COMPSCI 311: Introduction to Algorithms

Lecture 20: Intractability: Polynomial-Time Reductions

Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon

8 April 2019

Midterm Review: Dijkstra, MST

- 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. $O(m \log n)$ w/ priority queue

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.
  - $O(m \log n)$ with priority queue, like Dijkstra.
  - Kruskal's algorithm: consider edges in order of cost. Add edge if it does not create a cycle.
  - $O(m \log n)$ with union-find data structure

Review: 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:

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

Does not cover everything: gaps between 1 and 2, and 2 and 3

Guess and prove by induction for other 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
  - To prove: inductively assume recursive call works.

Review: Dynamic Programming

- Another design technique based on recursion
- Identify recursive structure of problem by writing recurrence for optimal value (optimal substructure property)
- Memoization or convert recurrence to iterative algorithm
- Weighted interval scheduling
  - Binary choice: $j \in O, j \notin O$
  - $\mathrm{OPT}(j) = \max\{\mathrm{OPT}(j-1), w_j + \mathrm{OPT}(p(j))\}$
  - 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
  - $\mathrm{OPT}(j) = \max_{1 \leq i \leq n} \{p_i + \mathrm{OPT}(j-i)\}$.
  - Running time $O(n^2)$ — n array entries, $O(n)$ per entry

Review: Dynamic Programming

- Sequence Alignment: 2D OPT array $\mathrm{OPT}(i, j)$
- Subset Sum: "add a variable"

$$\mathrm{OPT}(j, w) = \max \left\{ \begin{matrix} \mathrm{OPT}(j-1, w), \\ w_j + \mathrm{OPT}(j-1, w-w_j) \end{matrix} \right\}$$

  * Running time $O(nW)$
  — nW array entries, constant time per entry

- Shortest paths with negative edge weights (Bellman-Ford)

$$\mathrm{OPT}(i, v) = \min \left\{ \mathrm{OPT}(i-1, v), \ \min_{w \in V} \{c_{v,w} + \mathrm{OPT}(i-1, w)\} \right\}$$

  - $O(n^3)$ — n^2 array entries, constant time per entry

- Know how to design, analyze DP algorithms.
  Compute shortest paths in graphs with negative edge weights.

Review: 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

Review: Max-Flow Min-Cut

- 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
  - Forward edges saturated, backward edges have no flow.
  - The cut (A, B) is found by letting $A =$ set of nodes reachable from s in residual graph

Algorithm Design

- Formulate the problem precisely
- Design an algorithm
- Prove correctness
- Analyze running time

Sometimes you can't find an efficient algorithm.

Example: Graph Searches / Network Design

- **Input**: undirected graph $G = (V, E)$ with edge costs
- **Minimum spanning tree problem**: find min-cost subset of edges so there is a path between any $u, v \in V$.
  - $O(m \log n)$ greedy algorithm
- **Minimum Steiner tree problem**: find min-cost subset of edges so there is a path between any $u, v \in W$ for specified set of nodes W (called terminals)
  - No polynomial-time algorithm is known.
  - but: for $W = V$: spanning tree $O(m \log n)$
  - for $W = \{u, v\}$: shortest path $O(m \log n)$

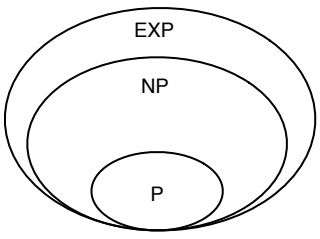
Example: Knapsack Problem

- **Input**: n items with costs and weights, capacity W
- **Goal**: select items to maximize total cost without exceeding W
- **Fractional knapsack**: select fraction in $[0, 1]$ of each item
  - $O(n \log n)$ greedy algorithm
- **0-1 Knapsack**: select all or none of each item
  - $O(nW)$ pseudo-polynomial time algorithm
  - No polynomial time algorithm known!
  - (Also none known for real weights)
- **Subset-Sum Problem** (Knapsack, no values)
  - maximum weight $\leq W$: $O(nW)$ pseudo-polynomial
  - No polynomial time algorithm known
  - same if we want weight sum *equal* to W:

