Chapter 10, 11, 12
Extending the Limits of Tractability

Coping With NP-Completeness

Q. Suppose I need to solve an NP-complete problem. What should I do?
A. Theory says you’re unlikely to find poly-time algorithm.

Must sacrifice one of three desired features.
- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

This lecture. Solve some special cases of NP-complete problems that arise in practice.

10.1 Finding Small Vertex Covers

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. What if k is small?

Brute force. $O(kn^{k+1})$.
- Try all $C(n, k) = O(n^k)$ subsets of size k.
- Takes $O(kn)$ time to check whether a subset is a vertex cover.

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

Ex. $n = 1,000, k = 10$.
- Brute. $k \cdot n^{k+1} = 10^{11} \Rightarrow$ infeasible.
- Better. $2^k \cdot n = 10^7 \Rightarrow$ feasible.

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

Finding Small Vertex Covers: Algorithm

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

```java
boolean Vertex-Cover(G, k) {
    if (G contains no edges) return true
    if (G contains $\geq kn$ 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
}
```

Pf.
- Correctness follows previous two claims.
- There are $\leq 2^{k-1}$ nodes in the recursion tree; each invocation takes $O(kn)$ time.

Finding Small Vertex Covers: Recursion Tree

$$T(n, k) = \begin{cases} 
  cn & \text{if } k = 1 \\
  \frac{cn}{2T(n, k-1) + ckn} & \text{if } k > 1
\end{cases} \Rightarrow T(n, k) \leq 2^k cn$$
10.2 Solving NP-Hard Problems on Trees

Independent Set on 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.
- If u \notin S and v \notin S, then S \cup \{v\} is independent \Rightarrow S not maximum.
- IF u \in S and v \notin S, then S \cup \{v\} \setminus \{u\} is independent.

Independent Set on Trees: Greedy Algorithm

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

```
Independent-Set-In-A-Forest(F) {
  S ← ∅
  while (F has at least one edge) {
    Let e = (u, v) be an edge such that v is a leaf
    Add v to S
    Delete from F nodes u and v, and all edges incident to them.
  }
  return S
}
```

**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 OPT includes u, or it includes all leaf nodes incident to u.

Dynamic programming solution. Root tree at some node, say r.
- \( OPT_{in}(u) = \text{max weight independent set rooted at } u, \text{ containing } u \)
- \( OPT_{out}(u) = \text{max weight independent set rooted at } u, \text{ not containing } u \)

\[
OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)
\]
\[
OPT_{out}(u) = \max_{v \in \text{children}(u)} \{ OPT_{in}(v), OPT_{out}(v) \}
\]

children(u) = \{ v, w, x \}
Independent Set on Trees: Greedy 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
      Max[u] = 0
    } else {
      Min[u] = Σ [v ∈ children(u)] Max[v] + w_u
      Max[u] = Σ [v ∈ children(u)] max(Max[v], Min[v])
    }
  }
  return max(Min[r], Max[r])
}
```

**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.

**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.

**Approximation Algorithms**

**Q.** Suppose I need to solve an NP-hard problem. What should I do?
**A.** Theory says you’re unlikely to find a poly-time algorithm.

**Must sacrifice one of three desired features.**
- Solve problem to optimality.
- Solve problem in poly-time.
- Solve arbitrary instances of the problem.

**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!
11.1 Load Balancing

Load Balancing

*Input.* m identical machines; n jobs, job j has processing time $t_j$.
- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.

*Def.* Let $J(i)$ be the subset of jobs assigned to machine i. The **load** of machine i is $L_i = \sum_{j \in J(i)} t_j$.

*Def.* The **makespan** is the maximum load on any machine $L = \max_i L_i$.

Load balancing. Assign each job to a machine to minimize makespan.

Load Balancing: List Scheduling

List-scheduling algorithm.
- Consider n jobs in some fixed order.
- Assign job j to machine whose load is smallest so far.

