

COMP202
Complexity of Algorithms
The Maximum Flow Problem
Bipartite Matchings

[See Chapter 8 in Goodrich and Tamassia.]

Learning outcomes

By the end of this set of lecture notes, a student 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*.

Connectivity information

Connectivity information can be defined by many kinds of relationships that exist between pairs of objects.

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the Internet, where the objects are computers.

Connectivity information

Connectivity information can be defined by many kinds of relationships that exist between pairs of objects.

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the Internet, where the objects are computers.

Connectivity information is also present in the parent-child relationships defined by a binary tree, where the objects are tree nodes.

Connectivity information

Connectivity information can be defined by many kinds of relationships that exist between pairs of objects.

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the Internet, where the objects are computers.

Connectivity information is also present in the parent-child relationships defined by a binary tree, where the objects are tree nodes.

Graphs are one way in which connectivity information can be stored, expressed, and utilized.

Graphs

A *graph* is a set of objects, called *vertices* (or *nodes*), together with a collection of pair-wise connections, called *edges*, between them.

Graphs

A *graph* is a set of objects, called *vertices* (or *nodes*), together with a collection of pair-wise connections, called *edges*, between them.

Graphs have applications in a number of different domains, including

- ▶ *mapping* (geographic information systems);
- ▶ *transportation* (road and flight networks);
- ▶ *electrical engineering* (circuit design);
- ▶ *process scheduling* (job makespans and assembly-line scheduling); and
- ▶ *computer networking* (connectivity of networks).

Graphs (cont.)

More formally, a graph $G = (V, E)$, is a set, V , of *vertices* and a collection, E , of pairs of vertices from V , called *edges*.

Graphs (cont.)

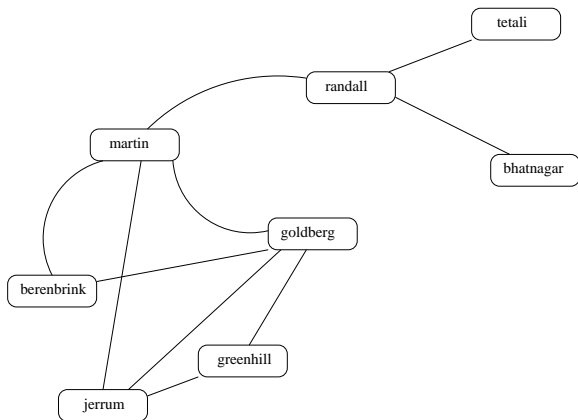
More formally, a graph $G = (V, E)$, is a set, V , of *vertices* and a collection, E , of pairs of vertices from V , called *edges*.

Edges in a graph are either *directed* or *undirected*.

- ▶ An edge (u, v) is said to be *directed* from u to v if the pair (u, v) is ordered. If all edges of a graph are directed, we usually refer to G as a *digraph*.
- ▶ An edge (u, v) is said to be *undirected* if the pair (u, v) is unordered. Typically, undirected edges are written as $\{u, v\}$ (using braces instead of parentheses).

Graph - Example

Graph of co-authorship



Graph Terminology

Two vertices are said to be *adjacent* if they are end-points of the same edge.

An edge is said to be *incident* to a vertex if the vertex is one of its end-points.

An *outgoing edge* of a vertex is a directed edge whose origin is that vertex.

