Lecture 1: Introduction

The Central Paradigm of Computer Science

The central paradigm in computer science is that an algorithm $\mathcal{A}$ is **good** if $\mathcal{A}$ runs in **polynomial time** in the input size n.

That is, $\mathcal{A}$ runs in time $T(n)=O(n^k)$ for some constant number k.

An algorithm is **bad** if it runs in exponential time.

Runtime of \ Input	10	100	1000
n			
n^2			
2^n			

For example, consider the problem of sorting n numbers.

A Good Algorithm: **MergeSort** runs in time O(n log n).

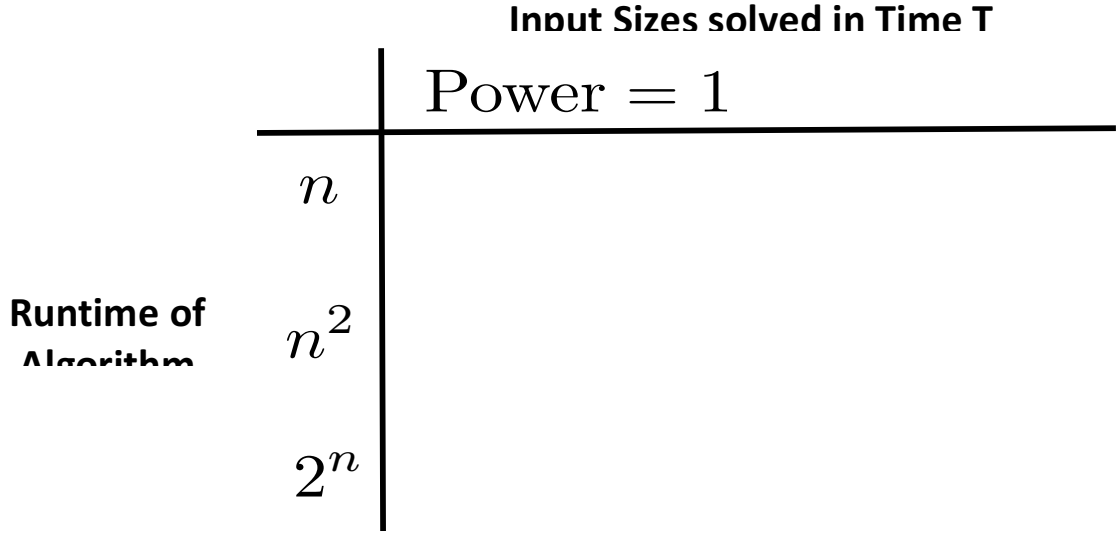
A Bad Algorithm: **BruteForce Search** runs in time $\Omega(2^n)$.

An Equivalent Characterization

This central paradigm has an equivalent formulation.

$\mathcal{A}$ runs in **polynomial time** in the input size n if and only if the input sizes that $\mathcal{A}$ can solve, *in a fixed amount T of time*, **scales multiplicatively** with increasing computational power.

The practical implications are perhaps simpler to understand with this latter formulation.



Moore's Law: Exponential time algorithms will never be able to solve large problems.
Thus, improvements in hardware will never overcome *bad algorithm design*.
Indeed, the current dramatic breakthroughs in computer science are based upon better (faster and higher performance) algorithmic techniques.

This polytime measure of quality or "goodness" is robust.
All *reasonable models* of algorithms are polynomial time equivalent.
The standard formal model is the **Turing Machine**. [See Comp 330]
Otherwise one model could perform, say, an *exponential number* of operations in the time another model took to perform just one.

Course Overview

We want to know what problems can be solved in **polytime**.
Towards this goal we will focus on learning and applying fundamental algorithmic techniques such as recursive algorithms, graph algorithms, greedy algorithms, dynamic programming and network flows.
The focus of Comp 251 was learning and applying fundamental techniques to solve applicatins in polynomial time.
The techniques included: recursive algorithms, graph algorithms, greedy algorithms, dynamic programming, network flows and (data structures).

Strong Warning!

But most problems **cannot** always be solved in polynomial time.
In this course our focus will be on understanding:

- The limits of polynomial solvability, e.g. linear programming.
- What problems cannot always be solved in polytime, e.g. NP-completeness.
- What algorithmic method can be used on these hard problems, e.g. approximation algorithms, heuristics, probabilistic methods, etc.

Cryptography

Alice wants to send Bob a message.
But she is worried that Eve might *intecept* the message.
What can she do about this?
She decides to **encypt** the message M as $\overline{M} =$ f(M).
Bob can then **decrypt** the message via $f^{-1}(\overline{M}) = f^{-1}(f(M)) = M$.
But now if Eve intercepts the message, all she sees is gibberish!

Locks and Keys

We can view this in the following way:
Alice **encrypts** the message M with the (encryption) lock f.
Bob **decrypts** the message $\overline{M}$ with the (decryption) key f^{-1}.

Because Eve does not have the key she cannot decipher the message.
However, there are two major problems here…

Problem 1. Eve might be able to **break** the code!
That is, given $\overline{M}$ she may be able to reconstruct f and then f^{-1}.
Techniques for code-breaking include: frequency analysis and cribs.

Problem 2.
Alice and Bob need to **agree** on what the encryption code (lock) f is.
But to do this they need to exchange a message discussing the code!!!

Public-Key Cryptography
Here is one way around Problem 2.
Bob tells Eve what the encryption code (lock) is!
In fact, Bob gives absolutely everyone a copy of the (encryption) lock.
They can use this lock to send him messages.
Thus, the lock is made **public**. This is called *public-key cryptography*.

But this *idea* sounds completely $\mathbf{c_r a^z y}$.
Doesn't this solution to Problem #2 make Problem #1 inevitable?
That is, if Eve has the lock f then won't she use it to decode $f(M)$?
No, not if f is hard to invert for anybody except Bob himself.
But do such functions f that are hard to invert **exist**? *Yes…*

RSA Encryption
Bob chooses two large *prime numbers* q_1, q_2 and a large number p that is co-prime to $(q_1-1)(q_2-1)$.
Bob's *public key* is (p, n) where $n=q_1 q_2$.
Encryption f: $\overline{M} = M^p \bmod n$.
Bob's *private key* is (q_1,q_2,x) where x is the inverse of p modular $(q_1-1)(q_2-1)$.

Decryption f^{-1}: M= $\overline{M}^x \bmod n$.

This method works – this is not hard to prove by verifying that:

$$\overline{M}^x \bmod n = (M^p)^x \bmod n = M$$

Public-key cryptography lies at the heart of the modern economy:
e.g. *Financial Services, Online Shopping, Secure Messaging, etc.*
We claim it is safe because:
Bob has a **good algorithm** for decryption.
Eve only has a **bad algorithm** for decryption.

Bob has a Good Decryption Algorithm

Initially, Bob can do the following in **polynomial time**.
Choose the primes q_1, q_2
Choose a number p that is coprime with $(q_1-1)(q_2-1)$.
Find, by Euclid's Algorithm, x the inverse of p modular $(q_1-1)(q_2-1)$.
Using *fast exponentiation*, encoding and decoding is polynomial time.

Eve has a Bad Decryption Algorithm

To decrypt Eve needs to find x the inverse of p modular $(q_1-1)(q_2-1)$.
She knows p, but does not know q_1, q_2
Instead she only knows $n=q_1 q_2$.
So to find q_1, q_2 she needs to find the **prime factorization** of n.
But it is believed that finding the *prime factors* of a b-bit number is hard.
In fact, if Eve could find x without q_1, q_2 then she could use x to find q_1, q_2.
That is, she could factor n.
This implies Eve only has an **exponential time** algorithm to decrypt.

