Games, Puzzles, & Computation Class Notes

Written by: Angel, Austin, Kevin, and Lillian

Week 3: Jan 30th, 2017 - Feb 3rd, 2017

Review of the previous week's lab

"dots and boxes"

In the last lab session, one of the games that was demonstrated was "dots" (a game where players try to make boxes by connecting dots). One of the strategies of the game is called double-dealing, which is when a player gives up some of their boxes so that one player can force the other one to make certain moves, leaving them with more boxes in the next turn.

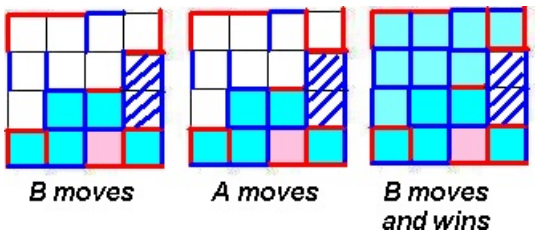

Figure 1: A simple example of B making a move to set themselves up for a win.

"hex"

This game has a distinct 1st player advantage. A good tip, when playing this game, is to pick blocks that will leave 2 other choices in the next round.
There can never be a tie in hex.

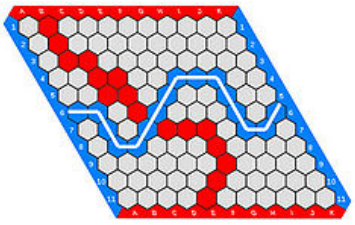

Figure 2: Red won because blue failed to block them.

There were 3 other games to play at the lab last week:
Niya, Gobblet, and Quantum tic-tac-toe.

Definitions

Computation - Can be thought of as the process of producing an output from a set of inputs and a finite number of steps.

Turing Machine - In its simplest form, a Turing machine can be thought of as a computer with infinite memory.

Decision Problems - Problems whose computed output is either *yes* or *no*. All of the problems referred to in complexity classes are decision problems.

Time Complexity of an Algorithm - The number of steps taken (to decide) based on the input.

Deterministic - A single path, sequential.

Nondeterministic - All paths at the same time.

Time - Let $t : \mathbb{N} \to \mathbb{R}$ be a function. Define the time complexity class $TIME(t(n))$ to be the set of all languages decidable by a $O(t(n))$ time Turing Machine. Similarly, $NTIME(t(n)) = \{L : L$ is a language decided by a $O(t(n))$ time nondeterministic Turing machine.$\}$

Decidable - A solution (yes/no) can always be found in a finite amount of time.

Verifier - A verifier for a language A is an algorithm V where $A = \{w : V$ accepts $< w, c >$ for some string $c\}$, where c is the certificate.

Polynomial Time - All reasonable deterministic models of computation are polynomially equivalent. This allows us to develop a theory and look at the complexity of problems that aren't specific to one model of computation.