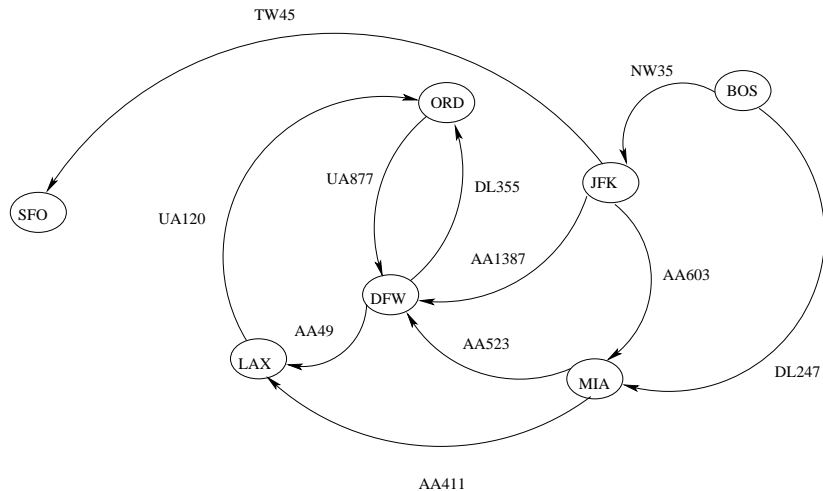
An *incoming edge* of a vertex is a directed edge whose destination is that vertex.

Graph terminology (cont.)

The *degree* of a vertex v , denoted $\text{deg}(v)$, is the number of edges incident to v .

In a directed graph, the *in-degree* (*out-degree*) of a vertex v is the number of *incoming* (*outgoing*) edges of v , and is denoted by $\text{indeg}(v)$ ($\text{outdeg}(v)$).

Digraph - Example



Graphs (cont.)

We have the following two elementary results about graphs.

Theorem: If G is an undirected graph with m edges then

$$\sum_{v \in V} \text{deg}(v) = 2m.$$

Theorem: If G is a directed graph with m edges, then

$$\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v) = m.$$

Graphs (cont.)

An undirected graph G is said to be *simple* if there is at most one edge between each pair of vertices u and v .

A digraph is *simple* if there is at most one directed edge from u to v for every pair of distinct vertices u and v .

Theorem: Let G be a *simple* graph with n vertices and m edges.

- ▶ If G is *undirected*, then $m \leq \frac{n(n-1)}{2}$.
- ▶ If G is *directed*, then $m \leq n(n-1)$.

More graph terminology

A *walk* in a graph is a sequence of alternating vertices and edges, starting at a vertex and ending at a vertex.

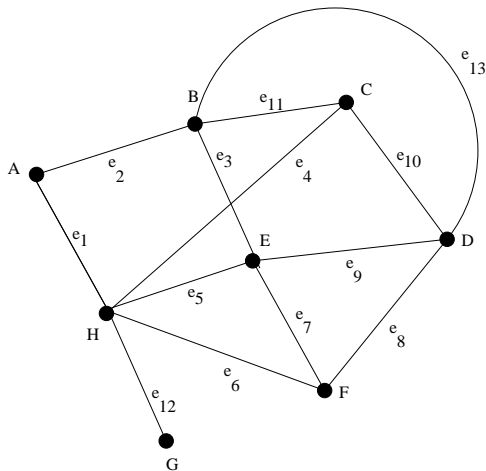
A *path* is a walk where each vertex in the walk is *distinct*.

A *circuit* is a walk with the same start and end vertex.

A *cycle* is a circuit where each vertex in the circuit is *distinct* (except for first and last vertex).

A *directed walk* is a walk in which all edges are directed and are traversed along their direction. Directed paths, circuits, and cycles are defined similarly.

An example



$A, e_2, B, e_{13}, D, e_{10}, C, e_{11}, B, e_3, E$ is a *walk* in this graph.

$G, e_{12}, H, e_5, E, e_9, D, e_{13}, B$ is a *path* (and also a walk) joining G and B .

$B, e_{11}, C, e_{10}, D, e_8, F, e_7, E, e_5, H, e_1, A, e_2, B$ is a *cycle*.

A small note...

If G is a simple graph, then giving the sequence of vertices is sufficient to describe a walk, path, circuit, or cycle (as then the edges are implied).

For example, the path joining G and G on the previous slide could be (more compactly) represented as

$G, H, E, D, B.$

Similarly the cycle could be written as

$B, C, D, F, E, H, A, B.$

Still more terminology

A *subgraph* of a graph G is a graph H whose vertices and edges are *subsets* of the vertices and edges of G .

A *spanning subgraph* of G is a subgraph of G that contains all the vertices of G .

