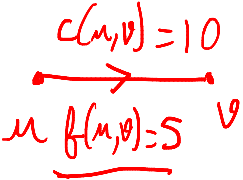
Network Flow - The basics

A *flow network* $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a *non-negative integer capacity* $c(u, v) \geq 0$.

We distinguish two vertices in the flow network: a *source* s and a *sink* t.

We assume that s has no in-edges, and that t has no out-edges.

Capacity constraint: Each edge (u, v) also has an associated flow value $f(u, v)$ which tells us how much flow has been sent along an edge. These values satisfy $0 \leq f(u, v) \leq c(u, v)$.

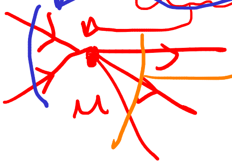


Network Flow - The basics

Capacity constraint: Each edge (u, v) also has an associated flow value $f(u, v)$ which tells us how much flow has been sent along an edge. These values satisfy $0 \leq f(u, v) \leq c(u, v)$.

Flow conservation: For every vertex other than s and t, the amount of flow *into* the vertex must equal the amount of flow *out* of the vertex:

$$\forall u \in V \setminus \{s, t\}, \quad \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$



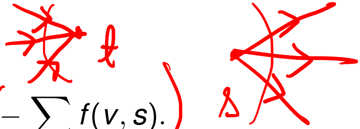
Capacity constraint: Each edge (u, v) also has an associated flow value $f(u, v)$ which tells us how much flow has been sent along an edge. These values satisfy $0 \leq f(u, v) \leq c(u, v)$.

Flow conservation: For every vertex other than s and t, the amount of flow *into* the vertex must equal the amount of flow *out* of the vertex:

$$\forall u \in V \setminus \{s, t\}, \qquad \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$

In the *maximum flow problem*, we are given a flow network G, with source, s and sink, t and we wish to find a flow of *maximum* value from s to t.

Value of flow:



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

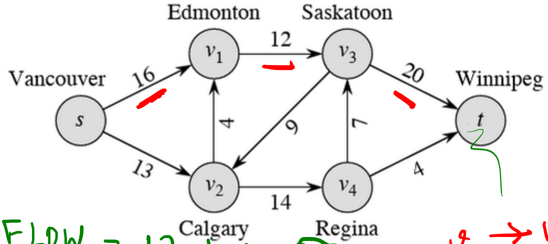
Flows in practice

Network flows can model many problems:

- Liquids flowing through pipes,
- parts through assembly lines,
- current through electrical networks,
- information through communication networks, etc.

Network Flow - Example of Max Flow

Input:



VAL OF FLOW = 12 + 11 = 23

$v_2 \to v_1$ 4

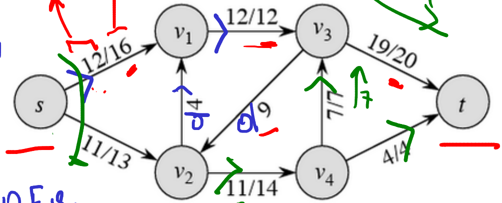
Output:

FLOW CAPACITY

19

= 0/4

FLOW INTO v_1
=
12+0=12
=
FLOW OUT OF v_1.



There are several algorithms for solving the max flow problem.

One method we will look at is the *Ford-Fulkerson method.*

There are several algorithms for solving the max flow problem.

One method we will look at is the *Ford-Fulkerson method.*

This algorithm searches for a *flow-augmenting path* from the *source* vertex s to the *sink* vertex t.

Network Flow Algorithms

There are several algorithms for solving the max flow problem.

One method we will look at is the *Ford-Fulkerson method.*

This algorithm searches for a *flow-augmenting path* from the *source* vertex s to the *sink* vertex t.

We then send as much flow as possible along the flow augmenting path, whilst obeying the *capacity constraints* of each edge.

Network Flow Algorithms

There are several algorithms for solving the max flow problem.

One method we will look at is the *Ford-Fulkerson method.*

This algorithm searches for a *flow-augmenting path* from the *source* vertex s to the *sink* vertex t.

We then send as much flow as possible along the flow augmenting path, whilst obeying the *capacity constraints* of each edge.

The maximum flow that we can send along the path is limited by the *minimum* of $c(u, v) - f(u, v)$ of an edge on this path.

Network Flows: Ford-Fulkerson Method

The Ford-Fulkerson method depends on three important ideas

- *residual networks*
- *augmenting paths*
- *cuts*

These three ideas are essential for the important *Max-Flow/Min-Cut* theorem.

