# CMPSCI 311: Introduction to Algorithms

## Dealing with NP-Completeness

Dan Sheldon

University of Massachusetts

Slides Adapted from Kevin Wayne
Last Compiled: May 3, 2017

## Please Complete Course Survey

`http://owl.oit.umass.edu/partners/courseEvalSurvey/uma/`

## Coping With NP-Completeness

Suppose you want to solve an NP-complete problem? What should you do?

You can't design an algorithm to do *all* of the following:

1. Solve arbitrary instances of the problem
2. Solve problem to optimality
3. Solve problem in polynomial time

Coping strategies

1. Design algorithms for special cases of problem. Independent set in path, tree.
2. Design approximation algorithms or heuristics. Today.
3. Design algorithms that may take exponential time. Not today.

## Approximation Algorithms

- $\rho$-approximation algorithm
  - Runs in polynomial time
  - Solves arbitrary instance of the problem
  - Guaranteed to find a solution within ratio $\rho$ of true optimum

## Greedy Vertex Cover

Here is a 2-approximation for Vertex Cover:

- Pick an arbitrary edge
- Take both endpoints
- Delete all covered edges and repeat

Analysis

- Result is a vertex cover, otherwise there is an edge remaining
- Final result corresponds to a matching: no two edges share an endpoint.
- Suppose the matching has $k$ edges.
- Then *any* cover has size at least $k$, because it must select at least one endpoint of every edge in the matching
- The selected cover has size $2k \Rightarrow$ at most twice the size of the optimal

## Knapsack Problem

- $n$ items, weights $w_i$, values $v_i$, total capacity $W$
- **Goal**: maximize total value without exceeding capacity

| item | value | weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

- Knapsack is NP-complete: $\textsc{Subset-Sum} \leq_P \textsc{Knapsack}$
- We already saw a $O(nW)$ dynamic programming algorithm

## Knapsack Problem Dynamic Programming v2

**Definition**. $\mathsf{OPT}(i, v)$ = min weight of a knapsack for which we can obtain a solution of value $\geq v$ using a subset of items $1, \ldots, i$

$$\mathsf{OPT}(i, v) = \min\{\mathsf{OPT}(i-1, v),\ w_i + \mathsf{OPT}(i-1, v-v_i)\}$$
$$\mathsf{OPT}(i, v) = 0 \quad \text{if } v \leq 0$$
$$\mathsf{OPT}(0, v) = \infty \quad \text{if } v > 0$$

- Running time is $O(nV)$ where $V$ is an upper bound on total value, e.g. $V = nv_{\max}$
- Not polynomial in input size
- Polynomial if values are small integers

## Knapsack Problem Approximation Algorithm

**Idea**: round all values (up) to coarser units and run dynamic programming algorithm

| item | value | weight |
|---|---|---|
| 1 | 934,221 | 1 |
| 2 | 5,956,342 | 2 |
| 3 | 17,810,013 | 5 |
| 4 | 21,217,800 | 6 |
| 5 | 27,734,384 | 7 |

| item | value | weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

The result will never exceed the weight capacity, but may lose some value due to rounding error.

## Rounding Details

Round to nearest multiple of $\theta$:

- $v_i$ = original value. 5,956,342
- $\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta$ = rounded value. 6,000,000
- $\left\lceil \frac{v_i}{\theta} \right\rceil$ = value actually used in DP. 6

## Analysis

- Let $S$ = rounded DP solution, $S^*$ = optimal solution
- We'll show that $S$ gets nearly as much value as $S^*$:

$$\begin{aligned}
\sum_{i \in S^*} v_i &\leq \sum_{i \in S^*} \bar{v}_i && \text{round up} \\
&\leq \sum_{i \in S} \bar{v}_i && S \text{ optimal for rounded values} \\
&\leq \sum_{i \in S} (v_i + \theta) && \text{rounding amount} \leq \theta \\
&\leq \sum_{i \in S} v_i + n\theta \\
&\leq (1+\epsilon) \sum_{i \in S} v_i
\end{aligned}$$

The last inequality is true if $n\theta \leq \epsilon \sum_{i \in S} v_i$. Set $\theta = \epsilon v_{\max}/n$.

## Running Time

- Recall $\theta = \epsilon v_{\max}/n$.
- Value of a single item in rounded DP is:

$$\left\lceil \frac{v_i}{\theta} \right\rceil = \left\lceil \frac{n v_i}{\epsilon v_{\max}} \right\rceil \leq \frac{n}{\epsilon}$$

- $V = n^2/\epsilon$ is an upper bound on total value
- Running time is $O(nV) = O(n^3/\epsilon)$

**Theorem**: For any $\epsilon > 0$, the rounding algorithm computes a solution whose value is within a $(1+\epsilon)$ factor of the optimum in $O(n^3/\epsilon)$ time.

