ECE365: Data Structures and Algorithms II (DSA 2)

Theoretical Computer Science

Give Credit Where Credit is Due

- Most of the content of this topic is not discussed in our textbook
- I have partially relied on the following resources:
  - "The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics" by Roger Penrose (possibly my favorite book, not really about CS!)
  - "Computational Complexity" by Papadimitriou
  - "Introduction to the Theory of Computation" (2nd or 3rd Ed.) by Michael Sipser
  - "Computational Complexity: A Modern Approach" by Arora and Barak
  - Various on-line resources, including pages from Wikipedia

Gödel's Theorem

- **Gödel's theorem** (1931) is really the first of two theorems often referred to, together, as *Gödel's incompleteness theorems*
- The theorem was in response to the efforts of David Hilbert, who was trying to prove that arithmetic was both **consistent** and **complete**
  - Informally, a *complete* system of arithmetic is one such that every true statement that can be expressed by the language can theoretically be proven within the language
  - Informally, a *consistent* system of arithmetic is one that does not contradict itself
- Gödel proved not only that arithmetic of his day wasn't consistent and complete, but further, that no system capable of expressing all basic facts of arithmetic can be both consistent and complete!
- For any consistent, formal system capable of expressing all basic arithmetical facts, there must be certain true statements expressible by the language that cannot be proven within the language
- This came as a blow to both mathematicians and philosophers
- Gödel's original proof included a very technical portion that, among other things, proved that symbols representing certain concepts (e.g., "there exists", "for all", etc.) must exist
- The second part of the proof is relatively simple, elegant, and clever, and we will discuss it

Things we can State in a Formal Language

- Imagine all the symbols that are part of a formal system ordered in some way
- It is then possible to express all possible finite strings of length one, ordered, followed by all possible finite strings of length two, ordered, etc.
- This is called a **lexicographical ordering**
- Some of these strings will have meaning and some will not
- Those strings with meanings in a formal mathematical language will include all true and false mathematical statements, definitions of functions, formal proofs, etc.
- **Propositions** are mathematical facts that can be stated in the language
- Some will be **propositional** functions (functions that evaluate to true or false) that depend on one or more variables (arguments to the functions)
- Let's denote the n^{th} propositional function on a single variable (according to the lexicographical ordering) as P_n
- Then $P_n(w)$ is that propositional function applied to w
- Let's denote the n^{th} proof in the language (according to the lexicographical ordering) as $\prod_n$

Gödel Sentences

- Now consider the following propositional function that depends on a single natural number w: $\sim\exists x\ [\prod_x \text{ proves } P_w(w)]$
- This is a propositional function on a single variable, w
- The more complex part of Gödel's proof shows that expressions such as these are expressible in the languages under consideration
- Note that this propositional function might be true for some values of w and false for others
- There must be some k for which $P_k(w)$ is equal to the above expression
- Therefore, we have: $P_k(w) = \sim\exists x\ [\prod_x \text{ proves } P_w(w)]$
- Applying this to k leads to an example of a *Gödel sentence*:

  $P_k(k) = \sim\exists x\ [\prod_x \text{ proves } P_k(k)]$

Implications of the Gödel Sentence

- Let’s think about the Gödel sentence: $P_k(k) = \sim\exists x\ [\prod_x \text{ proves } P_k(k)]$
- The meaning of this statement, more or less, is "There does not exist a proof of this statement", or alternatively, "This statement is not provable"
- If there is a proof of the statement within the formal system, the statement is false, and therefore the formal system must contain contradictions
- Otherwise, the statement is a true statement that is expressible in the formal system but cannot be proved within the system
- As mentioned earlier, Gödel's theorem came as a blow to mathematicians and philosophers
- It proves that no matter how many axioms are added, there will be certain true statements that cannot be proven (assuming the system is consistent)

Entscheidungsproblem

- Alan Turing invented the notion of the **Turing machine** in response to a related problem Hilbert was also working on; this problem is sometimes referred to as the *Entscheidungsproblem*
- Turing also had a remarkable career in cryptography, and he invented the Turing Test, which I discuss in my AI class
- The Entscheidungsproblem was a challenge to mathematicians to produce a mechanical procedure that could, in principle, solve all formally expressible yes/no mathematical questions
- The English term for the Entscheidungsproblem is "decision problem"
- We will further discuss the term *decision problem* later in the topic, which implies that a problem has a yes or no answer
- In 1936, Turing proved that a solution to the problem is impossible
- Alonzo Church independently proved the same result earlier the same year, involving his development of lambda calculus
- The two are now generally given joint credit for solving the problem

