CSEP 521- Applied Algorithms

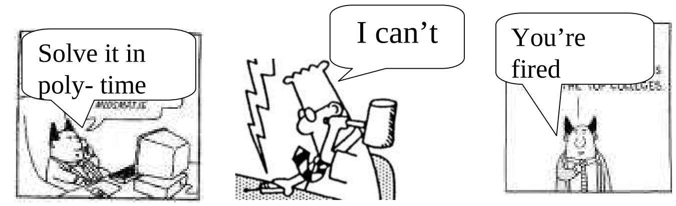
NP-hardness

Reading:

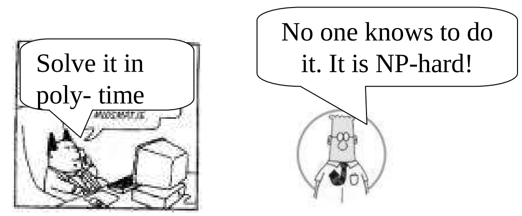
- Skiena, chapter 6
- CLRS, chapter 36 (1^{st} Ed.)
  chapter 34 (2^{nd} Ed.)

NP-Completeness Theory

I.



II.



NP-Completeness Theory

- Explains why some problems are hard and probably not solvable in polynomial time.
- Invented by Cook in 1971.
- Talks about the problems, independent of the implementation the machine, or the algorithm.

Polynomial-Time Algorithms

- Some problems are intractable:
  as they grow large, we are unable to solve them in reasonable time.
- What constitutes reasonable time? Standard working definition: polynomial time
  - On an input of size n the worst-case running time is $O(n^k)$ for some constant k
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$
  - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - Of course: most of the algorithms we've studied so far provide polynomial-time solution to some problems.
  - We define **P** to be the class of problems solvable in polynomial time.
- Are all problems solvable in polynomial time?
  - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
  - Such problems are clearly intractable, not in **P**

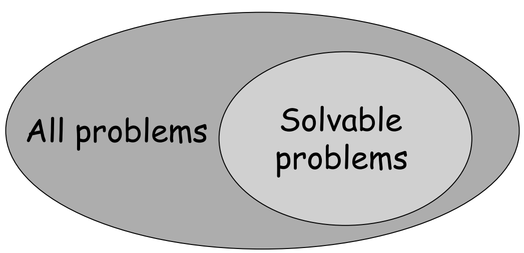
The Unsolvable Halting Problem

- For a given program P and input x, does P halt on x?
- Suggested solution: Let's run P on x and check.
- But what if P doesn't halt after 2 minutes? 10 days? A year?

Turing: The halting problem cannot be solved!

Proof: In bonus slides.

So some problems cannot be solved at all



We will explore the 'solvable area', and will distinguish between problems that can be solved efficiently and those that cannot be solved efficiently.

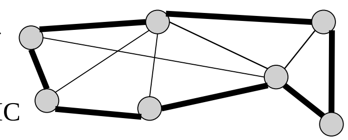
NP-Complete Problems

- The *NP-Complete* problems are an interesting class of solvable problems whose status is unknown
  - No polynomial-time algorithm has been discovered for an NP-Complete problem.
  - No above-polynomial lower bound has been proved for any NP-Complete problem, either.
- We call this the *P = NP question*
  - The biggest open problem in CS.

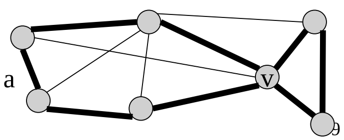
An NP-Complete Problem: Hamiltonian Cycles

- An example of an NP-Complete problem:
  - A *hamiltonian cycle* of an undirected graph is a simple cycle that contains every vertex.
  - The hamiltonian-cycle problem: given a graph *G*, does it have a hamiltonian cycle?
  - A naive algorithm for solving the hamiltonian-cycle problem: check all paths.
  - Running time? Exponential in size of *G.*

✓
a HC



Not a HC



P and NP

- As mentioned, **P** is the set of problems that can be solved in polynomial time
- **NP** (*nondeterministic polynomial time*) is the set of problems that can be solved in polynomial time by a *nondeterministic* computer

Non-determinism

- Think of a non-deterministic computer as a computer that magically "guesses" a solution, then has to verify that it is correct.
  - If a solution exists, the computer always guesses it
  - One way to imagine it: a parallel computer that can freely spawn an infinite number of processes.
    - Have one processor work on each possible solution.
    - All processors attempt to verify that their solution works.
    - a processor that finds it has a working solution announce it.
  - So: **NP** = problems *verifiable* in polynomial time.

