First Application of Network Flows: Bipartite Matching

- Given a bipartite graph $G = (L \cup R, E)$, a subset of edges $M \subseteq E \subseteq L \times R$ is a matching if each node appears in at most one edge in $M$.
- The maximum matching problem is to find the matching with the most edges.
- We’ll design an efficient algorithm for maximum matching in a bipartite graph.

Formulating it as a network flow problem

- **Goal**: given matching instance $G = (L \cup R, E)$, create a flow network $G'$, find a maximum flow $f$ in $G'$, and use $f$ to construct a maximum matching $M$ in $G$. Exercise.
  - Add a source $s$ and sink $t$
  - For each edge $(u, v) \in E$, add a directed edge from $u$ to $v$ with capacity 1
  - Add an edge with capacity 1 from $s$ to each node $u \in L$
  - Add an edge with capacity 1 from each node $v \in R$ to $t$
  - Run F-F to get an integral max-flow $f$
  - Set $M$ to the set of edges from $L$ to $R$ with flow $f(e) = 1$
  - **Claim**: The set $M$ is a maximum matching.

Analysis

Let’s prove that:

1. Integer flow $f$ in $G'$ $\implies$ matching $M$ in $G$ with $|M| = v(f)$
2. Matching $M$ in $G$ $\implies$ flow $f$ in $G'$ with $v(f) = |M|$

Therefore, max-flow $f$ in $G'$ $\iff$ maximum matching $M$ in $G$

**Proof of 1**: given $f$, construct $M$

- $M =$ edges from $L$ to $R$ carrying one unit of flow
- Capacity constraints $\implies$ at most 1 unit of flow leaving $u \in L$
- Flow values are all 0 or 1 $\implies$ $M$ has at most one edge incident to $u$
- Similar argument for $v \in L$
- Therefore, $M$ is a matching, and clearly $|M| = v(f)$.

**Second Application of Network Flows: Image Segmentation**

- Using an expensive camera and appropriate lenses, you can get a “bokeh” effect on portrait photos in which the background is blurred and the foreground is in focus.

  ![Image Segmentation Example](image.png)

- But using cheap cameras in phones and appropriate software you can fake this effect...
Formulating the problem

**Problem:** given set $V$ of pixels, classify each as foreground or background. Assume you have:
- Numeric “cost” for assigning each pixel foreground/background
- Numeric penalty for assigning neighboring pixels to different classes

**Sketch of approach:** other slides, board work, demo.

Divide-And-Conquer

- Solving recurrences, e.g., $T(n) \leq 2T(n/2) + O(n)$
  - Recursion tree, unrolling
  - “Guess and verify”: proof by induction
  - Master theorem Suppose $T(n) = aT(n/b) + O(n^d)$. Then:
    $$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{d/n}) & \text{if } d < \log_b a \end{cases}$$
- Designing algorithms
  - Often: divide input into equal sized chunks, solve each recursively, combine to solve original problem
  - Can be more subtle—e.g., integer multiplication
  - Tip: don’t think about what happens inside recursion. “Magic”

Sequence Alignment: 2D OPT array $OPT(i, j)$

- Subset Sum: “add a variable”
  $$OPT(j, w) = \max \left\{ \begin{array}{l} OPT(j - 1, w), \\ w_j + OPT(j - 1, w - w_j) \end{array} \right\}$$
  - Running time $O(nW) - nW$ array entries, constant time per entry
- Shortest paths with negative edge weights (Bellman-Ford)
  $$OPT(i, v) = \min \left\{ OPT(i - 1, v), \ min_{w \in V} (c_{v,w} + OPT(i - 1, w)) \right\}$$
  - $O(n^3) - n^2$ array entries, constant time per entry
- Know how to design, analyze DP algorithms. Know about shortest paths in graphs with negative edge weights.

Network Flow

- Problem formulation and definitions
  - Flow network: directed graph, capacities, sources $s$, sink $t$
  - Flow: assign flow $f[e]$ on each edge; capacity and flow conservation constraints
- Ford-Fulkerson
  - Initialize flow $f$ to all zeros
  - Residual graph $G_f$
  - Repeatedly find $s \to t$ path $P$ in $G_f$, use to augment $f$, update $G_f$
  - Stop when no $s \to t$ paths remain in $G_f$
- Analysis
  - Always maintain a flow: use facts of residual graph and augment operation, verify that definition of flow still holds
  - Termination and running time: flow increases by one in each iteration, and cannot exceed total capacity leaving $s$
  - Correctness: Max-Flow Min-Cut Theorem

Dynamic Programming

- Another design technique based on recursion
- Identify recursive structure of problem by writing recurrence for optimal value
- “Turn the crank” to convert recurrence to iterative algorithm
- Weighted interval scheduling
  - Binary choice: $j \in O, j \not\in O$
  - $OPT(j) = \max\{OPT(j - 1), w_j + OPT(p(j))\}$
  - Running time $O(n) - n$ array entries, constant time per entry
- Rod cutting
  - Multi-way choice: position $i \in \{1, \ldots, n\}$ of first cut
  - $OPT(j) = \max_{1 \leq i \leq a} \{p_i + OPT(j - i)\}$
  - Running time $O(n^2) - n$ array entries, $O(n)$ time per entry

Midterm Review

- Sketch of approach: other slides, board work, demo.
Max-Flow Min-Cut Theorem

- $v(f) \leq c(A, B)$ for any flow $f$ and any $s$-$t$ cut $c(A, B)$
- Upon termination, Ford-Fulkerson produces a flow $f$ and cut $(A, B)$ such that $v(f) = c(A, B)$, so $f$ is a max-flow and $(A, B)$ is a min-cut
- The cut $(A, B)$ is found by letting $A$ = set of nodes reachable from $s$ in residual graph

Know content of this week’s discussion. Be able to reason about flows, cuts in specific graphs. Understand principles and implications of Max-Flow Min-Cut Theorem.