Ford-Fulkerson Method

The Ford-Fulkerson method is *iterative*.

- Start with $f(u, v) = 0$ for all $u, v \in V$.

Ford-Fulkerson Method

The Ford-Fulkerson method is *iterative*.

- Start with $f(u, v) = 0$ for all $u, v \in V$.
- At each iteration, we increase flow by finding an *augmenting path* from s to t along which we can *push* more flow. (We consider the *residual network* when we search for augmenting paths.)

Ford-Fulkerson Method

The Ford-Fulkerson method is *iterative*.

- Start with $f(u, v) = 0$ for all $u, v \in V$.
- At each iteration, we increase flow by finding an *augmenting path* from s to t along which we can *push* more flow. (We consider the *residual network* when we search for augmenting paths.)
- This process is repeated until no more augmenting paths can be found.

Ford-Fulkerson Method

The Ford-Fulkerson method is *iterative*.

- Start with $f(u, v) = 0$ for all $u, v \in V$.
- At each iteration, we increase flow by finding an *augmenting path* from s to t along which we can *push* more flow. (We consider the *residual network* when we search for augmenting paths.)
- This process is repeated until no more augmenting paths can be found.
- The *Max-Flow Min-cut theorem* shows us that this process yields a maximum flow.

Ford-Fulkerson - Algorithm

Ford-Fulkerson-Method(G, s, t)

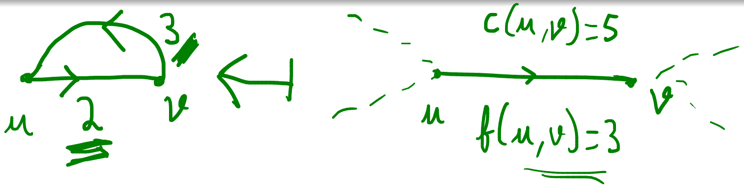
```
▷ Input: A flow network G and source s, sink t
▷ Output: Max flow through network
f ← 0
while augmenting path, P exists (in the residual network)
      do augment flow along P
         f ← f + new flow
         update residual network
return f
```

$f(u,v) = 0 \quad \forall$ EDGES (u,v)

Residual Networks

A *residual network* consists of **edges that can admit more net flow.**

Residual Networks



A *residual network* consists of **edges that can admit more net flow.**

Given the flow network G, and a flow f in that network, we define the residual network G_f. (So this depends upon the given flow f.)

Residual Networks (cont.)

The edges of G_f are of two types:

Residual Networks (cont.)

The edges of G_f are of two types:

- "Forwards edges"

  For any edge (u, v) in G for which $f(u, v) < c(u, v)$, there is an edge (u, v) in G_f.

Residual Networks (cont.)

The edges of G_f are of two types:

- “Forwards edges”

  For any edge (u, v) in G for which $f(u, v) < c(u, v)$, there is an edge (u, v) in G_f.

  The *residual capacity* $\Delta_f(u, v)$ of (u, v) in G_f is defined as $\Delta_f(u, v) = c(u, v) - f(u, v)$.

Residual Networks (cont.)

The edges of G_f are of two types:

- "Forwards edges"

  For any edge (u, v) in G for which $f(u, v) < c(u, v)$, there is an edge (u, v) in G_f.

  The *residual capacity* $\Delta_f(u, v)$ of (u, v) in G_f is defined as $\Delta_f(u, v) = c(u, v) - f(u, v)$.

- "Backwards edges"

  For any edge (u, v) in G for which $f(u, v) > 0$, there is an edge (v, u) in G_f.

Residual Networks (cont.)

The edges of G_f are of two types:

- "Forwards edges"

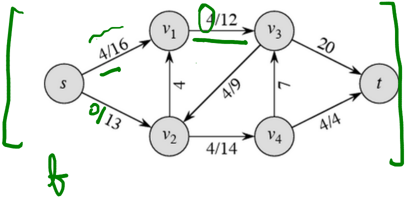
  For any edge (u, v) in G for which $f(u, v) < c(u, v)$, there is an edge (u, v) in G_f.

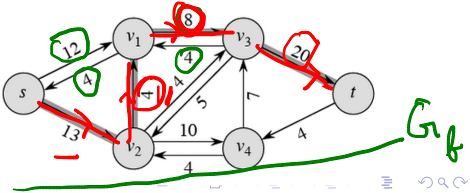
  The *residual capacity* $\Delta_f(u, v)$ of (u, v) in G_f is defined as $\Delta_f(u, v) = c(u, v) - f(u, v)$.