Tractability

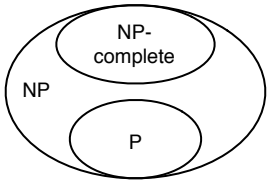
- Working definition of efficient: polynomial time
  - $O(n^d)$ for some d.
- Huge class of natural and interesting problems for which
  - We don't know any polynomial time algorithm
  - We can't prove that none exists
- **Goal**: develop mathematical tools to say when a problem is hard or "intractable"

Preview of Landscape: Classes of Problems



- P: solvable in polynomial time
- NP: includes most problems we don't know about
- EXP: solvable in exponential time

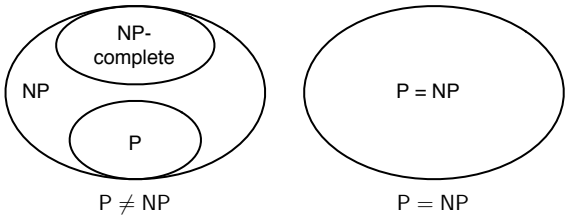
NP-Completeness



- NP-complete: problems that are "as hard as" every other problem in NP.
- A polynomial time algorithm for any NP-complete problem implies one for *every problem in NP*

$P \neq NP$?

Two possibilities:

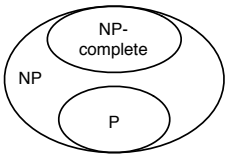


$P \neq NP$ $\quad$ $P = NP$

- We don't know which is true, but think $P \neq NP$
- \$1M prize to find out (Clay Institute Millenium Problems)

Outline

Goal: develop technical tools to make this precise



- **Polynomial-time reductions**: one problem is "as hard as" another (what does this mean?)
- **Define NP**: characterize this class of problems
- **NP-completeness**: some problems in NP are "as hard as" all others

Polynomial-Time Reduction

- Problem Y is **polynomial-time reducible** to Problem X

```
solveY(yInput)
  Construct xInput           // poly-time
  foo = solveX(xInput)       // poly # of calls
  return yes/no based on foo // poly-time
```

- . . . if any instance of Problem Y can be solved using
  1. A polynomial number of standard computational steps
  2. A polynomial number of calls to a black box that solves problem X
- **Notation** $Y \leq_P X$

Clicker Question

Suppose that $Y \leq_P X$. Which of the following can we infer?

A. If Y can be solved in polynomial time, then so can X.

B. If X can be solved in polynomial time, then so can Y.

C. If Y cannot be solved in polynomial time, then neither can X.

D. If X cannot be solved in polynomial time, then neither can Y.

Polynomial-Time Reduction

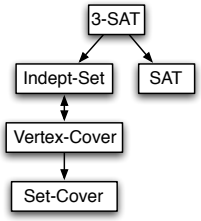
- $Y \leq_P X$

```
solveY(yInput)
  Construct xInput           // poly-time
  foo = solveX(xInput)       // poly # of calls
  return yes/no based on foo // poly-time
```

- Statement about relative hardness
  1. If $Y \leq_P X$ and $X \in P$, then $Y \in P$
  2. If $Y \leq_P X$ and $Y \notin P$ then $X \notin P$
- 1: design algorithms, 2: prove hardness

Preview

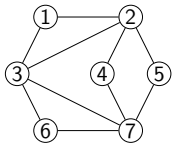
Partial map of problems we can use to solve others in polynomial time, through transitivity of reductions:



- $Y \longrightarrow X$ means $Y \leq_P X$.

First Reduction: Independent Set and Vertex Cover

Given a graph $G = (V, E)$,



- $S \subset V$ is an **independent set** if no nodes in S share an edge. Examples: $\{3, 4, 5\}, \{1, 4, 5, 6\}$.
- $S \subset V$ is a **vertex cover** if every edge has at least one endpoint in S. Examples: $\{1, 2, 6, 7\}, \{2, 3, 7\}$

INDEP-SET. Does G have independent set of size at least k?
VERTEX-COVER. Does G have a vertex cover of size at most k?

Intractability: quiz 3

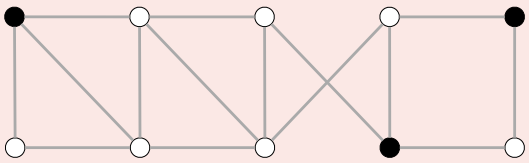
Consider the following graph G. Which are true?

