CS38
Introduction to Algorithms
Lecture 17
May 27, 2014

Outline
• coping with intractibility
  – NP-completeness
  – special cases
  – fixed parameter complexity
  – approximation algorithms

Hardness and completeness
• Reasonable that can efficiently transform one problem into another.
• Surprising:
  – can often find a special language L so that every language in a given complexity class reduces to L!
  – powerful tool

Definition: a language L is C-hard if for every language A ∈ C, A poly-time reduces to L; i.e., A ≤_P L.
meaning: L is at least as “hard” as anything in C

Lots of NP-complete problems
• logic problems
  – 3-SAT = {φ : φ is a satisfiable 3-CNF formula}
  – NAE3SAT, (3,3)-SAT
  – Max-2-SAT
• finding objects in graphs
  – independent set
  – vertex cover
  – clique
• sequencing
  – Hamilton Path
  – Hamilton Cycle and TSP
• problems on numbers
  – subset sum
  – knapsack
  – partition
• splitting things up
  – max cut
  – min/max bisection
Example: Integer programming

**Definition:** Integer Linear Program (ILP) = {LPs with integer variables that have a feasible solution}

**Theorem:** ILP is NP-complete.
- Proof:
  - Part 1: ILP ∈ NP. Proof? (try just for 0/1)
  - Part 2: ILP is NP-hard.
    - reduce from?

May 27, 2014 CS38 Lecture 17

---

Integer programming

- \( \varphi = (x \lor y \lor \neg z) \land (\neg x \lor w \lor z) \land \ldots \land (\ldots) \)
- ILP variable \( x \) for each Boolean variable \( x \)
- \( 0 \leq x \leq 1 \)
- represent \( \neg x \) by \( (1 - x) \)
- each clause has a natural linear expression:
  - e.g. \( (x \lor y \lor \neg z) \rightarrow (x + y + (1 - z)) \)
- constrain each such expression to be \( \geq 1 \)

is this reduction polynomial time?

YES maps to YES? NO maps to NO?

May 27, 2014 CS38 Lecture 17

---

Coping with intractability

- NP-complete problem cannot have a polynomial-time algorithm, unless \( P = NP \)
- considered unlikely

NP-complete problems are everywhere!

we need strategies to deal with them
Coping with intractability

- Strategies for coping with intractability
  - consider special case or more restrictive version of the problem
  - parameterized complexity
    - problem size n, parameter k
    - find $O(\exp(k) \cdot \text{poly}(n))$ instead of $O(n^k)$ algorithm
  - approximation algorithms: for optimization problems, find an approximate solution
  - heuristics...

Special case example

Independent set on trees

Given a tree, find a maximum cardinality subset of nodes such that no two are adjacent.

**Fact.** A tree has at least one node that is a leaf (degree = 1).

**Key observation.** If node v is a leaf, there exists a max cardinality independent set containing v.

**Pf.** (exchange argument)
- Consider a max cardinality independent set $S$.
- If $v \in S$, we’re done.
- Let $(u, v)$ be some edge.
  - If $u \in S$ and $v \notin S$, then $S \cup \{v\}$ is independent $\Rightarrow$ $S$ not maximum
  - If $u \notin S$ and $v \in S$, then $S \cup \{v\} - \{u\}$ is independent

Independent set on trees: greedy algorithm

**Theorem.** The following greedy algorithm finds a max cardinality independent set in forests (and hence trees).

**Pf.** Correctness follows from the previous key observation.

```
INDEPENDENT-SET-IN-A-FOREST (F)
S ← ∅
WHILE (F has at least 1 edge)
    φ ← (u, v) such that v is a leaf.
    S ← S ∪ {v}, delete u and v and all incident edges.
RETURN S.
```

**Remark.** Can implement in $O(n)$ time by considering nodes in postorder.

Weighted independent set on trees

Given a tree and node weights $w > 0$, find an independent set $S$ that maximizes $\sum_{v \in S} w_v$.

