Randomization + Approximation: Max-3-Sat

Max-3-Sat. Given a 3-Sat formula, find a truth assignment that satisfies as many clauses as possible
(Note: three distinct variables per clause)

\[ C_1 = x_2 \lor \bar{x}_3 \lor \bar{x}_4 \]
\[ C_2 = x_2 \lor x_3 \lor \bar{x}_4 \]
\[ C_3 = \bar{x}_1 \lor x_2 \lor x_4 \]
\[ C_4 = \bar{x}_1 \lor \bar{x}_2 \lor x_3 \]
\[ C_5 = x_1 \lor \bar{x}_2 \lor \bar{x}_4 \]

Remark: NP-hard search problem

Simple idea. Set each variable true with probability \( \frac{1}{2} \), independently

Randomized Max-3-Sat

For any clause \( C_i \):

\[ \Pr[\text{don't satisfy } C_i] = \left( \frac{1}{2} \right)^3 = \frac{1}{8} \]
\[ \Pr[\text{satisfy } C_i] = \frac{7}{8} \]

Assume \( k \) clauses \( \implies \) expected number of satisfied clauses \( \geq \frac{7}{8}k \)
(linearity of expectation)

Corollary: expected number of clauses satisfied by a random assignment is \( \geq \frac{7}{8}k \) of optimum (since optimum \( \leq k \))

A randomized approximation algorithm (guarantee for expected value)

Probabilistic Method

Prove an object exists by showing that a randomized procedure finds it with nonzero probability.

Corollary: For every 3-Sat instance with \( k \) clauses, there is a truth assignment that satisfies \( \geq \frac{7}{8}k \) clauses.

Proof: Expected number of satisfied clauses is \( \frac{7}{8}k \); a random variable is at least expected value with nonzero probability

Corollary: Every 3-SAT instance with \( \leq 7 \) clauses is satisfiable!

Proof: There is some assignment that satisfies \( \geq \frac{7}{8}k \) clauses. Then

\[ \# \text{ unsatisfied clauses} < \frac{k}{8} \leq \frac{7}{8} < 1 \]

There are no unsatisfied clauses.

Clicker Question

For what number of clauses can we guarantee that a 2-Sat formula is satisfiable?

A. 2 or fewer
B. 3 or fewer
C. 3 or more
D. 4 or fewer

Example:

\( (x_1 \lor x_2) \land (x_1 \lor \bar{x}_2) \land (\bar{x}_1 \lor x_2) \land (\bar{x}_1 \lor \bar{x}_2) \)
### What if We Want to Guarantee to Satisfy $\frac{7}{8}k$ clauses?

**Idea.** Generate random assignments until one satisfies $\geq \frac{7}{8}k$ clauses

How many tries do we need?

**Claim.** The probability a random assignment satisfies $\geq \frac{7}{8}k$ clauses is at least $\frac{1}{8k}$

**Proof.** Use fact that expected value $= \frac{7}{8}k$, algebra

$\implies$ expected number of trials to satisfy $\frac{7}{8}k$ clauses is at most $8k$

**Fact.** Can derandomize $\Rightarrow$ deterministic poly-time algorithm to satisfy $\geq \frac{7}{8}k$ clauses.

**Fact.** No poly-time algorithm can find an assignment satisfying $\geq (\frac{7}{8} + \epsilon)k$ for every satisfiable formula unless $P = NP$.

### What if We Want to Guarantee to Satisfy $\frac{7}{8}k$ clauses?

**Idea.** Generate random assignments until one satisfies $\geq \frac{7}{8}k$ clauses

How many tries do we need?

**Claim.** The probability a random assignment satisfies $\geq \frac{7}{8}k$ clauses is at least $\frac{1}{8k}$

**Proof.** Use fact that expected value $= \frac{7}{8}k$, algebra

$\implies$ expected number of trials to satisfy $\frac{7}{8}k$ clauses is at most $8k$

**Fact.** Can derandomize $\Rightarrow$ deterministic poly-time algorithm to satisfy $\geq \frac{7}{8}k$ clauses.

**Fact.** No poly-time algorithm can find an assignment satisfying $\geq (\frac{7}{8} + \epsilon)k$ for every satisfiable formula unless $P = NP$.

