Augmenting Paths

So how do we find augmenting paths?

Given a flow network G and a flow f, an *augmenting path* p for f is just a path from s to t in G_f.

- By definition, each edge (u, v) in G_f admits some additional positive net flow from u to v.

What's the maximum flow that you can push through an augmenting path p?

- Depends on the edge that admits the least flow.
  - A chain is only as strong as its weakest link
- Define the *residual capacity of* p:

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ on } p\}.$$

Lemma: If G is a flow network, f is a flow in G, and p is an augmenting path in G_f, define

$$f_p = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p \\ -c_f(p) & \text{if } (v,u) \text{ is on } p \\ 0 & \text{otherwise.} \end{cases}$$

Then f_p is a flow in G_f and $|f_p| = c_f(p) > 0$.

Key point $f + f_p$ is a flow in G, and $|f + f_p| = |f| + |f_p| > |f|$.

Cuts in flow networks

We can use the Ford-Fulkerson method by starting with the a flow of 0 on every node, computing an augmenting path, and updating the flow.

- We keep going until there are no more augmenting paths.

We need to prove that we then have the maximum flow.

To prove this, we use cuts:

- Given a flow network $G = (V, E)$, a *cut* consists of a partition S and $T = V - S$ such that $s \in S$ and $t \in T$.
  - like a cut in MST, except that $s \in S$ and $t \in T$, and now the network is directed.

So why do we care about cuts?

Time Out: Working with Flows

It makes life easier if we let the flow take sets as arguments.

$$f(X, Y) = \Sigma_{x \in X} \Sigma_{y \in Y} f(x, y).$$

This simplifies equations:

$f(X, X) = 0$:

- Proof: $f(X, X) = \Sigma_{x,x' \in X} f(x, x') = \frac{1}{2} \Sigma_{x,x' \in X}(f(x, x') + f(x', x)) = 0$

$f(X, Y) = -f(Y, X)$

- Proof: See homework.

If $X \cap Y = \emptyset$, then:
$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$
$f(X, Y \cup Z) = f(X, Y) + f(Y, Z)$

Why we care about cuts

- If f is a flow, the *flow of* f *across the cut* is $f(S, T)$.
- The *capacity* of the cut is $c(S, T)$.

Lemma: If f is a flow in G with source s and sink t, and (S, T) is a cut of G, then $f(S, T) = |f|$.

- The flow of f across the cut = the value of f

Proof:

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) \\ &= f(s,V) + f(S-s,V) \\ &= f(s,V) \\ &= |f| \end{aligned}$$

Corollary: If (S, T) is a cut of G, then $|f| \le c(S, T)$.

Proof:

$$|f| = f(S,T) = \Sigma_{u \in S, v \in T} f(u, v) \le \Sigma_{u \in S, v \in T} c(u, v) = c(S, T).$$

Key point: If $|f| = c(S, T)$ for any cut (S, T), then f must be a maximum flow.

Max-flow min-cut Theorem: If f is a flow in G with source s and sink t, then the following are equivalent:

1. f is a maximum flow
2. G_f contains no augmenting paths
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Proof: (1) $\Rightarrow$ (2): if G_f has an augmenting path p, then $|f| + |f_p| > |f|$, so f can't be a maximum flow.

(2) $\Rightarrow$ (3): Suppose that G_f has no augmenting path. We want to show that $|f| = c(S, T)$ for some cut (S, T). Define

$$S = \{v \in V : \text{ there is a path from } s \text{ to } v \text{ in } G_f\}.$$

Clearly $t \in T = V - S$ (otherwise there would be an augmenting path in G_f). Thus, (S, T) is a cut. If $u \in S$ and $v \in T$, then $f(u, v) = c(u, v)$ (otherwise there would be an edge (u, v) in G_f, and v would be in S). Therefore, $|f| = f(S, T) = c(S, T)$.

(3) $\Rightarrow$ (1): If $|f| = c(S, T)$, we've already seen that f must be a maximum flow.

Key point: If f is a flow in G and G_f has no augmenting paths, then f is a maximum flow in G.

