CSE 417
Branch & Bound (pt 2)
Coping with NP-Completeness

UNIVERSITY *of* WASHINGTON

Reminders

- **HW8 due Friday**
  - network flow coding
  - model the problem of rounding table entries as max flow
    - you are given a library that solves basic max flow

Review of last lecture

- Complexity theory: P & NP
  - answer can be found vs checked in polynomial time
- NP-completeness
  - hardest problems in NP
  - solvable iff P = NP
- Reductions
  - reducing from Y to X proves $Y \leq X$
    - if you can solve X, then you can solve Y
  - X is NP-hard if every Y in NP is $Y \leq X$

NP-Compete Problems

> "Easiest" NP-complete problems (reduce from these):

Packing	independent set
Covering	vertex cover
Constraint Satisfaction	3-SAT
Sequencing	Hamiltonian cycle
Partitioning	3D matching
Numerical	partition

Outline for Today

- More Reductions
- Coping with NP-Completeness
- Small Vertex Covers
- Independent Set on Trees
- Approximate Knapsack

More Reductions…

- We have not yet proven that all of these are NP-hard
  - assumed (3-)SAT is NP-hard (Cook-Levin Theorem)
  - showed Vertex Cover $\equiv_P$ Independent Set

- Still need to show:
  - 3-SAT ≤ Independent Set
  - 3-SAT ≤ Hamiltonian Cycle (or Vertex Cover ≤ Hamiltonian Cycle)
  - 3-SAT ≤ 3D matching
  - 3-SAT ≤ Subset Sum / Partition

- We'll just do a couple… (Textbook has more.)

3-SAT $\leq_p$ Independent Set

> Recall: formula is an **and** of m clauses,
> where each clause is an **or** of three literals,
> where each literal is of the form "x_k" or "not x_k".

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

> Example: variables x, y, z and 3 clauses
- $\wedge$ = and, $\vee$ = or

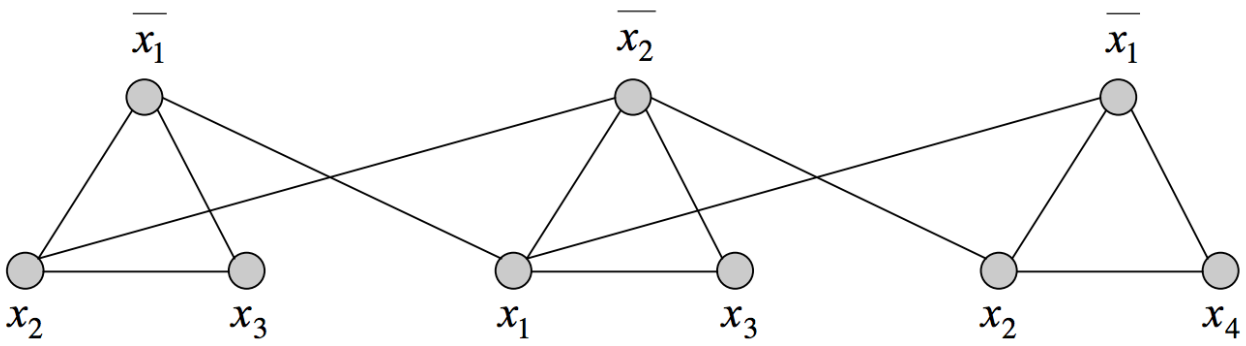
3-SAT $\leq_p$ Independent Set

- Reduce to Independent Set
- Idea: get independent set to choose the literals that are true
  - must set constraints so that it is possible for those literals to all be true

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

3-SAT $\leq_p$ Independent Set

- Reduction:
  - create a graph with nodes for every literal of every clause (3m total)
  - connect every x_i and not x_i by an edge
  - look for an independent set of size exactly m



3-SAT $\leq_p$ Subset Sum

> Reduce to Subset Sum

> Idea:
  - want subset sum to choose either x_i or not x_i
  - want every clause to have ≥ 1 literal chosen

3-SAT $\leq_p$ Subset Sum

- Idea:
  - want subset sum to choose either x_i or not x_i
  - want every clause to have ≥ 1 literal chosen

- Reduction:
  - numbers have 2k + 2m digits
    - use 2 instead of 1 to ensure no carries!
  - first 2k digits have 1s to indicate variable used
    - corresponding W digit means either x_i or not x_i used
  - last 2m digits indicate use in a clause
    - dummy rows let it get from 1-3 in clause to sum of 4

	x	y	z	C_1	C_2	C_3
x	1	0	0	0	1	0
¬ x	1	0	0	1	0	1
y	0	1	0	1	0	0
¬ y	0	1	0	0	1	1
z	0	0	1	1	1	0
¬ z	0	0	1	0	0	1
	0	0	0	1	0	0
	0	0	0	2	0	0
	0	0	0	0	1	0
	0	0	0	0	2	0
	0	0	0	0	0	1
	0	0	0	0	0	2
W	1	1	1	4	4	4