Turing Machines

- There are various ways that Turing machines can be defined; we will consider one valid definition
- A **Turing machine** consists of a *tape* that is infinitely long in both directions
  - The tape is comprised of an infinite sequence of squares, and each square is either marked or blank
  - We can imagine a read-write head that starts over some starting location
  - In Turing's original formulation, he allowed any finite collection of symbols to be used (i.e., an *alphabet*), plus a special symbol representing a blank cell
  - At most a finite number of squares on the tape will be marked with a symbol from the alphabet; the rest will be blank
- A Turing machine also consists of a finite (but potentially very large) set of possible *internal states*
  - One state is indicated as the *initial state*
  - Some states are *final states*, a.k.a. *accepting states*
- Finally, the Turing machine includes a *transition function* indicating a set of *rules*
  - Each rule has a very specific format: If we are in state x and read symbol s_1, then write symbol s_2, move one square (either left or right will be specified) on the tape, and go to state y
  - Example rule: If we are in state 17 and read ‘Q’, then write ‘Z’ and move one square left and jump to state 92
  - This is the only type of action a Turing machine is capable of!
  - Whenever a Turing machine follows a rule that leads to an accepting state, it stops

Turing Machine Variations

- When you look up the definition of a Turing machine in different sources, the definitions will vary, but if correctly specified, all definitions will be equivalent
- For example, it turns out that you don’t need a full alphabet or blank cells
- Limiting the symbols to 0 and 1, and constraining the tape to always contain a finite number of 1s, does not decrease the **power** of the machine
- The definition of "power" that we are using here has nothing to do with speed or efficiency; it only implies what is (or is not) possible to perform
- At any given time, the read-write head will always be above a 0 or a 1
- Depending on this value and the current internal state of the machine, it writes a 0 or 1, moves one space left or right, and changes to a new internal state
- Then, an example of a specific rule might be expressed like this: $_{100}0 \rightarrow {}_{10101}1R$
- Once again, this is the only type of action that such a Turing machine is capable of
- Other variations of Turing machines might only allow the tape to be infinitely long in one direction, or might add rejecting states in addition to accepting states

The Church-Turing Thesis

- The hypothesis known as **the Church-Turing thesis** states that any calculation that can be performed by any device is computable on a Turing machine
  - Sources differ as to what exactly counts as a calculation
  - For example, some sources say that the Church-Turing thesis applies specifically to functions of natural numbers
  - Others say it applies to calculations involving any number that can be generated one digit at a time by a Turing machine; this includes all rational numbers and some irrational numbers such as e and pi
  - In practice, calculations involving strings of characters are also generally allowed
- Turing showed that many basic calculations and various other standard procedures could be implemented with a Turing machine
- He reasoned that these were the necessary building blocks for what we now refer to as **algorithms**
- Today, this viewpoint that all algorithms can be performed by Turing machines is widely accepted by computer scientists
- All efforts to produce a mechanical formulation that is more powerful have failed
- In fact, most theoretical computer scientists accept this as a *formal definition of an algorithm*
- That is, *an algorithm is a computational process that can be reproduced by some Turing machine* (or an equivalent device)

Consequences of the Church-Turing Thesis

- A modern computer is not capable of any calculation that is not reproducible on a Turing machine
- In fact, modern computers are, technically speaking, less powerful than Turing machines because they have finite memory
- A modern computer is certainly able to perform many calculations more efficiently (in terms of big-O, not just real time) than a Turing machine
- However, *any polynomial-time algorithm that is implementable on a modern computer can also be implemented as a polynomial-time algorithm on a Turing machine*
- This does not necessarily mean that nothing is more powerful than a Turing machine
- Quantum computers, at the very least, should be able to solve more problems in polynomial time
- Technically speaking, any analog device is doing something that a Turing machine cannot exactly reproduce, but the Turing machine can simulate it with infinite precision
- Some people believe that the human mind is capable of things that a Turing machine is not (e.g., Roger Penrose has this view), and I discuss this possibility in my AI class, but most scientists seem to reject this
- However, no known discrete augmentation of Turing machines has been shown to increase the power of Turing machines
- For example, allowing multiple tapes does not increase the power of a Turing machine