Ford-Fulkerson again

FORD-FULKERSON(G, s, t)

```
1  for each edge (u, v) ∈ E[G]
2        do f[u, v] ← 0
3           f[v, u] ← 0
4  while there exists a path p from s to t in G_f
5        do c_f(p) = min{c_f(u, v) : (u, v) is on p}
6           for each edge (u, v) in p
7               do f[u, v] ← f[u, v] + c_f(p)
8                  f[v, u] ← f[v, u] − c_f(p)
```

Comments:

- Lines 1–3 initialize f
- Don't need to set $f[u, v] \leftarrow 0$ unless one of $(u, v), (v, u)$ is in E, since we we never touch these edges.

Problems:

- How do we check whether there is a path from s to t in G_f
  - Could use, e.g., BFS or DFS.
- Which path do we choose if there is more than one?
- How often do we go through the loop?
- Do we terminate?
  - If capacities are integers, each step gives an improvement of at least one, so we must terminate.
  - This means that the running time is $O(E|f^*|)$, where f^* is the maximum flow.

This is OK if $|f^*|$ is small, can be pretty horrible if it's not:

Can we do better by choosing a better augmenting path?

Edmonds-Karp Algorithm

Use BFS to find the *shortest* augmenting path.

- Each edge counts as 1.

Claim: The Edmonds-Karp algorithm runs in time $O(VE^2)$.

- We'll skip the proof (see pp. 597–598).
- The hard part is showing that using BFS guarantees that we do no more than $O(VE)$ iterations.
- It's easy to see that each iteration takes at most $O(E)$.
  - BFS takes time $O(V + E)$, but $V \leq E - 1$, since each vertex is on a path from s to t (so each vertex other than t must have an outgoing edge).

Can find fancier algoriths that run in time $O(V^3)$ (Section 27.5) and even $O(VE \lg(V^2/E))$ (the current champ).

Bipartite Matching

Consider a graph partitioned into two sets A and B:

- men and women
- task and machine/person to perform it
- lots of other examples

Model this using a *bipartite* graph $G = (V, E)$ where

- $V = A \cup B$
- edges go between nodes in A and nodes in B
  - there is an edge between a job and a machine if the machine can perform the job.
  - One machine can perform several jobs
  - One job can be performed by several machines

A *matching* is a subset M of edges in E such that each vertex has at most one edge in M incident on it.

- Everything is matched wtih at most one other thing.

A *maximum matching* has as many edges as possible.

- As many jobs as possible are done; as many machines as possible are working

Maximum matching and maximum flow

We can construct a flow network that corresponds to a bipartite graph $G = (V, E)$

- Add two vertices: a source s and a sink t.
- Add an edge with capacity 1 from s to every node in A.
- Add an edge with capacity 1 from every node in B to t.
- Give each edge in E capacity 1.

Call the flow network G'

Lemma: If M is a matching in G, then there is an integer-valued flow f in G' with $|f| = |M|$. Conversely, if there is an integer-valued flow f in G', then there is a matching M in G with $|f| = |M|$.

Proof: Suppose that M is a matching. Define a flow f such that if $u \in A$, $v \in B$, and $(u, v) \in M$, then $f(s, u) = f(u, v) = f(v, t) = 1$ and $f(u, s) = f(v, u) = f(t, v) = -1$; $f(u', v') = 0$ otherwise. It is easy to see that $|f| = M$.

Conversely, given f, let

$$M = \{(u, v) : u \in A, v \in B, f(u, v) > 0\}.$$

Why is M a matching?

- For $u \in A$, at most 1 unit of flow comes in (from s), so at most 1 unit can go out (conservation).
- For $v \in B$, at most one unit can go out (to t) so at most one unit can come in.

Why is $|M| = |f|$?

- $(A \cup \{s\}, B \cup \{t\})$ is a cut of G', so
  $|f| = f(A \cup \{s\}, B \cup \{t\}) = \Sigma_{(u,v) \in M} f(u, v) = |M|$.
- Since f is integer-valued and all capacities are at most 1, $f(u, v) = 1$ for $(u, v) \in M$ and $f(u, v) = 0$ for $(u, v) \notin M$. (Can't have $f(u, v) < 0$, since $f(v, u) \le c(v, u) = 0$.)