- "Backwards edges"

  For any edge (u, v) in G for which $f(u, v) > 0$, there is an edge (v, u) in G_f.

  The *residual capacity* $\Delta_f(v, u)$ of (v, u) in G_f is defined as $\Delta_f(v, u) = f(u, v)$.





Augmenting Paths

Given a flow network $G = (V, E)$ and a flow f, an *augmenting path* P is a (directed) path from s to t in the residual network G_f.

Augmenting Paths

Given a flow network $G = (V, E)$ and a flow f, an *augmenting path* P is a (directed) path from s to t in the residual network G_f.

Informally, an augmenting path is a path from the source to the sink in which we can send more net flow, i.e. flow along each edge has not reached the capacity.

Augmenting Paths

Given a flow network $G = (V, E)$ and a flow f, an *augmenting path* P is a (directed) path from s to t in the residual network G_f.

Informally, an augmenting path is a path from the source to the sink in which we can send more net flow, i.e. flow along each edge has not reached the capacity.

The Ford-Fulkerson method sends flow along augmenting paths until no more flow augmenting paths exist.

You can use any path-finding method to find these augmenting paths (e.g., BFS, DFS). An example of an augmenting path is highlighted in the last figure of the previous frame.

Updating the flow

Once an augmenting path P has been identified, we need to update the flow. How is this done?

Updating the flow

Once an augmenting path P has been identified, we need to update the flow. How is this done?

First of all, the amount of flow to send along P is limited by the minimum residual capacity of the edges on P, i.e. define

$$\Delta_f(P) = \min_{(u,v)\in P} \Delta_f(u, v).$$

Updating the flow (cont.)

Then we send this flow along P. How do we interpret this, and update the values of $f(u,v)$ for the edges in the path P?

Updating the flow (cont.)

Then we send this flow along P. How do we interpret this, and update the values of $f(u, v)$ for the edges in the path P?

1. If (u, v) is a "forwards edge", we set

$$f'(u, v) = f(u, v) + \Delta_f(P).$$

Updating the flow (cont.)

Then we send this flow along P. How do we interpret this, and update the values of $f(u, v)$ for the edges in the path P?

1. If (u, v) is a "forwards edge", we set

$$f'(u, v) = f(u, v) + \Delta_f(P).$$

2. If (u, v) is a "backwards edge", we set

$$f'(v, u) = f(v, u) - \Delta_f(P).$$

   (In other words, we *decrease* the flow along the original edge (v, u) in G).

Updating the flow (cont.)

Then we send this flow along P. How do we interpret this, and update the values of $f(u, v)$ for the edges in the path P?

1. If (u, v) is a "forwards edge", we set

$$f'(u, v) = f(u, v) + \Delta_f(P).$$

2. If (u, v) is a "backwards edge", we set

$$f'(v, u) = f(v, u) - \Delta_f(P).$$

   (In other words, we *decrease* the flow along the original edge (v, u) in G).

3. For all edges e not in P, we set $f'(e) = f(e)$.

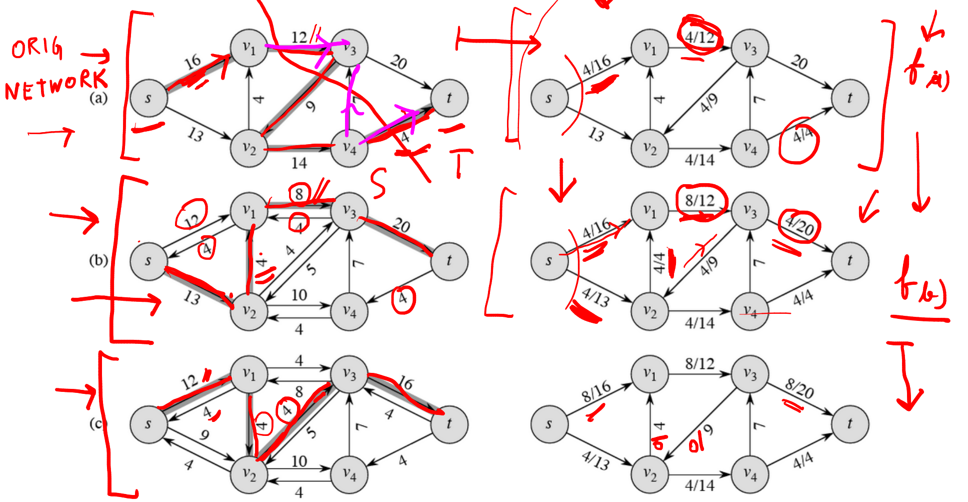
Updating the flow (cont.)

