CSCI 270 Lecture 18: The Return of Sequence Alignment

The **edit distance** between two strings is the minimal distance possible between two strings after inserting your choice of spaces. Our goal is to efficiently calculate the edit distance between X and Y.

Define: $SA[i, j]$ is the min cost of aligning strings $x_i x_{i+1} ... x_n$ and $y_j y_{j+1} y_m$.

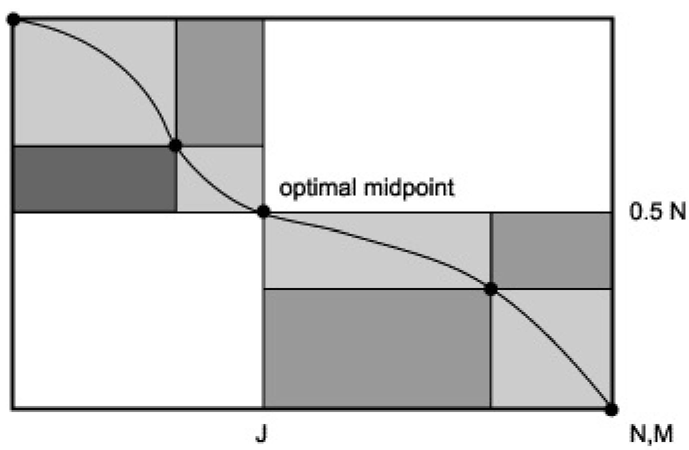
$SA[i, j] = SA[i + 1, j + 1]$, if $x_i = y_j$
$SA[i, j] = 1 + \min(SA[i + 1, j], SA[i, j + 1], SA[i + 1, j + 1])$, if $x_i \neq y_j$.
$SA[i, m + 1] = n - i + 1$
$SA[n + 1, j] = m - j + 1$

1: For all $i = n + 1$ to 1
2: For all $j = m + 1$ to 1
3: Calculate $SA[i, j]$

Runtime: $\theta(mn)$
Space: $\theta(mn)$

To improve the space requirements, we can toss out old columns when they no longer are relevant. We keep the last two columns only, thereby reducing the space requirement to $\theta(m + n)$ (the size of the input, which you can't improve on). The drawback is you cannot reconstruct the answer, which is a deal-breaker for certain applications.

We will instead use Divide and Conquer to get the best of both worlds. The high level idea is to find the optimal midpoint:



Run Sequence Alignment on $X = x_{\frac{n}{2}+1} x_{\frac{n}{2}+2} ... x_n$ and all of Y. We save the first column of data (the column which is calculated last), which tells us the optimal matching of the second half of X with any possible suffix of Y.

We want to run Sequence Alignment on $X = x_1 ... x_{\frac{n}{2}}$ and all of Y to get the other half of the equation. We want to find the optimal matching of the first half of X with any possible prefix of Y.

- If we run this normally, you would find the optimal matching of the first half of X with any possible **suffix** of Y.
- We'll reverse Y so that we're matching a prefix of Y, rather than a suffix of Y.
- This twists everything around, however. The first half of X will be matched with the mirror image of Y.
- We have to reverse X too!

Run Sequence Alignment on $X = x_{\frac{n}{2}} x_{\frac{n}{2}-1} ... x_1$ and $Y = y_m y_{m-1} ... y_1$. Save the first column of data (the column which is calculated last), which tells us the optimal matching of the first half of X with any possible prefix of Y.

Find the optimal midpoint, and recursively repeat the algorithm!

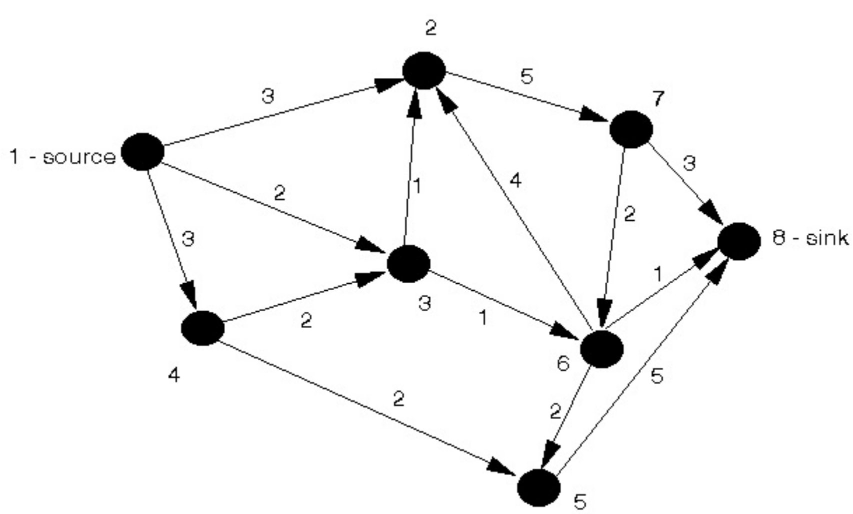
Network Flow

We have a weighted directed graph $G = (V, E)$, where the edge are "pipes" and their weight is their flow capacity. These values measure the rate in which fluid/data/etc can flow through the pipe. What is the maximum rate that flow can be pushed from s to t in the above graph?

Each edge e has capacity $c(e)$. An **s-t flow** is a function f which satisfies:

1. $0 \leq f(e) \leq c(e)$, for all e (capacity).
2. $\sum_{(x,v)} f((x, v)) = \sum_{(v,y)} f((v, y))$, for all nodes $v - \{s, t\}$ (conservation).

The value of a flow is $v(f) = \sum_{(s,x)} f((s, x))$



Minimum Cut

An $s - t$ **cut** is a partition of the nodes into sets $(A, V - A)$, where $s \in A$ and $t \in V - A$.

The **cutset** is the set of edges whose origin is in A and destination is in $V - A$.

The value of an $s - t$ cut is equal to the sum of the capacities of the edges in the cutset.

Problem Statement: Find the minimum cost $s - t$ cut.

1. What is the value of the mincut in the previous graph?
2. Anyone think this is a coincidence?

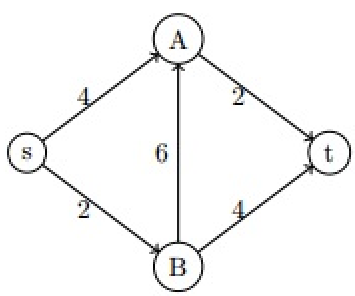
Weak Duality: Given a flow f and an $(A, V - A)$ cut, $v(f) \leq$ the value of the cut.

Corollary to Weak Duality: If $v(f) =$ the value of the cut, then f is a max-flow, and $(A, V - A)$ is a mincut.

The remaining question is, can this always be done? Is there always a flow and cut with equal values?

Let's try to solve the max-flow problem using a greedy algorithm. We'll just choose arbitrary $s - t$ paths that have a remaining capacity, and route as much flow as possible along this path.

Given the original graph G, and the flow so far f, you can construct the residual graph G_f, which contains two types of edges.



- Forward edges (u, v), which have capacity equal to the original capacity $c(u, v)$ minus the flow along the edge $f(u, v)$.
- Backward edges (v, u), which have capacity equal to the flow along the edge in the opposite direction $f(u, v)$.

Forward edges indicate you can still augment the flow along that edge. Backward edges indicate that you can "take back" your choice to push flow along that edge and find a better solution.

Why was this a valid thing to do? Are the two rules of flow maintained if we push flow "backwards"?

The **Ford-Fulkerson** algorithm looks for an **augmenting path** on the residual graph (that is, a path from s to t where you can push more flow), pushes the maximum possible amount of flow along that path, and repeats until there are no more paths.