This means that the size of the maximum matching is the same as the largest value for an integer-valued flow.

- So how do we construct integer-valued flows?
- We get one using Ford Fulkerson!

Lemma: Since all the capacities in G' are integer-valued, the maximum flow in G' is too.

Proof: By induction can show that all the flows in Ford-Fulkerson are integer-valued at every step of the way.

Bottom line: size of maximum matching = value of maximum flow.

There are better methods for maximum bipartite matching:

- Hopcroft and Karp have a $O(\sqrt{V}E)$ algorithm

Dynamic Programming

Dynamic programming is a technique for designing algorithms that's used in *optimization* problems.

- many possible solutions
- each solution has a value (payoff)
- we want to find the optimal solution (the one with the best payoff)

We can apply dynamic programming to optimization problems if, as choices are made, subproblems with a similar structure arise.

Key steps in using dynamic programming:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct the optimal solution from the computed information.

Seems pretty mysterious until you see examples ...

Matrix-chain multiplication

Suppose we want to multiply three matrices: $A_1A_2A_3$. Matrix multiplication is associative, so we have two ways of doing this:

$$(A_1A_2)A_3 \text{ or } A_1(A_2A_3)$$

Both ways give us the same answer. Which is better?

- How much does it cost to multiply an $n \times m$ matrix by an $m \times k$ matrix?
  - $n \times m \times k$ multiplications

Why this can matter:

- Suppose that A_1 is 10×100, A_2 is 100×5, and A_3 is 5×100.
  - A_1A_2 uses $10 \times 100 \times 5 = 5000$ multiplications
  - BA_3 uses $10 \times 5 \times 100 = 5000$ multiplications, where $B = A_1 \times A_2$ (a 10×5 matrix)
    - $(A_1A_2)A_3$ uses 10,000 mults altogether
  - A_2A_3 uses $100 \times 5 \times 100 = 50000$ mults
  - A_1C uses $10 \times 100 \times 100 = 100,000$ mults, where $C = A_2A_3$ (a 100×100 matrix)
    - $A_1(A_2A_3)$ uses 150,000 mults

That's a huge difference!

How Many Choices Are There?

With 2 matrices: 1 choice.

With 3 matrices: 2 choices

$$(A_1A_2)A_3 \text{ or } A_1(A_2A_3)$$

With 4 matrices: 5 chioices

$$(A_1((A_2A_3)A_4))$$
$$(A_1(A_2(A_3A_4)))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1A_2)A_3)A_4)$$
$$((A_1(A_2A_3))A_4)$$

In general, if $P(n)$ is the number of choices with n matrices,

- Choose k; figure out all the ways of grouping $A_1 \dots A_k$ and all the ways of grouping $A_{k+1} \dots A_n$: $P(k)P(n-k)$
- Thus, $P(n) = \Sigma_{k=1}^{n-1} P(k)P(n-k)$.
- It can be shown that $P(n) = \Omega(4^n/n^{3/2})$

Bottom line: $P(n)$ is exponential in n; you can't try all solutions to pick the best one.

Matrix Multiplication with Dynamic Programming

Notation:

- $A_{i..j}$ be the result of multiplying $A_i \dots A_j$.
- A_i is a $p_{i-1} \times p_i$ matrix.
- $m[i,j]$ is the number of multiplications involved in the cheapest algorithm for computing $A_{i..j}$.

Clearly $m[i,i] = 0$.

Claim: If $j > i$, then

$$m[i,j] = \min_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j)$$

Key point:

- This tells us the structure of the optimal solution.
- We get a recursive definition of the optimal solution, obtained by solving similar subproblems.

Could write a naive recursive algorithm based on the claim:

- **Problem:** this still takes exponential time.

A better way:

- Write a table whose entries are $m[i, j]$
  - There are only $n^2/2$ entries in the table.
  - We compute them inductively, starting with all entries where $i - j = 0$, then $i - j = 1$, $i - j = 2, \ldots$