COMPSCI 311: Introduction to Algorithms

Lecture 26: Randomized Algorithms. Review

Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon

1 May 2019

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)

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

How hard is MAX-3-SAT ?

Reformulate as decision problem:

Given formula ϕ and $k \in \mathbf{N}$, is there an assignment satisfying $\geq k$ clauses?

Is this NP-complete? Yes!
Taking k = number of clauses, we obtain 3-SAT!
medskip

Simple idea: Flip a coin for each $x_i \implies$ set to 1 or 0
(set each variable true with probability $\frac{1}{2}$, independently)

How many clauses do we expect to satisfy?

Randomized MAX-3-SAT

For any clause C_i:

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

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)

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 $\frac{1}{2}$. What is the expected number of satisfied clauses?

A. $\frac{1}{4}k$

B. $\frac{1}{2}k$

C. $\frac{3}{4}k$

D. $\frac{7}{8}k$

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 $\geq \frac{7}{8}k$ clauses.

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

An **existence proof** based on randomization!

Corollary. Every 3-SAT instance with ≤ 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:

$$(x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$$

How Many Tries to Satisfy $\frac{7}{8}kk$ Clauses?

Claim:
Probability of random assignment satisfying $\geq \frac{7}{8}k$ clauses is $\geq \frac{1}{8k}$

Proof: Let p_j = probability that j clauses are satisfied.
Group sum by terms $j < \frac{7}{8}k$ and $j \geq \frac{7}{8}k$.

$$\frac{7}{8}k = \sum_{j<\frac{7}{8}k} j \cdot p_j + \sum_{j \geq \frac{7}{8}k} j \cdot p_j$$

largest j in left sum is $< \frac{7}{8}k \leq \frac{7k-1}{8}$

$$\leq (\tfrac{7}{8}k - \tfrac{1}{8}) \sum_{j<\frac{7}{8}k} p_j + k \sum_{j \geq \frac{7}{8}k} p_j$$

$$\leq (\tfrac{7}{8}k - \tfrac{1}{8}) \cdot 1 + k p_{\text{suc}}$$

Thus, $p_{\text{suc}} \geq \frac{1}{8k} \implies$ **expected** tries to satisfy $\frac{7}{8}k$ clauses is $\leq 8k$

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

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

Review

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

Algorithmic Complexity

$f(n) = O(g(n))$ (and Ω, Θ) 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

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)

Guess solution ($f(n) \leq c \cdot g(n)$), prove by strong induction

Use Master Theorem: $T(n) = aT(n/b) + O(n^d)$. Then:

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

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

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

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).

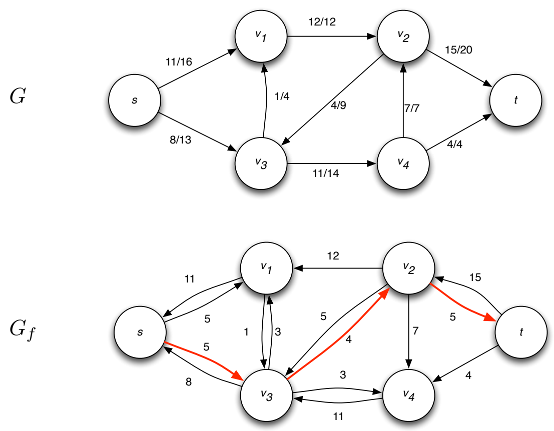
Max flow: forward edges saturated, backward edges have no flow.

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

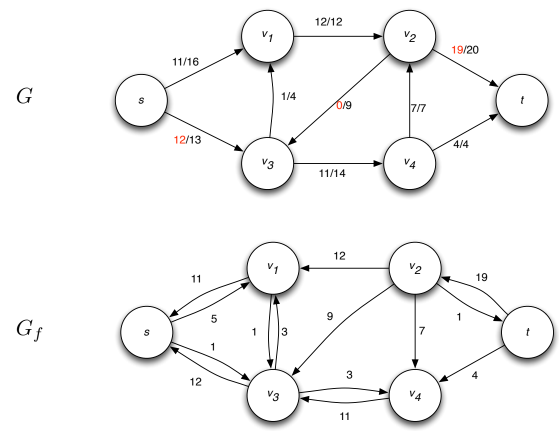
Solve: node capacities, node-disjont paths, edge-disjoint paths, etc.

Maximum bipartite matching: $O(mn)$ time

Augmenting Path



New Flow



Polynomial Time Reductions

- We focus on decision problems, e.g., for input $\langle G, k\rangle$, 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

$$\textsc{VertexCover} \leq_P \textsc{IndependentSet}$$

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

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

P and NP / Solver vs. Certifier

- P: Decision problems with a polynomial time algorithm.
- NP: Decision problems with a polynomial time certifier.

Intuition: A correct solution can be certified in polynomial time.

Let X be a decision problem and s be problem instance (e.g., $s = \langle G, k\rangle$ for $\textsc{Independent Set}$)

Poly-time solver. Algorithm $A(s)$ such that $A(s) = \textsc{Yes}$ iff correct answer is $\textsc{Yes}$, and running time polynomial time in $|s|$

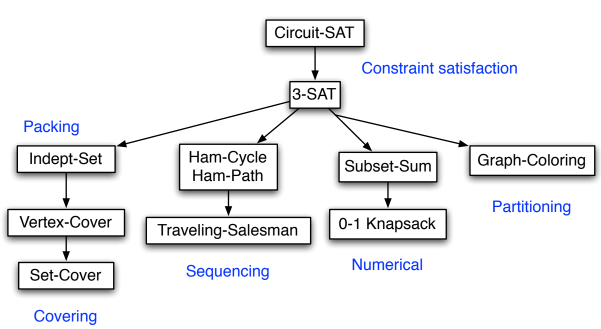
Poly-time certifier. Algorithm $C(s,t)$ such that for every instance s, there is some t such that $C(s,t) = \textsc{Yes}$ iff correct answer is $\textsc{Yes}$, and running time is polynomial in $|s|$.

- t is the "certificate" or hint. Must also be polynomial-size in $|s|$

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 the hardest problems in NP.
  If you can solve one in polynomial time then you prove P=NP
  Generally believed not possible.

NP Complete Problems



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

Numerical Problems

- Subset-Sum   numbers with precise sum

reduce from SAT: construct numbers digit-by-digit

Partitioning / Coloring Problems

- 3-coloring   no edge with same-color nodes
- k-coloring

Polynomial

2-coloring (bipartite graph)

Approximation Algorithms

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

Sometimes non-obvious (spanning tree to get cycle in TSP),
both greedy and non-greedy (choose both nodes for vertex cover).

Examples:

- 1.5-approximation for Load Balancing
- 2-approximation for Clustering
- 2-approximation for Vertex Cover

Randomized Algorithms

- Efficient in expectation
- Optimal with high probability
- Break (undesired) symmetry
- 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)

Techniques used in proof:
expected value, union bound, write sum in two ways