CSE 101 Final Exam Review
NP-Completeness (Ch 8)

- NP-Problems
- Reductions
- NP-Completeness & NP-Hardness
- SAT
- Hamiltonian Cycle
- Zero-One Equations
- Knapsack
Problems with brute force search algorithms are said to be in **Nondeterministic Polynomial** time (NP).

**NP-Decision** problems ask if there is some object that satisfies a polynomial time-checkable property.

**NP-Optimization** problems ask for the object that maximizes (or minimizes) some polynomial time-computable objective.
Examples of NP Problems

- SAT
- TSP
- Hamiltonian Cycle
- Knapsack
- Maximum Independent Set
SAT

Problem: Formula-SAT

Given a logical formula in a number of Boolean variables, is there an assignment to the variables that causes the formula to be true?
Hamiltonian Cycle (in text as Rudruta Path)

Given an undirected graph G is there a cycle that visits every vertex exactly once?
$1,000,000 \text{ Question: } \text{Is } P = NP? \\
Is it the case that every problem in NP has a polynomial time algorithm?

• If yes, every NP problem has a reasonably efficient solution.
• If not, some NP problems are fundamentally difficult

Most computer scientists believe $P \neq NP$. 
(But proving anything is very very hard)
Reductions

Reductions are a method for proving that one problem is at least as hard as another.

We show that if there is an algorithm for solving A, then we can use this algorithm to solve B. Therefore, B is no harder than A.
Hamiltonian Cycle $\rightarrow$ TSP

Hamiltonian Cycle Instance

TSP Instance

Cost = 1

Cost = 2
Reduction $A \rightarrow B$

- Instance of problem $A$
  - Solution to $A$ instance
  - Polynomial time reduction algorithm

- Instance of problem $B$
  - Solution to $B$ instance
  - Hypothetical algorithm for $B$
Reduction $A \rightarrow B$

If we have algorithms for reduction and interpretation:

• Given an algorithm to solve $B$, we can turn it into an algorithm to solve $A$.

• This means that $A$ might be easier to solve than $B$, but cannot be harder.
Circuit SAT

**Problem:** Given a circuit C with several Boolean inputs and one Boolean output, determine if there is a set of inputs that give output 1.

![Circuit Diagram]

**Important Reduction:**
Any NP decision problem $\rightarrow$ Circuit SAT
Any NP Decision Problem → Circuit SAT

• Any NP decision problem asks if there is some X that satisfies a polynomial-time checkable property.
• In other words, for some polynomial-time computable function F, it asks if there is an X so that F(X) = 1.
• Create a circuit C that computes F. The problem is equivalent to asking if there is an input for which C outputs 1.
NP-Complete

Circuit-SAT is our first example of an NP-Complete problem. That is a problem in NP that is at least as hard as any other problem in NP.

• **Good news:** If we find a polynomial time algorithm for Circuit-SAT, we have a polynomial time algorithm for all NP problems!

• **Bad news:** If any problem in NP is hard, Circuit-SAT is hard.

Note: Decision problems can be NP-Complete. For optimization problems, it is called NP-Hard.
Other NP-Complete/Hard Problems

The following are all NP-Complete/Hard:

• Formula SAT
• Maximum Independent Set
• TSP
• Hamiltonian Cycle
• Knapsack

How do we show this? By finding reductions from other NP-Hard/Complete Problems.
3-SAT

3-SAT is a special case of formula-SAT where the formula is an AND of clauses and each clause is an OR of at most 3 variables or their negations.

Example:

\[(x \lor y \lor z) \land (\bar{x} \lor u) \land (w \lor \bar{z} \lor u) \land (\bar{u} \lor w \lor \bar{z}) \land (\bar{y})\]
Circuit-SAT $\rightarrow$ 3-SAT

- Start with circuit

- Create variable for each wire
- Create formula with clause for each gate and output

$$(v \iff y \lor z) \land (u \iff x \land y) \land (w \iff u \land v) \land (t \iff \bar{w}) \land t)$$
These Aren’t 3-SAT Clauses

We have 3-variable clauses that aren’t 3-SAT clauses. Write it in terms of them.

• Write truth table
• Each 3-SAT clause sets one output to false.

\[(x \lor y \lor \bar{z}) \land (x \lor \bar{y} \lor z) \land (\bar{x} \lor y \lor z) \land (\bar{x} \lor \bar{y} \lor z)\]

\[= (z \iff x \lor y)\]
Another Look at 3-SAT

**Lemma:** A 3-SAT instance is satisfiable if and only if it is possible to select one term from each clause without selecting both a variable and its negation.
Proof

If satisfiable:

• Satisfying assignment causes at least one term in each clause to be true.
• Select one such term from each clause.
• Cannot contradict each other.
Proof

If there is a way to select terms:

• Set those variables to be true
  – Can do this without contradiction

• Set other variables arbitrarily

• Causes whole statement to be true
3-SAT \rightarrow Maximum Independent Set

Want to encode this select one term from each clause as a graph.