**Dynamic programming solution.** Root tree at some node, say $r$.
- $OPT_r(u) = \text{max}$ weight independent set of subtree rooted at $u$, containing $u$.
- $OPT_{\text{root}}(u) = \text{max}$ weight independent set of subtree rooted at $u$, not containing $u$.
- $OPT = \max \{ OPT_r(r), OPT_{\text{root}}(r) \}$.

```
OPT_r(u) = \begin{cases} 
    w_u + \sum_{v \in \text{children}(u)} OPT_v(v) & \text{if } u \text{ is a leaf} \\
    \max \{ OPT_v(v), OPT_{\text{root}}(v) \} & \text{otherwise}
\end{cases}
```

Weighted independent set on trees: dynamic programming algorithm

**Theorem.** The dynamic programming algorithm finds a max weighted independent set in a tree in $O(n)$ time.

```
WEIGHTED-INDEPENDENT-SET-IN-A-TREE (T)
Root the tree at node $r$. 
S ← ∅.
FOREACH (node $u$ of $T$ in postorder)
    IF (u is a leaf) 
        $M[u] = w_u$, ensures a node is visited after all its children
        $M[u] = 0$.
    ELSE
        $M[u] = w_u + \sum_{v \text{in } \text{children}(u)} \max \{ M[v], M_{\text{root}}(v) \}$.
RETURN $\max \{ M[r], M_{\text{root}}[r] \}$.
```

Can also find independent set itself (not just value)
NP-hard problems on trees: context

Independent set on trees. Tractable because we can find a node that breaks the communication among the subproblems in different subtrees.

Linear-time on trees. VERTEX-COVER, DOMINATING-SET, GRAPH-ISOMORPHISM, ...

Parameterized complexity example

Finding small vertex covers

Q. VERTEX-COVER is NP-complete. But what if \( k \) is small?

Brute force. \( O(k \cdot n^k) \).

- Try all \( O(n \cdot k^k) \) subsets of size \( k \).
- Takes \( O(k \cdot n) \) time to check whether a subset is a vertex cover.

Goal. Limit to exponential dependency on \( k \), say to \( O(2^k \cdot k \cdot n) \).

Ex. \( n = 1,000, k = 10 \).

Brute. \( 2^{10} \cdot 10^2 \) is infeasible.

Better. \( 2^k \cdot k \cdot n \) is feasible.

Remark. If \( k \) is a constant, then the algorithm is poly-time; if \( k \) is a small constant, then it’s also practical.

Finding small vertex covers: algorithm

Claim. The following algorithm determines if \( G \) has a vertex cover of size \( \leq k \) in \( O(2^k \cdot k \cdot n) \) time.

Pf. Correctness follows from previous two claims.

- There are \( 2^k \) nodes in the recursion tree; each invocation takes \( O(n) \) time.

```
Vertex-Cover(G, k) {
    if (G contains no edges) return true
    if (G contains \( \geq k \) edges) return false
    let \((u, v)\) be any edge of \( G \)
    a = Vertex-Cover(G - \{u\}, k-1)
    b = Vertex-Cover(G - \{v\}, k-1)
    return a or b
}
```
Finding small vertex covers: recursion tree

\[
T(n, k) = \begin{cases} 1 & \text{if } k = 0 \\ \min \{ c n, \overline{T(n, k-1)} \} & \text{if } k = 1 \\ \overline{T(n, k-2)} & \text{if } k > 1 \end{cases}
\]

Optimization Problems

- many hard problems (especially \textbf{NP}-hard) are optimization problems
  - e.g. find \textit{shortest} TSP tour
  - e.g. find \textit{smallest} vertex cover
  - e.g. find \textit{largest} clique

- may be minimization or maximization problem
- “OPT” = value of optimal solution

Approximation Algorithms

- often happy with approximately optimal solution
  - warning: lots of heuristics
  - we want approximation algorithm with guaranteed approximation ratio of \( r \)
  - meaning: on every input \( x \), output is guaranteed to have value
    at most \( r \cdot \text{OPT} \) for minimization
    at least \( \frac{\text{OPT}}{r} \) for maximization

Approximation Algorithm for VC:

- Example approximation algorithm:
  \textbf{Vertex Cover (VC)}: given a graph \( G \), what is the \textit{smallest} subset of vertices that touch every edge?

  \textbf{Theorem}: decision version of VC is \textbf{NP}-complete

  \textbf{Proof}: in \textbf{NP} (why?)
  - reduce from?

Approximation Algorithm for VC:

- Approximation algorithm for VC:
  - pick an edge \((x, y)\), add vertices \( x \) and \( y \) to VC
  - discard edges incident to \( x \) or \( y \); repeat.

- Claim: approximation ratio is 2.

