Randomization + Approximation: Max-3-Sat

3-Sat: Given set of clauses, is there a satisfying truth assignment? What if we can’t satisfy all clauses (constraints)? Do next best

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)

How hard is Max-3-Sat?
Reformulate as decision problem:
Given formula φ and k ∈ N, is there an assignment satisfying ≥ k clauses?
Is this NP-complete? Yes!
Taking k = number of clauses, we obtain 3-SAT!

Simple idea: Flip a coin for each x_i ⇒ set to 1 or 0 (set each variable true with probability 1/2, independently)
How many clauses do we expect to satisfy?

For any clause C_i:
Pr[don’t satisfy C_i] = (1/2)^3 = 1/8
Pr[satisfy C_i] = 7/8
Assume k clauses ⇒ expected number of satisfied clauses ≥ 7/8 k
(linearity of expectation)

Corollary: expected number of clauses satisfied by a random assignment is ≥ 7/8 k of optimum (since optimum ≤ k)
A randomized approximation algorithm (guarantee for expected value)

Clicker Question
Consider a 2-Sat instance with k clauses where each clause has two distinct variables. Suppose each variable is set to true independently with probability 1/2. What is the expected number of satisfied clauses?
A. 1/2 k
B. 1/4 k
C. 3/4 k
D. 7/8 k

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 ≥ 7/8 k clauses.
Proof: Expected number of satisfied clauses is ≥ 7/8 k; a random variable is at least expected value with nonzero probability
An existence proof based on randomization!

Corollary: Every 3-Sat instance with ≤ 7 clauses is satisfiable!
Proof: There is some assignment that satisfies ≥ 7/8 k clauses. Then
# unsatisfied clauses < k/8 < 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:

\[(x_1 \lor x_2) \land (x_1 \lor \bar{x}_2) \land (\bar{x}_1 \lor x_2) \land (\bar{x}_1 \lor \bar{x}_2)\]

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 \(k^{th}/\text{median}/\text{quicksort}, \text{MAX-3-SAT}\)

Given Las Vegas algorithm, place time bound \(\implies\) get Monte Carlo
In general, can’t do the other way around

Algorithmic Complexity

\(f(n) = O(g(n))\) (and \(\Omega, \Theta\)) are relations between functions

Can also see \(O(g(n))\) as a class of functions that grow asymptotically not faster than \(g\)

\(f(n) = O(g(n))\) (upper bound) means
- there exist \(c > 0\) and \(n_0\) s.t. \(f(n) \leq cg(n) \ \forall n \geq n_0\)

Can choose \(c\) and \(n_0\) as needed (arbitrarily large)

\(f(n) = \Omega(g(n))\) (lower bound)
- there exist \(c > 0\) and \(n_0\) s.t. \(f(n) \geq cg(n) \ \forall n \geq n_0\)

*equivalent* to \(g(n) = O(f(n))\)

\(f(n) = \Theta(g(n))\) equivalent to \(f(n) = O(g(n))\) and \(f(n) = \Omega(g(n))\)

Know definitions and apply them

Compare two functions

Analyze running time

How Many Tries to Satisfy \(\frac{x}{k}\) Clauses?

**Claim:** Probability of random assignment satisfying \(\geq \frac{x}{k}\) clauses is \(\geq \frac{1}{x}\)

**Proof:** Let \(p_j = \text{probability that \(j\) clauses are satisfied.}

Group sum by terms \(j < \frac{x}{k}\) and \(j \geq \frac{x}{k}\).

\[
\frac{x}{k} = \sum_{j < \frac{x}{k}} j \cdot p_j + \sum_{j \geq \frac{x}{k}} j \cdot p_j
\]

largest \(j\) in left sum is \(< \frac{x}{k}\)

\[
\leq (\frac{x}{k} - \frac{1}{2}) \sum_{j < \frac{x}{k}} p_j + k \sum_{j \geq \frac{x}{k}} p_j
\]

\[
\leq (\frac{x}{k} - \frac{1}{2}) \cdot 1 + kp_{\text{max}}
\]

Thus, \(p_{\text{max}} \geq \frac{1}{x} \implies \text{expected}\) tries to satisfy \(\frac{x}{k}\) clauses is \(\leq 8k\)