• Create one vertex for each term in each clause.
• Edges between terms in same clause.
• Edges between contradictory terms.

Example:

$$(x \lor y \lor z) \land (\bar{x} \lor y) \land (\bar{y} \lor x)$$
Analysis

An independent set in this graph has:

• At most one vertex from each clause.

• No vertices representing contradictory terms.

It has an independent set of size \#Clauses if and only if, you can select one term form each clause without a contradiction.

Therefore, \(|MIS| = \#Clauses\) if and only if the 3-SAT has a solution.
Zero-One Equations

**Problem:** Given a matrix $A$ with only 0 and 1 as entries and $b$ a vector of 1s, determine whether or not there is an $x$ with 0 and 1 entries so that

$$Ax = b.$$
3-SAT → ZOE

Basic Idea:

• Use the one term from each clause formulation of 3-SAT.

• Create one variable for each term to denote whether or not it has been selected.

• Add equations to enforce exactly one term from each clause, no contradictory terms selected.
General Construction

• Create one variable per term
• For each clause, create one equation
• For each pair of contradictory term, create an equation with those two and a new variable
Another Way of Looking at ZOE

Recall if $A = [v_1 \ v_2 \ v_3 \ ... \ v_n ]$,

$Ax = x_1 \ v_1 + x_2 \ v_2 + x_3 \ v_3 + \ ... + x_n \ v_n$.

Example:

$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$

$x_1 \times [1 \ 0 \ 0 \ 1] + \ x_2 \times [0 \ 0 \ 1 \ 1] + \ x_3 \times [1 \ 1 \ 1 \ 0]$

$= [1 \ 1 \ 1 \ 1 \ 1]$
Subset Sum

**Problem:** Given a set $S$ of numbers and a target number $C$, is there a subset $T \subseteq S$ whose elements sum to $C$.

**Alternatively:** Can we find $x_y \in \{0,1\}$ so that
\[
\sum_{y \in S} x_y y = C.
\]

Reduction: ZOE $\rightarrow$ Subset Sum.
Subset Sum → Knapsack

- Create Knapsack problem where for each item \( \text{Value(item)} = \text{Weight(item)} \).
- Maximizing value is the same as maximizing weight (without going over capacity).
- We can achieve value = capacity if and only if there is a subset of the items with total weight equal to capacity.
ZOE -> Hamiltonian Cycle

- Start with a cycle
- Double up some edges
- Cycle must pick one edge from each pair.
  - This provides a nice set of binary variables
- Need a way to add restrictions so that we can’t just use any choices.
Gadget

- Must use these edges.
- Two ways to fill out.
Gadget Use

• Hook gadget up between a pair of edges.
• Hamiltonian Cycle must use exactly one of the connected edges.
• This allows us to force logic upon our choices.
Construction

By doing this for several pairs of edges we can construct Hamiltonian Cycle problems equivalent to the following:

• You are given a number of choices where you need to pick one from several options (of multi-edges).

• You have several constraints, that say of two choices you must have picked exactly one of them.
Full Construction

Choices:
• For each variable, choose either 0 or 1.
• For each equation, choose one variable.

Constraints:
• For each variable that appears in an equation, exactly one of the following should be selected:
  – That variable in that equation
  – That variable equal to 0
Reduction Summary

Any NP Decision Problem → Circuit SAT

Circuit SAT → Maximum Independent Set

Maximum Independent Set → 3-SAT

3-SAT → Zero-One Equations

Zero-One Equations → Subset Sum

Subset Sum → Knapsack

Hamiltonian Cycle → Travelling Salesman Problem
Dealing With NP-Completeness (Ch 9)

• Backtracking/Branch and Bound
• Heuristic Search
• Approximation Algorithms
Deductions

One way to progress is so make deductions.

• Use the rules to show that some square can only be filled out in one way.

• Use that information to help fill out more squares.

• Hopefully, you can keep going until the entire problem is solved.
Getting Stuck

Deductions are very useful when you can make them, but for hard problems, you will often get stuck quickly and be unable to make more deductions.

Option 1: Stronger deductive rules.
Option 2: Guess and Check
Guess and Check

• Make a guess for some entry.
• Try to solve the resulting puzzle (perhaps doing more guessing).
• If you find a solution, great!
• If not, you have deduced that your original guess was wrong.
Backtracking

Backtracking(P,S)

If you can deduce unsolveable
   Return ‘no solutions’
Split S into parts S₁,S₂,...
For each i,
   Run Backtracking(P,Sᵢ)
Return any solutions found
Splitting

How do you split S into parts?

• Pick variable \( x_i \) and set \( x_i = \text{True} \), or \( x_i = \text{False} \)
• Try all possible numbers in a square in Sudoku
• Try all possible edges in Hamiltonian Cycle

Which variable do we guess?