The Universal Turing Machine

- Up to this point in our discussion, it seems that we would need a separate Turing machine, each with their own sets of internal states and rules, to implement any particular algorithm
- This leads to a question: Might there be a single Turing machine that can simulate all other Turing machines?
- Turing showed that the answer to this question is yes, and a Turing machine that can achieve this is referred to as a **universal Turing machine**
- Imagine a methodology to encode the specification of a Turing machine using 0s and 1s (or a more general alphabet)
- A universal Turing machine reads from its input tape a description, or encoding, of a Turing machine to emulate; the rest of the tape stores the input to this Turing machine
- The universal Turing machine simulates the specified Turing machine and applies it to the appropriate input
- Note that we can imagine ordering all possible Turing machines using a principle similar to the lexicographical ordering described during the discussion of Gödel's theorem
- If we were to iterate though all the possible Turing machines, according to the ordering scheme just mentioned, one of them would be the universal Turing machine

The Halting Problem

- Now we will turn our discussion back to the Entscheidungsproblem; Turing invented the notion of a Turing machine to explore this problem
- Turing proposed a problem now known as **the halting problem**, which can be states as follows: Devise an algorithm to decide whether the n^{th} Turing machine eventually halts on the m^{th} possible input
- Today, it might make more sense to consider this in terms of figuring out whether a given computer program (with infinite memory available) will fall into an infinite loop when applied to a particular input
- The halting problem is an example of what is known as a **decision problem**, which challenges us to find an algorithmic solution to answer a yes/no (or otherwise Boolean) question
- If such an algorithm exists, the problem is said to be *decidable*, *solvable*, or *computable*; otherwise, it is **undecidable**
- Turing proved that it is impossible to devise an algorithm which solves the halting problem for all cases; that is, *the halting problem is undecidable*
- Of course, there will be some program/input combinations that you can prove will halt, and some program/input combinations that you can prove will fall into an infinite loop
- For others, the best you can do is run the program on the input; if it is taking a really long time, you don't know if it is in an infinite loop, or if it just hasn't finished yet

Trouble

- One way to prove that the halting problem is undecidable is with an indirect proof
- First, assume it is possible to solve the halting problem
- Then you could create a function, halt(a, i), which accepts two strings representing the encoding of a Turing machine (or a computer program, or more generally, an algorithm) and its input
- This should return true if the algorithm, a, halts on the input, i, and it should return false otherwise
- Now consider the following procedure, which should be implementable if halt exists:

```
function Trouble(string s)
   if halt(s, s) == false
     return true
   else
     loop forever
```

- If it is possible to write such a function, then it must be implemented by some Turing machine; let t be the string encoding of that Turing machine
- Imagine, then, calling Trouble(t); this leads to a paradox!
- In order to avoid the paradox, it must not be possible to solve the halting problem

Diagonalization

- Turing's use of the Turing machine concept to prove that the halting problem is undecidable was also related to the Trouble algorithm, but the proof was different
- Turing's proof relied on a method known as **diagonalization**
- There are different variations of this proof; one goes like this:
  - Assume that we use binary to represent Turing machines and inputs
  - Consider a matrix, M, where the number in the i^{th} row and j^{th} column is 1 (representing true) if the Turing machine represented by i in binary halts on the input represented by j in binary, and 0 (representing false) otherwise
  - Assume that any invalid Turing machine halts right away on all inputs, so that every entry of M is well-defined
  - Then the value in row i of column j of M is just halt(i, j)
  - Now consider the Trouble algorithm; assume it can be implemented by some Turing machine with binary encoding t (which would be possible if the halting problem could be solved)
  - We know that halt(t, i) returns true if the Trouble algorithm halts for input i, and it returns false if the Trouble algorithm does not halt for input i
  - Looking back at the pseudo-code for the Trouble algorithm, we see that Trouble(i) should return true if halt(i, i) returns false, and Trouble(i) should loop forever if halt(i, i) returns true
  - Therefore, halt(t, i) must differ from every row in the matrix in at least one position (the main diagonal, where the row number equals the column number), so no row of the matrix can represent the Trouble algorithm
  - Recall that each row in M represents a Turing machine, and every Turing machines is represented by some row in the matrix
  - However, if implementing a solution to halt were possible, then implementing Trouble would be possible, and we just reasoned that no row in the matrix represents the Trouble algorithm; therefore, implementing halt is not possible