## A Real-World Application

- First, an extension to the Knapsack problem
- Suppose there are $k$ different actions available for each item (e.g., type of repair to apply to infrastructure element)
  - $w_{ij}$ = cost of action $j$ for item $i$
  - $v_{ij}$ = value of action $j$ for item $i$

$$\mathsf{OPT}(i, v) = \min_j \{w_{ij} + \mathsf{OPT}(i-1, v-v_{ij})\}$$

- Still $O(nV)$

## A Real-World Application

- Now, suppose the item values are not independent, but ordered from 1 to $n$ such that the value from $\{1, \ldots, i\}$ depends on value from $\{1, \ldots, i-1\}$ and action taken at $i$

$$\text{OPT}(i, v) = \min_j \{w_{ij} + \text{OPT}(i-1, f_{ij}(v))\}$$

- $f_{ij}(v)$ = value needed from $\{1, \ldots, i-1\}$ to realize $v$ units of value, assuming action $j$ taken, e.g.:
  - $w_{ij}$ = cash fee for investment type $j$
  - $f_{ij}(v)$ = starting account value needed to get balance of $v$ dollars in next time step if take action $j$
- Still $O(nV)$
- Interpretation: $\text{OPT}(i, v)$ = ROI curve.

## A Real-World Application

- Can be extended further, to case where there is a rooted tree on items
- Let $T_i$ be subtree rooted at $i$
- $\text{val}(T_i)$ depends on $\{\text{val}(T_\ell) : \ell \text{ is a child of } i\}$, action at $i$
- Barrier-removal in river networks.

## Summary

Topics

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

## Polynomial Time Reductions

- We focus on decision problems, e.g., for input $(G, k)$, the question is 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 $X \leq_P Z$.

## 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 the 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 doesn't such an 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

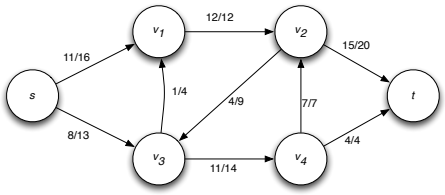
- Flow network
  - Directed graph
  - Source node $s$ and target node $t$
  - Edge capacities $c(e) \geq 0$
- Flow
  - Capacity Constraints: $0 \leq f(e) \leq c(e)$ on each edge
  - Flow conservation: for all $v \notin \{s, t\}$,

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

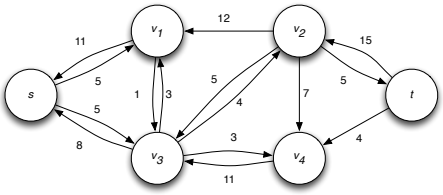
  - Value $v(f)$ of flow $f$ = total flow on edges leaving source
  - **Max flow problem**: find a flow of maximum value
- Residual network encodes how you can change the current flow without violating the capacity constraints.
- Ford Fulkerson Algorithm: Repeatedly increases the flow by finding augmenting paths in the residual network.

## Augment Example

$G$

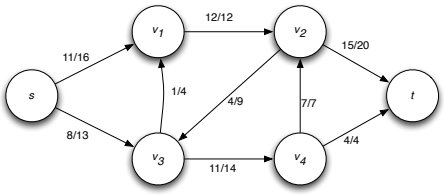



$G_f$

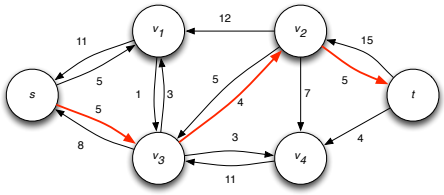



## Augmenting Path

$G$

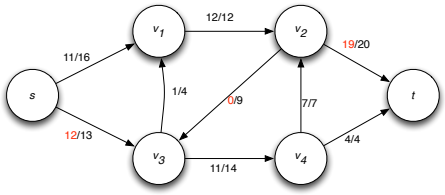



$G_f$

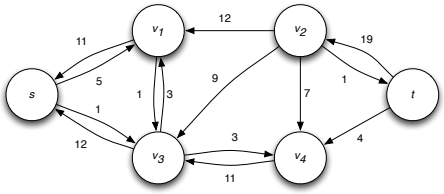



## New Flow

$G$




$G_f$




## 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

- $\mathsf{OPT}(j) = \max\{\mathsf{OPT}(j-1), w_j + \mathsf{OPT}(p(j))\}$
- $\mathsf{OPT}(0) = 0$
- Compute $\mathsf{OPT}(j)$ 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 $T(n) = aT(n/b) + O(n^d)$. Then:

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

## 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.

## 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.

## Greedy Algorithms

- Things to note:
  - If a greedy algorithm requires first sorting the input, remember to include the running time of sorting in your overall analysis.
  - Focus on correctness proofs: "greedy stays ahead", "exchange argument", induction, contradiction
- What to know:
  - Apply/adapt proof techniques for scheduling problems; solve similar problems
  - Working knowledge of MST algorithms, Dijkstra: apply to concrete examples, understand principles and proof techniques

## 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 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 $t$

## 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 $d(u) + \ell(u, v)$ 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.

## 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.

## Asymptotic Analysis

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

- $f(n)$ is $O(g(n))$
  - if any only if $\exists c \geq 0, n_0 \geq 0$ s.t. $f(n) \leq cg(n)$ for all $n \geq n_0$
- $f(n)$ is $\Omega(g(n))$
  - if and only if $\exists c \geq 0, n_0 \geq 0$ s.t. $f(n) \geq cg(n)$ for all $n \geq n_0$
  - if and only if $g(n)$ is $O(f(n))$
- $f(n)$ is $\Theta(g(n))$
  - if and only if $f(n)$ 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