### Monte Carlo vs. Las Vegas Algorithms

**Monte Carlo:**
- **guaranteed:** runs in polynomial time
- **likely:** finds correct answer

Example: Contraction algorithm for global min-cut.

**Las Vegas:**
- **guaranteed:** finds correct answer
- **likely:** runs in polynomial time

Example: Randomized $k^{th}$/median/quicksort, MAX-3-SAT

### Summary

**Topics**
- Randomized, approximation algorithms
- Polynomial-time reductions
- NP-completeness
- Network flows
- Dynamic programming
- Divide-and-conquer
- MST
- Greedy
- Graph algorithms and definitions
- Asymptotic analysis

### Approximation Algorithms

- $\rho$-approximation algorithm
- Runs in polynomial time
- Solves arbitrary instance of the problem
- Guaranteed to find a solution within ratio $\rho$ of optimum:
  - $\frac{\text{value of our solution}}{\text{value of optimum solution}} \leq \rho$

Examples:
- 1.5-approximation for Load Balancing
- 2-approximation for Vertex Cover

### Randomized Algorithms

- Efficient in expectation
- Optimal with high probability
- Show some solution exists, or derive bound on number

Types of randomized algorithms:
- Fail with some small probability (Monte Carlo)
- Always succeed, but running time is random (Las Vegas)

### Polynomial Time Reductions

- We focus on decision problems, e.g., for input $(G, k)$, does there exist a vertex cover with at most $k$ nodes?

- Given two decision problems $X$ and $Y$, $X \leq_P Y$ means that it’s possible to transform an input $I$ of $X$ into an input $f(I)$ of $Y$ in polynomial time such that

  $I$ is a yes instance of $X$ iff $f(I)$ is a yes instance of $Y$

  The transformation is a reduction from $X$ to $Y$.

- We saw examples such as

  - $\text{VertexCover} \leq_P \text{IndependentSet}$
  - $3\text{-SAT} \leq_P \text{IndependentSet}$

- Useful property: If $X \leq_P Y$ and $Y \leq_P Z$ then $X \leq_P Z$. 
NP Completeness

- \( P \) = set of problems you can solve in polynomial time, e.g., minimum spanning tree, matchings, flows, shortest path.
- \( NP \) = set of problems you can verify in polynomial time:
  - If the answer should be yes then there's some extra input (a "witness" or "certificate" or "hint") that you can be given that makes it easy (i.e., in poly time) to check answer is yes
  - If the answer should be "no" then there is no such input.
- \( Y \) is \( NP \)-Complete if \( Y \in NP \) and \( X \leq_P Y \forall X \in NP \).
- Useful Properties: Suppose \( X \leq_P Y \). Then
  - If \( Y \in P \) then \( X \in P \).
  - If \( Y \in NP \) and \( X \) is \( NP \)-complete then \( Y \) is also \( NP \) complete
  - If \( Y \in P \) and \( X \) is \( NP \)-complete then \( P = NP \).
- \( NP \)-Complete problems are in some sense the hardest problems in \( NP \). If you can solve one of them in polynomial time then you prove \( P = NP \). But very few people believe this is possible.

Network Flows

- Flow network
  - Directed graph
  - Source node \( s \) and target node \( t \)
  - Edge capacities \( c(e) \geq 0 \)
- Flow
  - Capacity Constraints: \( 0 \leq f(e) \leq c(e) \) on each edge
  - Flow conservation: for all \( v \notin \{s, t\} \),
    \[
    \sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)
    \]
  - Value \( v(f) \) of flow \( f \) = total flow on edges leaving source
- Max flow problem: find a flow of maximum value
- Residual network encodes how you can change the current flow without violating the capacity constraints.
- Ford Fulkerson Algorithm: Repeatedly increases the flow by finding augmenting paths in the residual network.
### 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

### Dynamic Programming

- Design technique based on recursion. Identify recursive structure by writing recurrence for optimal value.
- The recurrence identifies all subproblems.
- Solve them in a systematic way starting from simplest ones first (base case)

**Example:** Weighted interval scheduling
- \( \text{OPT}(j) = \max\{\text{OPT}(j-1), w_j + \text{OPT}(p(j))\} \)
- \( \text{OPT}(0) = 0 \)
- Compute \( \text{OPT}(j) \) iteratively for \( j = 0 \) to \( n \)
- Running time \( O(n) \)