A graph is *connected* if, for any two distinct vertices, there is a path between them.

If a graph G is not connected, its maximal connected subgraphs are called the *connected components* of G .

Graphs (cont.)

A *forest* is a graph without cycles.

A *tree* is a *connected forest*, i.e. a connected graph without cycles.

A tree with a distinguished node (*root*) is called a *rooted tree*, otherwise it is called a *free tree* (or, often, simply a *tree*).

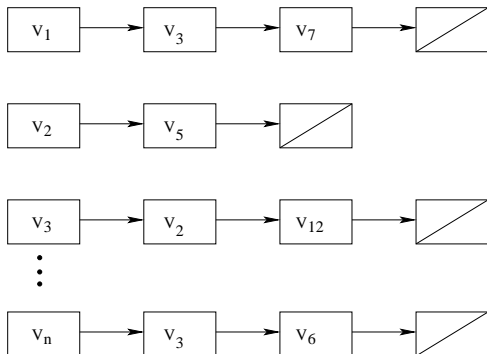
A *spanning tree* of a graph is a spanning subgraph that is a free tree.

Graphs (cont.)

Let G be an undirected graph with n vertices and m edges. We have the following observations:

- ▶ If G is *connected*, then $m \geq n - 1$.
- ▶ If G is a tree then, $m = n - 1$.
- ▶ If G is a forest, then $m \leq n - 1$.

Representing Graphs - Adjacency List (linked lists)



Representing Graphs - Adjacency Matrix

Here we represent the structure of the graph with a $\{0, 1\}$ matrix, the ones signifying that there is an edge present between the two vertices.

	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	0	1	1	0	1	0	0
V_2	1	0	1	0	1	0	0
V_3	1	1	0	0	0	0	1
V_4	0	0	0	0	1	1	0
V_5	1	1	0	1	0	0	1
V_6	0	0	0	1	0	0	0
V_7	0	0	1	0	1	0	0

Digraphs

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.

Graph search methods

Two common methods for exploring graphs are the *Depth-First Search* (DFS) and *Breadth-First Search* (BFS) methods.

Graph search methods

Two common methods for exploring graphs are the *Depth-First Search* (DFS) and *Breadth-First Search* (BFS) methods.

As a very brief reminder, DFS starts at a vertex, chooses an edge from that vertex and “walks out as far as possible”, finding new vertices until it encounters one it has already seen, then it backs up (as little as possible) to find other un-encountered vertices.

Graph search methods

Two common methods for exploring graphs are the *Depth-First Search* (DFS) and *Breadth-First Search* (BFS) methods.

As a very brief reminder, DFS starts at a vertex, chooses an edge from that vertex and “walks out as far as possible”, finding new vertices until it encounters one it has already seen, then it backs up (as little as possible) to find other un-encountered vertices.

In contrast to the DFS method, the Breadth First Search algorithm starts at a vertex and first explores the entire neighborhood of that vertex before moving onto another vertex.

Graph search methods (cont.)

Therefore, the DFS method generates “long, skinny” search trees, while the BFS method generates “short, bushy” ones.

The running time of BFS and DFS is the same, namely $O(n + m)$.

Graph search methods (cont.)

Therefore, the DFS method generates “long, skinny” search trees, while the BFS method generates “short, bushy” ones.

The running time of BFS and DFS is the same, namely $O(n + m)$.

Generally speaking, we may utilize a *stack* data structure to control a DFS search, while a *queue* data structure may be used for a BFS search.

Graph search methods (cont.)

Therefore, the DFS method generates “long, skinny” search trees, while the BFS method generates “short, bushy” ones.

The running time of BFS and DFS is the same, namely $O(n + m)$.

Generally speaking, we may utilize a *stack* data structure to control a DFS search, while a *queue* data structure may be used for a BFS search.

We will not go into great detail about these search methods here. Any book on (graph) algorithms should contain more detailed descriptions of these procedures.