- Proof:
  - an optimal VC must include at least one endpoint of each edge considered
  - therefore \( 2 \cdot \text{OPT} \geq \text{actual} \)
Weighted vertex cover

Given a graph \( G = (V, E) \) with vertex weights \( w_i \geq 0 \), find a min weight subset of vertices \( S \subseteq V \) such that every edge is incident to at least one vertex in \( S \).

\[
\text{total weight} = 6 + 23 + 7 + 9 + 10 = 55
\]

Weighted vertex cover: IP formulation

Integer programming formulation.

- Model inclusion of each vertex \( i \) using a 0/1 variable \( x_i \).
  
  \[
x_i = \begin{cases} 
0 & \text{if vertex } i \text{ is not in vertex cover} \\
1 & \text{if vertex } i \text{ is in vertex cover}
\end{cases}
\]

Vertex covers in 1-1 correspondence with 0/1 assignments: \( S = \{ i \in V : x_i = 1 \} \).

- Objective function: maximize \( \sum w_i x_i \).
- Must take either vertex \( i \) or \( j \) (or both): \( x_i + x_j \geq 1 \).

Observation. If \( x^* \) is optimal solution to (ILP), then \( S = \{ i \in V : x^*_i = 1 \} \) is a min weight vertex cover.

Linear programming

Given integers \( a_{ij}, b_i, \) and \( c_j \), find real numbers \( x_j \) that satisfy:

\[
(P) \quad \max c^T x \\
\text{s.t.} \quad \sum_{i \in \mathcal{I}} a_{ij} x_i = b_j \\
\quad x \geq 0
\]


LP feasible region

LP geometry in 2D.

The region satisfying the inequalities:

\[
\begin{align*}
2x_1 + 3x_2 & \leq 6 \\
3x_1 + 2x_2 & \leq 6 \\
x_1 + 2x_2 & \leq 6 \\
x_1 \geq 0, x_2 \geq 0
\end{align*}
\]
Weighted vertex cover: LP relaxation

Linear programming relaxation.

\[(LP) \min \sum_{i \in V} w_i x_i \]
subject to:
\[x_i \geq 0 \quad \forall i \in V \]
\[x_i + x_j \geq 1 \quad \forall (i, j) \in E \]
\[x_i \leq 1 \quad \forall i \in V \]

Observation. Optimal value of (LP) is optimal value of (ILP).

Pf. LP has fewer constraints.

Note. LP is not equivalent to vertex cover.

Q. How can solving LP help us find a small vertex cover?
A. Solve LP and round fractional values.

Weighted vertex cover: LP rounding algorithm

Lemma. If \(x^*\) is optimal solution to (LP), then \(S = \{i \in V : x_i^* \geq 1/2\}\) is a vertex cover whose weight is at most twice the min possible weight.

Pf. [\(S\) is a vertex cover]
- Consider an edge \((i, j) \in E\).
- Since \(x_i^* + x_j^* \geq 1\), either \(x_i^* \geq 1/2\) or \(x_j^* \geq 1/2\) \(\Rightarrow\) \((i, j)\) covered.

Pf. [\(S\) has desired cost]
- Let \(S^*\) be optimal vertex cover. Then

\[\sum_{i \in S^*} w_i \geq \frac{1}{2} \sum_{i \in S} w_i \]

LP is a relaxation
\[\sum_{i \in S} x_i^* \leq \frac{1}{2} \sum_{i \in S} w_i\]

Theorem. The rounding algorithm is a 2-approximation algorithm.

Pf. Lemma + fact that LP can be solved in poly-time.

Approximation Algorithms

- diverse array of ratios achievable
- some examples:
  - (min) Vertex Cover: 2
  - MAX-3-SAT (satisfy max # clauses): 8/7
  - (min) Set Cover: \(\ln n\)
  - (max) Clique: \(n/\log^2 n\)
  - (max) Knapsack: \((1 + \varepsilon)\) for any \(\varepsilon > 0\)
- many known to be "correct" unless \(P = NP\)

Knapsack problem

Knapsack problem.
- Given \(n\) objects and a knapsack.
- Item \(i\) has value \(v_i \geq 0\) and weighs \(w_i \geq 0\). \(-\cdot\) we assume \(w_i \leq W\) for each \(i\)
- Knapsack has weight limit \(W\).
- Goal: fill knapsack so as to maximize total value.

Ex: \([3, 4]\) has value 40.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

original instance \((W = 11)\)

Knapsack is NP-complete

**Knapsack**. Given a set \(X\), weights \(w_i \geq 0\), values \(v_i \geq 0\), a weight limit \(W\), and a target value \(V\), is there a subset \(S \subseteq X\) such that:

\[\sum_{i \in S} w_i \leq W\]
\[\sum_{i \in S} v_i \geq V\]

**Subset-Sum**. Given a set \(X\), values \(w_i \geq 0\), and an integer \(U\), is there a subset \(S \subseteq X\) whose elements sum to exactly \(U\)?