Cryptography and the Secret Services

RSA encryption was designed by Rivest, Shamir and Edelman in 1978.
However, the method was actually invented by Clifford Cocks in 1973.
He worked for the British Intelligence Service at GCHQ.
But this information was not **declassified** until 1997.

Lecture 2: Network Flows – the Ford Fulkerson Algorithm

Maximum Flows in a Network

In the **shortest path problem**, we want to find a path (of minimum length/cost) in a directed graph $G=(V,A)$ from a source s to a sink t.

In the **maximum flow problem**, we essentially want to pack as many s-t paths as possible that "fit" into the graph.

Specifically:

- A good is produced at a *source vertex* s.
- We must send as much of the good as possible to a *sink vertex* t.
- Each arc $a=(i,j)$ has a capacity $u_a=u_{ij}$.
- The **arc capacity** is the maximum amount of the good that can be sent via that arc (over a given time period).

A network flow **f** from a source s to a sink t satisfies two properties:

Capacity Constraints: The flow on an arc a is non-negative and at most its *capacity*: $0 \leq f_a \leq u_a$

Flow Conservation: The flow into a vertex $v \neq \{s,t\}$ *equals* the flow out of the vertex:

$$\sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a$$

The Value of a Flow

So a network flow **f** satisfies *flow conservation* and the *capacity constraints*.

The **value** of a flow **f**, is the *quantity of flow* that reaches* the sink t:

$$|f| = \sum_{a \in \delta^-(t)} f_a = \sum_{a \in \delta^+(s)} f_a$$

* We may assume the network has no arcs into s and no arcs out of t.

A fundamental problem then is to find an s-t flow of **maximum value**.

The Maximum Flow Problem

The Maximum Flow Problem. Find an s-t flow **f*** of maximum value.

Mathematically, we can formulate the maximum flow problem as:

$$\max \sum_{a \in \delta^-(t)} f_a$$

$$s.t \quad \sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a \qquad \forall v \in V \setminus \{s,t\}$$

$$0 \leq f_a \leq u_a \qquad \forall a \in A$$

Examples of Flows

Simple examples: *Oil through Pipelines*, *Data Packets in an Information Network*, *Current in an Electrical Network*, *Goods in a Transportation Network, Currency Flows in a Financial Network.*

Less Obvious Examples: *Image Segmentation, Open Pit Mining, Flight Scheduling.*

Paths and Flows

What do *s-t* paths have to do with *s-t* flows?

An *s-t* path satisfies **flow conservation**.

Thus an *s-t* path is flow of **value** one.

Path Unions and Flows

Similarly the union of a set of *s-t* paths satisfies **flow conservation**.

- Note in the union the same path may also be used *more than once.*

If this union of paths also satisfies the **capacity constraint** $f_a \leq u_a$ on each arc then they form an *s-t* flow.

Cycles and Flows

A directed cycle also satisfies **flow conservation**.
Thus, the union of a set of cycles satisfies **flow conservation**.
If this union of cycles also satisfies the **capacity constraint** on each arc then they form an *s-t* flow.

The Flow Decomposition Theorem

Thus the union of a set of path and a set of cycles satisfies **flow conservation**.
If this union of paths and cycles also satisfies the **capacity constraint** on each arc then they form an *s-t* flow.
The converse is also true: *every s-t flow is made up of s-t flows and cycles.*

> **The Flow Decomposition Theorem.** Any *s-t* flow can be decomposed into a collection of *s-t* paths and directed cycles.

We'll prove this in a later lecture. *For now let's just see an example of this…*

Trucking Syrup

Imagine sending maple syrup on a network from Montreal to Vancouver. (See xxample in Class)
For each link a in the network:

- Let u_a be the trucking capacity in tonnes on the link.
- Let f_a be the amount of syrup we are transporting on that link.

The Flow Constraints

First let's verify that this example is indeed an *s-t* flow.
Capacity Constraints: clearly $u_a \leq f_a$ on every arc a.
Flow Conservation: it is easy to verify that the flow into each vertex (non- source/non-sink) equals the flow out of the vertex.
So we do have an *s-t* flow and the flow **value** is 19 tonnes.

A Flow Decomposition

We can find a flow decomposition by taking out paths and cycles one step at a time.

But the flow decomposition is not unique.

An Observation

If a flow puts positive weight on a directed cycle then we can remove this weight from the cycle and still have a flow of the same value.

- *Flow conservation still holds.*
- *Capacity constraints still hold.*

The Structure of Maximum Flows

The Flow Decomposition Theorem. Any *s*-*t* flow can be decomposed into a collection of *s*-*t* paths and directed cycles.

Observation. For any *s*-*t* flow there is an *s*-*t* flow of the same value that contains no directed cycles.

$\Longrightarrow$ There is a maximum flow that contains no directed cycles.

$\Longrightarrow$ There is a maximum flow whose flow decomposition contains only directed paths.

Finding Maximum Flows

Thus there is a maximum flow whose flow decomposition contains only directed paths.

So we can search for maximum flows by searching for directed paths.

The **greedy approach** to do this would be to:

- Find a directed path P from s to t.
- Send as much flow on the path P as possible.
- Repeat until there is no surplus capacity on any *s*-*t* path.

Does this work?

NO. See example in class.

What Went Wrong?

In each step, we tried to increase the flow value (the flow out of *s*) whilst maintaining flow conservation.

We know adding flow along an *s-t* path achieves this.

But *s-t* paths are not the only graphical objects that achieve this…

Augmenting Paths

An *s-t* augmenting path is a path P from *s* to *t* where some of the arcs can be in the reverse direction.

Suppose we *increase* the flow on a **forward arc** but *decrease* the flow on a **backward arc** on the path P.

Then **flow conservation** is still maintained!

The Use of Backward Arcs

Using a **backwards arc** corresponds to reducing the amount of flow used on that arc in the forward direction.

So we are correcting a mistake we made earlier.

- We are sending the maple syrup back where it came from as we have previously sent too much syrup down the arc.

Maintaining Feasibility

So using an augmenting path has two nice properties:

- It maintains **flow conservation**.
- It increases the **flow value**.

But we must ensure the **capacity constraints** remain satisfied.

In particular, we must ensure that:

- The flow on an arc does not exceed its capacity: $f_a \leq u_a$
- The flow on an arc is non-negative: $f_a \geq 0$

By sending flow along backward arcs, the latter condition can be violated.

The Bottleneck Capacity

How much can we increase the flow value on an augmenting path?

Given the current flow **f**:

- We can increase the flow on a forward arc *(i,j)* by $u_{ij} - f_{ij}$.
- We can decrease the flow on a backward arc *(j,i)* by f_{ij}.

In particular, we can increase the flow on **augmenting path** P by

$$b(P, \mathbf{f}) = \min \left[\min_{(i,j) \text{ forward arc}} u_{ij} - f_{ij}, \min_{(j,i) \text{ backward arc}} f_{ij} \right]$$

We call *b*(*P*, **f**) the **bottleneck capacity** of P w.r.t. flow **f**.

The Ford-Fulkerson Algorithm

Thus, we have derived the famous **Ford-Fulkerson algorithm**:

Set **f=0**

Repeat

Find an augmenting path *P* w.r.t to **f**

Augment flow on path *P* by *b*(*P*, **f**)

So we have a prospective algorithm, but we know very little about it:

- *Does the algorithm actually find a maximum flow?*
- *Is it efficient?*
- *What else does it tell us about maximum flow problems?*

When and How can the Algorithm use an Arc?

Before answering these questions, it will be useful to understand exactly how and when the algorithm is allowed to use an arc.

Recall the Ford-Fulkerson algorithm uses arcs in both the *forward* and *backward* directions. But when is it allowed to do this?