P and NP

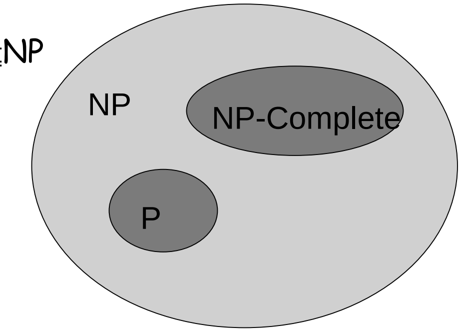
- Summary so far:
  - **P** = problems that can be solved in polynomial time
  - **NP** = problems for which a solution can be verified in polynomial time
  - Unknown whether **P = NP** (most suspect not)
- Hamiltonian-cycle problem is in **NP**:
  - Cannot solve in polynomial time.
  - Easy to verify solution in polynomial time.

NP-Complete Problems

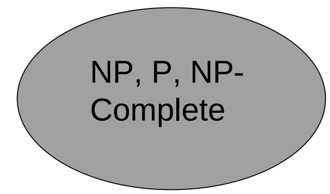
- We will see that NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time…
  - …then *every* NP-Complete problem can be solved in polynomial time…
  - …and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)
  - Thus: solve hamiltonian-cycle in $O(n^{100})$ time, you've proved that **P = NP**. Retire rich & famous.

NP Problems

For sure $P \subseteq NP$



But maybe P=NP ??



Why Prove NP-completeness?

- Though nobody has proven that **P** != **NP**, if you prove a problem is NP-Complete, most people accept that it is probably intractable.
- Therefore it can be important to prove that a problem is NP-Complete
  - Don't need to come up with an efficient algorithm.
  - Can instead work on *approximation algorithms.*

Reduction

- The crux of NP-Completeness is *reducibility*
  - Informally, a problem P can be reduced to another problem Q if *any* instance of P can be "easily rephrased" as an instance of Q, the solution to which provides a solution to the instance of P
    - *What do you suppose "easily" means?*
    - This rephrasing is called *transformation*
  - Intuitively: If P reduces to Q, P is "no harder to solve" than Q.

Reducibility - An example

- P: Given a set of Booleans {$x_i \in$ TRUE, FALSE}, is at least one TRUE?
- Q: Given a set of integers, is their sum positive?
- Transformation: given ($x_1, x_2, \ldots, x_n$) booleans, let ($y_1, y_2, \ldots, y_n$) be a set of integers where $y_i = 1$ if x_i = TRUE, and $y_i = 0$ if x_i = FALSE.
- P is no harder than Q: if we can solve Q we can run the transformation to get a solution to P.

Using Reductions

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-complete:
  - P is NP-complete if $P \in NP$ and P is NP-hard.
- Definition of NP-Hard:
  - P is NP-hard if all problems R of NP are reducible to P. Formally: $R \leq_p P, \forall R \in NP$
- If $P \leq_p Q$ and P is NP-hard, Q is also NP-hard.

Using Reductions

- Given one NP-Complete problem, we can prove that many interesting problems NP-Complete. This includes:
  - Graph coloring
  - Hamiltonian path/cycle
  - Knapsack problem
  - Traveling salesman
  - Job scheduling
  - Many, many, many more (see the compendium)

Optimization v.s. Decision

To simplify things, we will worry only about *decision problems* with a yes/no answer

- Many problems are *optimization problems*, but we can often re-cast them as decision problems

Example: Graph coloring.

- Optimization problem: what is the minimal number of colors needed to color G?
- Reporting problem: Can G be colored using k colors? If so, report a legal k-coloring.
- Decision problem: Can G be colored using k colors?

Subset Sum

- Input: Integers $a_1, a_2, \ldots, a_n, b$
- Output: Determine if there is subset

$$X \subseteq \{1,2,\ldots, n\}$$

with the property $\sum_{i \in X} a_i = b$

- Non-deterministic algorithm: Guess the subset *X* and check the sum adds up to *b.*

Decision Problems are Polynomial Time Equivalent to their Reporting Problems

- Example: Subset sum
  - Decision Problem: Determine if a subset sum exists.
  - Reporting Problem: Determine if a subset sum exists and report one if it does.
- Using decision to report
  - Let subset-sum(A,b) returns true if some subset of A adds up to b. Otherwise it returns false.