**Theorem.** Subset-Sum \(\leq\) Knapsack.

Pf. Given instance \((w_1, \ldots, w_n, U)\) of Subset-Sum, create Knapsack instance:

\[\forall i \quad w_i = u_i\]
\[V = W = U\]
\[\sum_{i \in S} v_i \leq U\]
Knapsack problem: dynamic programming I

Def. $\text{OPT}(i, w) = \max$ value subset of items $1, \ldots, i$ with weight limit $w$.

Case 1. $\text{OPT}$ does not select item $i$.
* $\text{OPT}$ selects best of $1, \ldots, i-1$ using up to weight limit $w$.

Case 2. $\text{OPT}$ selects item $i$.
* New weight limit $w - w_i$.
* $\text{OPT}$ selects best of $1, \ldots, i-1$ using up to weight limit $w - w_i$.

Theorem. Computes the optimal value in $O(nW)$ time.
* Not polynomial in input size.
* Polynomial in input size if weights are small integers.

Knapsack problem: dynamic programming II

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

Case 1. $\text{OPT}$ does not select item $i$.
* $\text{OPT}$ selects best of $1, \ldots, i-1$ that achieves value $v$.

Case 2. $\text{OPT}$ selects item $i$.
* Consumes weight $w_i$, need to achieve value $v - v_i$.
* $\text{OPT}$ selects best of $1, \ldots, i-1$ that achieves value $v - v_i$.

Theorem. Dynamic programming algorithm II computes the optimal value in $O(n^2v_{\text{max}})$ time, where $v_{\text{max}}$ is the maximum of any value.
* The optimal value $V^* \leq n v_{\text{max}}$.
* There is one subproblem for each item and for each value $v \leq V^*$.
* It takes $O(1)$ time per subproblem.

Remark 1. Not polynomial in input size!
Remark 2. Polynomial time if values are small integers.

Knapsack problem: polynomial-time approximation scheme

Intuition for approximation algorithm.
* Round all values up to lie in smaller range.
* Run dynamic programming algorithm II on rounded instance.
* Return optimal items in rounded instance.

Round up all values:
* $v_{\text{max}} = \text{largest value in original instance}$.
* $\epsilon = \text{precision parameter}$.
* $\theta = \text{scaling factor} = \epsilon v_{\text{max}} / n$.

Observation. Optimal solutions to problem with $\Psi$ are equivalent to optimal solutions to problem with $\Phi$.

Intuition. $\Psi$ close to $v$ so optimal solution using $\Psi'$ is nearly optimal; $\Phi$ small and integral so dynamic programming algorithm II is fast.

Knapsack problem: polynomial-time approximation scheme

Round up all values:
* $v_{\text{max}} = \text{largest value in original instance}$.
* $\epsilon = \text{precision parameter}$.
* $\theta = \text{scaling factor} = \epsilon v_{\text{max}} / n$.

Theorem. If $S$ is solution found by rounding algorithm and $S'$ is any other feasible solution, then
* $\sum_{i \in S'} v_i = \sum_{i \in S} v_i$ always rounded up.
* $\sum_{i \in S'} v_i = \text{solve rounded instance optimally}$.
* $\sum_{i \in S'} (v_i + \theta)$ never rounded up by more than $\theta$.
* $\sum_{i \in S'} v_i + \theta \geq \max \text{OPT}(S)$ if $|S| \leq \theta$.
* $\sum_{i \in S'} v_i + \theta \geq \max \text{OPT}(S)$ if $|S| > \theta$.

Pf. Let $S'$ be any feasible solution satisfying weight constraint.
Knapsack problem: polynomial-time approximation scheme

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

Proof.
* We have already proved the accuracy bound.
* Dynamic program II running time is $O(e^\varepsilon \ln(1/\varepsilon))$, where

$$v_{\text{run}} = \left\lceil \frac{\text{time}}{\Theta} \right\rceil - \left\lfloor \frac{n}{\varepsilon} \right\rfloor$$

PTAS. $(1 + \varepsilon)$-approximation algorithm for any constant $\varepsilon > 0$.
* Produces arbitrarily high quality solution.
* Trades off accuracy for time.
* But such algorithms are unlikely to exist for certain problems...