```
List-Scheduling(m, n, t_1, t_2, ..., t_n) {
    for i = 1 to m {
        L_i = 0  // load on machine i
        J(i) = ∅  // jobs assigned to machine i
    }
    for j = 1 to n {
        i = argmin_k L_k  // machine i has smallest load
        J(i) = J(i) ∪ {j}  // assign job j to machine i
        L_i = L_i + t_j  // update load of machine i
    }
}
```

Implementation. $O(n \log n)$ using a priority queue.

Load Balancing: List Scheduling Analysis

- First worst-case analysis of an approximation algorithm.
- Need to compare resulting solution with optimal makespan $L^*$.

**Lemma 1.** The optimal makespan $L^* = \max_j t_j$.
**Pf.** Some machine must process the most time-consuming job.

**Lemma 2.** The optimal makespan $L^* = \frac{1}{m} \sum_j t_j$.
**Pf.**
- The total processing time is $\sum_j t_j$.
- One of m machines must do at least a $1/m$ fraction of total work.
Theorem. Greedy algorithm is a 2-approximation.

Proof. Consider load $L_i$ of bottleneck machine $i$.

- Let $j$ be last job scheduled on machine $i$.
- When job $j$ assigned to machine $i$, $i$ had smallest load. Its load before assignment is $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.

Sum inequalities over all $k$ and divide by $m$:

$$L_i - t_j \leq \frac{1}{m} \sum_{k \in C_i} t_k \leq L^*$$

Now

$$L_i = (L_i - t_j) + t_j \leq 2L^*.$$
Gradient Descent: Vertex Cover

**VERTEX-COVER.** Given a graph $G = (V, E)$, find a subset of nodes $S$ of minimal cardinality such that for each $u-v$ in $E$, either $u$ or $v$ (or both) are in $S$.

**Neighbor relation.** $S \sim S'$ if $S'$ can be obtained from $S$ by adding or deleting a single node. Each vertex cover $S$ has at most $n$ neighbors.

**Gradient descent.** Start with $S = V$. If there is a neighbor $S'$ that is a vertex cover and has lower cardinality, replace $S$ with $S'$.

**Remark.** Algorithm terminates after at most $n$ steps since each update decreases the size of the cover by one.

Local Search

**Local search.** Algorithm that explores the space of possible solutions in sequential fashion, moving from a current solution to a "nearby" one.

**Neighbor relation.** Let $S \sim S'$ be a neighbor relation for the problem.

**Gradient descent.** Let $S$ denote current solution. If there is a neighbor $S'$ of $S$ with strictly lower cost, replace $S$ with the neighbor whose cost is as small as possible. Otherwise, terminate the algorithm.

12.2 Metropolis Algorithm
Metropolis Algorithm

Metropolis algorithm. [Metropolis, Rosenbluth, Rosenbluth, Teller, Teller 1953]
- Simulate behavior of a physical system according to principles of statistical mechanics.
- Globally biased toward "downhill" steps, but occasionally makes "uphill" steps to break out of local minima.

Gibbs-Boltzmann function. The probability of finding a physical system in a state with energy E is proportional to \( e^{-E/(kT)} \), where \( T > 0 \) is temperature and \( k \) is a constant.
- For any temperature \( T > 0 \), function is monotone decreasing function of energy E.
- System more likely to be in a lower energy state than higher one.
  - \( T \) large: high and low energy states have roughly same probability
  - \( T \) small: low energy states are much more probable

Simulated Annealing

Simulated annealing.
- \( T \) large \( \Rightarrow \) probability of accepting an uphill move is large.
- \( T \) small \( \Rightarrow \) uphill moves are almost never accepted.
- Idea: turn knob to control \( T \).
- Cooling schedule: \( T = T(i) \) at iteration \( i \).

Physical analog.
- Take solid and raise it to high temperature, we do not expect it to maintain a nice crystal structure.
- Take a molten solid and freeze it very abruptly, we do not expect to get a perfect crystal either.
- Annealing: cool material gradually from high temperature, allowing it to reach equilibrium at succession of intermediate lower temperatures.

12.3 Hopfield Neural Networks
Hopfield Neural Networks

**Hopfield networks.** Simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states.

**Input:** Graph \( G = (V, E) \) with integer edge weights \( w \).

**Configuration.** Node assignment \( s_u = \pm 1 \).

**Intuition.** If \( w_{uv} < 0 \), then \( u \) and \( v \) want to have the same state; if \( w_{uv} > 0 \) then \( u \) and \( v \) want different states.

**Note.** In general, no configuration respects all constraints.

**Def.** With respect to a configuration \( S \), edge \( e = (u, v) \) is **good** if \( w_{uv} s_u s_v < 0 \). That is, if \( w_{uv} < 0 \) then \( s_u = s_v \); if \( w_{uv} > 0 \), \( s_u \neq s_v \).

**Def.** With respect to a configuration \( S \), a node \( u \) is **satisfied** if the weight of incident good edges \( \geq \) weight of incident bad edges.

\[ \sum_{(e \in E) \cap \{u\}} w_{uv} s_u s_v \geq 0 \]

**Def.** A configuration is **stable** if all nodes are satisfied.

**Goal.** Find a stable configuration, if such a configuration exists.

---

**State Flipping Algorithm**

**Goal.** Find a stable configuration, if such a configuration exists.

**State-flipping algorithm.** Repeatedly flip state of an unsatisfied node.

```plaintext
Hopfield-Flip(G, w) {
    S ← arbitrary configuration
    while (current configuration is not stable) {
        u ← unsatisfied node
        s_u = -s_u
    }
    return S
}
```

---

**Hopfield Neural Networks**

Unsatisfied node: \(-10 - 8 \geq 0\)

Satisfied node: \(-4 - 4 - 4 \leq 0\)

Stable configuration.
12 Local Search Methods

Hopfield Neural Networks

Claim. State-flipping algorithm terminates with a stable configuration after at most $W = \sum_{e} |w_e|$ iterations.

Pf attempt. Consider measure of progress $\Phi(S) = \#$ satisfied nodes.

$\Phi(S) = \sum_{e \in E} |W_e|$

- Clearly $0 \leq \Phi(S) \leq W$.
- We show $\Phi(S)$ increases by at least 1 after each flip.
  When u flips state:
  - all good edges incident to u become bad
  - all bad edges incident to u become good
  - all other edges remain the same

$\Phi(S') - \Phi(S) = \sum_{e \in E} I(w_e) + \sum_{e \in E} I(w_e) > \Phi(S) + 1$

- if u is satisfied
- if u is unsatisfied

Complexity of Hopfield Neural Network

Hopfield network search problem. Given a weighted graph, find a stable configuration if one exists.

Hopfield network decision problem. Given a weighted graph, does there exist a stable configuration?

Remark. The decision problem is trivially solvable (always yes), but there is no known poly-time algorithm for the search problem.

$\Omega(n \log W)$