Recitation 7

Logic and Complexity

Review

Operations on propositions:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \oplus Q$	$P \implies Q$	$P \iff Q$
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	T	F
F	F	T	F	F	F	T	T

where the notation is interpreted as follows

- $\neg$ means "not"
- $\wedge$ means "and"
- $\vee$ means "or"
- $\oplus$ means "or, but not both of", which we call "xor"
- $\implies$ means "implies"
- $\iff$ means "if and only if"

Warm-Up

a. Answer true or false for all of the following. ($\equiv$ means that each propositions always have the same truth value).

i. $p \vee q \equiv \neg p \wedge \neg q$

ii. $(p \wedge q) \vee r \equiv p \wedge (q \vee r)$

iii. $p \Rightarrow \neg q \equiv p \vee q$

iv. $p \Rightarrow q \equiv p \wedge \neg q$

v. $p \Leftrightarrow q \equiv (p \wedge q) \vee (\neg p \wedge \neg q)$

vi. $p \wedge \neg q \equiv \neg(\neg p \vee q)$

vii. $(p \vee q) \wedge r \equiv (p \wedge r) \vee (q \wedge r)$

b. Give an assigment to the variables x_1, x_2, x_3 which makes the following logical expression evaluate to true.

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge x_3$$

c. Is the following logical expression is always true? Explain your answer.

$$(x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_3 \wedge \neg x_3)$$

d. Come up with a logical expression with three variables which has **only one** assignment to the variables which makes it true.

e. Given inputs p and q, create a circuit which outputs $p \oplus q$ using only **not gates**, **or gates**, and **and gates**.

f. Given inputs x_1, x_2 and x_3, create a circuit which outputs true if and only if exactly one input is true.

Section Lesson - P vs. NP

Everything past this point is outside the scope of CS22, though it will reinforce concepts from the class. These concepts are covered further in CS1010 and CS157.

An algorithm is simply a series of steps. It can be expressed as a computer program, instructions in a cookbook, directions on a map, as a plain sentence, etc. For example, consider the following `maximum algorithm` that finds the maximum element given `list` as input.

```
v = -infinity
for each element x in list:
    v = max(v, x)
output v
```

It is often helpful to decribe the **run time** of an algorithm. The **run time** can be thought of as the number of steps that the algorithm takes, and is often expressed as a function as the size of the input to the algorithm

For example, if n is the size of the input list to the `maximum algorithm` then the algorithm has a run time of $f(n) = n$ steps.

Give an informal inductive argument as to why this is true. (Hint: Start by considering the base case of a list with one item. The one item is clearly the maximum so we return it in 1 step.)

An algorithm is considered *fast* if its run time is a **polynomial**. This polynomial can be n^2, or even $n^{100} + 4n^{63}$, it doesn't matter. If the run time is polynomial then it is feasible for a computer. If the run time is instead exponential, like 2^n, then it is infeasible for a computer if n is large. The terms **polynomial time** and **fast** are used synonymously for the rest of this handout.

Consider the following algorithm which determines if there is any assignment to a logical expression with variables $x_1, ..., x_n$ that is ever true. (The problem of determining if a logical expression evaluates to true is called `SAT`).

```
Try every assignment to the variables.

If the logical expression ever evaluates to true, return true.
```

j. What is the run time of this algorithm in terms of n, the number of variables.

   Hint: How many assignments do you have to try? Think in terms of 0/1 strings.

P is the set of problems that can be solved in **polynomial time**. In other words, a problem in P can be solved by an algorithm that has a polynomial run time.

Finding the maximum element in a list is a P problem, as is sorting a list.

Unfortunately, the algorithm given above is the best known algorithm for determining if a logical expression ever evaluates to true (solving SAT). Therefore it is not a P problem (as far as we know).

NP

A problem is in NP if, given a candidate solution to the problem, there exists a fast algorithm to determine whether the candidate is in fact a valid solution.

For example, determining if a logical expression ever evaluates to true (SAT) is an NP problem. We don't know how to quickly find an assignment to the variables which makes the expression true, but given an assignment to the variables we can quickly check that this assignment makes the expression true by just constructing the corresponding circuit.

k. *Challenge:* Another NP problem is the following: Given a number n where $n = pq$ for two large primes p and q, can you find p and q? Explain why this is an NP problem.

At first it looks like this is a P problem as well because you can simply check all pairs of numbers between 0 and $\sqrt{n}$. However, when you encode a number using 0's and 1's in a computer, it takes $\log_2(n)$ bits. Therefore looping through $\sqrt{n}$ numbers is exponential in the size of your input (we understand that this is confusing so please call over a TA if you wish to discuss this further).

Most computer scientists beleive that P $\neq$ NP but no one can come up with a proof. There is a million dollar reward for finding one. If you were to prove that P = NP then RSA would be useless.

Challenge: NP-Hard

It turns out that for any problem in NP, you can come up with a logic circuit for it where determining if the circuit ever evaluates true corresponds to solving the original problem.

Therefore, if you can find a "fast" algorithm for determining if a logical circuit ever outputs true then P = NP.

Changing one problem into another problem is called a "reduction".

A problem p is NP-Hard if there is a reduction from every problem in NP to p.

Another example of a problem that is NP-Hard is determining if a graph is 3-colorable, which you will learn about soon.

Challenge: Not in NP

There are problems that we think are even harder then NP problems. For example, given a logical expression, is it always true?