• Often helps to pick a variable that shows up a lot. Then guessing it’s value will make later deductions easier.
These problems are still NP-Hard. Worst case, backtracking will still take exponential time. But it is usually much better than brute force.

SAT Solvers can use these ideas to solve problems with hundreds of variables, many many more than would be practical by brute force.
Optimization Version

Backtracking works well for decision/search problems (where a potential solution works or doesn’t work), but not so well for optimization problems (where many solutions work, but you need to find the best one).

If most solutions work, how do you weed out bad paths?
Branch & Bound

To get rid of bad paths do two things:

1) Keep track of the best solution you have found so far.

2) Try to prove upper bounds on your subproblems.

If an upper bound is smaller than your best solution so far, it cannot contain the optimum.
Branch and Bound

BranchAndBound(Best, S)
    If UpperBound(S) ≤ Best
        Return ‘no improvement’
    If S a full solution
        Return value of S
    Split S into S₁, S₂, ...
    For each Sᵢ
        New ← BranchAndBound(Best, Sᵢ)
        Best = Max(New, Best)
    Return Best
Many optimization problems have a structure where solutions “nearby” a good solution will likely also be good.

This leads to a natural algorithmic idea:

• Find an OK solution
• Search nearby for better solutions
• Repeat
Local Search

LocalSearch(f)
\\ Try to maximize f(x)
   x ← Random initial point
   Try all y close to x
      If f(y) > f(x) for some y
         x ← y
      Repeat
 Else Return x
**Problem:** Given a graph G find a way to color the vertices of G black and white so that as many edges as possible have endpoints of different colors.

This is NP-Hard.
How to Get Unstuck

• Randomized Restart
  – If you try many starting points, hopefully, you will find one that finds you the true maximum.

• Expand Search Area
  – Look for changes to 2 or 3 vertices rather than 1.
    • Larger area means harder to get stuck
    • Larger area also takes more work per step

• Still no guarantee of finding the actual maximum in polynomial time.
Simulated Annealing

• At the start of algorithm take big random steps.
  – Hopefully, this will get you onto the right “hill”.
• As the algorithm progresses, the “temperature” decreases and the algorithm starts to fine tune more precisely.
• Works well in practice on a number of problems.
Approximation Algorithms

An $\alpha$-approximation algorithm to an optimization problem is a (generally polynomial time) algorithm that is guaranteed to produce a solution within an $\alpha$-factor of the best solution.

Our local search algorithm for MAXCUT is a 2-approximation algorithm.
Vertex Cover

**Problem (Vertex Cover):** Given a graph G find a set S of vertices so that every edge of G contains a vertex of S and so that $|S|$ is as small as possible.

Also, NP-Hard.
Greedy Algorithm

GreedyVertexCover(G)

S ← {}
While(S doesn’t cover G)
    (u,v) ← some uncovered edge
    Add u and v to S
Return S
Analysis

Algorithm finds $k$ edges and $2k$ vertices.

- Edges are vertex-disjoint.
- **Any** cover must have at least one vertex on each of these edges.
- Optimum cover has size at least $k$.
- We have a 2-approximation.
Knapsack

Even though general knapsack is NP-Hard, we have a polynomial time algorithm if all weights are small integers (or more generally small integer multiples of some common value).
Small Values Dynamic Program

Let \( \text{Lightest}_{\leq k}(V) \) be the weight of the lightest collection of the first \( k \) items with total value \( V \).

We have a recursion

\[
\text{Lightest}_{\leq k}(V) = \min\{\text{Lightest}_{\leq k-1}(V), Wt(k) + \text{Lightest}_{\leq k-1}(V - \text{Val}(k))\}
\]

This gives a DP.

\( \#\text{subprobs} = [\text{Total Value}][\#\text{items}] \)

Time/Subproblem = \( O(1) \).
Approximation Algorithm

1) Throw away items that don’t fit in sac.
2) Let $V_0$ be highest value of item.
3) Round each item’s value to closest multiple of $\delta V_0$.
4) Run the small integer values DP.

**Runtime:** Values integer multiples of $\delta V_0$. Total value at most $[\#\text{items}] V_0 = ([\#\text{items}]/\delta) \delta V_0$. Total runtime $O([\#\text{items}]^2/\delta)$. 
Optimal value at least $V_0$.

Rounding changes the value of any set of items by at most $\delta V_0$.

The solution we find is at least as good as the optimal after round.

Our solution is within $\delta V_0$ of optimal.
Combining

Let $\delta = \frac{\varepsilon}{\#\text{items}}$.

$\text{OPT} \geq V_0$.

Our solution is at most $\varepsilon V_0$ worse.

Have a $(1+\varepsilon)$-approximation algorithm.

$\text{Runtime} = O((\#\text{items})^3/\varepsilon)$

For any $\varepsilon > 0$, have a $(1+\varepsilon)$-approximation in polynomial time. (known as a PTAS).