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 Chapter. 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 $|S| = k$, and for each edge $(u, v)$ either $u \in S$, or $v \in S$, or both.

---

**Finding Small Vertex Covers**

**Q.** What if $k$ is small?

**Brute force.** $O(k^m n^k)$.
- Try all $\binom{n}{k} = O(n^k)$ 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 $O(2^k n^{k+1})$.

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

- **Brute.** $k^m n^k = 10^{104}$ ⇒ infeasible.
- **Better.** $2^k k^m n^{k+1}$ ⇒ feasible.

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

---

**Claim.** If $G$ has $n$ nodes, the maximum degree of any node is $d$, and there is a vertex cover of size $k$, then $G$ has at most $kd$ edges.

**Pf.**
- Suppose $G$ has a vertex cover $S$ of size $k$.
- Each node of $S$ can cover at most $d$ edges.
- The total covered edges is $kd$.

**Claim.** If $G$ has a vertex cover of size $k$, it has $\leq k(n-1)$ edges.

**Pf.** Each vertex covers at most $n-1$ edges.
Finding Small Vertex Covers

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

\[ \text{Pf. } \Rightarrow \]
- Suppose \( G \) has a vertex cover \( S \) of size \( \leq k \).
- \( S \) contains either \( u \) or \( v \) (or both). Assume it contains \( u \).
- \( S - \{u\} \) is a vertex cover of \( G - \{u\} \).

\[ \text{Pf. } \Leftarrow \]
- Suppose \( S \) is a vertex cover of \( G - \{u\} \) of size \( \leq k-1 \).
- Then \( S \cup \{u\} \) is a vertex cover of \( G \). \( \Box \)

Finding Small Vertex Covers: Algorithm

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

```java
boolean Vertex-Cover(G, k) {
    if (G contains no edges) return true
    if (k == 0) 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
}
```

\[ \text{Pf.} \]
- Correctness follows from the last claim.
- There are \( \leq 2^{k+1} \) nodes in the recursion tree; each invocation takes \( O(kn) \) time. \( \Box \)

Finding Small Vertex Covers: Recursion Tree

The diagram shows the recursion tree for the algorithm. The function \( T(n,k) \) describes the complexity, with different cases for different values of \( k \). The tree recursively breaks down into subproblems, each of which is solved by the algorithm, until the base case is reached, which is solving for a vertex cover of size 0 in a graph with no edges.

The tree structure is as follows:
- For \( k = 0 \): \( T(n,0) = cn \)
- For \( k = 1 \): \( T(n,1) = 2T(n-1,k-1) + cn \)
- For \( k > 1 \): \( T(n,k) \) follows the same pattern as for \( k = 1 \), with the base case being \( T(n,0) = cn \).

The tree is symmetric around the middle, with each level representing a decision point in the algorithm, leading to subproblems that are solved by recursive calls.

For a given graph \( G \) of size \( n \), the tree grows in a way that each level represents a potential split in the vertex cover problem, leading to a total of \( 2^k \) nodes at depth \( k \).

The blue nodes represent the final decision or base case in the recursion tree, which is determined by the value of \( k \) and the complexity of the graph.

This visualization helps in understanding the exponential growth of the number of subproblems as \( k \) increases, which is why the algorithm's time complexity is \( O(2^k n) \) for graphs with a vertex cover of size \( k \).
Finding Small Vertex Covers: Algorithm

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

```java
boolean Vertex-Cover(G, k) {
    if (G contains no edges)   return true
    if (G contains $\geq k(n-1)$ 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
}
```

10.2 Solving NP-Hard Problems 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 $u \in S$ 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. □

---

---
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 ← φ
    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.

- \( \text{OPT}_\text{in}(u) \) = 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.

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

\[
\text{OPT}_\text{out}(v) = \max \{ \text{OPT}_\text{in}(v), \text{OPT}_\text{out}(v) \}
\]

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) {
            \( \text{Min}[u] = w_u \)
            \( \text{Mout}[u] = 0 \)
        } else {
            \( \text{Min}[u] = w_u + \sum \max(\text{Min}[v], \text{Mout}[v]) \)
            \( \text{Mout}[u] = \sum \max(\text{Min}[v], \text{Mout}[v]) \)
        }
    }
    return \max(\text{Min}[r], \text{Mout}[r])
}
```

**Pf.** Takes O(n) time since we visit nodes in postorder and examine each edge exactly once. □
10.3 Circular Arc Coloring

**Wavelength-Division Multiplexing**

Wavelength-division multiplexing (WDM). Allows 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^n) time by trying all k-colorings.

**Goal.** O(f(k)) ⋅ poly(m, n) on rings.

\[ n = 4, m = 6 \]
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 ≥ depth.
- Strong duality does not hold.

(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.
Circular Arc Coloring: Dynamic Programming Algorithm

CircularArcColoring \( (n, k) \) {
    Let \( f' \) be a coloring of the paths having edge \((v_n, v_1)\).
    Let \( F_0 = \{ f' \} \);
    for \( i = 1, 2, \ldots, n \) {
        foreach \( f \) in \( F_{i-1} \) {
            Add extensions of \( f \) for the paths having edge \((v_i, v_{i+1})\) to \( F_i \).
        }
        if \( f' \) in \( F_n \) return true else return false
    }
}

Circular Arc Coloring: Running Time

CircularArcColoring \( (n, k) \) {
    Let \( f' \) be a coloring of the paths having edge \((v_n, v_1)\).
    Let \( F_0 = \{ f' \} \);
    for \( i = 1, 2, \ldots, n \) {
        foreach \( f \) in \( F_{i-1} \) {
            Add extensions of \( f \) for the paths having edge \((v_i, v_{i+1})\) to \( F_i \).
        }
        if \( f' \) in \( F_n \) return true else return false
    }
}

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.

Vertex Cover in Bipartite Graphs
Vertex Cover

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.

$S = \{3, 4, 5, 1', 2'\}$
$|S| = 5$

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

Pf. For each edge in $M$, we need a distinct vertex in $S$.

$M = 1-2', 3-1', 4-5'$
$|M| = 3$

Vertex Cover: 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.

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

$S = \{1, 3, 2', 5\}$
$|S| = 4$

$M^* = 1-1', 2-2', 3-3', 5-5'$
$|M^*| = 4$
Kőnig-Egerváry Theorem. In a bipartite graph $G = (L, R, E)$, the max cardinality of a matching is equal to the min cardinality of a vertex cover.

- Suffices to find matching $M$ and vertex cover $S$ such that $|M| = |S|$.
- Formulate max flow problem as for bipartite matching.
- Let $M$ be max matching and let $(A, B)$ be min cut.

**Claim 1.** $S = L \cup R$ is a vertex cover.
- Consider $(u, v) \in E$.
- $u \in L_A$, $v \in R_A$ impossible since infinite capacity.
- Thus, either $u \in L_B$ or $v \in R_B$ 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 $\text{cap}(A, B)$.
- $|M| = \text{cap}(A, B) = |L_A| + |R_A| = |S|$.
Register Allocation

Register. One of k of high-speed memory locations in computer’s CPU.

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.

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.

Proof. Delete node v and all incident edges.

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.

\[ k = 3 \quad k = 2 \]

G is 3-colorable even though all nodes have degree 3
Chaitin's Algorithm

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

Chaitin's Algorithm

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

Proof. 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.