An Impossible Task

- I have seen diagonalization used to demonstrate that there are well-defined sequences of 0s and 1s that cannot be produced by any algorithm
- Imagine ordering all possible inputs to a universal Turing machine (binary version)
- Consider a matrix with rows representing each input that leads to an output with infinite length (i.e., it produces an infinite sequence of 0s and 1s)
- Now consider the main diagonal, but change all the 0s to 1s and all the 1s to 0s
- This is a well-defined sequence of 0s and 1s that is not produced by any input to the universal Turing machine, and therefore, not produced by any algorithm
- You might ask: Why can't you just design a Turing machine to produce this sequence?
- The reason is that this would require determining whether the universal Turing machine halts for each of its input (to determine whether this represents a row of the matrix)
- That turns out to be equivalent to the halting problem; it is not solvable

Nondeterministic Turing Machines

- All the Turing machines that we can produce are deterministic; given the machine and a specific input, there is a direct course of action that will follow
- We can hypothesize an extension of the Turing machine model that is (in a sense) nondeterministic; this will be called a **nondeterministic Turing machine**
  - As with regular Turing machines, there are multiple ways to define this, but all valid definitions are ultimately equivalent; we will consider one valid definition
  - For every internal state, symbol combination, instead of one *action* that is always taken (indicating a symbol to write, a direction to move, and a destination state), there will be a set of actions
  - The set may contain no actions, one action, or many actions
  - A nondeterministic Turing machine takes all applicable actions at once (i.e., in parallel), leading to a superposition of Turing machine configurations
  - Note that the nondeterministic Turing machine is not randomly choosing a single action; it is taking all applicable actions at the same instant
  - As soon as any path leads to an accepting state, the entire machine stops
- A nondeterministic Turing machine can be simulated by a regular Turing machine, but the slowdown (at least using an obvious approach) would be exponential
- Everything that is computable by a nondeterministic Turing machine is also computable by a regular Turing machine, but some things might be much faster on a nondeterministic Turing machine

Optimization Problems

- When talking about different problems that may or may not be solvable by a regular or a nondeterministic Turing machine, there are often two versions of the problem
- The simpler version of the problem is called a *decision problem* which, as we have already discussed, asks a yes/no question
- The harder version is called an *optimization problem*, which asks us to find the best way of doing something
- For example, *the Hamiltonian cycle problem* asks whether there is any Hamiltonian cycle in a graph; this is a decision problem
- The related optimization problem allows weighted edges and asks us to find the shortest Hamiltonian cycle in the graph; this is also known as *the traveling salesman problem*
- We can typically recast optimization problems as a sequence of decision problems
- For example, we can ask if a solution exists with an answer better than some chosen upper-bound or lower-bound, and we can vary the bound iteratively

Complexity Classes P and NP

- A **complexity class** defines a set of computational problems with similar big-Oh time or memory bounds for deterministic or nondeterministic Turing machines
- The class **P** defines all decision problems that can be solved by a regular Turing machine (i.e., an algorithm) in polynomial time
- The class **NP** defines all decision problems that can be solved by a nondeterministic Turing machine in polynomial time
- It is equivalent to define NP (roughly speaking) as the set of decision problems for which a yes solution to the problem can be checked by a regular Turing machine in polynomial time
- This is not obvious; we are not going to prove it formally in this class, but here is a (hopefully) intuitive explanation
  - If a regular Turing machine can check a yes solution in polynomial time, a non-deterministic polynomial Turing machine can try every possible solution in parallel and check them
  - If a non-deterministic Turing machine finds an accepting state in polynomial time, a regular Turing machine can check the path leading to an accepting state and verify that it is valid
- It is a common misconception for people to think that "NP" stands for "non-polynomial", but it does not; it stands for "non-deterministic polynomial time"

Does P = NP?