Forward Direction: an arc *a=(i,j)* can be used forwards if there is spare capacity on the arc: $f_{ij} < u_{ij}$

Backward Direction: an arc *a=(i,j)* can be used backwards if it has already been used forwards, and so flow can be sent back: $f_{ij} > 0$

The Residual Graph

Given G=*(V, A)* and a flow **f**, our previous observations allow us to define the **residual graph** G_f.

For each arc *a=(i,j)*

- The residual graph G_f contains an arc *(i,j)* of capacity $u_a - f_a$.
- The residual graph G_f contains an arc *(j,i)* of capacity f_a.

Using the residual graph is the best way to both implement and understand the Ford-Fulkerson algorithm.

See class for example.

The Bottleneck Capacity

Observe that the arcs in the residual graph only have capacities.

The **bottleneck capacity** of a path is now just the minimum capacity of an arc in the corresponding directed path in the residual graph.

$$b(P, \mathbf{f}) = \min \left[\min_{(i,j) \text{ forward arc}} u_{ij} - f_{ij} , \min_{(i,j) \text{ backward arc}} f_{ij} \right]$$

The Ford-Fulkerson Algorithm

Thus, we can restate the algorithm as:

Set **f=0**

Repeat

Find a directed s-t path *P* in the residual graph G_f

Augment flow on path *P* by its bottleneck capacity *b*(*P*, **f**)

Does this algorithm output a maximum flow?

To determine this we must study graph cuts…

Lecture 3: Network Flows – the Maxflow-Mincut Theorem

Recap on Network Flows

Given a graph G=*(V, A)* with an integral capacity u_a on each arc a.

A network flow **f** from a source *s* to a sink *t* satisfies two properties:

Capacity Constraints: The flow on an arc *a* is non-negative and at most its *capacity*: $0 \leq f_a \leq u_a$

Flow Conservation: The flow into a vertex $v \neq \{s,t\}$ *equals* the flow out of the vertex:

$$\sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a$$

The Maximum Flow Problem. Find an *s-t* flow **f*** of maximum value.

Mathematically, we can formulate the maximum flow problem as:

$$\max \sum_{a \in \delta^-(t)} f_a$$

$$s.t \quad \sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a \qquad \forall v \in V \setminus \{s,t\}$$

$$0 \leq f_a \leq u_a \qquad \forall a \in A$$

The Ford-Fulkerson Algorithm

We found a prospective algorithm for the maximum flow problem.

Thus, we have derived the famous **Ford-Fulkerson algorithm**:

Set **f=0**

Repeat

Find a directed s-t path *P* in the residual graph G_f

Augment flow on path *P* by its bottleneck capacity *b*(*P*, **f**)

We were left with 3 questions:

- *Is it efficient?*
- *Does the algorithm actually find a maximum flow?*
- *What else does it tell us about maximum flow problems?*

We begin to address all three questions in this lecture.

s-t Cuts

We say $(\mathcal{S}, V \setminus \mathcal{S})$ is an *s-t* cut if $s \in \mathcal{S}$ and $t \notin \mathcal{S}$.

Let **f*** be the flow output when the Ford-Fulkerson algorithm terminates.

Let $\mathcal{S}^*$ be set of vertices reachable from *s* in the residual graph G_{f^*}.

$$\mathcal{S}^* = \{ v : \exists \text{ directed path from } s \text{ to } v \text{ in } G_{\mathbf{f}^*} \}$$

Note that:

- $(\mathcal{S}^*, V \setminus \mathcal{S}^*)$ is an s-t cut.
- There are no arcs leaving $\mathcal{S}^*$ in G_{f^*} : $\delta^+_{G_{f*}}(\mathcal{S}^*) = \emptyset$.

These facts will be key to proving the algorithm works.

The Cut Lemma

The Cut Lemma. Given a flow **f** and any *s-t* cut (*S*, V*S*). Then the value of the flow is:

$$|f| = \sum_{a \in \delta^+(\mathcal{S})} f_a - \sum_{a \in \delta^-(\mathcal{S})} f_a$$

Proof.

The value of the flow is the amount of flow leaving the source *s*:

$$|f| = \sum_{a \in \delta^+(s)} f_a$$

$$= \sum_{a \in \delta^+(s)} f_a - \sum_{a \in \delta^-(s)} f_a$$

$$= \left(\sum_{a \in \delta^+(s)} f_a - \sum_{a \in \delta^-(s)} f_a \right) + \sum_{v \in \mathcal{S} \setminus \{s\}} \left(\sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a \right)$$

$$= \sum_{v \in \mathcal{S}} \left(\sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a \right)$$

Now take any arc *a=(i,j)* in the graph. There are four cases:

1) $(i,j) \in \delta^+(\mathcal{S})$
   - f_a appears once in sum with coefficient +1.
2) $(i,j) \in \delta^-(\mathcal{S})$
   - f_a appears once in sum with coefficient -1.
3) $\{i,j\} \subseteq V \setminus \mathcal{S}$
   - f_a does not appear in sum.
4) $\{i,j\} \subseteq \mathcal{S}$
   - f_a appears twice in sum with coefficients +1 and -1.

$$\Longrightarrow |f| = \sum_{a \in \delta^+(\mathcal{S})} f_a - \sum_{a \in \delta^-(\mathcal{S})} f_a$$

The Capacity of a Cut

We define the capacity of an *s-t* cut $(\mathcal{S}, V\setminus\mathcal{S})$ as:

$$\text{cap}(\mathcal{S}) = \sum_{a \in \delta^+(\mathcal{S})} u_a$$

The capacity of any *s-t* cut upper bounds the value of the maximum flow.

Corollary. Given any flow **f** and any *s-t* cut $(\mathcal{S}, V\setminus\mathcal{S})$ we have:

$$|f| \leq \text{cap}(\mathcal{S})$$

Proof.

By the Cut Lemma, we have:

$$\begin{aligned} |f| &= \sum_{a \in \delta^+(\mathcal{S})} f_a - \sum_{a \in \delta^-(\mathcal{S})} f_a \\ &\leq \sum_{a \in \delta^+(\mathcal{S})} u_a + 0 \end{aligned}$$

The Maxflow-Mincut Theorem

By the corollary, the value of a maximum flow is at most the capacity of the **minimum cut**.

In fact, they are equal.

Maxflow-Mincut Theorem. The maximum value of an *s-t flow* is equal to the minimum capacity of an *s-t cut*.

Proof.

Recall **f*** is the flow output by the Ford-Fulkerson algorithm.

Recall S^* is the vertices reachable from *s* in the residual graph G_{f^*}

$$\mathcal{S}^* = \{ v : \exists \text{ directed path from } s \text{ to } v \text{ in } G_{\mathbf{f}^*} \}$$

We know, by Lemma 1, that: $|f^*| \leq \text{cap}(\mathcal{S}^*)$

So we will now show that: $|f^*| \geq \text{cap}(\mathcal{S}^*)$

This will imply that **f*** is a maximum *s-t* flow and $(\mathcal{S}^*, V \setminus \mathcal{S}^*)$ is a minimum capacity *s-t* cut.

Recall that $\delta^+_{G_{f*}}(\mathcal{S}^*) = \emptyset$ otherwise we could have grown S*.

But what does this imply about the flow **f*** ?