Outline for Today

> More Reductions
> Coping with NP-Completeness
> Small Vertex Covers
> Independent Set on Trees
> Approximate Knapsack

Coping with NP-Completeness

1. Your problem could lie in a special case that is easy
   - (alternatively, reduce the special case to the general case to prove it's NP-hard)

2. Look for approximate solutions
   - work well in worst-case or just on your data
     - (how do you know they work well if you can't solve it?)

3. Look for "fast enough" exponential time algorithms
   - next time…

Outline for Today

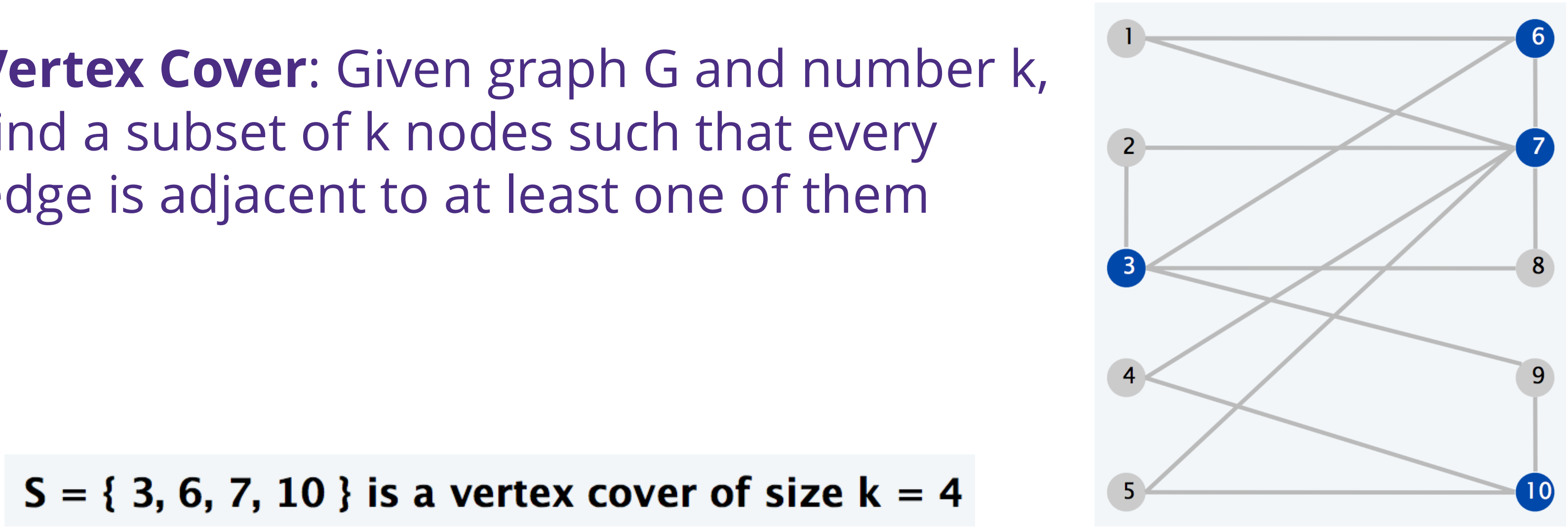
> More Reductions
> Coping with NP-Completeness
> Small Vertex Covers
> Independent Set on Trees
> Approximate Knapsack

Easy Special Cases

- Some problems are easy if certain inputs are small
- Example: Knapsack is easy if weight limit (W) is small
  - likewise for subset sum and partition
  - likewise for any problem with a pseudo-polynomial time algorithm
- Other inputs being small can help too...
  - next: Vertex Cover is easy if node set size (k) is small

Recall: Vertex Cover

> **Vertex Cover**: Given graph G and number k, find a subset of k nodes such that every edge is adjacent to at least one of them



Vertex Cover: Brute Force

- Brute Force solution is try every combination of k nodes
  - n^k choices
  - can check in O(k n) time if this is a solution
    - no nodes repeated
    - no edges between chosen nodes
  - running time is $O(k\, n^{k+1})$

```
for (Node n1 : nodes)
  for (Node n2 : nodes)
    ...
        for (Node nk in nodes)
            // check if {n1, n2, ..., nk} is solution
```