Reporting Reduces to Decision

```
Assume that subset-sum ({a_1,…,a_n},b) is true
X := the empty set;
for i = 1 to n do
   if subset-sum({a_{i+1},…,a_n},b - a_i) then
      add i to X;
      b := b - a_i;
```

Example: {**3**, 5, 2, 7, 4, 2}; b = 11
{**5**, 2, 7, 4, 2}; b = 11-**3** ? True, X = {3}, b = 8
{**2**, 7, 4, 2}, b = 8-**5** ? False
{**7**, 4, 2}, b = 8-**2** ? True, X = {3,2}, b = 6
{**4**, 2}, b = 6-**7** ? False
{**2**}, b = 6-**4** ? True, X = {3,2,4}, b = 2
b = 4-**2** ? True, X ={3,2,4,2}

Optimization Reduces to Decision

Example: Graph coloring

•k=1, repeat:

•Is G k-colorable?

•If yes, k is the answer to the optimization problem.

•If no, k := k+1.

•Can do even better with binary search.

•In both cases, the number of iterations is polynomial (G is clearly n-colorable)

Proving NP-Completeness

- How do we prove a problem P is NP-Complete?
  - Pick a known NP-Complete problem Q
  - Reduce Q to P (show $Q \leq_p P$, use P to solve Q)
    - Describe a transformation that maps instances of Q to instances of P, s.t. "yes" for P = "yes" for Q
    - Prove the transformation works
    - Prove it runs in polynomial time
  - and yeah, prove $P \in$ **NP**
- We need at least one problem for which NP-hardness is known. Once we have one, we can start reducing it to many problem.

The SAT Problem

- The first problems to be proved NP-Complete was *satisfiability* (SAT):
  - Given a Boolean expression on *n* variables, can we assign values such that the expression is TRUE?
  - Ex: $((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
  - **Cook's Theorem:** The satisfiability problem is NP-Complete
    - Note: Argue from first principles, not reduction (any computation can be described using SAT expressions)
    - Proof: not here

Conjunctive Normal Form

- Even if the form of the Boolean expression is simplified, the problem may be NP-Complete
  - *Literal*: an occurrence of a Boolean or its negation
  - A Boolean formula is in *conjunctive normal form*, or *CNF*, if it is an AND of clauses, each of which is an OR of literals
    - Ex: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5)$

  3-CNF: each clause has exactly 3 distinct literals
  - Ex: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5 \vee x_3 \vee x_4)$
  - Note: true if at least one literal in each clause is true

The 3-CNF Problem

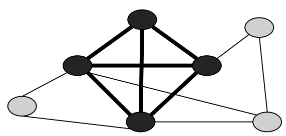
- Theorem: Satisfiability of Boolean formulas in 3-CNF form (the *3-CNF Problem*) is NP-Complete
  - Proof: not here
- The reason we care about the 3-CNF problem is that it is relatively easy to reduce to others.
  - Thus, knowing that 3-CNF is NP-Complete we can prove many seemingly unrelated problems are NP-Complete.

The k-clique Problem

- *A clique in a graph G is* a subset of vertices fully connected to each other, i.e. a complete subgraph of G.
- The *clique problem*: how large is the maximum-size clique in a graph?
- *Can we turn this into a decision problem?*
- A: Yes, we call this the *k-clique problem*
- *Is the k-clique problem within **NP**?*

  Yes: Nondeterministic algorithm: guess k vertices then check that there is an edge between each pair of them.

4-clique:



3-CNF → Clique

- How can we prove that k-clique is NP-hard?
- We need to show that if we can solve k-clique then we can solve a problem which is known to be NP-hard.
- We will do it for 3-CNF:
- Given a 3-CNF formula, we will transform it to an instance of k-clique (a graph and a number k), for which a k-clique exists iff the 3-CNF formula is satisfiable.

3-CNF → Clique

- The reduction:
  - Let F = $C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a 3-CNF formula with k clauses, each of which has 3 distinct literals.
  - For each clause, put three vertices in the graph, one for each literal.
  - Put an edge between two vertices if they are in different triples and their literals are consistent, meaning not each other's negation.

