Lecture 26

NP-completeness as a Design Guide

- Algorithms for special cases
  - Finding small vertex covers
- Independent Set in Trees
- Approximation algorithms

Sofya Raskhodnikova
"I can't find an efficient algorithm, I guess I'm just too dumb."
"I can't find an efficient algorithm, because no such algorithm is possible!"
"I can't find an efficient algorithm, but neither can all these famous people."
NP-Completeness as a Design Guide

Q. Suppose I need to solve an NP-complete problem. What should I do?
A. You are unlikely to find poly-time algorithm that works on all inputs.

Must sacrifice one of three desired features.
- Solve problem in polynomial time (⇒ e.g., fast exponential algorithms)
- Solve arbitrary instances of the problem
- Solve problem to optimality (⇒ approximation algorithms)

Most of this lecture. Solve some special cases of NP-complete problems that arise in practice.
10.1 Finding Small Vertex Covers

• Suppose vertex cover describes warehouse “placement” problem,
  • e.g.: how many warehouses (placed in cities) are needed so there is one at an endpoint of every designated highway segment?

• Not interested if the answer is larger than 10

• This is vertex cover (of the highway graph) with $k < 10$
**Vertex Cover**

**VERTEX COVER:** Given a graph $G = (V, E)$ and an integer $k$, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge $(u, v)$ either $u \in S$, or $v \in S$, or both.

$k = 4$

$S = \{ 3, 6, 7, 10 \}$
Finding Small Vertex Covers

Q. How fast can we solve Vertex Cover for small k?

First attempt: Brute force.

- Try all $\binom{n}{k} = \Theta(n^k)$ subsets of size k.
- Takes $O(kn)$ time to check whether a subset is a vertex cover.
- Crude time bound: $O(kn^{k+1})$.
- Slow even for k=10.

Second attempt. Limit exponential dependency on k, e.g., to $O(2^k kn)$.

Ex. n = 1,000, k = 10.

Brute. $kn^{k+1} = 10^{34} \Rightarrow$ infeasible.

Better. $2^k kn = 10^7 \Rightarrow$ feasible.

Remark. If k is a constant, algorithm is poly-time; if k is a small constant, then it's also practical.

Important. The algorithm is still exponential, and hence scales badly (e.g., consider k=40). However, it is better than brute force.
Finding Small Vertex Covers

Idea: Recursive solution similar to self-reducibility argument.

Claim 1. Let u-v be an edge of G. G has a vertex cover of size $\leq k$ iff at least one of $G - \{u\}$ and $G - \{v\}$ has a vertex cover of size $\leq k-1$.

Pf. $\Rightarrow$
- Suppose $G$ has a vertex cover $S$ of size $\leq k$.
- $S$ contains either $u$ or $v$ (or both).
- Without loss of generality, assume it contains $u$.
- $S - \{u\}$ is a vertex cover of $G - \{u\}$.

Pf. $\Leftarrow$
- Suppose $S$ is a vertex cover of $G - \{u\}$ of size $\leq k-1$.
- Then $S \cup \{u\}$ is a vertex cover of $G$. $\blacksquare$
Finding Small Vertex Covers: Algorithm

**Claim 2.** The following algorithm find a vertex cover of size \( \leq k \) in \( G \) if it exists and runs in \( O(2^k n) \) time.

```plaintext
SmallVC(G, k) {
    if k=0
        if (G contains no edges) return Ø
        else return false
    let (u, v) be any edge of G
    S_u = SmallVC(G - {u}, k-1)
    if S_u ≠ false return S_u ∪ {u}

    S_v = SmallVC(G - {v}, k-1)
    if S_v ≠ false return S_v ∪ {v}
    else return false
}
```

**Pf.**

- Correctness follows from Claim 1.
- There are \( \leq 2^{k+1} \) nodes in the recursion tree; each invocation takes \( O(n) \) time.
Finding Small Vertex Covers: Recursion Tree

\[ T(n, k) \leq \begin{cases} 
  cn & \text{if } k = 0 \\
  2T(n, k-1) + cn & \text{if } k > 1 
\end{cases} \implies T(n, k) \leq 2^{k+1} c \cdot n \]
Another algorithm for small VC

As before. Remove or include one vertex u (as in fast exponential algorithm)

- If u is in VC, remove adjacent edges
  - T(n-1,k-1)
- If u not in VC, remove adjacent edges, add all neighbors of u to VC
  - T(n-1-deg(u), k-deg(u))

Idea. By choosing vertex u with largest degree (at least |E|/k, otherwise no VC of size k exists), get better exponent in k

Exercise: how small an exponent in k can you get while maintaining linear scaling in n?
Summary

Often input size is too crude a measure of complexity
- e.g., VC algorithm linear in n, exponential in k

Parameterized complexity
- General theory of such problems
- Clever algorithms, hardness arguments

Take away message:
- When facing a seemingly hard problem, look for what “really” makes it hard
10.2 Solving NP-Hard Problems on Trees
Maximum Independent Set: Special Case

When is Independent Set hard?
The graphs that come from the reduction from 3-SAT look funny

Can we select a family of graphs on which it is solvable?
  • What about trees?
Independent set on trees. Given a tree, find a maximum cardinality subset of nodes such that no two share an edge.