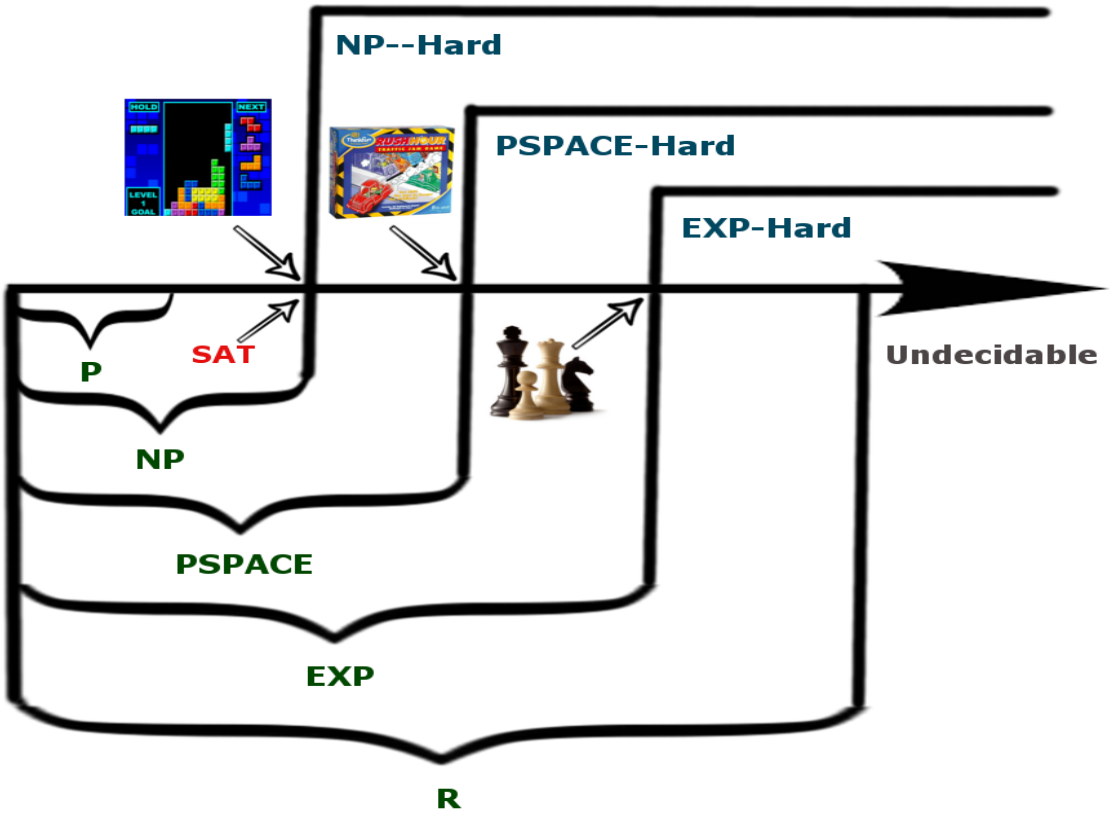

Figure 3: Complexity classes and Examples of some games in those categories

Classes of Complexity

P - *Polynomial-Time*

- P is the class of languages decidable in Polynomial time on a deterministic Turing machine.

$$P = U_k TIME(n^k)$$

  Ex: $O(n^2)$, $O(nlogn)$, $O(n^9)$, $O(n^{100})$

NP - *Nondeterministic Polynomial-Time*

- NP is the class of languages decidable in Polynomial time on a nondeterministic Turing machine.

$$NP = U_k NTIME(n^k)$$

- Equivalently, if the answer to a problem is *yes*, then there is a *proof* of this that can be verified by a deterministic polynomial-time algorithm.

  Ex: Legend of Zelda, Super Mario Bros, Candy Crush Saga, and Battleship are NP-Complete, which means they are in both NP and NP-Hard.

PSPACE *Polynomial-Space*

- This class of decision problems contains all problems are solvable by a Turing machine in polynomial space.

  Ex: Games that are PSPACE-complete are Hex, Reversi, Mahjong, Atomix, and Sokoban, when generalized to be played on a n x n board.

EXP - *Exponential-Time*

- This class of decision problems contains all problems in *exponential* time by a Turing machine.

  Ex: $O(2^n)$

R - *Recursive Languages*

- This class of decision problems includes all problems that are solvable by a Turing machine.
- This class is often identified with the class of "effectively computable" functions.

Other things to know...

co-NP - *compliment of NP*

- This class can be thought of as the "opposite" of NP. If the answer to a problem is *no*, then there is a *proof* of this that can be verified by a deterministic polynomial-time algorithm.

X-hard - (*NP*-hard, *EXP*-hard, etc...)

- A problem is X-hard for some complexity class X if it is at least as hard as the hardest problems in X. Proving a problem is X-hard is often done through reductions.

X-complete - (*NP*-complete, *EXP*-complete, etc...)

- A problem is X-complete for some complexity class X if it:
  1. Is in class X.
  2. Is X-hard.

