Final Exam Review

CSE 101 Winter 2022
Exam Details

- 6 Questions in 180 minutes, in class
- Randomized Assigned Seating
- No books or electronic aids. 10 one-sided pages of notes.

- Exam is comprehensive. This review only covers the new material. See previous exam reviews for the rest.
Independent Set

**Definition:** In an undirected graph $G$, an **independent set** is a subset of the vertices of $G$, no two of which are connected by an edge.

**Problem:** Given a graph $G$ compute the largest possible **size** of an independent set of $G$.

Call answer $I(G)$. 
Simple Recursion

Is vertex v in the independent set?

**If not:** Maximum independent set is an independent set of G-v. 
\[ I(G) = I(G-v). \]

**If so:** Maximum independent set is v plus an independent set of G-N(v). 
\[ I(G) = 1+I(G-N(v)). \]

**Recursion:** 
\[ I(G) = \max(I(G-v), 1+I(G-N(v))) \]
**Lemma:** If $G$ has connected components $C_1, C_2, \ldots, C_k$ then

$$I(G) = I(C_1) + I(C_2) + \ldots + I(C_k).$$
Independent Sets of Trees

Subproblems are all subtrees!
Recursion

Root not used:
\[ I(G) = \Sigma I(\text{children’s subtrees}) \]

Root is used:
\[ I(G) = 1 + \Sigma I(\text{grandchildren’s subtrees}) \]
Travelling Salesman Problem

**Problem:** Given a weighted (undirected) graph $G$ with $n$ vertices find a cycle that visits each vertex exactly once whose total weight is as small as possible.
Naïve Algorithm

• Try all possible paths and see which is cheapest.
• Runtime \( \approx n! \)
Setup

Need partial solutions for subproblems.
• Look for $s$-$t$ paths instead of cycles.

$\text{Best}_{st}(G) =$ Best value of a path starting at $s$ and ending at $t$ that visits each vertex exactly once.
• Answer is minimum of $\text{Best}_{st}(G) + \ell(s,t)$. 
Recursion

• What happens if we undo the last step?
  – Last step some edge \((u, t)\).
  – Value is \(\ell(u, t)\)+length of rest of path.
  – Want best \(s\)-\(u\) that uses every vertex except for \(t\).
• \(\text{Best}_{st}(G) = \min_u [\text{Best}_{su,-t}(G) + \ell(u,t)]\).
• Now we need a recursion for \(\text{Best}_{su,-t}(G)\).
Recursion II

- How do we solve for $\text{Best}_{su,-t}(G)$?
- Remove last edge $(v,u)$.
- Need best $s$-$v$ path that uses all vertices except for $t$ and $u$.
- Need more complicated subproblems to solve for that.
Recursion III

$\text{Best}_{st,L}(G) = \text{Best s-t path that uses exactly the vertices in L.}$

- Last edge is some $(v,t) \in E$ for some $v \in L$.
- Cost is $\text{Best}_{sv,L-t}(G) + \ell(v,t)$.

**Full Recursion:**

$\text{Best}_{st,L}(G) = \min_v [\text{Best}_{sv,L-t}(G) + \ell(v,t)]$. 
Runtime Analysis

**Number of Subproblems:**
L can be any subset of vertices (2\(^n\) possibilities)
s and t can be any vertices (n\(^2\) possibilities)
n\(^2\)2\(^n\) total.

**Time per Subproblem:**
Need to check every v (O(n) time).

**Final Runtime:**
O(n\(^3\)2\(^n\))
[can improve to O(n\(^2\)2\(^n\)) with a bit of thought]
NP-Completeness (Ch 8)

- NP-Problems
- Reductions
- NP-Completeness & NP-Hardness
- SAT
- Hamiltonian Cycle
- Zero-One Equations
- Knapsack
Such problems 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.
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?

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

\[x = \text{True}, \ y = \text{True}, \ z = \text{False}\]

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

No satisfying assignment.
Hamiltonian Cycle
(in text as Rudruta Path)
Given an undirected graph G is there a cycle that visits every vertex exactly once?
General Knapsack

Recall knapsack has a number of items each with a weight and a value. The goal is to find the set of items whose total value is as much as possible without the total weight going exceeding some capacity.