Vertex Cover: Brute Force

- Brute Force solution is try every combination of k nodes
  - running time is $O(k\ n^{k+1}) = O(k\ 2^{(k+1) \lg n})$
- Would like to improve this to, e.g., $O(n\ 2^k)$
  - exponential part depends only on k
  - time increases proportionally with n for any value of k
- Can make a real improvement...
- Example: n = 1,000 and k = 10
  - $k\ n^{k+1} = 10^{34}$
  - $2^k\ n = 10^7$

Vertex Cover: Improved

- **Proposition**: Let (u,v) be an edge of G. Then G has a vertex cover of size k iff G – {u} or G – {v} has a vertex cover of size k – 1

- Notation: graph G – {u}…
  - has all edges of G except u
  - has all edges of G except those to/from u

Vertex Cover: Improved

- **Proposition**: Let (u,v) be an edge of G. Then G has a vertex cover of size k iff G – {u} or G – {v} has a vertex cover of size k – 1

- Proof (⟹):
  - Let S be a vertex cover of G of size k
  - S must include either u or v
    - assume S includes u (without loss of generality)
  - S – {u} covers every edge except possibly those adjacent to u
  - S – {u} covers every edge of G – {u}

Vertex Cover: Improved

- **Proposition**: Let (u,v) be an edge of G. Then G has a vertex cover of size k iff G – {u} or G – {v} has a vertex cover of size k – 1

- Proof (⟸):
  - Let S be a vertex cover of G – {u} of size k – 1
  - only edges of G not covered by S are (potentially) those adjacent to u
  - so S + {u} is a vertex cover of G of size k

Vertex Cover: Improved

> **Proposition**: Let (u,v) be an edge of G. Then G has a vertex cover of size k iff G – {u} or G – {v} has a vertex cover of size k – 1

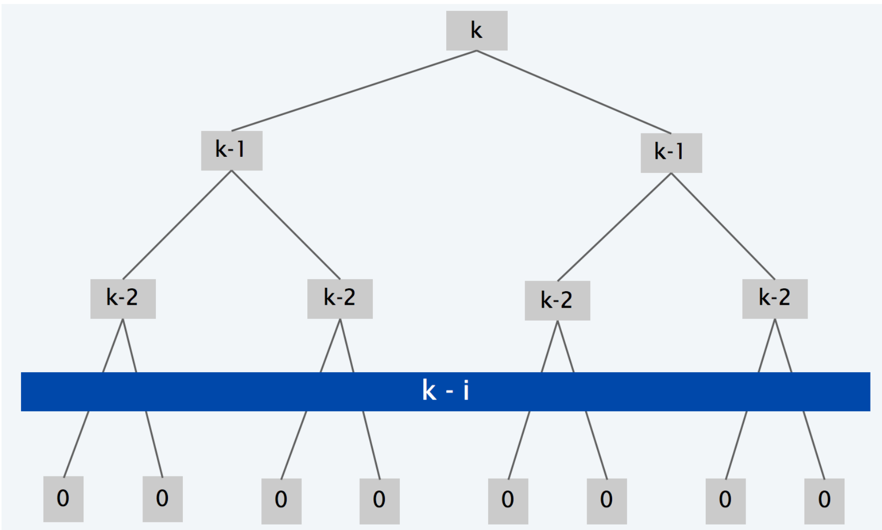
```
VertexCover(G, k):

  if k > 0:
    pick an edge (u,v)
    return VertexCover(G – {u}, k-1) || VertexCover(G – {v}, k-1)

  else:
    return true iff G has no edges
```

Vertex Cover: Improved

```
if k > 0:
  pick an edge (u,v)
  return VertexCover(G – {u}, k-1) || VertexCover(G – {v}, k-1)
```



Vertex Cover: Improved

- Running time is $O(2^k (n+m))$
  - takes $O(n + m)$ time to construct $G - \{u\}$ and $G - \{v\}$

```
VertexCover(G, k):

  if k > 0:
    pick an edge (u, v)
    return VertexCover(G – {u}, k-1) || VertexCover(G – {v}, k-1)

  else:
    return true iff G has no edges
```

Vertex Cover: Improved

- Running time is $O(2^k (n+m))$
  - takes $O(n + m)$ time to construct $G - \{u\}$ and $G - \{v\}$

- Easily improved to $O(2^k\, n\, k)$:
  - reject any instance with $m > nk$
    - at most n edges are removed in each recursive call
    - if $m > nk$, then we cannot end up with 0 edges after k recursive calls
  - with $m \leq nk$, running time is now $O(2^k\, nk)$

