Chapter 10
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. \( O(k \, n^{k+1}) \).
- Try all \( C(n, k) = O(n^k) \) subsets of size \( k \).
- Takes \( O(k \, n) \) time to check whether a subset is a vertex cover.

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

Ex. \( n = 1,000, k = 10 \).
Brute. \( k \, n^{k+1} = 10^{34} \Rightarrow \) infeasible.
Better. \( 2^k \, k \, n = 10^7 \Rightarrow \) 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

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

Pf. ⇒
- Suppose G has a vertex cover S of size ≤ k.
- S contains either u or v (or both). Assume it contains u.
- S - {u} is a vertex cover of G - {u}.

Pf. ⇐
- Suppose S is a vertex cover of G - {u} of size ≤ k-1.
- Then S ∪ {u} is a vertex cover of G.

Claim. If G has a vertex cover of size k, it has ≤ k(n-1) edges.
Pf. Each vertex covers at most n-1 edges.
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 from 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

\[ T(n, k) \leq \begin{cases} 
  c & \text{if } k = 0 \\
  cn & \text{if } k = 1 \\
  2T(n, k-1) + ckn & \text{if } k > 1 
\end{cases} \Rightarrow T(n, k) \leq 2^k cn \]
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 $S \cup \{v\} - \{u\}$ is independent. □
**Theorem.** The following greedy algorithm finds a maximum cardinality independent set in forests (and hence trees).

```plaintext
Independent-Set-In-A-Forest(F) {
    S ← Ø
    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. 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 $r$.
- $OPT_{in}(u) = \max$ weight independent set of subtree rooted at $u$, containing $u$.
- $OPT_{out}(u) = \max$ weight independent set of subtree rooted at $u$, not containing $u$.

$$OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)$$

$$OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max \{OPT_{in}(v), OPT_{out}(v)\}$$

children($u$) = \{ $v$, $w$, $x$ \}
Weighted Independent Set on Trees: Dynamic Programming Algorithm

**Theorem.** The dynamic programming algorithm finds a maximum weighted independent set in a tree 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) {
            \( M_{\text{in}}[u] = w_u \)
            \( M_{\text{out}}[u] = 0 \)
        }
        else {
            \( M_{\text{in}}[u] = w_u + \sum_{v \in \text{children}(u)} M_{\text{out}}[v] \)
            \( M_{\text{out}}[u] = \sum_{v \in \text{children}(u)} \max(M_{\text{in}}[v], M_{\text{out}}[v]) \)
        }
    }
    return \( \max(M_{\text{in}}[r], M_{\text{out}}[r]) \)
}
```

**Pf.** Takes $O(n)$ time since we visit nodes in postorder and examine each edge exactly once. □ can also find independent set itself (not just value)
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 a tree with a node u breaking communication](image1)

**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.
10.3 Circular Arc Coloring
Wavelength-Division Multiplexing

Wavelength-division multiplexing (WDM). Allows \( m \) communication streams (arcs) to share a portion of a fiber optic cable, provided they are transmitted using different wavelengths.

Ring topology. Special case is when network is a cycle on \( n \) nodes.

Bad news. NP-complete, even on rings.

Brute force. Can determine if \( k \) colors suffice in \( O(k^m) \) time by trying all \( k \)-colorings.

Goal. \( O(f(k)) \cdot \text{poly}(m, n) \) on rings.
Wavelength-Division Multiplexing

**Wavelength-division multiplexing (WDM).** Allows m communication streams (arcs) to share a portion of a fiber optic cable, provided they are transmitted using different wavelengths.

**Ring topology.** Special case is when network is a cycle on n nodes.

**Bad news.** NP-complete, even on rings.

**Brute force.** Can determine if k colors suffice in $O(k^m)$ time by trying all k-colorings.

**Goal.** $O(f(k)) \cdot \text{poly}(m, n)$ on rings.
**Review: Interval Coloring**

**Interval coloring.** *Greedy algorithm finds coloring such that number of colors equals depth of schedule.*

Circular arc coloring.
- Weak duality: number of colors $\geq$ depth.
- Strong duality does not hold.

```
 max depth = 2
 min colors = 3
```
(Almost) Transforming Circular Arc Coloring to Interval Coloring

Circular arc coloring. Given a set of n arcs with depth \( d \leq k \), can the arcs be colored with k colors?

Equivalent problem. Cut the network between nodes \( v_1 \) and \( v_n \). The arcs can be colored with k colors iff the intervals can be colored with k colors in such a way that "sliced" arcs have the same color.
Circular Arc Coloring: Dynamic Programming Algorithm

Dynamic programming algorithm.

- Assign distinct color to each interval which begins at cut node $v_0$.
- At each node $v_i$, some intervals may finish, and others may begin.
- Enumerate all $k$-colorings of the intervals through $v_i$ that are consistent with the colorings of the intervals through $v_{i-1}$.
- The arcs are $k$-colorable iff some coloring of intervals ending at cut node $v_0$ is consistent with original coloring of the same intervals.