Have algorithm that runs in polynomial time in the weights.

If weights are allowed to be large (written in binary), don’t have a good algorithm.
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 problem A instance**
  - **Solution to A**
  - Polynomial time reduction algorithm

- **Instance of problem B**
  - **Solution to problem B instance**
  - **Solution to B instance**
  - Hypothetical algorithm for B

- Polynomial time interpretation algorithm
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.

**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.

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

$\begin{align*}
x & \rightarrow \neg x \\
y & \rightarrow \neg y \\
z & \rightarrow \neg z \\
u & \rightarrow \neg u \\
v & \rightarrow \neg v \\
w & \rightarrow \neg w \\
\end{align*}$

• Create variable for each wire

• Create formula with clause for each gate and output

$$(v \iff y \lor z) \land (u \iff x \land y) \land (w \iff u \land v) \land (t \iff \neg w) \land (t)$$
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.

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

\[= (z \iff x \lor y)\]
Note

This means that 3-SAT is also NP-Complete since we have:

Any problem in NP $\rightarrow$ Circuit SAT $\rightarrow$ 3-SAT
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.

**Example:**

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

Example:
\[(x \lor y \lor z) \land (\overline{x} \lor y) \land (\overline{y} \lor \overline{z}) \land (\overline{x} \lor \overline{z})\]
\[x = \text{False}, \ y = \text{True}, \ z = \text{False}\]
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

Example: \( (x \lor y \lor z) \land (\bar{x} \lor \bar{y}) \)

\( x = \text{True}, \ y = \text{True}, \ z = \text{False} \)
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:

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

$\left( x \lor y \lor z \right) \land \left( \overline{x} \lor y \right) \land \left( \overline{y} \lor \overline{x} \right)$

<table>
<thead>
<tr>
<th>x_1</th>
<th>x_2</th>
<th>x_3</th>
<th>x_4</th>
<th>x_5</th>
<th>x_6</th>
<th>x_7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One term per clause:

- $x_1 + x_2 + x_3 = 1$
- $x_4 + x_5 = 1$
- $x_6 + x_7 = 1$

No Contradictions:

- $x_1 + x_4 + x_8 = 1$
- $x_1 + x_7 + x_9 = 1$
- $x_2 + x_6 + x_{10} = 1$
- $x_5 + x_6 + x_{11} = 1$
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 $A = [v_1 \ v_2 \ v_3 \ldots \ v_n]$, 

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

**Example:**

$$A = \begin{bmatrix}
1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0
\end{bmatrix} \quad \begin{array}{c}
\times_1 \times [1 \ 0 \ 0 \ 1] + \\
\times_2 \times [0 \ 0 \ 1 \ 1] + \\
\times_3 \times [1 \ 1 \ 1 \ 0]
\end{array}$$

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

What if we treated these as numbers rather than vectors?
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

$$\sum_{y \in S} x_y = C.$$

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

• Create Knapsack problem where for each item Value(item) = 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.
Ham. Cycle Strategy

- 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.
• Must use these edges.
• Two ways to fill out.
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
Example

\[ x_1 + x_2 + x_3 = 1 \]
\[ x_2 + x_4 = 1 \]

\[ x_1 = 1 \]
\[ x_2 = 0 \]
\[ x_3 = 0 \]
\[ x_4 = 1 \]
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

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.
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 $S_1, S_2, \ldots$
For each i,
Run Backtracking(P, S_i)
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
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 S_1, S_2, ...
   For each S_i
      New ← BranchAndBound(Best, S_i)
      Best = Max(New, Best)
   Return Best
Local Search

Many optimization problems have a structure where solutions “nearby” a good solution will likely also be good.
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
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.

Often approximation algorithms can produce good enough solutions.
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.
Greedy Algorithm

GreedyVertexCover(G)

\[ S \leftarrow \{\} \]

While (S doesn’t cover G)

\[ (u,v) \leftarrow \text{some uncovered edge} \]

Add u and v to S

Return S

2 Approximation!