Take any arc $(i,j) \in \delta^+(\mathcal{S}^*)$. This arc is **not** in the residual graph.

$$\implies \quad f^*_{ij} = u_{ij}$$

Take any arc $(i,j) \in \delta^-(\mathcal{S}^*)$. The reverse of this arc is **not** in the residual graph.

$$\implies \quad f^*_{ij} = 0$$

Now by Lemma 1:

$$\begin{aligned} |f^*| &= \sum_{a \in \delta^+(\mathcal{S}^*)} f_a - \sum_{a \in \delta^-(\mathcal{S}^*)} f_a \\ &= \sum_{a \in \delta^+(\mathcal{S}^*)} u_a - \sum_{a \in \delta^-(\mathcal{S}^*)} f_a \\ &= \sum_{a \in \delta^+(\mathcal{S}^*)} u_a - 0 \\ &= \text{cap}(\mathcal{S}^*) \end{aligned}$$

This completes the proof.

This deals with 2 of our 3 questions:

- *Does the algorithm find a maximum flow?*
- *What else does it tell us about maximum flow problems?*

Let's examine the remaining question…

- *Is the Ford-Fulkerson algorithm efficient?*

The running time is *O*(*m* #iterations).
So we must calculate the number of iterations.

The Number of Iterations

The algorithm terminates when $b(P, \mathbf{f})=0$ for every *s*-*t* path P.
Thus, as the capacities are integral, we have $b(P, \mathbf{f}) \geq 1$.

- In the worst case, we increase the flow value by 1 in each iteration.

But the maximum flow value equals the minimum cut value, C.

- If $U = \max_a u_a$ then $C \leq n \cdot U$.

Thus the number of iteration is at most nU.

$$\Longrightarrow \quad \text{Runtime} \;=\; O(mn \cdot U)$$

Pseudo-Polynomial Time

The running time of $O(mn \cdot U)$ is only **pseudo-polynomial time**.
Recall, this is not truly polynomial time in the input size.

- If the integers have b bits then U could be 2^b.

But maybe the algorithm really is polynomial time, and our running time analysis was just not precise enough to prove this?
No. *The Ford-Fulkerson algorithm is a pseudo-polynomial time algorithm…*
See class for an example where the Ford-Fulkerson algorithm takes 2U iterations!
We remark that in that example if the augmenting paths were chosen more carefully then we could find the maximum flow in just 2 iterations.

Proofs of the Maxflow-Mincut Theorem

We have given an **algorithmic proof** of the maxflow-mincut theorem using the Ford-Fulkerson algorithm.

> **Maxflow-Mincut Theorem.** The maximum value of an *s-t flow* is equal to the minimum capacity of an *s-t cut*.

This is an important theorem and you will discover many very different proofs of it in other courses. For example:

- An **optimization proof** using Linear Programming duality. [COMP 360]
- A **graph theoretic proof** using Menger's Theorem. [MATH 350]

Matchings

A **matching** is a set of vertex-disjoint edges.
Hence, each vertex is incident to *at most* one edge in the matching.
A matching is **perfect** if every vertex is incident to an edge in the matching.

Bipartite Graphs

In a **bipartite graph** the vertex set can be partitioned as $V = X \cup Y$ such that every edge has one end-vertex in X and one end-vertex in Y.
Is there an efficient algorithm to find a maximum cardinality matching in a bipartite graph? *Yes, using maximum flows.*

The Auxiliary Network

Take an undirected bipartite graph $G = (X \cup Y, E)$.
Direct each edge (x_i, y_j) from x_i to y_j.
Add a **source** vertex s with an outgoing arc to each vertex in X.
Add a **sink** vertex t with an incoming arc from each vertex in Y.
Give every arc a *capacity* of one.
Then a flow of value k in the auxiliary network consists of k arc-disjoint paths.
But then each path gives an edge in a **matching** in the original graph.
So a *maximum* flow corresponds to a *maximum* matching!

The Running Time

Recall the runtime of the Ford-Fulkerson algorithm is $O(m \cdot \#\text{iterations})$.
But the maximum cardinality of a **matching** is at most

$$\min \{ |X|, |Y| \} \leq n$$

Thus the number of iterations is at most n.

$$\implies \quad \text{Runtime} = O(m \cdot n)$$

> **Theorem.** *There is a polynomial time algorithm to find a maximum cardinality matching in a bipartite graph.*

In fact, there is a polynomial time algorithm even in *non-bipartite* graphs, but that requires techniques beyond the scope of this course.

The Maxflow-Mincut Theorem

Recall the Maxflow-Mincut theorem:

> **Maxflow-Mincut Theorem.** The maximum value of an *s-t flow* is equal to the minimum capacity of an *s-t cut*.

Therefore, we can also find a minimum cut in the auxiliary network in polynomial time.
So what?
To answer this question, let's consider a slightly different formulation of the auxiliary network…

The Auxiliary Network (Revised)

Let's give the arcs between the *X*-vertices and the *Y*-vertices **infinite** capacity.
This changes nothing as we can still only use these arcs once.

- This is because the *unique arc* into each *X*-vertex still has capacity one.

So the maximum flow value and the minimum cut capacity remain the same.

The Minimum Cut

Recall, the maximum flow has value at most n, since s has n outgoing arcs.

Thus, by the *maxflow-mincut theorem*, the capacity of the minimum cut S^* is almost at most n.

In particular the capacity of S^* is *finite*.

$\Longrightarrow$ $\delta^+(S^*)$ contains **no** infinite capacity arcs.

The Structure of the Minimum Cut

Let $X^* = X \cap \mathcal{S}^*$.

Then $\Gamma(X^*) \subseteq \mathcal{S}^* \cap Y$ otherwise:

$$\text{cap}(\mathcal{S}^*) = \sum_{a \in \delta^+(\mathcal{S}^*)} u_a = \infty$$

So all the arcs in $\delta^+(\mathcal{S}^*)$ are of the form (s, x_i) and the form (y_j, t).

A Digression…

The Vertex Cover Problem

Take an undirected graph G=*(V,E)*.

A **vertex cover** is a set $C \subseteq V$ that touch <u>every</u> edge in the graph.

- For each edge *e=(u,v)* either $u \in C$ or $v \in C$ (or both are).

A basic problem in graph theory is the following:

> **Vertex Cover Problem.** *Find a minimum cardinality vertex cover in the graph.*

Now back to the matching problem…

A Vertex Cover

So there are only three types of edge in the bipartite graph:

- The fourth type, edges between X^* and $Y \setminus \Gamma(X^*)$, cannot occur.

This means $X \setminus X^* \cup \Gamma(X^*)$ is a vertex cover.

But the cardinality of this vertex cover is the **capacity** of the cut.

Similarly, any vertex cover induces an s-t cut whose capacity is the same as the cardinality of the vertex cover.

Maxflow-Mincut Theorem. The maximum value of an *s-t flow* is equal to the minimum capacity of an *s-t cut*.
Corollary. In a bipartite graph, the maximum cardinality of a matching is equal to the minimum cardinality of a vertex cover.

Efficient Algorithms for Vertex Cover
When we run the maximum flow algorithm in the auxiliary network we also find a **minimum cut**.
The minimum cut gives a **minimum vertex cover** in the original graph.

> **Theorem.** *There is a polynomial time algorithm to find a minimum cardinality vertex cover in a bipartite graph.*

In contrast to the matching problem, we do **not** believe there is a polytime algorithm for the vertex cover problem in non-bipartite graphs:

> **Theorem.** *There is a no polynomial time algorithm to find a minimum cardinality vertex cover in a non-bipartite graph, unless P=NP.*

Certificates of Optimality
Therefore, in a bipartite graph, if a **vertex cover** C and a **matching** M have the same cardinality then they are both optimal.

- C *is a minimum vertex cover.*
- M *is a maximum matching.*

Thus they form a **certificate of optimality**.

Lecture 4: Network Flows – Applications

The Maximum Flow Problem. Find an *s-t* flow $\mathbf{f^*}$ of maximum value.