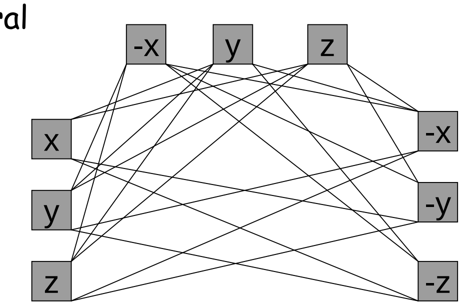
Construction by Example

$$F = (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

literal

clause

G

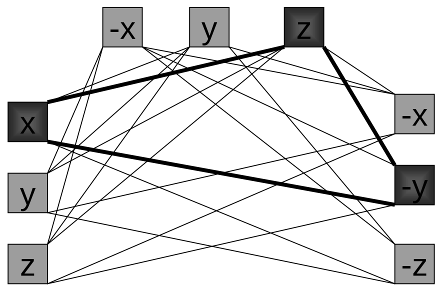


An edge means 'these two literals do not contradict each other'.

Construction by Example

$$F = (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$
$$x = 1, y = 0, z = 1$$

G



Any clique of size k must include exactly one literal from each clause.

General Construction

$$F = \bigcap_{i=1}^{k} \bigcup_{j=1}^{3} a_{ij} \quad \text{where } a_{ij} \in \{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$$

literals

$G = (V,E)$ where

$$V = \{a_{ij} : 1 \le i \le k, 1 \le j \le 3\}$$

$$E = \{\{a_{ij}, a_{i',j'}\}: i \neq i' \text{ and } a_{ij} \neq \neg a_{i'j'}\}$$

k is the number of clauses

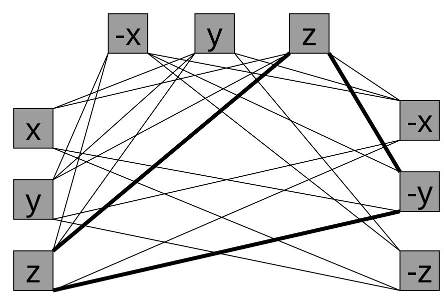
The Reduction Argument

- We need to show
  - F satisfiable implies G has a clique of size k.
    - Given a satisfying assignment for F, for each clause pick a literal that is satisfied. Those literals in the graph G form a k-clique.
  - G has a clique of size k implies F is satisfiable.
    - Given a k-clique in G, assign TRUE to each literal in the clique. This yields a satisfying assignment to F (why?).

Clique to Assignment

$$F = (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

G

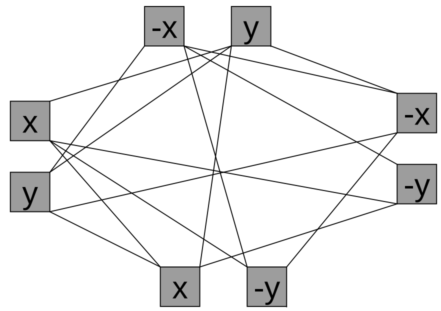


$$y = 0, z = 1$$

Assignment to Clique (2-CNF)

$$F = (x \vee y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg y)$$

G

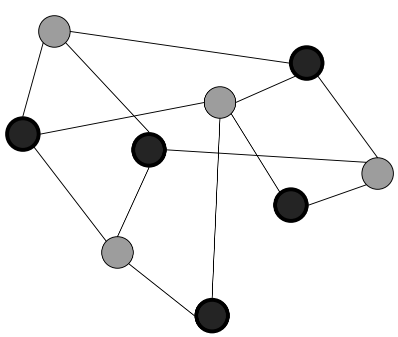


G has no 4-clique → no assignment exists.
What is the max-clique size?
How does this value related to the formula?

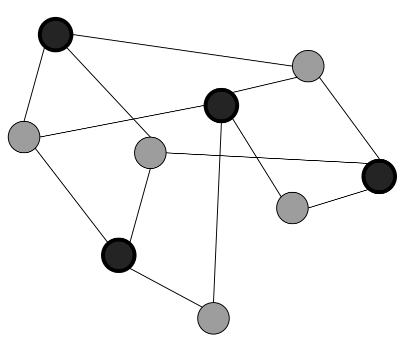
The Vertex Cover Problem

- A vertex cover for a graph G is a set of vertices incident to every edge in G
- The vertex cover problem: what is the minimum size vertex cover in G?
- Restated as a decision problem: does a vertex cover of size k exist in G?
- Theorem: vertex cover is NP-Complete

Vertex Cover (Example)



