ENEE 459E/CMSC 498R

In-class exercise

In this in-class exercise, we will explore what it means for a problem to be intractable (i.e. it cannot be solved by an efficient algorithm). There are two parts:

1. In the first part, we will look at problems that have *no* algorithmic solution. This means that any computer program that we write for solving this problem will have some inputs on which it will *never* terminate. Such problems are called *undecidable*. As we will see, the fact that such problems even exist was a big surprise for mathematicians.

2. In the second part, we will look at problems which are *decidable*, that is, there is a computer program that will solve the problem, given enough time. However, we will see that for some fundamental problems, there is strong evidence that these problems require extremely large amounts of time to solve. Indeed, the problems that we will look at in this exercise are known as *NP-complete*.

Much of the text in the following is taken from "Introduction to the Theory of Computation" by Sipser. Thomson Brooks/Cole, 1997.

1. In 1900, mathematician David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris. In his lecture, he identified 23 mathematical problems and posed them as a challenge for the coming century. Hilbert's tenth problem was to devise an algorithm that tests whether a polynomial has an integral root. He did not use the term *algorithm* but rather "a process according to which it can be determined by a finite number of operations." Interestingly, in the way he phrased this problem, Hilbert explicitly asked that an algorithm be "devised." Thus he apparently assumed that such an algorithm must exist–someone need only find it.

   As we now know, no algorithm exists for this task; it is algorithmically unsolvable. For mathematicians of that period to come to this conclusion with their intuitive concept of algorithm would have been virtually impossible. The intuitive concept may have been adequate for giving algorithms for certain tasks, but it was useless for showing that no algorithm exists for a particular task. Proving that an algorithm does not exist requires having a clear definition of algorithm. Progress on the tenth problem had to wait for that definition.

   The definition came in the 1936 papers of Alonzo Church and Alan Turing. Church used a notational system called the λ-calculus to define algorithms. Turing did it with his "machines." These two definitions were shown to be equivalent. This connection between the informal notion of algorithm and the precise definition has come to be called the *Church-Turing thesis*.

   The Church-Turing thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem. In 1970, Yuri Matijasevic, building on work of Martin Davis, Hilary Putnam, and Julia Robinson, showed that no algorithm exists for testing whether a polynomial has integral roots.

In the following, we will consider the famous undecidable problem known as the Halting problem. Informally, the Halting problem asks, given a program M and input w, to determine whether the program M "halts" on input x or whether it enters an infinite loop. Formally, to solve the Halting problem, we must design an algorithm O, which takes as input a sequence of characters $\langle M, w\rangle$, where M is interpreted as a program and w is interpreted as the input to the program, and does the following:

- $O(\langle M, w\rangle)$ outputs 1 in the case that $M(w)$ outputs 1.
- Otherwise, $O(\langle M, w\rangle)$ outputs 0.

We will prove that there is no algorithm for the Halting problem via a proof by contradiction. In other words, we assume that an algorithm O exists for the Halting problem and obtain a contradiction. This implies that an algorithm for O cannot exist. To prove this, we will use the *diagonalization* method, which is also used to prove that the set of real numbers is uncountable.

We assume that algorithm O solves the Halting problem. This means that on input $\langle M, w\rangle$, $O(\langle M, w\rangle)$ outputs 1 in the case that $M(w)$ outputs 1 and $O(\langle M, w\rangle)$ outputs 0 otherwise.

Consider a new algorithm D which uses O as a subroutine. This new algorithm D gets input $\langle M\rangle$ (interpreted as a program). D runs $O(\langle M, \langle M\rangle\rangle)$ and outputs the opposite of what O

outputs. I.e. it runs O to determine the output of the program M on input $\langle M \rangle$ (so M is being run on its own description) and outputs the opposite.

We will examine tables of behaviors for O and D. We list all programs down the rows, $M_1, M_2, \ldots$ and all their descriptions across the columns, $\langle M_1 \rangle, \langle M_2 \rangle, \ldots$. The entries tell whether the program in a given row outputs 1 in a given column. The entry is 1 if the program outputs 1 but is *blank* if it outputs some other value or enters an infinite loop. We made up the entries in the following table to illustrate the idea.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	1		1		
M_2	1	1	1	1	
M_3					
M_4	1	1			
$\cdots$					
$\cdots$					

Figure 1: Behaviors of machines $M_1, M_2, \ldots$.

(a) For each row M_i and column $\langle M_j \rangle$ of Figure 2, consider the output of algorithm $O(\langle M_i, \langle M_j \rangle \rangle)$ on that pair. Use Figure 1 to fill in the following table where each entry now contains the output of O (either 0 or 1) on the corresponding pair.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1					
M_2					
M_3					
M_4					
$\cdots$					
$\cdots$					

Figure 2: Behavior of machine O.