Mathematically, we can formulate the maximum flow problem as:

$$\max \sum_{a \in \delta^-(t)} f_a$$

$$\text{s.t.} \sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a \qquad \forall v \in V \setminus \{s,t\}$$

$$0 \leq f_a \leq u_a \qquad \forall a \in A$$

Extensions and Applications

We saw some simple applications of network flows.

Here we will see some more complex applications: *Image Segmentation, Flight Scheduling, Open-Pit Mining.*

To do this let's see how to extend the model to solve problems with:

- *Lower Bounds of arc flows.*
- *Supply and Demand.*
- *Multiple Sources and Multiple Sinks.*

Extensions: Supply/Demand Version

Suppose the source *s* has a fixed **supply** amount equal to b.

- The sink *t* has a **demand** of b (equivalently, a supply of $-b$).

Is there a flow that satisfies the supply and demand constraints:

$$\text{flow-out}(s) = \text{flow-in}(t) = b$$

Yes *if and only if* there is an *s-t* flow of value at least b.

Extensions: Multiple Sources/Sinks

Suppose there are sources $\{s_1, s_2, \ldots s_k\}$ and sinks $\{t_1, t_2, \ldots t_\ell\}$.

Source s_i has a **supply** of b_i. Sink t_j has a **demand** of b_j.

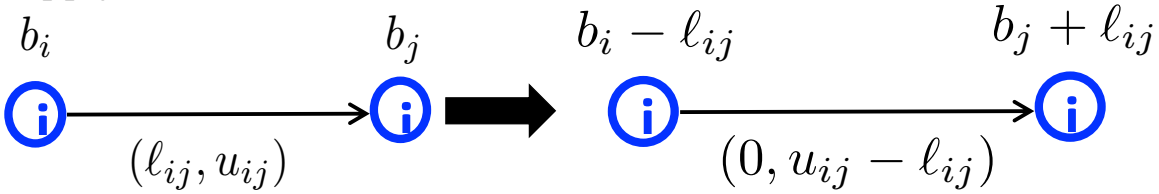
Is there a flow that satisfies the supply and demand constraints:

$$\text{flow-out}(v) - \text{flow-in}(v) = b_v \qquad \forall v \in V$$

We can **convert** this to a single-source and single-sink flow problem:
Add a super-source *s* and arcs (s, s_i) with capacities b_{s_i}
Add a super-sink *t* and arcs (t_j, t) with capacities b_{t_j}
There is a feasible flow to the multi-source problem *if and only if* there is a flow from s to t of value $\sum_{v:b_v>0} b_v$

Lower Bounds

Suppose we must send ℓ_{ij} units of flow on an arc (*i*,*j*).
Thus the capacity constraint on (*i*,*j*) becomes: $\ell_{ij} \leq f_{ij} \leq u_{ij}$
To deal with this we force ℓ_{ij} units of flow along the arc and edit the supply/demand values as follows:



After applying this for each arc we have:

$$b'_i = b_i + \sum_{(k,i)\in A} \ell_{ki} - \sum_{(i,k)\in A} \ell_{ik} \quad \forall i \in V$$

$$0 \leq f_{ij} \leq u_{ij} - \ell_{ij} \quad \forall (i,j) \in A$$

Flight Scheduling

Suppose we have a collection of flights:

Origin-Destination	Time
Boston – San Francisco	7am – 9am
Toronto - Boston	7am – 8am
Montreal - Toronto	8am – 9am
Toronto – San Francisco	10am – 1pm
Montreal - Toronto	10am – 11am
San Francisco – Montreal	10am – 5pm
San Francisco – Los Angeles	3pm – 4pm
Boston - Toronto	3pm – 4pm

How do we service all these flights using as few planes as possible?
e.g. Can we schedule all these flights using at most $k=3$ planes?

We answer this question using network flows…

The Network Flow Model

We create the network flow as follows:

- For each flight i, we have a vertex for the origin o_i and the destination d_i.
- We add a **source** vertex s with supply k.
- We add an arc (s, o_i) with capacity *1* for every flight i
- We add a **sink** vertex t with demand k.
- We add an arc (d_i, t) with capacity 1 for every flight i.
- We add an arc (d_i, o_j) with capacity 1 if it is feasible for a plane to service flight i and then service flight j.
- Finally we have an arc (o_i, d_i) with a **lower bound** of 1.

See class for a feasible flow; so there is a schedule that uses exactly three planes.

Remarks

In reality, airline scheduling is more complex that this as we have to also take into account:

Crew Constraints: *health and safety, crew locations, etc.*
Costs: *crew, fuel, servicing, landing slots, etc.*
Flexible Schedules: *the ability to simply make minor modifications.*
Plane Capacities.

Multiple Days: *where planes/crews are at the end of the day is important.*

These constraints cans also be incorporated into network flow models. Moreover, fast algorithms are necessary to deal with problems caused by flight delays or bad weather.

Open Pit Mining

We have a set V of blocks.

- Each block $i \in V$ has a profit π_i.
- This profit is the estimated value of the ore contained in the block minus the cost of digging up and processing the block.

We want to maximize the **profit** of the pit.

This sounds easy but there are topological constraints on the mine…

The pit cannot be too **steep**.

- In particular, to dig a block i we must first remove the 3 closest blocks in the layer above.

It is trivial to extend this 2D model into a realistic 3D model.

The Network Flow Model

Again, we solve problem this using a network flow:

- There is a vertex for each block $i \in V$.
- There is a **source** vertex s and a **sink** vertex t.
- There is an arc (s, i) of capacity π_i if and only if $\pi_i > 0$.
- There is an arc (j, t) of capacity $|\pi_i|$ if and only if $\pi_i < 0$.
- There is an arc from i to the 3 blocks above it of capacity ∞.

Observe that:

$$\text{cap}(\{s\}) = \sum_{i:\pi_i>0} \pi_i$$

This is **finite**, so the capacity of the *minimum cut* S^* is finite.

$\Longrightarrow$ The maximum flow value is **finite**.

Now suppose the minimum cut S^* contains a block i.

- Then the 3 blocks above *i* are in $\mathcal{S}^*$ as the corresponding arcs have **infinite** capacity.
- Then the 3 blocks above each of those blocks are also in $\mathcal{S}^*$ etc.

So $\mathcal{S}^*$ corresponds to a feasible pit!

The Capacity of a Cut

What exactly is the capacity of an *s-t* cut $\mathcal{S}$?

$$\begin{aligned}
\text{cap}(\mathcal{S}) &= \sum_{a:\in\delta^+(\mathcal{S})} u_a \\
&= \sum_{i\notin\mathcal{S}:\pi_i>0} \pi_i + \sum_{i\in\mathcal{S}:\pi_i<0} |\pi_i| \\
&= \left(\sum_{i\in V:\pi_i>0} \pi_i - \sum_{i\in\mathcal{S}:\pi_i>0} \pi_i\right) + \sum_{i\in\mathcal{S}:\pi_i<0} |\pi_i| \\
&= \left(\sum_{i\in V:\pi_i>0} \pi_i - \sum_{i\in\mathcal{S}:\pi_i>0} \pi_i\right) - \sum_{i\in\mathcal{S}:\pi_i<0} \pi_i \\
&= \sum_{i\in V:\pi_i>0} \pi_i - \sum_{i\in\mathcal{S}} \pi_i
\end{aligned}$$

The Minimum Cut is an Optimal Pit

So: $\text{cap}(\mathcal{S}) = \sum_{i\in V:\pi_i>0} \pi_i - \sum_{i\in\mathcal{S}} \pi_i = \Phi - \sum_{i\in\mathcal{S}} \pi_i$

Observe that Φ is a *constant* independent of the cut $\mathcal{S}$.