Then we send this flow along P. How do we interpret this, and update the values of $f(u, v)$ for the edges in the path P?

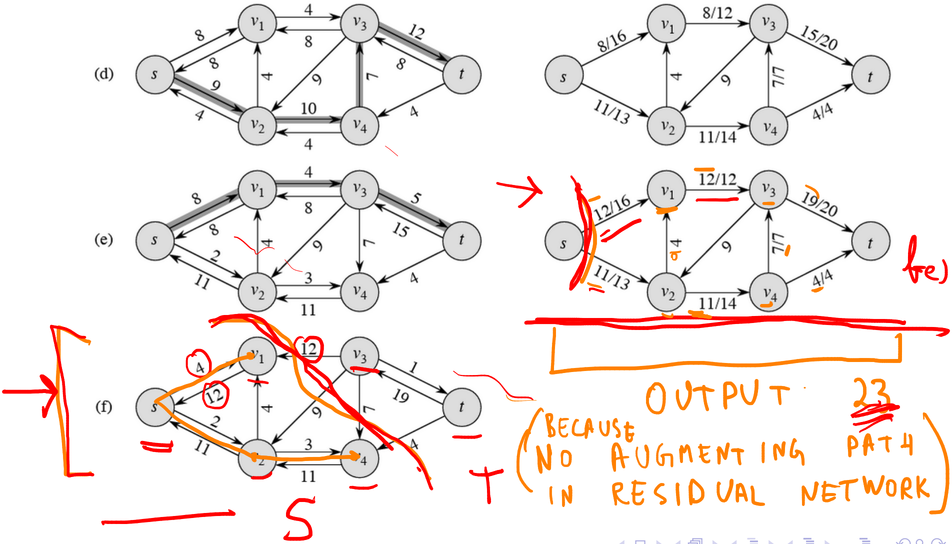
1. If (u, v) is a "forwards edge", we set
$$f'(u, v) = f(u, v) + \Delta_f(P).$$
2. If (u, v) is a "backwards edge", we set
$$f'(v, u) = f(v, u) - \Delta_f(P).$$
(In other words, we *decrease* the flow along the original edge (v, u) in G).
3. For all edges e not in P, we set $f'(e) = f(e)$.
4. Finally, we update the residual network to get the new one that corresponds to the new flow f'.

An example

4+12+7 = 23



An example (cont.)



OUTPUT : 23

(BECAUSE NO AUGMENTING PATH IN RESIDUAL NETWORK)

Cuts in Networks

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until a maximum flow has been found.

Cuts in Networks

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until a maximum flow has been found.

The *Max-Flow/Min-Cut Theorem* tells us that a flow is maximum if and only if no augmenting path exists.

Cuts in Networks

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until a maximum flow has been found.

The *Max-Flow/Min-Cut Theorem* tells us that a flow is maximum if and only if no augmenting path exists.

s , t

A *cut* (S, T) in a flow network $G = (V, E)$ is a partition of the vertices V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

Cuts in Networks (cont.)

If f is a flow, then the *net flow* across the cut (S, T) is defined to be

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

Cuts in Networks (cont.)

If f is a flow, then the *net flow* across the cut (S, T) is defined to be

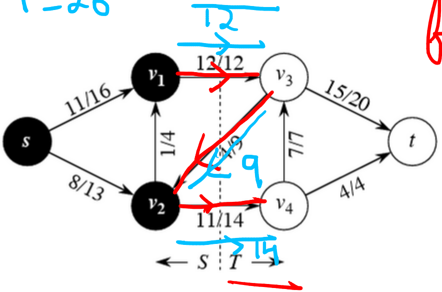
$$f(S,T) = \sum_{u \in S, v \in T} f(u,v) - \sum_{u \in T, v \in S} f(u,v).$$

The *capacity* of a cut (S, T) is

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

$c(S,T) = 12 + 14 = 26$

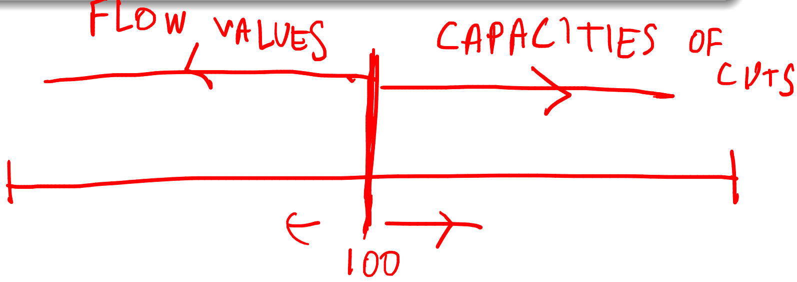
$f(S,T) = 12 + 11$
$- 4$
19



Max-Flow/Min-Cut Theorem

Theorem

The maximum flow ~~is~~ in a network is equal to capacity of a minimum cut in the network.