Showing Membership in NP

There are two different equivalent ways to show that a problem is in the class NP; either through a polynomial time verification algorithm, or a polynomial time nondeterministic solution algorithm. Here are some examples:

Problem: - Subset-Sum
Input: A set of integers S and an integer k.
Output: Does there exist a set $T \subseteq S$ s.t. the sum of the elements of T is k?

Thm. The Subset-Sum problem is in the class NP.

- Verifier $\Rightarrow << S, k >, c >$ - c is a list of numbers.
  1. Check that $c \subseteq S$.
  2. Check that $\sum_{i=1}^{|c|} c_i = k$.
  3. If 1 and 2 are true, output *yes*, otherwise, *no*.
- Solve the problem nondeterministically:
  1. Nondeterministically select a set $T \subseteq S$.
  2. Check that $\sum_{i=1}^{|T|} T_i = k$.
  3. If 2 is true, output *yes*, otherwise, *no*.

A **clique** is a subset of vertices of an undirected graph s.t. its induced subgraph is complete.

Problem: - Clique
Input: An undirected graph $G =< V, E >$ and an integer k.
Output: Does G contain a k-clique?

Thm. The Clique problem is in the class NP.

- Verifier $\Rightarrow << G, k >, c >$ - c is a list of vertices.
  1. Is $c \subseteq V(G)$?
  2. Are all vertices in c are connected to each other?
  3. If 1 and 2 are true, output *yes*, otherwise, *no*.
- Solve the problem nondeterministically:
  1. Nondeterministically select k vertices in G.
  2. Do they form a clique?
  3. If 2 is true, output *yes*, otherwise, *no*.

Reductions

Reductions are algorithms that are used to transform one problem into another. Reduction from one problem to another problem is a way to show that the second problem is at least as difficult as the first. We say problem A is reducible to problem B if there exist an algorithm that solves problem B efficiently and can be used to solve problem A. Although there are many methods of algorithm reductions, we focus on a specific and very popular kind of reduction often used to prove NP-Completeness.

0.1 Satisfiability

Satisfiability (also known as Boolean Satisfiability Problem) is a decision problem of determining whether there exist a combination of literals (TRUE/FALSE) such that the expression is satisfied (OUTPUT = TRUE). If true, we call such an expression satisfiable (and similarly, unsatisfiable if the output is FALSE).

- Problem: Satisfiability
- Input: A set of Boolean Variables V and a set of clauses C over V
- Output: Does there exist a satisfying truth assignment for C, i.e, a way to set the literals true or false so that each clause contains at least one true literal?

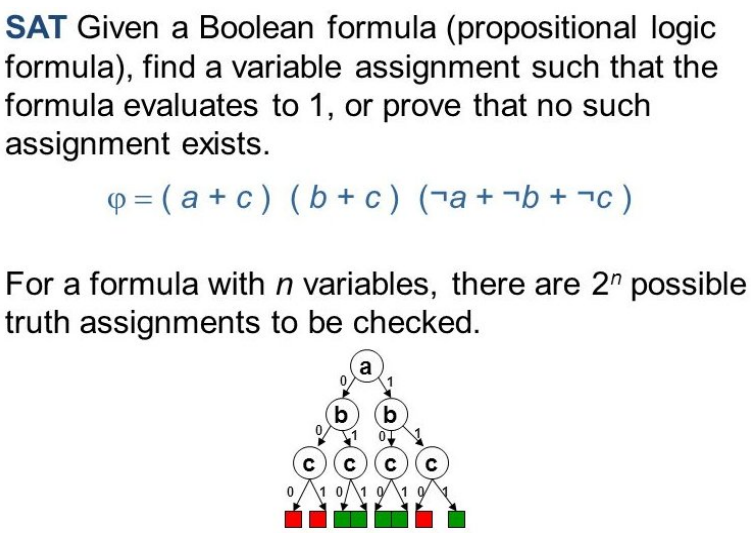

Figure 4: Adapted From 'The Quest for Efficient Boolean Satisfiability Solvers' - Sharad Malik

0.2 3-Satisfiability (3SAT)

It is easy to test Satisfiability on problems containing only two pairs of clauses, however, we are interested in problems containing of a larger class. How many literals per clause do you need to turn a problem from polynomial to hard?