A vertex cover of size 5

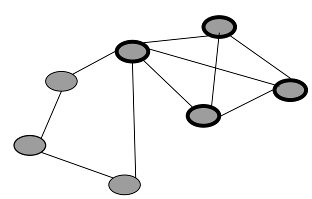


A vertex cover of size 4

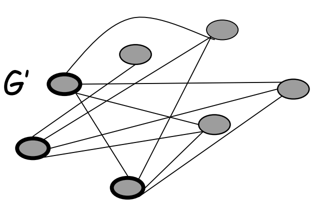
Clique → Vertex Cover

- First, show vertex cover in NP (How?)
- Next, reduce k-clique to vertex cover:
  - The complement G_C of a graph G contains exactly those edges not in G
  - Compute G_C in polynomial time
  - Claim: G has a clique of size k iff G_C has a vertex cover of size |V| - k

G



G'

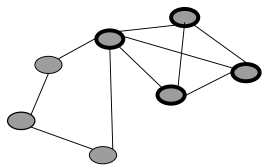


Clique → Vertex Cover

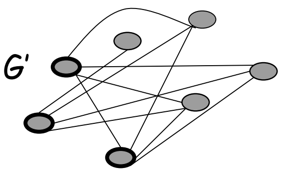
Claim: If G has a clique of size k, then G_C has a vertex cover of size |V| - k

- Let V' be the k-clique
- Then V-V' is a vertex cover in G_C
  - Let (u,v) be any edge in G_C
  - Then u and v cannot both be in V' (*why?*)
  - Thus at least one of u or v is in V-V' (*why?*), so the edge (u,v) is covered by V-V'
  - Since true for *any* edge in G_C, V-V' is a VC.

G



G'

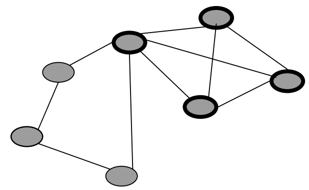


Clique → Vertex Cover

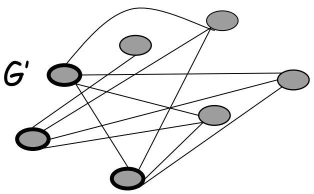
Claim: If G_C has a vertex cover $V' \subseteq V$, with |V'|=|V| - k, then G has a clique of size k

- For all u,v ∈ V, if (u,v) ∈ G_C then u ∈ V' or v ∈ V' or both (Why?)
- In other words: if u ∉ V' and v ∉ V', then (u,v) ∈ E
- Therefore, all vertices in V-V' are connected by an edge, thus V-V' is a clique
- Since |V| - |V'| = k, the size of the clique is k

G



G'



The Traveling Salesman Problem:

- A well-known optimization problem:
  - Optimization variant: a salesman must travel to n cities, visiting each city exactly once and finishing where he begins. How to minimize travel time?
  - Model as complete graph with cost c(i,j) to go from city i to city j
- How would we turn this into a decision problem?
  - Answer: ask if there exists a path with cost < k

The Traveling Salesman Problem:

- Asides:
  - TSPs (and variants) have enormous practical importance
    - E.g., for shipping and freighting companies
    - Lots of research into good approximation algorithms
  - Recently made famous as a DNA computing problem

Hamiltonian Cycle ⇒ TSP

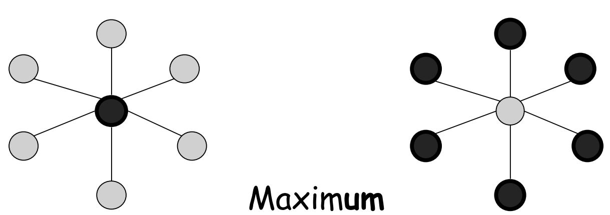
- The hamiltonian-cycle problem: given a graph *G*, is there a simple cycle that contains every vertex?
- To transform ham. cycle problem on graph *G* = (V,E) to TSP, create graph *G'* = (V,E'):
- *G'* is a complete graph
- Edges in E' also in E have cost 0
- All other edges in E' have cost 1
- TSP: is there a TS cycle on *G'* with cost 0?
  - If G has a ham. cycle, *G'* has a TS cycle with cost 0
  - If *G'* has TS cycle with cost 0, every edge of that cycle has cost 0 and is thus in *G*. Thus, *G* has a ham. cycle.

