COMPSCI 311: Introduction to Algorithms
Lecture 20: Intractability: Intro and Polynomial-Time
Reductions
Dan Sheldon
University of Massachusetts Amherst

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

Midterm Review

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: $j \in O, j \notin O$
  - $OPT(j) = \max\{OPT(j-1), w_j + OPT(p(j))\}$
  - 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
  - $OPT(j) = \max_{i \leq j} \{p_i + OPT(j-i)\}$
  - Running time $O(n^2) - n$ array entries, $O(n)$ time per entry

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

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

Divide-And-Conquer
- "Guess and verify": proof by induction
- 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:

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: $j \in O, j \notin O$
  - $OPT(j) = \max\{OPT(j-1), w_j + OPT(p(j))\}$
  - 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
  - $OPT(j) = \max_{i \leq j} \{p_i + OPT(j-i)\}$
  - Running time $O(n^2) - n$ array entries, $O(n)$ time per entry

- Sequence Alignment: 2D OPT array $OPT(i, j)$
- Subset Sum: “add a variable”
  $$OPT(i, w) = \max \left\{ \begin{array}{l} OPT(j-1, w) \\ w_j + OPT(j-1, w-w_j) \end{array} \right\}$$
  - Running time $O(nW) - nW$ array entries, constant time per entry
- Shortest paths with negative edge weights (Bellman-Ford)
  $$OPT(i, v) = \min_{w \in V} \{c_{v,w} + OPT(i-1, w)\}$$
  - $O(n^3) - n^2$ 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 $s \rightarrow t$ path $P$ in $G_f$, use to augment $f$, update $G_f$.
  - Stop when no $s \rightarrow t$ 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

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
- Be able to reason about flows, cuts in specific graphs.
  Understand principles and implications of Max-Flow Min-Cut Theorem.

Algorithm Design

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

Sometimes you can’t find an efficient algorithm.

Example: 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 terminal set $W$.
  - No polynomial-time algorithm is known.

Example: Subset Sum / Knapsack

- Input: $n$ items with weights, capacity $W$
- Goal: maximize total weight without exceeding $W$
  - $O(nW)$ pseudo-polynomial time algorithm (DP)
  - No polynomial time algorithm known!

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 Lansdscape: Classes of Problems

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

$P \neq NP$?

Two possibilities:

- $P = NP$
- $NP$-complete

▶ We don’t know which is true, but think $P \neq NP$
▶ $1M prize if you can find out (Clay Institute Millenium Problems)

Outline

Goal: develop technical tools to make this precise

- Polynomial-time reductions: what it means for one problem to be “as hard as” another
- Define $NP$: characterize mystery 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 $X$ can be solved in polynomial time, then so can $Y$.
B. If $Y$ cannot be solved in polynomial time, then neither can $X$.
C. Both A and B.
D. Neither A nor B.
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.

Independent Set and Vertex Cover

Claim: S is independent set if and only if V - S is a 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:
1. On Independent Set instance \((G, k)\)
2. Construct Vertex Cover instance \((G, n - k)\)
3. Return Yes if solveVC\((G, n - k)\) = Yes

Correctness for Yes output:
1. Suppose G has independent set S with \(\geq k\) nodes
2. Then T = V - S is a vertex cover with \(\leq n - k\) nodes
3. The algorithm correctly outputs Yes

Correctness for No output:
1. Suppose G has no independent set S with \(\geq k\) nodes
2. 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.
3. The algorithm correctly outputs No
**Vertex Cover \(\leq_P\) Independent Set**

- **Claim:** \(\text{Vertex Cover} \leq_P \text{Independent Set}\)
- **Reduction:**
  - On \(\text{Vertex Cover}\) input \((G,k)\)
  - Construct \(\text{Independent Set}\) input \((G,n-k)\)
  - Return Yes if \(\text{solveIS}(G,n-k) = \text{Yes}\)
- **Proof:** similar

**Aside: Decision versus Optimization**

- For intractability and reductions we will focus on decision problems (Yes/No answers)
- Algorithms have typically been for optimization (find biggest/smallest)
- Can reduce optimization to decision and vice versa. Discuss.

**Reduction Strategies**

- Reduction by equivalence
  - \(\text{Vertex Cover}\) and \(\text{Independent Set}\)
- Reduction to a more general case
- Reduction by “gadgets”

**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:
  - \(S_1 = \{A, C\}\)
  - \(S_2 = \{B, E\}\)
  - \(S_3 = \{A, C, E\}\)
  - \(S_4 = \{D\}\)
  - \(S_5 = \{B, C, E\}\)
- Find a small team that has all skills. \(S_1, S_4, S_5\)

**Theorem.** \(\text{Vertex Cover} \leq_P \text{Set Cover}\)

**Intractability: quiz 4**

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.** 1
- **B.** 2
- **C.** 3
- **D.** None of the above.

\(U = \{1, 2, 3, 4, 5, 6, 7\}\)
- \(S_a = \{1, 4, 6\}\)
- \(S_b = \{1, 6, 7\}\)
- \(S_c = \{1, 2, 3, 6\}\)
- \(S_d = \{1, 3, 5, 7\}\)
- \(S_e = \{2, 6, 7\}\)
- \(S_f = \{3, 4, 5\}\)

**Clicker**

Vertex Cover is a special case of Set Cover with:

- **A.** \(U = V\) and \(S_e =\) the two endpoints of \(e\) for each \(e \in E\).
- **B.** \(U = E\) and \(S_v =\) the set of edges incident to \(v\) for each \(v \in V\).
- **C.** \(U = V \cup E\) and \(S_v =\) the set of neighbors of \(v\) together with edges incident to \(v\) for each \(v \in V\).
Reduction of Vertex Cover to Set Cover

**Theorem.** \textsc{VertexCover} $\leq_p$ \textsc{SetCover}

**Reduction.**
- Given \textsc{Vertex Cover} instance $(G, k)$
- Construct \textsc{Set Cover} instance $(U, S_1, \ldots, S_m, k)$ with $U = E$, and $S_v$ = the set of edges incident to $v$
- Return Yes iff $\text{solveSC}(U, S_1, \ldots, S_m, k) = \text{Yes}$

**Proof**
- Straightforward to see that $S_{v_1}, \ldots, S_{v_\ell}$ is a set cover of size $\ell$ if and only if $v_1, \ldots, v_\ell$ is a vertex cover of size $\ell$
- This implies the algorithm correctly outputs Yes if $G$ has a vertex cover of size $\leq k$ and No otherwise
- Polynomial $\#$ of steps outside of solveSC
- Only one call to solveSC