- Clearly, *the class P must be a subset of the class NP*, because every regular Turing machine is also a valid non-deterministic Turing machine
- This leads to the question: Are there any problems in NP that are not in P? Or, are the two complexity classes equivalent?
- This is **the P = NP question**, a.k.a. *the P versus NP problem*
- The answer, to this day, is not known!
- It is believed by a majority of computer scientists (as verified by a semi-recent poll) that P ≠ NP, but this has never been proven
- This is probably the most important open question in computer science
- If there is any problem that is in NP but not P, such a problem is solvable by a regular Turing machine in theory, but only with an exponential-time algorithm
- Such problems are said to be **intractable**
- Sometimes, theoretical computer scientists say a problem is "easy" if it has a polynomial time solution, and "hard" if it requires an exponential solution
- By this convention, intractable problems are hard

Reductions

- Remember the notion of **reduction**, which refers to the situation when an instance of one problem can be mapped to an instance of another problem
- Typically, we are interested in polynomial-time reductions (i.e., for which the mapping itself takes polynomial time)
- For example, in a previous topic, we saw that the maximum bipartite matching problem can be reduced to the network flow problem
- If P_1 can be reduced to P_2 in polynomial time, you can express this as $P_1 \leq_P P_2$
- Then you can use an algorithm that solves P_2 to solve P_1
- If P_1 and P_2 are decision problems, the answer to the mapped instance of P_2 is the same as the answer to the original instance of P_1
- When such a reduction is possible, we can formally say that P_1 is at least as easy P_2 and P_2 is at least as hard as P_1
- It should be fairly obvious that reduction is transitive

Complexity Classes NP-hard and NP-complete

- We are now ready to define two additional complexity classes
- The class **NP-hard** includes every decision problem for which every problem in NP can be reduced to it in polynomial time
- A problem in the class NP-hard is called an *NP-hard problem*
- The class **NP-complete** includes every decision problem that is in NP and for which every problem that is in NP can be reduced to it in polynomial time
- A problem in the class NP-complete is called an *NP-complete problem*
- An alternative definition of an NP-complete problem is a problem that it is NP-hard and is also in NP
- Note that *if we can solve any NP-complete problem in polynomial time, then we can solve all NP problems in polynomial time*!
- This would mean that P = NP
- Note that not all sources define NP-hard and NP-complete as complexity classes, but rather as properties that some problems might have

Cook’s Theorem

- In 1971, Stephen Cook proved *Cook's Theorem*, a.k.a. the Cook–Levin theorem, which states that *the Boolean satisfiability problem (SAT) is NP-Complete*
- The problem: Given a Boolean expression containing Boolean variables, ands, ors, nots, and parentheses, is there any assignment of truth values that makes the expression true?
- To prove that SAT is NP-complete, we need to prove that it is in NP, and that it is NP-hard
- Clearly, a potential solution to an instance of SAT can be verified by a regular Turing machine in polynomial time
- Similarly, a nondeterministic Turing machine could try out all possible solutions simultaneously, and solve the decision problem in polynomial time
- So, SAT is in NP; this is one of two conditions that must be true for an NP-complete problem
- The second condition is much more complicated to prove
- We are going to discuss a version of the proof that I found years ago on Wikipedia (although it has since been edited to some extent)

Cook’s Proof (high-level details first)

- Consider any given NP problem, X
- By the definition of NP, there is some nondeterministic Turing machine M = (Q, Σ, s, F, δ) that solves the problem X in polynomial time
- Here, Q is the set of states including an initial state s and accepting states F; Σ is the alphabet; and δ is the transition function
- This machine will accept some instances of input to the problem and reject others in polynomial time; that is, for a given input, it decides whether there is a solution
- Cook showed that it is possible to construct, in polynomial time, a Boolean expression, B, based on M and its input, I, such that B is satisfiable if and only if M accepts I
- This means that the expression, B, is satisfiable if and only if the answer to the given instance of the decision problem, X, is yes
- Thus, if SAT is solvable in polynomial time, we can determine the answer to any NP problem, X, in polynomial time
- We could do this by formulating M (the hypothetical non-deterministic Turing machine to solve X), creating B, and applying the SAT solution to B

The Variables of B

- In what follows, $q \in Q$, $-p(n) \leq i \leq p(n)$, $j \in \Sigma$, $0 \leq k \leq p(n)$, and $p(n)$ is the (polynomial) time it takes M to solve a problem instance of size n
- Note that $|\Sigma|$ and $|Q|$ are technically constants
- The Boolean formula B relies on the following variables:
  - $T_{i,j,k}$ is true iff tape cell i contains symbol j at step k of the computation; there are $O(p(n)^2)$ such variables
  - $H_{i,k}$ is true iff the M's read/write head is at tape cell i at step k of the computation; there are $O(p(n)^2)$ such variables
  - $Q_{q,k}$ is true iff M is in state q at step k of the computation; there are $O(p(n))$ such variables
- Note that these are intended interpretations of the variables, not assigned values