Coping with NP-Completeness

- Some problems are easy if certain inputs are small
  - (these cases are studied in “parameterized complexity”)
- Example: Knapsack is easy if weight limit (W) is small
- Example: Vertex Cover is easy if node set size (k) is small
  - running time is $O(\text{poly}(n)\ 2^k)$
  - same approach works for large independent sets
    - recall: S is a vertex cover iff V – S is independent

Foreword: Tree Width

> Some problems are easy if certain inputs are small
  – (these cases are studied in "parameterized complexity")

> Example: Knapsack is easy if weight limit (W) is small

> Example: Vertex Cover is easy if node set size (k) is small

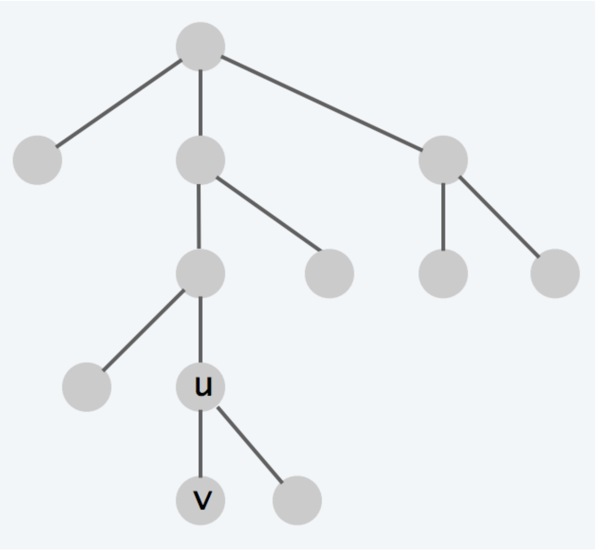
> Example: many problems on graphs are easy if "tree width" is small
  – intuition: problems on trees are easy (dynamic programming)
  – tree width measures how "tree-like" a graph is
  – will discuss more later… (if time)

Outline for Today

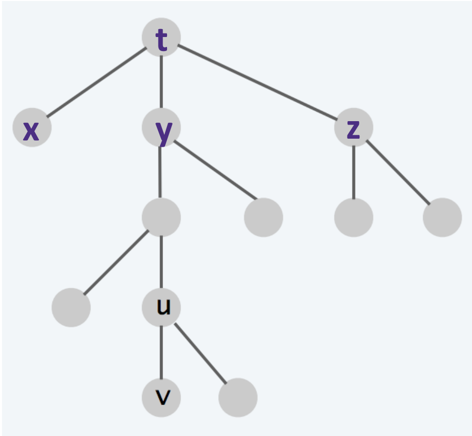
> More Reductions
> Coping with NP-Completeness
> Small Vertex Covers
> Independent Set on Trees
> Approximate Knapsack

Recall: Independent Set

- **Independent Set**: Given graph G and number k, find a subset of k nodes such that no two are connected by an edge

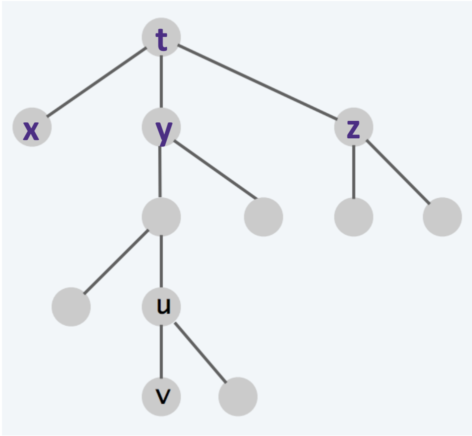


Independent Set on Trees



- Apply dynamic programming...
  - optimal solution on tree rooted at t = larger of
    optimal solution with t excluded
    (optimal solution to which t can be legally added) + 1
  - optimal solution with t excluded =
    (opt solution on x) + (opt solution on y) + (opt solution on z)
    - no problem from edges (t,x), (t,y), (t,z) since t is not included
  - optimal solution with t included =
    (opt solution on x with x excluded) +
    (opt solution on y with y excluded) +
    (opt solution on z with z excluded)
    - no problem from edges (t,x), (t,y), (t,z) since x, y, z not included

Independent Set on Trees



- Apply dynamic programming…
  - optimal solution on tree rooted at t = larger of
    optimal solution with t excluded
    (optimal solution to which t can be legally added) + 1
  - solve 2n problems: one with node included, with with node excluded
  - takes O(n) time all together
- Can be generalized to included weights on nodes
  - as usual, problems on trees are easy with dynamic programming

Special Types of Graphs

- Trees (have seen)
- Bipartite graphs (have seen)
- Planar Graphs (see textbook)
- Chordal Graphs
- Graphs of bounded tree-width (more later...)

some NP-complete problems become easy for each type...