**Fact.** Can **derandomize** \(\Rightarrow\) deterministic poly-time algorithm to satisfy \(\geq \frac{x}{k}\) clauses.

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

Review

- Asymptotic analysis
- Graph algorithms
- Greedy
- Minimum Spanning Trees
- Divide-and-conquer
- Dynamic programming
- Network flows
- Polynomial-time reductions
- NP-completeness
- Randomized, approximation algorithms

Graph Searches: BFS and DFS

- **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 \(s\)
  - Use for: shortest path from \(s\), test bipartiteness

- **DFS from node \(s\):**
  - Recursively call on each unvisited node
  - 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 \(t\)
### Graph Searches: BFS and DFS

- Any search will construct a **tree**
  - tree edge: when first visiting a node from a neighbor
  - tree shape may differ with choice of neighbor order
- **Undirected graphs**
  - DFS has **tree** edges and **back** edges (to indirect ancestor)
  - BFS has **tree** edges and **non-tree** edges (as most ±1 difference)
- **Directed graphs**
  - DFS edges: **tree**, **back** (to ancestor), **cross** (between subtrees) and **forward** (to descendant)
  - BFS non-tree edges:
    - go at most 1 level down, same level, or any level up
  - Cycle detection: use DFS, only **back** edges
  - Detect for directed graphs: mark nodes unvisited/open/closed

### Directed Acyclic Graphs

- DFS has no back edges (only tree, cross and forward edges)
- Topological Ordering / Sorting: iff graph is DAG keep taking nodes with no incoming edges in linear time: \( O(V + E) \)
  - Some algorithms are efficient for special case of DAGs
  - e.g. find longest path (dynamic programming)

### Bipartite Graphs

- An undirected graph \( G \) is bipartite if nodes partitioned in two sets, no edges within each set
  - (color nodes red, blue, no edge with endpoints of same color)
- Two equivalent conditions:
  - \( G \) bipartite if and only if it has no odd cycle
  - \( G \) bipartite if and only no edge within same layer of BFS

### Flavors of Graph Traversal

- 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 \( d(u) + \ell(u,v) \) 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.

### Amortized Analysis

- Often, useful to count total work rather than work per iteration
  - naive analysis of BFS and DFS: \( O(V^2) \), real bound \( O(V + E) \)
  - more complex: Union-Find, negative cycle detection
- Minor data structure changes can improve runtime bound
  - e.g., updating indegree for topological sorting

### Greedy

- Make local choice that seems best now
  - earliest deadline for jobs
  - shortest edge for Kruskal, Prim
  - closest node for Dijkstra
- For problems with **optimal substructure** property
- **Correctness Arguments**
  - Greedy stays ahead
  - Exchange argument (compare to assumed optimum)
    - careful if several optimal solutions
## Minimum Spanning Trees

- 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

## Divide and Conquer

- Divide problem into several parts
- Solve each instance
- Combine solutions to solve original problem

### Recurrences

Unroll (draw recursion tree)

Use Master Theorem:

$$T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a
\end{cases}$$

## Strengthening Assumptions

Solve more than was asked for
- sort-and-count for counting inversions

Return more than was asked for
- crowd increase problem on midterm 2

Avoid recomputations!

## Dynamic Programming

- Overlapping subproblems: avoid recomputing common partial results
- Often: computing optimum: optimal substructure
  - but evaluates multiple choices, unlike greedy
- Binary choice (choose or don’t choose an item)
- $n$-ary choice (multiple options): rod cutting

Adding one more dimension (subset sum, knapsack)

Example: Weighted interval scheduling

- $OPT(j) = \max\{OPT(j - 1), w_j + OPT(p(j))\}$
- $OPT(0) = 0$
- Compute $OPT(j)$ iteratively for $j = 0$ to $n$
- Running time $O(n)$

Pseudopolynomial cases: proportional to one of input values
- actually exponential in number of bits for that input value

## Space-Time Tradeoff

Use more time to save some space

Sometimes, same asymptotic time (more rarely)

Hirschberg sequence alignment, $T(n) = 2T(n/2) + O(n^2) \Rightarrow O(n^2)$

More often: higher time complexity for smaller space

## Network Flows: Ford-Fulkerson

Flow networks directed, source-sink, edge capacities

Maximum flow = minimum cut.