The Clauses of B

- Now let's consider the following clauses:
  - $T_{i,j,0}$ for all i and j such that tape cell i of the input I contains symbol j; this represents the initial contents of the tape; there are O(p(n)) such clauses
  - $Q_{s,0}$, which represents the initial state of M; there is one such clause
  - $H_{0,0}$, which represents the initial position of the read/write head; there is one such clause
  - $T_{i,j,k} \rightarrow \neg\, T_{i,j',k}$ for all j and j' such that $j \neq j'$ (and for all i and k); i.e., there can be only one symbol per tape cell; there are $O(p(n)^2)$ such clauses
  - $T_{i,j,k} \wedge T_{i,j',(k+1)} \rightarrow H_{i,k}$ for all j and j' such that $j \neq j'$ (and for all i and k); i.e., each tape symbol remains unchanged unless written; there are $O(p(n)^2)$ such clauses
  - $Q_{q,k} \rightarrow \neg\, Q_{q',k}$ for all q and q' such that $q \neq q'$ (and for all k); i.e., there can be only one state at a time; there are O(p(n)) such clauses
  - $H_{i,k} \rightarrow \neg\, H_{i',k}$ for all $i \neq i'$ (and for all k); i.e., there can be only one head position at a time; there are $O(p(n)^3)$ such clauses
  - $(H_{i,k} \wedge Q_{q,k} \wedge T_{i,\sigma,k}) \rightarrow \vee_{(q,\, \sigma,\, q',\, \sigma',\, d) \in \delta} [H_{i+d,k+1} \wedge Q_{q',k+1} \wedge T_{i,\sigma',k+1}]$ (for all i and k); i.e., all transitions must be valid; d is +1 or -1, and to generate the right-hand side, we need to loop through all valid transitions; there are $O(p(n)^2)$ such clauses
  - $\vee_{f \in F} Q_{f,p(n)}$; this is a single disjunction (logical OR) of clauses representing possible accepting states; i.e., the machine must end in an accepting state; there is one such clause
- The Boolean expression, B, is the conjunction (logical AND) of all these clauses
- The number of clauses is polynomial with respect to the input size, so B can be formed in polynomial time

What Does B Mean?

- The meaning of B roughly translates to the following in English (this is my own wording):

  "We start with the initial contents of the tape, in the initial state, at the initial position of the read-write head, and there is only one symbol in each tape slot at all times, and no symbol changes unless it is written, and we are in only one state at a time, and we are at only one tape position at a time, and every change represents a valid transition, and we end up at an accepting state."

- Any of assignment of values to the variables that makes the expression B true is analogous to a polynomial-length path for M leading to an accepting state for the original problem instance
- Similarly, if M contains a polynomial-length path that leads to an accepting state, that path can be used to fill in the variables of B (assigning them their intended values) to make B true
- Thus, B is satisfiable if an only if M accepts I
- Instead of solving an instance of the NP problem with M applied to I, you can translate M and I to a Boolean expression, B, in polynomial time, and then determine if B is satisfiable
- If the Boolean satisfiability problem can be solved in polynomial time, this then shows that any NP problem can be solved in polynomial time by an algorithm (in which case, P = NP)

One Year Later...

- In 1972, the year after Cook proved his theorem, Richard Karp published a very influential paper titled, "Reducibility Among Combinatorial Problems"
- This paper showed that 21 problems are NP-complete
- However, Karp's methodology was nothing like Cook's
- Once Cook shows that SAT is NP-complete, it became much simpler to prove that other problems are NP-complete
- *To show that a new problem is NP-complete, you need to reduce a known NP-complete problem to the new problem in polynomial time*; note that the direction of the reduction is important
- A successful reduction means that a polynomial-time solution to the new problem can be used to solve an NP-complete problem (and thus all NP problems) in polynomial time
- The reduction proves that the new problem is NP-hard
- *You also need to prove that the new problem is in NP* (this is usually simple)
- Karp started with Cook's result and used a series of reductions to form a hierarchy of problems that are NP-complete
- Many additional problems have been shown to be NP-complete since then