- Problem: 3-Satisfiability
- Input: A collection of clauses C where each clause contains exactly 3 literals, over a set of Boolean Variables V.
- Output: Is there a truth assignment to V such that each clause is satisfied?

Clique Problem

A clique is an undirected graph G = (V, E) is a subset V' $\subseteq$ V of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The size of a clique is the number of vertices it contains. The clique problem is the optimation problem of finding a clique of maximum size in a graph. As a decision problem, we ask simply whether a clique of a given size K exists in the graph. the formal definition is

CLIQUE = { $\langle$ G, E $\rangle$: G is a graph containing a clique of size k }

A naive algorithm for determining whether a graph G = (V, E) with —V— vertices has a clique size of k is to list all k-subsets of V, and check each one to se whether it forms a

clique. An efficient algorithm for the clique problem is unlikely to exist.

Theorem The clique problem is NP-complete.

Proof To show that CLIQUE $\in$ NP, for a given graph G = (V, E), we use the set V' $\subseteq$ V of vertices in the clique as a certificate for G. We can check whether V' is a clique in polynomial time by checking whether, for each pair u, v $\in$ v', the edge (u,v) belongs to E. We next prove that CLIQUE is reducable to 3-CNF SAT, which shows that the clique problem is NP-hard. You might be surprised that we should be able to prove such a result, since on the surface logical formulas seem to have little to do with graphs. The reduction algorithm begins with an instance of 3-CNF-SAT.

Let $\Phi = C_1 \wedge C_2 \wedge C_3 \cdots C_k$ be a boolean formula in 3-CNF with k clauses. for r = 1, 2, $\cdots$ k, each clause C_r has exactly three distinct literals l_1^r, l_2^r, and l_3^r. We shall construct a graph G such that Φ is satisfiable if and only if G has a clique of size k. We construct the graph G = (V, E) as follows. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in Φ, we place a triple of vertices v_1^r, v_2^r, v_3^r into V. We put an edge between two vertices v_i^r and v_j^s if bot hof the following hold:

- v_i^r and v_j^s are in different triples, that is, r $\neq$ s, and
- their corresponding literals are consistent, that is, l_i^r is not the negation of l_j^s

We can easilty build this graph from Φ in polynomial time. As an example of this construction, if we have
$\Phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$,
then G is the graph shown in figure 6.

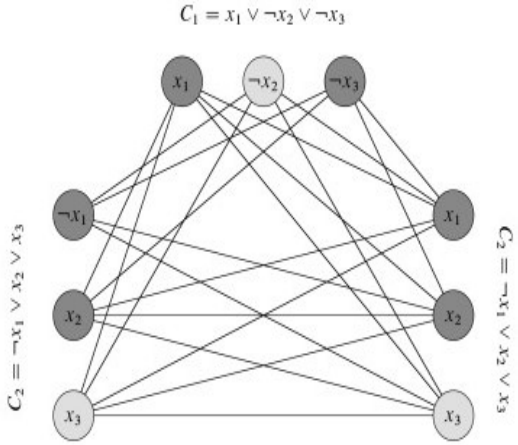

Figure 5: CNF to Clique

We must show that this transformation of Φ into G is a reduction. First, suppose that Φ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r. Picking one such "true" literal fro meach clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices v_i^r, $v_j^s \in$ V', where r $\neq$ s, both corresponsding literals l_i^r and l_j^s map to 1 by the given satisfying assignment, and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Good Reference Material

- Computational Complexity: A Modern Approach
  Princeton University
  pdf

- MIT OpenCourseWare - Video and Class Notes
  Lecture 23: Computational Complexity
  link

- University of Illinois - Class Notes
  Lecture 21: NP-Hard Problems
  pdf