Other NP-Complete Problems

- *Partition:* Given a set of integers, whose total sum is 2*S*, can we partition them into two sets, each adds up to *S*?
- *Subset-sum*: Given a set of integers, does there exist a subset that adds up to some target *T*?
- *Graph coloring*: can a given graph be colored with *k* colors such that no adjacent vertices are the same color?

Independent Set

- Input: A graph G=(V,E), k
- Problem: Is there a subset S of V of size at least k such that no pair of vertices in S has an edge between them.
- Maximum independent set problem: find a maximum size independent set of vertices.



Maxim**al** independent set

Maxim**um** independent set

Steiner Tree

- Input: A graph G=(V,E), a subset T of the vertices V, and a bound B
- Problem: Is there a tree connecting all the vertices of T of total weight at most B?

- Application: Network design and wiring layout.
- The case T=V is polynomially solvable (this is the MST problem).

Exact Cover

- Input: A set $U = \{u_1, u_2, \ldots u_n\}$ and subsets

$$S_1, S_2, \ldots, S_m \subseteq U$$

- Output: Determine if there is a set of disjoint sets that union to U, that is, a set X such that:

$$X \subseteq \{1,2,\ldots,m\}$$

$$i, j \in X \text{ and } i \neq j \text{ implies } S_i \cap S_j = \Phi$$

$$\bigcup_{i \in X} S_i = U$$

Example of Exact Cover

$$U = \{a,b,c,d,e,f,g,h,i\}$$

$$\{a,c,e\},\{a,f,g\},\{b,d\},\{b,f,h\},\{e,h,i\},\{f,h,i\},\{d,g,i\}$$

Exact Cover:

$$\{a,c,e\},\{b,f,h\},\{d,g,i\}$$

3-Partition

- Input: A set of numbers $A = \{a_1, a_2, \ldots, a_{3m}\}$ and a number B such that $B/4 < a_i < B/2$ and

$$\sum_{i=1}^{3m} a_i = mB.$$

- Output: Determine if A can be partitioned into $S_1, S_2, \ldots, S_m$ such that for all i

$$\sum_{j \in S_i} a_j = B.$$

Note: each S_i must contains exactly 3 elements.

Example of 3-Partition

- A = {26, 29, 33, 33, 33, 34, 35, 36, 41}
- B = 100, m = 3
- 3-Partition:
  - 26, 33, 41
  - 29, 36, 35
  - 33, 33, 34

Bin Packing

- Input: A set of numbers $A = \{a_1, a_2, \ldots, a_m\}$ and numbers B (capacity) and K (number of bins).
- Output: Determine if A can be partitioned into $S_1, S_2, \ldots, S_K$ such that for all i

$$\sum_{j \in S_i} a_j \leq B.$$

Bin Packing Example

- A = {2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5}
- B = 10, K = 4
- Bin Packing:
  - 3, 3, 4
  - 2, 3, 5
  - 5, 5
  - 2, 4, 4

Perfect fit!

Comments on NP-completeness proofs

- hardest part -- choosing a good problem from which to do reduction
- must do reduction from arbitrary instance
- common error -- backwards reduction. Remember that you are using your problem as a black box for solving known NPC problem
- freedom in reduction: if problem includes parameter, can set it in a convenient way
- size of problem can change as long as it doesn't increase by more than polynomial

Comments cont.

- When a problem is generalization of known NP-complete problem, a reduction is usually easy.
- Example: Set Cover
  - given U, set of elements, and collection $S_1, S_2, \ldots, S_n$ of subsets of U, and an integer k
  - determine if there is a subset W of U of size at most k that intersects every set S_i
- Reduction from Vertex Cover
  - U set of vertices
  - S_i is the i^{th} edge

The Unsolvable Halting Problem

- For a given program P and input x, does P halt on x?

Turing: The halting problem cannot be solved!

Proof: Assume that there is an algorithm *Halt*(*a*, *i*) that decides if the algorithm encoded by the string *a* will halt when given as input the string *i*,

The Halting Problem

Consider the following program

```
Funny (s)  // s is a string decoding a program.
  if (Halt(s, s) = "no") return ("yes")
  else {some infinite loop}
```

Note: Funny(s) halts ⇔ Halt(s, s)=no.

Let T be the string decoding the program Funny.
What is the output of Halt(T, T)?
If the output is 'No' then Halt (T,T)= Yes
If the output is 'Yes' then Halt (T,T)= No