So the capacity of $\mathcal{S}$ is **minimized** exactly when the *profit* associated with $\mathcal{S}$ is **maximized**!

$\Longrightarrow$ The minimum cut $\mathcal{S}^*$ gives the optimum pit design.

Of course, we can now find the *minimum cut* by running a maximum flow algorithm.

Remarks

In practice, an important factor is how to dig the pit in **stages**.

- Huge costs and interest payments mean that incorporating **time** aspects are vital.

This network flow formulation is actually not far from the *state of the art* in the field of **mining**.

This model applies to numerous problems in manufacturing and scheduling with time or priority constraints.

Image Segmentation

A fundamental problem in computer vision is *image segmentation*.

A basic task there is **foreground/background segmentation**:

- Label the pixels according to whether they belong to the foreground or background.

The model has two parameters for each pixel:

- Let f_i be the likelihood that pixel i is in the **foreground**.
- Let b_i be the likelihood that pixel i is in the **background**.

In general, if $f_i > b_i$ then we want pixel i in the foreground.

But we also want a *smooth boundary* between the foreground and background. To incorporate this aim there is:

A **penalty** ρ_{ij} for separating adjacent pixels i and j.

The Network Flow Model

Again, we solve this problem using a network flow:

- There is a vertex for each *pixel i*.
- There is a **source** (foreground) vertex s and a **sink** (background) vertex t.
- There is an arc (s, i) of capacity f_i.
- There is an arc (j, t) of capacity b_j.
- For **adjacent pixels**, there are arcs (i, j) and (j, i) both of capacity ρ_{ij}.

The Capacity of a Cut

What exactly is the capacity of an *s-t* cut $\mathcal{S}$?

$$\begin{aligned}
\mathrm{cap}(\mathcal{S}) &= \sum_{a:\in\delta^+(\mathcal{S})} u_a \\
&= \sum_{i\notin\mathcal{S}} f_i + \sum_{j\in\mathcal{S}} b_j + \sum_{(i,j)\in\delta^+(\mathcal{S})} \rho_{ij} \\
&= \left(\sum_{i\in V} f_i - \sum_{i\in\mathcal{S}} f_i\right) + \left(\sum_{j\in V} b_j - \sum_{j\notin\mathcal{S}} b_j\right) + \sum_{(i,j)\in\delta^+(\mathcal{S})} \rho_{ij} \\
&= \sum_{i\in V}(f_i + b_i) - \sum_{i\in\mathcal{S}} f_i - \sum_{j\notin\mathcal{S}} b_j + \sum_{(i,j)\in\delta^+(\mathcal{S})} \rho_{ij} \\
&= \Phi - \left(\sum_{i\in\mathcal{S}} f_i + \sum_{j\notin\mathcal{S}} b_j - \sum_{(i,j)\in\delta^+(\mathcal{S})} \rho_{ij}\right)
\end{aligned}$$

The Minimum Cut is an Optimal Segmentation

So: $$\mathrm{cap}(\mathcal{S}) = \Phi - \left(\sum_{i\in\mathcal{S}} f_i + \sum_{j\notin\mathcal{S}} b_j - \sum_{(i,j)\in\delta^+(\mathcal{S})} \rho_{ij}\right)$$

Observe that Φ is a *constant* independent of the cut $\mathcal{S}$.

So the capacity of $\mathcal{S}$ is **minimized** exactly when term in the bracket is **maximized**!

But this is exactly the *segmentation problem*:

$$\max_{\mathcal{S}} \sum_{i\in\mathcal{S}} f_i + \sum_{j\notin\mathcal{S}} b_j - \sum_{(i,j)\in\delta^+(\mathcal{S})} \rho_{ij}$$

To see this is indeed the **segmentation problem** observe that:

- There is a bonus f_i if pixel i is placed in the *foreground*.
- There is a bonus b_j if pixel j is placed in the *background*.
- There is a **penalty** ρ_{ij} for separating adjacent pixels i and j.

So we can solve the segmentation problem by finding a **minimum cut** in the graph!

Flow Decomposition

Recall the **Ford-Fulkerson algorithm**:

Set **f=0**

Repeat

Find an augmenting path *P* w.r.t to **f**

Augment flow on path *P* by *b*(*P*, **f**)

Previously, we asked three questions about the Ford-Fulkerson algorithm:

1) *Does the algorithm actually find a maximum flow?*
2) *Is it efficient?*
3) *What else does it tell us about maximum flow problems?*

We answered all three questions but our answer to the second question was unsatisfactory.

The Running Time

Is the Ford-Fulkerson algorithm **efficient**?

The Ford-Fulkerson algorithm is *pseudo-polynomial time*.

The runtime is *O*(*m* #iterations).

The # iterations could be nU where $U = \max_{a \in A} u_a$

Can we do better?

Which Path?

To answer this question observe that, as stated, the **Ford-Fulkerson algorithm** is not completely specified.

There may be many augmenting paths we can choose in each iteration.

Does it matter which path we select in each iteration?

To answer this question, let's first prove a theorem we encountered earlier in the course…

The Flow Decomposition Theorem

Recall, the union of a set of path and a set of cycles satisfies **flow conservation**.

Moreover, if this union of paths and cycles also satisfies the **capacity constraint** on each arc then they form an *s-t* flow.

We claimed the converse was also true: *every s-t flow is made up of s-t paths and cycles.*

> **The Flow Decomposition Theorem.** Any *s-t* flow can be decomposed into a collection of *s-t* paths and directed cycles.

In fact we actually have something stronger:

> **The Flow Decomposition Theorem.** Any *s-t* flow can be decomposed into a collection of *s-t* paths and directed cycles of cardinality at most m.

To prove the flow decomposition theorem we need the following claim…

Claim. Any *s-t* flow $\mathbf{f \neq 0}$ contains a path or a cycle.

Proof. If $\mathbf{f}$ contains a directed cycle C we are done.

So we may assume $\mathbf{f}$ contains no directed cycles.

$\Longrightarrow$ $\mathbf{f}$ has flow value at least one as $\mathbf{f \neq 0}$.

$\Longrightarrow$ $\sum_{a \in \delta^+(\{s\})} f_a \geq 1$

In particular, there is a least one arc $a_1 = (s, v_1)$ with $f_{a_1} \geq 1$.

But then, *by flow conservation*, there must be an arc in $\mathbf{f}$ leaving v_1.

As $\mathbf{f}$ contains no cycles this arc $a_2 = (v_1, v_2)$ goes to a new vertex v_2.

But then, *by flow conservation*, there must be an arc in $\mathbf{f}$ leaving v_2.

As $\mathbf{f}$ contains no cycles this arc $a_3 = (v_2, v_3)$ goes to a new vertex v_3.

The graph is finite so this process must terminate.

$\Longrightarrow$ The process must terminate at the sink vertex *t*.

$\Longrightarrow$ We have found an *s-t* path P with $\min_{a \in P} f_a \geq 1$.

Thus $\mathbf{f}$ contains an *s-t* path.

Proof of the Flow Decomposition Theorem

The Flow Decomposition Theorem. Any *s-t* flow $\mathbf{f}$ can be decomposed into a collection of *s-t* paths and directed cycles of cardinality at most m.

Proof. By induction on the number of arcs m in the graph.

Base Cases:

If $m=0$ then $\mathbf{f}$ is empty and trivially consists of zero paths and cycles.

If $m=1$ then $\mathbf{f}$ is a simply the arc (s,t).

$\Longrightarrow$ $\mathbf{f}$ can be decomposed into **one** *s-t* path.

Induction Hypothesis: Assume any flow with $k<m$ arcs can be decomposed into a collection of at most k cycles and *s-t* paths.

