CSE 417
Branch & Bound (pt 1)
NP-Completeness
Reminders

> HW7 is due today

> HW8 will be posted shortly
  – 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 previous topics

> Modeling techniques
  - shortest paths (intersection of both network flows and dynamic programming)
  - binary search
  - network flows (max flow & min cost flow)

> Design techniques
  - divide and conquer
  - dynamic programming
  - branch and bound
    > applies to problems too hard to solve with the other techniques
    > (in particular, it applies to NP-complete problems, defined shortly...)
Outline for Today

- P and NP
- Reductions
- Some NP-complete Problems
**Definition:** P is the set of problems that can be solved in polynomial time by a sufficiently large computer
- (one with enough memory)

**Theoretical details:**
- polynomial time in the number of bits of input
  - excludes pseudo-polynomial time algorithms
- only decision problems
  - equivalent to optimization due to binary search
- algorithm must run on a Turing machine
  - equivalent to usual machines
P: History

> “Invented” by Jack Edmonds (1965)
  > earlier work often focused on actual running times on real machines
  > Edmonds wanted to explain the significance of his matching algorithm
    > solved general matching (harder than bipartite matching) in polynomial time
    > paper was rejected multiple times

> (Note: von Neumann and others also helped “invent” P)
Polynomial time algorithms are typically more useful in practice:
- some pseudo-poly and exponential time algorithms are useful (more later...)
- some polynomial algorithms are not useful (e.g., $O(n^8)$)
- in general, though, it is a good dividing line

Need this definition to get a reasonable theory:
- want the following pieces:
  - linear time is fast
    - if function $g$ is fast and $f$, which calls $g$ as a subroutine, is fast if we count calls to $g$ as one operation, then $f$ is fast (composability)
  - those two imply polynomial time is fast
Definition: NP is the set of problems for which a correct answer can be verified in polynomial time
- i.e., the problem of checking whether an answer is correct is in P
- \( P \subseteq NP \) but NP is believed to be strictly larger
- (implies that a correct answer must be polynomial size...
  > so these are problems with small (polynomial size) proofs of correctness)

This is not true of all problems, e.g.:
- testing equivalence of regular expressions
- solving (generalized) chess or go
  > proof of a winning strategy is very large
NP-Completeness

> **Definition:** Problem X is **NP-hard** if *any* problem in NP could be solved in polynomial time if given a function that solves X

> Shows that X is as hard as any problem in NP

> **Definition:** Problem X is **NP-complete** if it is in NP and is NP-hard

> NP-complete problems are the hardest problems in NP
  - can solve these problems iff P = NP
    > solving these would solve them all
**NP-Completeness**

> Boolean Satisfiability (SAT): given a logical formula on variables $x_1, ..., x_n$ using only **and**, **or**, & **not**, determine whether there is a setting of the variables to T/F so that the formula evaluates to T
>  - in NP
>  - short proof of correctness: give the T variables and the F variables

> **Theorem** (Cook–Levin): SAT is NP-complete
>  - (see textbook for a proof)
P vs NP

Can win $1,000,000 by proving that $P \neq NP$ (or $P = NP$)...

Proving $P = NP$ would be easier (if it were true)
- just invent a polynomial time algorithm for any NP-complete problem
- unfortunately, we don’t think such an algorithm exists
  > since so many smart people have been trying hard for decades

Proving $P \neq NP$ is deviously difficult...
P vs NP

> Can win $1,000,000 by proving that P ≠ NP (or P = NP)...

> Proving P ≠ NP is deviously difficult...
  > if P = NP, then we would have a poly time algorithm to find the proof
    > (finding a proof of certain logical statements is NP-complete)
    > unfortunately, there would be no proof in that case
  > if P ≠ NP, then we could hope for a “natural proof”
    > (formalizes the idea of how most would normally try to prove this)
    > unfortunately, Razborov & Rudich proved that such a proof would actually imply that P = NP
  > most other reasonable ideas for proofs have been ruled out
    > hard to find an approach that seems workable & hasn't been ruled out
In mathematics, P vs NP is unsettled
   though most believe they are unequal

In physics, P ≠ NP is often taken as a physical law
   (see recent work on black holes etc.)
   simple version: “this Ising model must take exponential time to cool down because if not you could use it to solve NP-complete problems in poly time”

(Actually, nearly all physicists believe BQP, not P, is the set of problems that can be solved physically...
   these are problems solvable on quantum computers)
Outline for Today

- P and NP
- Reductions
- Some NP-complete Problems
Reductions

> **Definition**: Problem Y is polynomial-time reducible to X, denoted $Y \leq_p X$, if there is a polynomial time algorithm that solves Y assuming a polynomial-time subroutine for solving X.
  
  - algorithm makes $\text{poly}(n)$ calls to the subroutine and does $\text{poly}(n)$ other work

> The algorithm here is called a (Cook) “reduction”
  
  - show that Y is no harder than X ($Y \leq_p X$) by giving a reduction from Y to X
Definition: Problem Y is polynomial-time reducible to X, denoted \( Y \leq_p X \), if there is a polynomial time algorithm that solves Y assuming a polynomial-time subroutine for solving X.