Outline for Today

- More Reductions
- Coping with NP-Completeness
- Small Vertex Covers
- Independent Set on Trees
- Approximate Knapsack

Coping with NP-Completeness

1. Your problem could lie in a special case that is easy
   – saw those above…

2. **Look for approximate solutions**
   – now…

3. Look for “fast enough” exponential time algorithms
   – next time…

Approximation Algorithms

- Large sub-field of algorithms
  - like Randomized Algorithms, it is a full course on its own
- For now, we will just look at one example...

Approximation Algorithms

- **Knapsack**: Given items of the form (w_i, v_i) and a number W, find the largest total value of any subset of total weight at most W
- Dynamic programming solves this in O(nW) time
  - great if W is small
- If W is large, we cannot solve it exactly,
  BUT we can solve it approximately

Approximation Algorithms

- First, need a slightly different algorithm for different version...
- **Knapsack**: Given items of the form (w_i, v_i) and a number W, find the largest total value of any subset of total weight at most W
- **Knapsack 2**: Given items of the form (w_i, v_i) and a number V, find the smallest total weight of any subset of total value at least V
  - can still solve first version with this (binary search on V)
  - more useful here: want to approximate values not weights

Approximation Algorithms

- **Knapsack 2**: Given items of the form (w_i, v_i) and a number V, find the smallest total weight of any subset of total value at least V

- Still solvable by dynamic programming
  - optimal solution on 1 .. n with V = minimum of
    optimal solution on 1 .. n-1 with V and
    (optimal solution on 1 .. n-1 with $V - v_n$) + w_n
  - running time is $O(nV) = O(n^2 (\max v_i))$

Approximation Algorithms

- **Knapsack 2**: Given items of the form (w_i, v_i) and a number V, find the smallest total weight of any subset of total value at least V

- Still solvable by dynamic programming
  - table also lets you answer the usual version of knapsack
  - running time is $O(nV) = O(n^2 (\max v_i))$
  - use this version if V << W

Approximation Algorithms

- **Knapsack 2**: Given items of the form (w_i, v_i) and a number V, find the smallest total weight of any subset of total value at least V
- Dynamic programming solves this in O(nV) time
  - great if V is small
- If W is large, we cannot solve it exactly, BUT we can solve it approximately...

Approximation Algorithms

- If W is large, we cannot solve it exactly,
  BUT we can solve it approximately

- Idea: round the values (up) to multiples of T
  - replace value v by ceil(v / T) T
  - result is between v and v + T

- Can solve in time O(n (V/T)) after dividing weights by T
  - correct since all values are multiples of T

Approximation Algorithms

- Would like to get a $(1 + \varepsilon)$ approximation
  - where ε can be chosen close to 0
  - e.g. $\varepsilon = 0.05$ to get within 5% of correct solution

- Will do so by choosing $T = \varepsilon (\max v_i) / n$
  - the smaller ε is, the less rounding we do

Approximation Algorithms

- Choose $T = \varepsilon (\max v_i) / n$
- **Proposition**: If subset U is optimal on rounded problem, then for any subset V on original problem, sum of values in V ≤ (1 + ε) (sum of values in U)
  - sum of values in V
    ≤ sum of rounded values in V *since we round up*
    ≤ sum of rounded values in U *since U was optimal*
    ≤ sum of values in U + nT
    ≤ (1 + ε) (sum of values in U) *$\max v_i \leq$ sum of values in U*

Approximation Algorithms

- **Proposition**: If subset U is optimal on rounded problem,
  then for any subset V on original problem,
  sum of values in V ≤ (1 + ε) (sum of values in U)

- Our answer is within a factor of (1 + ε) of the true max value
  - take V to be the true optimum above

- Running time is $O(n^2 \text{ (max rounded } v_i)) = O(n^3 / \varepsilon)$
  - recall $T = \varepsilon \text{ (max } v_i) / n$
  - since rounded max $v_i = \text{ceil}(\max v_i / T) \leq \text{ceil}(n / \varepsilon)$

Pseudo-Polynomial Time Algorithms (out of scope)

> We have shown the following for Knapsack:
  - for any $\varepsilon > 0$, there is an algorithm for approximately solving Knapsack, within a factor of $1 + \varepsilon$, in time polynomial in n and $1/\varepsilon$

> Such a result is called an "FPTAS"
  - a fully polynomial-time approximation scheme

> **Theorem**: Almost any optimization problem with an FPTAS has a pseudo-polynomial time solution
  - assumes the answer is integer and polynomially bounded
  - choose ε small enough that ε x (solution) < 1