(b) In the following table, we add a row and column corresponding to the algorithm D defined above. By our assumption, O is an algorithm and so is D. Therefore it must occur on the list $M_1, M_2, \ldots$ of all algorithms. Use Figure 2 to fill in the following table where each entry again contains the output of O on the corresponding pair. What happens when you get to the entry corresponding to row D and column $\langle D \rangle$?

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1							
M_2							
M_3							
M_4							
$\cdots$							
D							
$\cdots$							

Figure 3: Behavior of machine O with input $\langle D, * \rangle$ included.

2. A literal is a Boolean variable or a negated Boolean variable, as in x or $\overline{x}$. A clause is several literals connected with $\vee$s, as in $(x_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee x_4)$. A Boolean formula is in conjunctive normal form, called a cnf-formula, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee x_4) \wedge (x_3 \vee \overline{x}_5 \vee x_6) \wedge (x_3 \wedge \overline{x}_6)$$

It is a 3cnf formula if all the clauses have *three* literals.

A Boolean formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.

The 3SAT problem asks, given a 3cnf formula, to determine whether the formula is satisfiable.

It turns out that determining whether a 3cnf formula is satisfiable is believed to be a hard problem. Specifically, for 3cnf formula on n variables, the best algorithms (that correctly solve the problem on all inputs) run in time $2^{c \cdot n}$ for constant $c < 1$. Since the problem can always be solved using exhaustive search in time 2^n, the best algorithms we have are not much better than exhaustive search. Additional evidence for the fact that 3SAT is a hard problem is that 3SAT is a type of problem that is called *NP-complete*. We won't get into the specifics of the definition here, but intuitively what this means is that if you had an efficient algorithm for solving $3SAT$, then there would be very many other hard problems that you could immediately solve efficiently.

(a) Determine if the following 3cnf formula is satisfiable:

$$\Phi_1 = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee \overline{x}_3 \vee x_4) \wedge (x_2 \vee x_3 \vee \overline{x}_4). \quad (1)$$

Note that in order for the formula to be satisfiable it is necessary and sufficient to have an assignment to the literals $x_1, \overline{x}_1, x_2, \overline{x}_2, x_3, \overline{x}_3, x_4, \overline{x}_4$ such that

- Each clause contains at least one literal set to 1.
- Two literals of the form x_i, $\overline{x}_i$ are set to opposite values.

(b) Specify an algorithm that solves 3SAT in time 2^n using exhaustive search (where n is the number of variables).

(c) In the following, we will see another hard problem known as the *graph clique* problem. We will show that if there is a polynomial-time algorithm that solves the graph clique problem, then can use it to construct a polynomial-time algorithm solving 3SAT. This means that if we believe there is no polynomial-time algorithm for 3SAT then there must also be no polynomial-time algorithm for graph clique.

The graph clique problem. A clique in an undirected graph is a subgraph, wherein every two nodes are connected by an edge (i.e. a subgraph which is a *complete* graph). A k-clique is a clique that contains k nodes. Figure 4 illustrates a graph having a 5-clique.

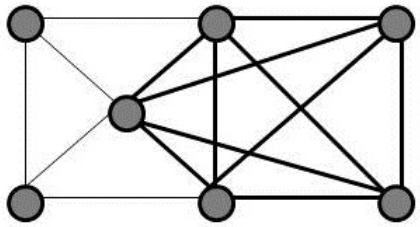

Figure 4: A graph with a 5-clique.

The graph clique problem is to determine whether a graph contains a clique of a specified size.

Constructing an Algorithm A for solving 3SAT. Assume that B is an algorithm solving the graph clique problem. We will use it as a subroutine to construct an algorithm A for solving 3SAT.
On input a 3CNF formula Φ with k clauses such as

$$\Phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k),$$

A will construct a graph G that is related to this 3CNF formula in the following ways:

- The nodes in graph G are organized into k groups of three nodes called *triples*. Each triple corresponds to one of the clauses in Φ, and each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in Φ.

- The edges (i.e. the connections between the nodes) in G must also be chosen by A. It will be your task to specify how A chooses these edges.
- The graph G will have a k-clique if and only if the original 3SAT formula (which has k clauses) was satisfiable.

A will then run B on input $\langle G, k \rangle$ to determine whether G contains a k-clique. If B determines that G contains a k-clique, then A concludes that Φ is satisfiable. If B determines that G does not contain a k-clique, then A concludes that Φ is not satisfiable.

Specify how the algorithm A chooses the edges of G:

- **Hint 1:** Whether or not there is an edge between two nodes in the constructed graph G will depend on whether the literals are (1) in the same *triple* (2) are labeled with literals of the form x_i, $\overline{x}_i$.
- **Hint 2:** Assume that Φ is satisfiable. Then there is some assignment of variables which sets at least one literal in each clause to 1. For each clause, pick one of the literals set to 1, to yield a multi-set which contains k literals. The graph you construct will have a k-clique that consists of the nodes corresponding to these literals.