Induction Step: Take any flow $\mathbf{f}$ with m arcs.

By the Claim, $\mathbf{f}$ contains a path or a cycle.

Case I: Let $\mathbf{f}$ contain a cycle C.

Let $\hat{\mathbf{f}}$ be the flow obtained by removing $\min_{a \in C} f_a$ units of flow from each arc in the cycle C.

But then $\hat{\mathbf{f}}$ has at least one fewer arc than $\mathbf{f}$.

$\Longrightarrow$ By the induction hypothesis, $\hat{\mathbf{f}}$ can be decomposed into at most $m-1$ paths and cycles.

$\Longrightarrow$ $\mathbf{f}$ can be decomposed into at most m paths and cycles.

Case II: Let $\mathbf{f}$ contain a path P.

Let $\hat{\mathbf{f}}$ be the flow obtained by removing $\min_{a \in P} f_a$ units of flow from each arc in the path P.

But then $\hat{\mathbf{f}}$ has at least one fewer arc than $\mathbf{f}$.

$\Longrightarrow$ By the induction hypothesis,$\hat{\mathbf{f}}$ can be decomposed into at most $m-1$ paths and cycles.

$\Longrightarrow$ $\mathbf{f}$ can be decomposed into at most m paths and cycles.

So what?

The Best Choice of Paths

The runtime of the Ford-Fulkerson algorithm is $O(m \cdot \#\text{iterations})$.
But the **Flow Decomposition Theorem** tells us that the maximum flow can be decomposed into at most m paths and cycles.
Moreover, since using a cycle does not increase the flow value, there is a maximum flow $\mathbf{f^*}$ that decomposes into at most m paths.
But if the Ford-Fulkerson algorithm selects, in turn, the paths in this decomposition then the running time will be $O(m^2)$.

A Good Choice of Paths

But we don't know what $\mathbf{f^*}$ is, so we don't know its flow decomposition.

- If we knew the flow decomposition then we wouldn't need to run the Ford-Fulkerson (or any other) algorithm at all to find $\mathbf{f^*}$!

But the **Flow Decomposition Theorem** is still useful to us.

- It tells us that, *in theory*, there is a good choice of paths we could use to give a fast implementation of the Ford-Fulkerson algorithm.
- Even if we can't find this exact set of paths, maybe we can find a different set of paths that is almost as good?

Lecture 5: Network Flows – Fast Algorithms

The Ford-Fulkerson Algorithm

Set **f=0**

Repeat

Find a directed *s-t* path *P* in G_f

Augment flow on path *P* by *b*(*P*, **f**)

But, how we chose the augmenting path P has major consequences for the number of iterations required and hence the running time.
For example, in a generic implementation the # iterations may be nU giving an exponential runtime.

The Flow Decomposition Theorem

The Flow Decomposition Theorem. Any *s-t* flow can be decomposed into a collection of *s-t* paths and directed cycles of cardinality at most *m*.
The Flow Decomposition Theorem tells us that it is possible for the Ford-Fulkerson algorithm to terminate in m iterations.
What happens if we select the paths *greedily?*

The Maximum Capacity Augmenting Path Algorithm

Perhaps the most natural approach is to select the paths greedily by residual capacity:

Set **f=0**

Repeat

Find the maximum capacity augmenting path *P* w.r.t to **f**

Augment flow on path *P* by *b*(*P*, **f**)

Thus, we choose the path that increases the flow value by the largest amount possible at that specific time.

An Efficient Algorithm

The maximum capacity augmenting path algorithm will give us a **(weakly) polynomial time** algorithm for finding a maximum flow.
The proof is almost identical to our proof that the greedy algorithm was an O(log n)–approximation algorithm for the set cover problem.
We begin with a simple observation:

Proof.

Observation 1. Let $\mathbf{f}^*$ be a maximum flow. Then there is a path P in G with capacity at least: $|\mathbf{f}^*| / m$.

Proof.

By the flow decomposition theorem, $\mathbf{f}^*$ consists of at most m paths.

Thus at least one of these paths carries a one mth fraction of the total flow value.

Observation 2. Let $\mathbf{f}$ be a flow and let $\mathbf{f}^*$ be a maximum flow. Then in the Residual Graph G_f there is path P with:

$$b(P, \mathbf{f}) \geq \frac{|\mathbf{f}^*| - |\mathbf{f}|}{m}$$

Proof.

Note that $(\mathbf{f}^* - \mathbf{f})$ satisfies the *flow conservation constraints*.

Thus $(\mathbf{f}^* - \mathbf{f})$ can be decomposed into at most m paths and cycles.*

* Technically $\mathbf{f}^* - \mathbf{f}$ may have $2m$ arcs, but we can only need use one of the two directions of any arc.

Thus we may assume $(\mathbf{f}^* - \mathbf{f})$ can be decomposed into at most m paths.

Thus at least one of these paths carries the a one mth fraction of the difference in their flow values.

The Number of Iterations

Theorem. The maximum capacity augmenting path algorithm terminates in at most $m \cdot (\ln n + \ln U)$ iterations where $U = \max_{a \in A} u_a$.

Proof.

Let the algorithm find the paths $\{P_1, P_2, \ldots, P_T\}$

We want to show that $T \leq m \cdot (\ln n + \ln U)$

Let $\mathbf{f_t}$ be the flow found after t iterations.

By Observation 2, the path P_{t+1} satisfies:

$$b(P_{t+1}, \mathbf{f_t}) \;\geq\; \frac{|\mathbf{f}^*| - |\mathbf{f_t}|}{m} = \frac{\Delta_{t+1}}{m}$$

Here Δ_{t+1} is the flow quantity still to be found at the start of Step $t+1$.

Observe that: $\Delta_1 \;=\; |\mathbf{f}^*| - |\mathbf{f_0}| \;=\; |\mathbf{f}^*| - |\mathbf{0}| \;=\; |\mathbf{f}^*|$

So: $b(P_{t+1}, \mathbf{f_t}) \;\geq\; \dfrac{\Delta_t}{m}$

Thus:

$$\begin{aligned}
\Delta_{t+1} \;&\leq\; \Delta_t - \frac{1}{m} \cdot \Delta_t \\
&=\; \left(1 - \frac{1}{m}\right) \cdot \Delta_t \\
&\leq\; \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{1}{m}\right) \cdot \Delta_{t-1} \\
&\leq\; \underbrace{\left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{1}{m}\right) \cdot \cdots\cdot \left(1 - \frac{1}{m}\right)}_{t \text{ times}} \cdot \Delta_1 \\
&=\; \left(1 - \frac{1}{m}\right)^t \cdot \Delta_1
\end{aligned}$$

So:

$$\Delta_{t+1} \;\leq\; \left(1 - \frac{1}{m}\right)^t \cdot \Delta_1 \;=\; \left(1 - \frac{1}{m}\right)^t \cdot |\mathbf{f}^*|$$

Key Fact. $1 - x \;<\; e^{-x} \quad \forall x \neq 0$

$$\implies\; \Delta_{t+1} \;\leq\; \left(1 - \frac{1}{m}\right)^t \cdot |\mathbf{f}^*| \;<\; \left(e^{-\frac{1}{m}}\right)^t \cdot |\mathbf{f}^*| \;=\; e^{-\frac{t}{m}} \cdot |\mathbf{f}^*|$$

Now setting $t = m \cdot \ln |\mathbf{f}^*|$ gives:

$$\Delta_{t+1} \;<\; e^{-\frac{m \cdot \ln |\mathbf{f}^*|}{m}} \cdot |\mathbf{f}^*| \;=\; e^{-\ln |\mathbf{f}^*|} \cdot |\mathbf{f}^*| \;=\; 1$$

So after $T = m \cdot \ln |\mathbf{f}^*|$ steps the quantity of flow remaining to be found is less than one.