Re-definition: X is NP-hard if, for every Y in NP, \( Y \leq_p X \), i.e., X is at least as hard as any problem in NP.
> **Definition:** Problem Y is polynomial-time reducible to X, denoted $Y \leq_P X$, if there is a polynomial time algorithm that solves Y assuming a polynomial-time subroutine for solving X.

> **Warning:** do not confuse the order of X and Y!
  - a reduction from Y to X shows...
    > if you could solve X efficiently, then you could solve Y efficiently
    > so Y is no harder than X ($Y \leq_P X$)
> Only need **one** reduction from an NP-hard problem to X to prove that X is NP-hard

  – suppose Y is NP-hard
    > i.e., $Z \leq_p Y$ for every $Z$ in NP
  – suppose $Y \leq_p X$
  – then X is NP-hard
    > $Z \leq_p Y \leq_p X$ (so $Z \leq_p X$) for every $Z$ in NP
Dick Karp popularized NP-completeness using reductions...
99% of known NP-complete problems are from reductions
- reductions seem to be much easier than direct proofs

Reductions are a useful tool in practice
- they let you prove that there is almost certainly no way to solve it efficiently
- so you can stop trying to find an exact solution
Reductions in Practice

> Reductions are a useful tool in practice
  – they let you prove that there is almost certainly no way to solve it efficiently
  – so you can stop trying to find an exact solution

> In practice, almost every NP problem is in P or is NP-complete
  – hence, you can either find an algorithm or prove there is none
  – on the other hand..

> Theorem (Ladner): if P ≠ NP, then there are infinitely many problems that are in NP, not in P, and not NP-complete.
Garey & Johnson

> "Computers and Intractability" by Garey & Johnson contains over 300 NP-complete problems
  – can give you a quick answer for many, many problems
  – book is from 1979

> More problems have been found since then
  – see the web
More Theory About Reductions

> **Definition**: Problem Y is polynomial-time reducible to X, denoted $Y \leq_p X$, if there is a polynomial time algorithm that solves Y assuming a polynomial-time subroutine for solving X.

> **Technical Details**
  - Karp used a weaker (more difficult to achieve) notion of reduction:
    > algorithm can only make one call to the subroutine AND
    > algorithm must simply return whatever that returns
    > i.e., algorithm constructs one problem for X to solve that has the property that it is solvable iff the Y problem given is solvable.
More Theory About Reductions

Definition: Problem Y is polynomial-time reducible to X, denoted $Y \leq_p X$, if there is a polynomial time algorithm that solves Y assuming a polynomial-time subroutine for solving X.

Technical Details

– Karp used a weaker (more difficult to achieve) notion of reduction
– It is (AFAIK) still an open problem whether this difference matters
  > Last I checked, every NP-completeness reduction could be made even weaker
    – (Specifically, they can be done as Karp reductions in log space)
– Like the textbook, I will ignore the difference
Outline for Today

> P and NP
> Reductions
> Some NP-complete Problems
NP-Complete Problems

> Have already seen some...

> Knapsack
  – pseudo-poly time algorithm shows only hard with big numbers

> 0/1 (integer) linear programming
  – reason why min cost flow problems (special case) are important

> Independent set
  – find set of k nodes in a graph with no edges between them
NP-Complete Problems

> Some are *opposites* of easy problems...

> Longest path
  – cannot simply negate edge lengths...
  – our algorithms assume no negative length cycles

> Max cut
  – can negate costs but not capacities (those were assumed $\geq 0$)
NP-Compete Problems

> Both textbook and Garey & Johnson look at six problem types
– most useful to reduce from ("easiest" of NP-hard problems)

<table>
<thead>
<tr>
<th>Packing</th>
<th>independent set</th>
<th>clique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covering</td>
<td>vertex cover</td>
<td></td>
</tr>
<tr>
<td>Constraint Satisfaction</td>
<td>3-SAT</td>
<td>SAT</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Hamiltonian cycle</td>
<td>TSP</td>
</tr>
<tr>
<td>Partitioning</td>
<td>3D matching</td>
<td></td>
</tr>
<tr>
<td>Numerical</td>
<td>partition</td>
<td>subset sum, knapsack</td>
</tr>
</tbody>
</table>
Packing Problems

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

- **Clique**: Given graph $G$ and number $k$, find a subset of $k$ nodes such that every pair are connected by an edge.

- **Reductions (Independent Set $\equiv_p$ Clique)**:
  - Let $G'$ be the opposite graph:
    - $N' = N$
    - $(u,v)$ in $E'$ iff $(u, v)$ not in $E$
  - nodes are independent in $G$ iff they are clique in $G'$.
Covering Problems

> **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.