Ford-Fulkerson - Algorithm

Ford-Fulkerson(G, s, t)

▷ Input: A network G and vertices, s, t
▷ Output: A maximum flow
for each edge $(u, v) \in E(G)$
    do $f[u, v] \leftarrow 0$
Initialize residual network $G_f = G$
while there exists and augmenting path P from s to t
    do $\Delta_f(p) \leftarrow \min \{\Delta_f(u, v) : (u, v) \in P\}$
        for each edge $(u, v) \in P$
            Update the flow (forwards and backwards edges)
            Update the residual network based on new flow

Complexity of the Ford-Fulkerson algorithm

What is the running time of the Ford-Fulkerson algorithm?

Complexity of the Ford-Fulkerson algorithm

What is the running time of the Ford-Fulkerson algorithm?

Let $|f^*|$ denote the value of a maximum flow f^* in a network with n vertices and m edges.

Complexity of the Ford-Fulkerson algorithm

What is the running time of the Ford-Fulkerson algorithm?

Let $|f^*|$ denote the value of a maximum flow f^* in a network with n vertices and m edges.

Finding an augmenting path in the residual network can be done using a DFS or BFS algorithm. These run in time $O(n + m) = \underline{O(m)}$.

Complexity of the Ford-Fulkerson algorithm

What is the running time of the Ford-Fulkerson algorithm?

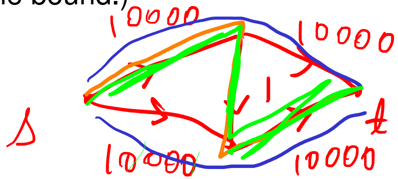
Let $|f^*|$ denote the value of a maximum flow f^* in a network with n vertices and m edges.

Finding an augmenting path in the residual network can be done using a DFS or BFS algorithm. These run in time $O(n+m) = O(m)$. $n = |V|$, $m \doteq |E|$

Each augmentation increases the flow by at least one unit (using the fact that the capacities are integers), so there are at most $|f^*|$ augmentation steps.

Complexity of the Ford-Fulkerson algorithm (cont.)

So the Ford-Fulkerson algorithm runs in time $O(|f^*|m)$.
(This isn't ideal, as a poor choice of augmenting paths can result in this large time bound.)



Complexity of the Ford-Fulkerson algorithm (cont.)

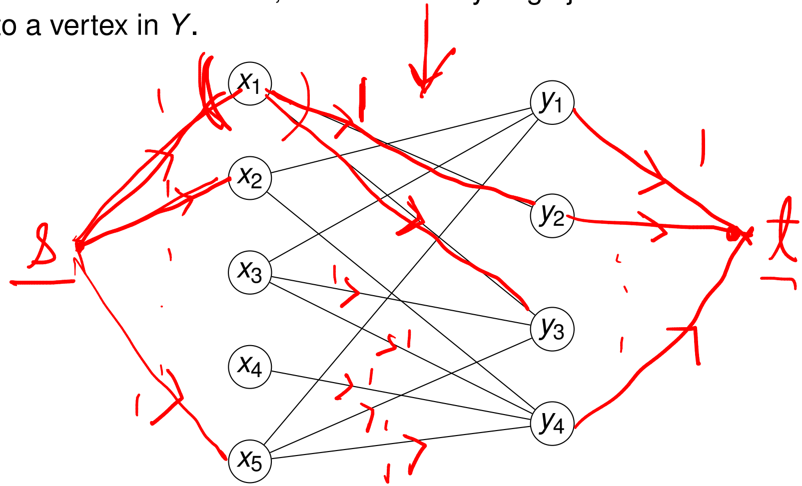
So the Ford-Fulkerson algorithm runs in time $O(|f^*|m)$. (This isn't ideal, as a poor choice of augmenting paths can result in this large time bound.)