A. The white vertices are a vertex cover of size 7.

B. The black vertices are an independent set of size 3.

C. Both A and B.

D. Neither A nor B.



15

slide credit: Kevin Wayne / Pearson

Independent Set and Vertex Cover

- **Claim**: S is independent set if and only if $V - S$ is vertex cover.

1. S independent set $\Rightarrow$ $V - S$ vertex cover
   - Consider any edge (u, v)
   - S independent $\Rightarrow$ either $u \notin S$ or $v \notin S$
   - i.e., either $u \in V - S$ or $v \in V - S$
   - $\Rightarrow$ $V - S$ is a vertex cover
2. $V - S$ vertex cover $\Rightarrow$ S independent set
   - Similar.

Independent Set $\leq_P$ Vertex Cover

Claim: INDEPENDENT SET $\leq_P$ VERTEX COVER. **Reduction**:

- On INDEPENDENT SET instance $\langle G, k \rangle$
- Construct VERTEX COVER instance $\langle G, n - k \rangle$
- Return YES iff `solveVC`$(\langle G, n - k \rangle) =$ YES

Correctness for YES output:

- Suppose G has independent set S with $\geq k$ nodes
- Then $T = V - S$ is a vertex cover with $\leq n - k$ nodes
- The algorithm correctly outputs YES

Correctness for NO output:

- Suppose G has no independent set S with $\geq k$ nodes
- Then there is no vertex cover with T with $\leq n - k$ nodes, otherwise $S = V - T$ is an independent set with $\geq k$ nodes.
- The algorithm correctly outputs NO

Vertex Cover $\leq_P$ Independent Set

- **Claim**: VERTEX COVER $\leq_P$ INDEPENDENT SET
- **Reduction**:
  - On VERTEX COVER input $\langle G, k \rangle$
  - Construct INDEPENDENT SET input $\langle G, n - k \rangle$
  - Return YES if `solveIS`$(\langle G, n - k \rangle) =$ YES
- **Proof**: similar

Decision versus Optimization

- For intractability and reductions we focus on decision problems (YES/NO answers)
- Algorithms have typically been for optimization (find biggest/smallest)
- Can reduce optimization to decision and vice versa.
- If we can solve MAXINDSET(G) and result is S then INDSET(G, k) has solution iff $k \leq |S|$
- solve MAXINDSET(G) by solving INDSET(G, k), $k = 1, \ldots, n$ or faster by doing binary search

Reduction Strategies

- Reduction by equivalence (Vertex Cover and Independent Set)
- Reduction to a more general case
- Reduction by "gadgets" (e.g., Satisfiability)

Reduction to General Case: Set Cover

Problem. Given a set U of n elements, subsets $S_1, \ldots, S_m \subset U$, and a number k, does there exist a collection of at most k subsets S_i whose union is U?

- Example: $U = \{A, B, C, D, E\}$ is the set of all skills, there are five people with skill sets:

$$S_1 = \{A, C\}, \quad S_2 = \{B, E\}, \quad S_3 = \{A, C, E\}$$

$$S_4 = \{D\}, \quad S_5 = \{B, C, E\}$$

- Find a small team that has all skills. S_1, S_4, S_5

Theorem. VERTEXCOVER $\leq_P$ SETCOVER

Clicker Question

Given the universe $U = \{1, 2, 3, 4, 5, 6, 7\}$ and the following sets, which is the minimum size of a set cover?

A. 2
B. 3
C. 4
D. None of the above

$S_1 = \{1, 4, 6\}$ $S_2 = \{2, 6, 7\}$
$S_3 = \{1, 2, 3, 6\}$ $S_4 = \{1, 3, 5, 7\}$
$S_5 = \{2, 6, 7\}$ $S_6 = \{3, 4, 5\}$

Analysis

- "YES" instance: G has a vertex cover of size $\leq k$
  - U has a set cover of size $\leq k$
  - Output is YES—correct
- "NO" instance: G does not have a vertex cover of size $\leq k$
  - U *does* have a set cover of size $\leq k$ for $k \geq 1$
  - Output is YES—incorrect

–>