> **Reduction (Independent Set $\equiv_p$ Vertex Cover):**
  
  - subset $S$ is independent iff $V - S$ is a vertex cover:
    
    > $S$ is independent iff for every $(u,v)$ in $E$, either $u$ not in $S$ or $v$ not in $S$ iff for every $(u,v)$ in $E$, either $u$ in $V - S$ or $v$ in $V - S$ iff $V - S$ is covering
  
  - reduction to vertex cover:
    
    > call vertex cover with $k$ replaced by $n - k$
  
  - reduction to independent set: same
Constraint Satisfaction Problems

> **SAT**: Given a logical formula on variables $x_1, ..., x_n$ using only **and**, **or**, & **not**, determine whether there is a setting of the variables to T/F so that the formula evaluates to T

> **3-SAT**: As above, but formula is of the form “$t_1 \text{ and } t_2 \ldots \text{ and } t_m$”, where each $t_i$ is of the form “$f_{i1} \text{ or } f_{i2} \text{ or } f_{i3}$”, where each $f_{ij}$ is either “$x_k$” or “not $x_k$” for some $k$

  - e.g.:  
    - $(\text{not } x_1) \text{ or } x_2 \text{ or } x_3$ and
    - $(x_1 \text{ or (not } x_2 \text{) or } x_3)$ and
    - $(\text{not } x_1 \text{) or (not } x_2 \text{) or (not } x_3))$
Constraint Satisfaction Problems

> **SAT**: Given a logical formula on variables $x_1, ..., x_n$ using only **and**, **or**, & **not**, determine whether there is a setting of the variables to T/F so that the formula evaluates to T
  >  – Cook proved directly that this is NP-complete

> **3-SAT**: As above, but formula is of the form “$t_1 \text{ and } t_2 \text{ and } ... \text{ and } t_m$”, where each $t_i$ is of the form “$f_{i1} \text{ or } f_{i2} \text{ or } f_{i3}$”, where each $f_{ij}$ is either “$x_k$” or “$\text{not } x_k$” for some $k$
  >  – can see that 3-SAT $\leq_p$ SAT because former is special case
  >  – requires work to show the opposite (will skip details)
Sequencing Problems

> **Hamiltonian Cycle**: Given a graph G, find a cycle that visits every node exactly once (a “simple” cycle of length n)

> **Traveling Salesperson Problem (TSP)**: Given weighted graph G and number v, find a Hamiltonian cycle of length at most v

> **Reduction (Hamiltonian Cycle \( \leq_p \) TSP)**:
  - take the weight of each edge to be 1
  - find a cycle of length n
Partitioning Problems

> **3D Matching**: Given disjoint sets $X$, $Y$, $Z$ and a set of $M$ of triples of the form $(x,y,z)$, with $x$ in $X$, $y$ in $Y$, and $z$ in $Z$ and a number $k$, find a set of $k$ triples with no $x$'s, $y$'s, or $z$'s in common
>  – could call this “tri-partite matching”

> **Reduction (bipartite matching $\leq_p$ 3D matching)**:
>  – node set is split into $X$ and $Y$
>  – let $Z$ be the set of edges
>  – triple for each edge $(u, v, (u,v))$
>    > no triples have the same $Z$ part, so intersection means $u$ or $v$ is same
>  – note: this does not show that bipartite matching is NP-hard!
Numerical Problems

> **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 \(W\).

> **Subset Sum**: Given weights \(w_1, \ldots, w_n\) and a number \(W\), find a subset whose total weight is exactly \(W\).

> **Reduction (Subset Sum \(\leq_p\) Knapsack)**:
  - choose the values equal to the weights
  - Knapsack gives the largest sum of weights \(\leq W\)
  - just check if it equals \(W\)
Numerical Problems

MY HOBBY:
Embedding NP-Complete Problems in Restaurant Orders

CHOTCHKIES RESTAURANT

APPLÉTIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80

SANDWICHES
BARBECUE 6.55

WE'D LIKE EXACTLY $15.05
WORTH OF APPETIZERS, PLEASE.

...EXACTLY? UHH...

HERE, THESE PAPERS ON THE KNAPSACK
PROBLEM MIGHT HELP YOU OUT.

LISTEN, I HAVE SIX OTHER
TABLES TO GET TO—

AS FAST AS POSSIBLE, OF COURSE. WANT
SOMETHING ON TRAVELING SALESMAN?
Numerical Problems

> **Partition**: Given set $W = \{w_1, ..., w_n\}$ of weights, find a subset $S$ such that total weight of $S$ is total weight of $W - S$
>  - i.e., does it split into two parts of equal weights?

> **Reduction (Subset Sum $\leq_P$ Partition)**:
>  - add two extra weights: $(\text{sum of weights}) + W$ and $2 \times (\text{sum of weights}) - W$
>  - total weight of all items is now $4 \times (\text{sum of weights})$
>  - two new elements cannot be in the same side of partition

\[
\begin{align*}
v_{n+1} &= 2 \sum_i w_i - W & W \\
v_{n+2} &= \sum_i w_i + W & \sum_i w_i - W
\end{align*}
\]