Fact. A tree on at least two nodes has at least two leaf nodes.

Key observation. If v is a leaf, there exists a maximum size independent set containing v.

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

\[ \text{degree} = 1 \]
Independent Set on Trees: Greedy Algorithm

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

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

**Remark.** Can implement in $O(n)$ time by considering nodes in postorder.
Weighted Independent Set on Trees

Weighted independent set on trees. Given a tree and node weights \( w_v > 0 \), find an independent set \( S \) that maximizes \( \sum_{v \in S} w_v \).

Observation. If \((u, v)\) is an edge such that \( v \) is a leaf node, then either \( \text{OPT} \) includes \( u \), or it includes all leaf nodes incident to \( u \).

Dynamic programming solution. Root tree at some node, say \( r \).

- \( \text{OPT}_{\text{in}}(u) = \max \) weight independent set of subtree rooted at \( u \), containing \( u \).
- \( \text{OPT}_{\text{out}}(u) = \max \) weight independent set of subtree rooted at \( u \), not containing \( u \).

\[
\text{OPT}_{\text{in}}(u) = w_u + \sum_{v \in \text{children}(u)} \text{OPT}_{\text{out}}(v)
\]

\[
\text{OPT}_{\text{out}}(u) = \sum_{v \in \text{children}(u)} \max \{ \text{OPT}_{\text{in}}(v), \text{OPT}_{\text{out}}(v) \}
\]
Independent Set on Trees: Algorithm

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

```
Weighted-Independent-Set-In-A-Tree(T) {  
    Root the tree at a node r  
    foreach (node u of T in postorder) {  
        if (u is a leaf) {  
            Min[u] = w_u  
            M_out[u] = 0  
        }  
        else {  
            Min[u] = \sum_{v \in \text{children}(u)} M_{out}[v] + w_v  
            M_out[u] = \sum_{v \in \text{children}(u)} \max(M_{out}[v], Min[v])  
        }  
    }  
    return \max(M_{in}[r], M_{out}[r])  
}
```

**Exercise.** Recover the independent set of maximum weight by tracing back.

**Pf.** Takes $O(n)$ time since we visit nodes in postorder and examine each edge exactly once. □
Context

**Independent set on trees.** This structured special case is tractable because we can find a node that *breaks the communication* among the subproblems in different subtrees.

![Diagram of independent set on trees](image)

**Graphs of bounded tree width.** Elegant generalization of trees that:
- Captures a rich class of graphs that arise in practice.
- Enables decomposition into independent pieces.

see Chapter 10.4
11 Approximation Algorithms
Approximation algorithm

Suppose a minimization problem is NP hard

- Cannot find a polynomial-time algorithm that finds optimal solution on every instance
- What if I can guarantee that my algorithm’s solution is within 5% of optimal? That is,
  - $\text{OPT} \leq \text{my-opt} \leq 1.05 \times \text{OPT}$

- Good enough?
  - Depends on context
  - If data is already noisy or we don’t know exact cost function, then approximation might be fine.
Q. Suppose I need to solve an NP-complete problem. What should I do?
A. You are unlikely to find poly-time algorithm that works on all inputs.

Must sacrifice one of three desired features.
- Solve problem in polynomial time (→ e.g., fast exponential algorithms)
- Solve arbitrary instances of the problem
- Solve problem to optimality (→ approximation algorithms)

ρ-approximation algorithm.
- Guaranteed to run in poly-time.
- Guaranteed to solve arbitrary instance of the problem
- Guaranteed to find solution within ratio ρ of true optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what optimum value is!
Simple Approximation Algorithms

Vertex cover
- size of any maximal matching $M$ is 2-approximation
- (Sandwich: $|M| \leq |\text{opt-VC}| \leq 2|M|$)

Metric Traveling Salesman
- Inorder traversal of MST is a 2-approx
- Christofides: 3/2 approx

Knapsack
- For any $\delta$: there is a $(1+\delta)$-approximation in time $O(n/\delta)$ by rounding weights to multiples of $\delta W$
Metric TSP

Input: undirected $G$, non-negative edge lengths $w$

Goal: path $p$ that visits all vertices (can visit some vertices twice) of minimum length

Why metric?
- Can always use the shortest path to go from $u$ to $v$
- So may as well assume all edges are present and $w(u,v) = d(u,v)$ (where $d =$ distance in original $G$)

Idea 1: look at MST.
- 2-approx
- Inefficiency: might have to reuse edges

Idea 2: look at set of odd-degree vertices $O$
- Add min-weight matching (Edmonds Algorithm) on $O$
- This adds $OPT/2$ to total weight since optimal tour of $O$ can be written as union of two disjoint matchings