Applications of the search methods

BFS and DFS can be used to answer a variety of questions about graphs including:

- ▶ Testing whether G is connected.
- ▶ Computing the connected components of G .
- ▶ Finding a spanning forest of G (or spanning tree if G is connected).
- ▶ Searching for a cycle in G , or reporting that G is acyclic.
- ▶ Given a start vertex x of G , computing, for every vertex v of G , a path with the minimum number of edges between x and v , or reporting that no such path exists (BFS).
- ▶ Testing for *strong connectivity* of digraphs. (Is there a directed path from u to v , for all u and v in D ?)

Weighted Graphs

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.

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.

Network Flow - The basics

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

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 $0 \leq f(u, v) \leq c(u, v)$.

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.

Network Flow - The basics

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 $0 \leq f(u, v) \leq c(u, v)$.

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.

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 .

Network Flow Algorithms

There are several algorithms for solving this maximum flow problem.

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

Network Flow Algorithms

There are several algorithms for solving this maximum flow problem.

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

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 this maximum flow problem.

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

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 this maximum flow problem.

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

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 Algorithm

The Ford-Fulkerson algorithm 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)

- 1 ▷ Input: A flow network G and source s , sink t
- 2 ▷ Output: Max flow through network
- 3 $f \leftarrow 0$
- 4 **while** augmenting path, P exists (in the residual network)
- 5 **do** augment flow along P
- 6 $f \leftarrow f + \text{new flow}$
- 7 update residual network
- 8 **return** f

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.)

G_f has the same vertices as G .

The edges of G_f are of two types:

Residual Networks (cont.)

G_f has the same vertices as G .

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.)

G_f has the same vertices as G .

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.)

G_f has the same vertices as G .

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.)

G_f has the same vertices as G .

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.

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' .

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.

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

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

Max-Flow/Min-Cut Theorem

Theorem

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

Ford-Fulkerson - Algorithm

Ford-Fulkerson(G, s, t)

- 1 ▷ Input: A network G and vertices, s, t
- 2 ▷ Output: A maximum flow
- 3 **for** each edge $(u, v) \in E(G)$
- 4 **do** $f[u, v] \leftarrow 0$
- 5 Initialize residual network $G_f = G$
- 6 **while** there exists an augmenting path P from s to t
- 7 **do** $\Delta_f(p) \leftarrow \min \{ \Delta_f(u, v) : (u, v) \in P \}$
- 8 **for** each edge $(u, v) \in P$
- 9 Update the flow (forwards and backwards edges)
- 10 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 $O(n + m) = O(m)$.

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 $O(n + m) = O(m)$.

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.)

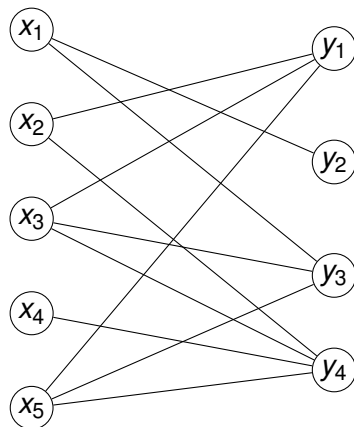
So the Ford-Fulkerson algorithm runs in time $O(|f^*|m)$.
(This isn't ideal, as a poor choice of augmenting paths can result in this large time bound.)

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 in 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 in 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 .

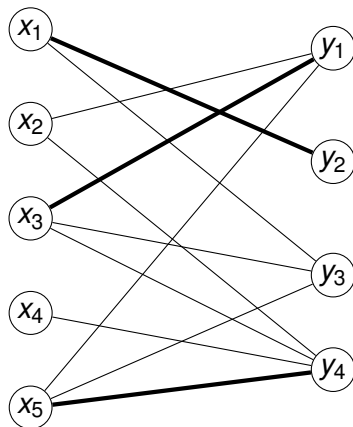
A bipartite graph could also represent relations between job applicants and available positions (i.e. people who are qualified for a particular job), customers and stores, houses and nearby police stations, etc, etc.

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 $O(n)$).