![Diagram of circular arc coloring]
Circular Arc Coloring: Running Time

**Running time.** $O(k! \cdot n)$.
- $n$ phases of the algorithm.
- Bottleneck in each phase is enumerating all consistent colorings.
- There are at most $k$ intervals through $v_i$, so there are at most $k!$ colorings to consider.

**Remark.** This algorithm is practical for small values of $k$ (say $k = 10$) even if the number of nodes $n$ (or paths) is large.
Extra Slides
Vertex Cover in Bipartite Graphs
**Vertex cover.** Given an undirected graph \( G = (V, E) \), a vertex cover is a subset of vertices \( S \subseteq V \) such that for each edge \((u, v) \in E\), either \( u \in S \) or \( v \in S \) or both.

\[
S = \{3, 4, 5, 1', 2'\} \quad |S| = 5
\]
**Vertex Cover**

**Weak duality.** Let $M$ be a matching, and let $S$ be a vertex cover. Then, $|M| \leq |S|$.

**Pf.** Each vertex can cover at most one edge in any matching.
**König-Egerváry Theorem.** In a bipartite graph, the max cardinality of a matching is equal to the min cardinality of a vertex cover.

\[ S^* = \{3, 1', 2', 5'\} \]
\[ |S^*| = 4 \]

\[ M^* = 1-1', 2-2', 3-3', 5-5' \]
\[ |M^*| = 4 \]
König-Egerváry Theorem. In a bipartite graph, the max cardinality of a matching is equal to the min cardinality of a vertex cover.
- Suffices to find matching $M$ and cover $S$ such that $|M| = |S|$.
- Formulate max flow problem as for bipartite matching.
- Let $M$ be max cardinality matching and let $(A, B)$ be min cut.
**Vertex Cover: Proof of König-Egerváry Theorem**

**König-Egerváry Theorem.** In a bipartite graph, the max cardinality of a matching is equal to the min cardinality of a vertex cover.

- Suffices to find matching $M$ and cover $S$ such that $|M| = |S|$.
- Formulate max flow problem as for bipartite matching.
- Let $M$ be max cardinality matching and let $(A, B)$ be min cut.
- Define $L_A = L \cap A, L_B = L \cap B, R_A = R \cap A, R_B = R \cap B$.

**Claim 1.** $S = L_B \cup R_A$ is a vertex cover.
- consider $(u, v) \in E$
- $u \in L_A, v \in R_B$ impossible since infinite capacity
- thus, either $u \in L_B$ or $v \in R_A$ or both

**Claim 2.** $|S| = |M|$.
- max-flow min-cut theorem $\Rightarrow |M| = \text{cap}(A, B)$
- only edges of form $(s, u)$ or $(v, t)$ contribute to $\text{cap}(A, B)$
- $|M| = \text{cap}(A, B) = |L_B| + |R_A| = |S|$. \[\square\]
Register Allocation
Register Allocation

**Register.** One of k of high-speed memory locations in computer's CPU. say 32

**Register allocator.** Part of an optimizing compiler that controls which variables are saved in the registers as compiled program executes.

**Interference graph.** Nodes are "live ranges." Edge u-v if there exists an operation where both u and v are "live" at the same time.

**Observation.** [Chaitin, 1982] Can solve register allocation problem iff interference graph is k-colorable.

**Spilling.** If graph is not k-colorable (or we can't find a k-coloring), we "spill" certain variables to main memory and swap back as needed.

typically infrequently used variables that are not in inner loops
A Useful Property

Remark. Register allocation problem is NP-hard.

Key fact. If a node \( v \) in graph \( G \) has fewer than \( k \) neighbors, \( G \) is \( k \)-colorable iff \( G - \{ v \} \) is \( k \)-colorable.

Pf. Delete node \( v \) from \( G \) and color \( G - \{ v \} \).
   - If \( G - \{ v \} \) is not \( k \)-colorable, then neither is \( G \).
   - If \( G - \{ v \} \) is \( k \)-colorable, then there is at least one remaining color left for \( v \).

\[ \text{delete } v \text{ and all incident edges} \]

\[ k = 2 \]
\[ G \text{ is } 2\text{-colorable even though all nodes have degree } 2 \]
Chaitin's Algorithm

\begin{algorithm}
\textbf{Vertex-Color}(G, k) \{
    \textbf{while} (G is not empty) \{
        Pick a node \( v \) with fewer than \( k \) neighbors
        \textbf{Push} \( v \) on stack
        \textbf{Delete} \( v \) and all its incident edges
    \}
    \textbf{while} (stack is not empty) \{
        \textbf{Pop} next node \( v \) from the stack
        \textbf{Assign} \( v \) a color different from its neighboring nodes which have already been colored
    \}
\}
\end{algorithm}
Chaitin's Algorithm

**Theorem.** [Kempe 1879, Chaitin 1982] Chaitin's algorithm produces a \( k \)-coloring of any graph with max degree \( k-1 \).

**Pf.** Follows from key fact since each node has fewer than \( k \) neighbors.

**Remark.** If algorithm never encounters a graph where all nodes have degree \( \geq k \), then it produces a \( k \)-coloring.

**Practice.** Chaitin's algorithm (and variants) are extremely effective and widely used in real compilers for register allocation.