### Divide-And-Conquer

- Design technique:
  - 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”
- Solving recurrences, e.g., \( T(n) \leq 2T(n/2) + O(n) \)
  - Recursion tree, unrolling
  - “Guess and verify”: proof by induction
  - Master theorem
    - Suppose \( T(n) = aT(n/b) + O(n^d) \).
    - \( T(n) = \begin{cases} 
    O(n^d) & \text{if } d > \log_b a \\
    O(n^d \log n) & \text{if } d = \log_b a \\
    O(n^{d\log a}) & \text{if } d < \log_b a 
    \end{cases} \)

### MST

- Definitions: spanning tree, MST, cut
- Cut property: lightest edge across any cut belongs to every MST
- Prim’s algorithm: maintain a set \( S \) of explored nodes. Add cheapest edge from \( S \) to \( V-S \). Repeat.
- Kruskal’s algorithm: consider edges in order of cost. Add edge if it does not create a cycle.
- Cycle property: most expensive edge in any cycle does not belong to MST

### Greedy Algorithms

- Greedy algorithms are “short sighted” algorithms that take each step based on what looks good in the short term.
  - **Example:** Kruskal’s Algorithm adds lightest edge that doesn’t complete a cycle when building an MST.
  - **Example:** When maximizing the number of non-overlapping TV shows we always added the show that finished earliest out of the remaining shows.
Graph Algorithms: BFS and DFS Trees

- BFS from node $s$:
  - Partitions nodes into layers $L_0 = \{s\}, L_1, L_2, L_3, \ldots$
  - $L_i$ defined as neighbors of nodes in $L_{i-1}$ that aren’t already in $L_0 \cup L_1 \cup \ldots \cup L_{i-1}$.
  - $L_i$ is set of nodes at distance exactly $i$ from $s$
  - Returns tree $T$: for any edge $(u, v)$ in graph, $u$ and $v$ are in the same layer or adjacent layer
  - Can be used to test whether $G$ is bipartite, find shortest path from $s$ to $t$

- DFS from node $s$
  - Returns DFS tree $T$ rooted at $s$
  - For any edge $(u, v)$, $u$ is an ancestor of $v$ in the tree or vice versa.
  - Both run in time $O(m + n)$
  - Both can be used to find connected components of graph, test whether there is a path from $s$ to $t$

Bipartite, Directed Graphs

- An undirected graph $G$ is bipartite if its nodes can be colored red and blue such that no edge has two endpoints of the same color
  - $G$ is bipartite if and only if it does not contain an odd cycle
  - $G$ is bipartite if and only if, after running BFS from any node, there is no edge between two nodes in the same layer
- A directed graph is acyclic (a DAG) if there is no directed cycle
  - There is no directed cycle if and only if there is a topological ordering.
  - Can find a topological order using the fact that a DAG has a node with no incoming edges.

Related “Traversal” Algorithms

Algorithms that grow a set $S$ of explored nodes from starting node $s$

- BFS (traversal): add all nodes $v$ that are neighbors of some node $u \in S$. Repeat.
- Dijkstra (shortest paths): add node $v$ with smallest value of $d(u) + f(u, v)$ for some node $u$ in $S$, where $d(u)$ is distance from $s$ to $u$. Repeat.
- Prim (MST): add node $v$ with smallest value of $c(u, v)$ where $u \in S$. Repeat.

Asymptotic Analysis

Given two positive functions $f(n)$ and $g(n)$:

- $f(n)$ is $O(g(n))$
  - iff $\exists c > 0, n_0 \geq 0$ s.t. $f(n) \leq cg(n)$ for all $n \geq n_0$
- $f(n)$ is $\Omega(g(n))$
  - iff $\exists c > 0, n_0 \geq 0$ s.t. $f(n) \geq cg(n)$ for all $n \geq n_0$
  - iff $g(n)$ is $O(f(n))$
- $f(n)$ is $\Theta(g(n))$
  - iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$
- Know how to apply definitions, compare functions, use to analyze running time of algorithms

Thanks!

Good luck on the final, and please fill out the course survey!
http://owl.umass.edu/partners/courseEvalSurvey/uma/