Other algorithms exist (such as the *Edmonds-Karp algorithm*) that run in time that is asymptotically better than the Ford-Fulkerson algorithm when $|f^*|$ is very large.

Edmonds-Karp works by selecting *shortest* augmenting paths in the residual network (considering each edge to have length 1 when finding an augmenting path). This algorithm has a running time of $O(nm^2)$.

Bipartite graphs

A *bipartite graph* is a graph whose vertex set can be partitioned into two sets X and Y, such that every edge joins a vertex in X to a vertex in Y.



Bipartite graphs (cont.)

Bipartite graphs arise naturally is many situations when objects are being assigned to other objects.

For example, the set X could represent jobs and the set Y might represent machines. An edge (x_i, y_j) means that job x_i is capable of being assigned to machine y_j.

Bipartite graphs (cont.)

Bipartite graphs arise naturally is many situations when objects are being assigned to other objects.

For example, the set X could represent jobs and the set Y might represent machines. An edge (x_i, y_j) means that job x_i is capable of being assigned to machine y_j.

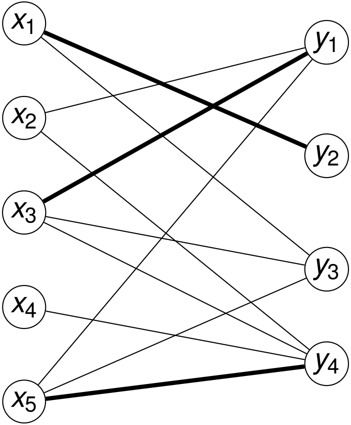
A bipartite graph could also represent relations between job applicants and available positions (i.e. people who are qualified for a particular job), customers and stores, houses and nearby police stations, etc, etc.

Matchings

A *matching* is a subset of the edges of a bipartite graph where each vertex appears in at most one edge (i.e. edges in the matching share no common endpoints).

Matchings

A *matching* is a subset of the edges of a bipartite graph where each vertex appears in at most one edge (i.e. edges in the matching share no common endpoints).



Matchings (cont.)

One of the oldest problems in combinatorial algorithms is that of determining the size of the largest *matching* in a bipartite graph.

Several algorithms have been developed for this task, as well as algorithms for graphs that are not bipartite (for which the problem is significantly more complicated).

Matchings (cont.)

One of the oldest problems in combinatorial algorithms is that of determining the size of the largest *matching* in a bipartite graph.

Several algorithms have been developed for this task, as well as algorithms for graphs that are not bipartite (for which the problem is significantly more complicated).

We can actually use an algorithm for the maximum flow problem to solve the problem of finding a matching of maximum size.

Finding a bipartite maximum matching

To use the augmenting path algorithm (or the Edmonds-Karp algorithm or some other maximum flow algorithm), we need to define our flow network.

Finding a bipartite maximum matching

To use the augmenting path algorithm (or the Edmonds-Karp algorithm or some other maximum flow algorithm), we need to define our flow network.

The flow network is obtained from the bipartite graph by adding two new vertices, a source vertex s and a sink vertex t.

Finding a bipartite maximum matching

To use the augmenting path algorithm (or the Edmonds-Karp algorithm or some other maximum flow algorithm), we need to define our flow network.

The flow network is obtained from the bipartite graph by adding two new vertices, a source vertex s and a sink vertex t.

Join all vertices in X to s and all vertices in Y to t. Direct all edges from s to X, from X to Y, and from Y to t.

Finding a bipartite maximum matching

To use the augmenting path algorithm (or the Edmonds-Karp algorithm or some other maximum flow algorithm), we need to define our flow network.

The flow network is obtained from the bipartite graph by adding two new vertices, a source vertex s and a sink vertex t.

Join all vertices in X to s and all vertices in Y to t. Direct all edges from s to X, from X to Y, and from Y to t.

Finally, give each edge a capacity of 1.

Finding a bipartite maximum matching (cont.)

Claim: The value of a maximum flow in the newly constructed flow network is equal to the size of a maximum matching in the original bipartite graph.

Finding a bipartite maximum matching (cont.)

Claim: The value of a maximum flow in the newly constructed flow network is equal to the size of a maximum matching in the original bipartite graph.

As a result, we can find a maximum matching (using, say, the Ford-Fulkerson augmenting path algorithm) in time $O(nm)$ (in this case the value of a maximum flow $|f^*|$ is $O(n)$).

$$O(\underbrace{|f^*|}_{\leq n} \cdot m) = O(nm).$$