$\implies$ We have found a maximum flow!

Now the maximum flow has value at most nU so

$$\ln |\mathbf{f}^*| \leq \ln n \cdot U = \ln n + \ln U$$

Thus the number if iterations is at most $m \cdot (\ln n + \ln U)$

The Time per Iteration

Theorem. The maximum capacity augmenting path algorithm takes $O(m^2)$ time per iteration.

Proof. It remains to calculate how long it takes to find the maximum capacity augmenting path.

Label the arcs $\{1,2,...,2m\}$ in the residual graph in decreasing order of residual capacity.

- We can test if there is an s-t path using only arcs in $\{1,2,...,k\}$ in $O(m)$ time.
- We can do this for all k in time $O(m^2)$.

The maximum capacity augmenting path is the path we find using the smallest k for which an s-t path exists using only arcs in $\{1,2,...,k\}$.

Using binary search gives a runtime of $O(m \cdot \log m)$ *per step.*

A Weakly Polynomial Time Algorithm

So we have proven:

Theorem. The maximum capacity augmenting path algorithm terminates in at most $m \cdot (\ln n + \ln U)$ iterations.

Theorem. The maximum capacity augmenting path takes time at most $O(m^2)$ per iteration.

Putting this together gives a (weakly) polynomial time algorithm:

Theorem. The maximum capacity augmenting path algorithm runs in time $O(m^3 \cdot (\ln n + \ln U))$

A Strongly Polynomial Time Algorithm

Is it possible to obtain a *strongly** polynomial time algorithm?
Yes if we choose the **shortest length path** in each step!

The Shortest Augmenting Path Algorithm

Set **f=0**
Repeat
    Find the shortest length s-t path *P* in G_f
    Augment flow on path *P* by *b*(*P*, **f**)

Theorem. The shortest length augmenting path algorithm finds a maximum flow in time $O(m^2 \cdot n)$.

The Running Time

The runtime is *O(m #iterations).*
So we must show that the # iterations is *O(mn).*

The Number of Iterations

Theorem. The shortest augmenting path algorithm terminates in at most *mn* iterations.
Proof.
*L*et the algorithm find the paths $\{P_1, P_2, \ldots, P_T\}$
We want to show that $T \leq m \cdot n$
To prove this we use a **potential energy function** argument.
Potential Energy Function Argument: *A rock rolling down a hill will eventually stop at the bottom of the hill.*

The Potential Function Argument

Formally, let **Φ** be a finite function with the following properties:

- It is lower bounded by ℓ.
- It decreases in each time period (by at least a minimum amount **δ**).

Then after a finite amount of time T the function Φ becomes fixed. Moreover, given the starting value Φ_0 and lower bound ℓ, we can easily calculate a maximum time T^* by which the function must be fixed.

Similar arguments apply if the function is non-increasing but must strictly decrease after a fixed number of steps.

We will actually apply a symmetric argument…

Let Φ be a finite function with the following properties:

- It is upper bounded by γ.
- It increases in each time period (by at least a minimum amount δ).

Then after a finite amount of time Φ becomes fixed.

<u>The Potential Function?</u>

But what is the potential function Φ for this network flow algorithm?

Let $\mathbf{f}_\tau$ be the flow found after τ iterations and let G_{f_τ} be its residual graph.

Let $d_\tau(v)$ be the distance of vertex v from the source s in G_{f_τ}.

The *potential (energy) function* is then defined as: $\Phi_\tau = \sum_{v \in V} d_\tau(v)$

Because $d_t(v)$ is a distance it is non-negative so $\Phi_0 = \sum_{v \in V} d_0(v) \geq 0$

As the graph contains n vertices, we have $d_\tau(v) \leq n - 1$.

In particular: $\Phi_\tau = \sum_{v \in V} d_\tau(v) \leq n^2$

*We may define $d_t(v) = n$ if there is no directed path from s to v in G_{f_τ}.

So to apply the potential function method we must prove:

- Φ is non-decreasing.
- Φ strictly increases regularly.

Theorem. The potential function $\Phi_\tau = \sum_{v \in V} d_\tau(v)$ is non-decreasing.
Proof.
It suffices to show that $d_\tau(v)$ is non-decreasing for each vertex v.
If not there is a first time τ and a vertex v such that $d_\tau(v) > d_{\tau+1}(v)$
Let vertex v be such a vertex with the smallest distance label $d_{\tau+1}(v)$
Let vertex u precede v on the shortest path $Q_{\tau+1}(v)$ from s to v in $G_{f_{\tau+1}}$.
$\Longrightarrow \qquad d_{\tau+1}(v) = d_{\tau+1}(u) + 1$
Now vertex u is not a counter-example so $d_\tau(u) \leq d_{\tau+1}(u)$
If $(u,v) \in G_{f_\tau}$ then we have a contradiction as follows:
$d_\tau(v) \leq d_\tau(u) + 1 \leq d_{\tau+1}(u) + 1 = d_{\tau+1}(v)$
So $(u,v) \in G_{f_{\tau+1}}$ but $(u,v) \notin G_{f_\tau}$
This means that $(v,u) \in P_\tau$.
But P_τ is a shortest path in G_{f_τ} so:
$d_\tau(u) = d_\tau(v) + 1 > d_{\tau+1}(v) + 1 = d_{\tau+1}(u) + 2 \geq d_\tau(u) + 2$
This is a contradiction.

Theorem. There are at most *mn* iterations.
Proof.
The algorithm finds the augmenting paths $\{P_1, P_2, \ldots, P_T\}$
We augment the path P_τ by its bottleneck capacity in iteration τ.
In particular there is at least one arc a_τ that gave the bottleneck capacity.
We will prove that each arc can be the bottleneck arc at most n/2 times.
$\Longrightarrow$ The number of iterations is at most $2m \cdot \frac{1}{2}n = mn$
Let $(u,v) \in G_{f_\tau}$ be a bottleneck arc in the augmenting path P_τ.
As P_τ is a shortest *s*-*t* path in G_{f_τ} we have: $d_\tau(v) = d_\tau(u) + 1$
But *(u,v)* is the bottleneck arc in iteration τ thus $(u,v) \notin G_{f_{\tau+1}}$

Suppose *(u,v)* is next a bottleneck arc in iteration $\hat{\tau} > \tau + 1$

$$\Longrightarrow \quad (u,v) \in G_{f_{\hat{\tau}}}$$

But to be in $G_{f_{\hat{\tau}}}$ the arc must be added back into the residual graph in some iteration $\bar{\tau}$ where $\tau + 1 \leq \bar{\tau} < \hat{\tau}$

$$\Longrightarrow \quad (v,u) \in G_{f_{\bar{\tau}}}$$

As $P_{\bar{\tau}}$ is a shortest *s-t* path in $G_{f_{\bar{\tau}}}$ we have:

$$d_{\bar{\tau}}(u) \;=\; d_{\bar{\tau}}(v) + 1 \qquad \geq\; d_{\tau}(v) + 1 \quad =\; d_{\tau}(u) + 2$$

So the distance label of *u* must have increased by at least 2.

This can happen at most *n/2* times otherwise we get $d_{\tau}(u) > n$.

A Strongly Polynomial Time Algorithm

So we have proven:

Theorem. The shortest augmenting path algorithm terminates in at most *mn* iterations.

Theorem. The shortest augmenting path algorithm takes time at most *O(m)* per iteration.

Putting this together gives a strongly polynomial time algorithm:

Theorem. The shortest augmenting path algorithm runs in time $O(m^2n)$.