Residual graph for max flow disconnects $s$ from $t$ (cut).

Max flow: forward edges saturated, backward edges have no flow.

Complexity: $O(mnC_{\max})$ (Ford-Fulkerson), $O(m^2n)$ (Edmonds-Karp), $O(mn^2)$ (Dinitz)

Solve: node capacities, node-disjoint paths, edge-disjoint paths, etc.

Maximum bipartite matching: $O(mn)$ time
Augmenting Path

NP Completeness

NP Complete Problems

Polynomial Time Reductions

P and NP / Solver vs. Certifier

New Flow
Finding Reductions

Problems are very close (map to one another)
- SetCover and HittingSet

Problems may be are duals:
- VertexCover and IndependentSet

Sometimes we construct gadgets
- 3-SAT to IndependentSet

A Classification of NP-Complete Problems

Useful Read: Kleinberg & Tardos, Ch. 8.10
Recall:
- Optimization problems (find the min or max number of ...)
- Decision problems (is there solution with \( \leq k \) or \( \geq k \) of ...)

Equivalent in complexity, for a given problem

Satisfiability problems

“Most general”: satisfy all constraints
- Circuit-SAT
- SAT
- 3-SAT

Covering Problems

Achieve some global goal with few elements
- Vertex Cover: cover edges with vertices
- Set Cover: cover entire set with subsets
- Hitting Set: cover subsets with elements
- Dominating Set: cover self and neighbor vertices

Packing Problems

Choose many elements while avoiding conflicts
- Independent Set vertices with no edges
- Set Packing non-intersecting subsets

Polynomial
Matching (edges with no common endpoints)
- Done: case of bipartite graphs (network flow)

Sequencing problems

- Hamiltonian Path (all nodes)
  - reduction to cycle: extra node, connected to all others
- Hamiltonian Cycle
  - reduction to path: split a node, add an endpoint to each half
- Traveling Salesman Problem: minimum-length tour
  - reduce from HAM-CYCLE
<table>
<thead>
<tr>
<th>Numerical Problems</th>
<th>Partitioning / Coloring Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Subset-Sum numbers with precise sum reduce from SAT: construct numbers digit-by-digit</td>
<td>▶ 3-coloring no edge with same-color nodes</td>
</tr>
<tr>
<td></td>
<td>▶ $k$-coloring</td>
</tr>
<tr>
<td></td>
<td><strong>Polynomial</strong></td>
</tr>
<tr>
<td></td>
<td>2-coloring (bipartite graph)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Approximation Algorithms</th>
<th>Randomized Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ $\rho$-approximation algorithm</td>
<td>▶ Efficient in expectation</td>
</tr>
<tr>
<td>▶ Runs in polynomial time</td>
<td>▶ Optimal with high probability</td>
</tr>
<tr>
<td>▶ Solves arbitrary instance of the problem</td>
<td>▶ Break (undesired) symmetry</td>
</tr>
<tr>
<td>▶ Guaranteed to find a solution within ratio $\rho$ of optimum: $\frac{\text{value of our solution}}{\text{value of optimum solution}} \leq \rho$</td>
<td>▶ Show some solution exists, or derive bound on number</td>
</tr>
<tr>
<td>Sometimes non-obvious (spanning tree to get cycle in TSP), both greedy and non-greedy (choose both nodes for vertex cover).</td>
<td>Types of randomized algorithms:</td>
</tr>
<tr>
<td>Examples:</td>
<td>▶ Fail with some small probability (Monte Carlo)</td>
</tr>
<tr>
<td>▶ 1.5-approximation for Load Balancing</td>
<td>▶ Always succeed, but running time is random (Las Vegas)</td>
</tr>
<tr>
<td>▶ 2-approximation for Clustering</td>
<td>Techniques used in proof:</td>
</tr>
<tr>
<td>▶ 2-approximation for Vertex Cover</td>
<td>expected value, union bound, write sum in two ways</td>
</tr>
</tbody>
</table>