3SAT

- *3SAT* is a variation of SAT in which the Boolean expression is restricted to expressions in conjunctive normal form (CNF) with exactly 3 literals in each clause
- CNF means that the Boolean expression is expressed as a conjunction of clauses
- Each clause is a disjunction of literals, and each literal is a variable or its negation
- A common notation is:
  - Disjunctions use the symbol "∨" meaning "or“
  - Conjunctions use the symbol "∧", meaning "and“
  - Negation uses the symbol "¬"
- There is a simple reduction from SAT to 3SAT
- For example, a disjunction with less than three literals can be converted to a disjunction with exactly three literals be repeating a term more than once
- This is important because it turns out to be simpler to reduce 3SAT (compared to SAT) to many other NP-complete problems

The Independent Set Problem

- Consider an undirected graph G = (V, E)
- A set of vertices I (a subset of V) is said to be *independent* if for all (i, j) such that i, j ε I, the edge (i, j) does not exist
- The *independent set problem*: Given a graph G and a number k, determine if there is any independent set of nodes, I, with at least k nodes
- This is a decision problem, and it is clearly in NP (because we can clearly verify a solution in polynomial time)
- The optimization version of the problem would ask to find the largest independent set in the graph
- To prove that the independent set problem is NP-complete, what remains is to reduce a known NP-complete problem to this problem

A Neat Reduction

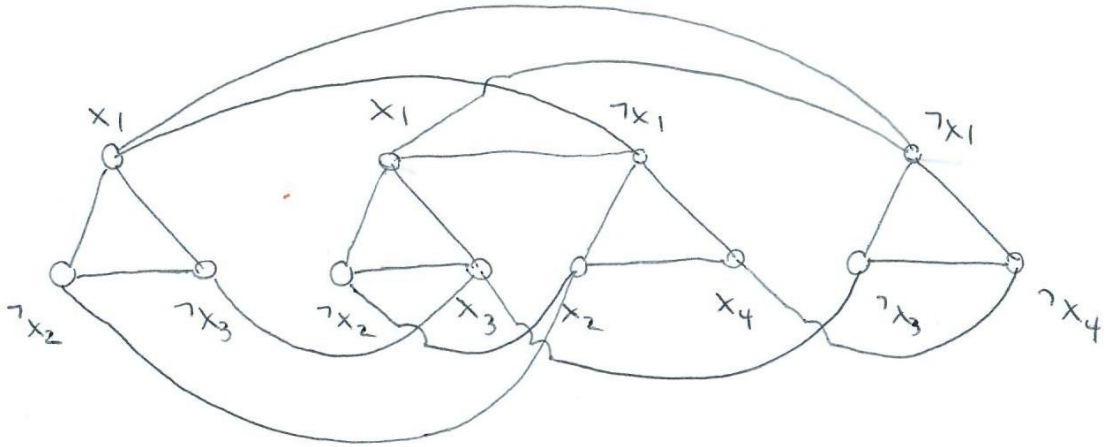
- There is a reasonably simple reduction from 3SAT to the independent set problem!
- Start with a Boolean expression in the conjunctive normal form with k clauses and three literals per clause
- For each clause, construct a triangle in a graph (i.e., give the graph three connected vertices, one for each literal in the clause); so, the graph will consist of k triangles overall
- Then, connect two vertices in separate triangles if and only if one represents the negation of the other; e.g., a vertex representing x_1 would be connected to all vertices representing $\neg x_1$
- If there were a polynomial solution for the independent set problem, this would mean that you could determine, in polynomial time, if there is an independent set of at least size k
- Since this independent set could not use two vertices in the same triangle, it must use exactly one vertex from each triangle (and the size of the independent set would be exactly k)
- This independent set would map to one literal from each of the clauses in the original 3SAT problem that could all be set to true, and thus it represents a solution to that problem
- Similarly, if there is any valid assignment of variables in the original 3SAT problem that makes the original expression true, by choosing one true variable in each clause, we could also find an independent set of size k in the constructed graph
- This reduction proves that the independent set problem is at least as hard as 3SAT, and thus is NP-hard
- We have already stated that the problem is clearly in NP, since a potential solution can be checked in polynomial time
- Therefore, the independent set problem is NP-complete

A Neat Reduction: Example

- Consider the following Boolean expression in 3SAT form:

  $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

- We could map this to the following graph:



- Any independent set of size 4 has a direct mapping to an assignment of Boolean values that make the original 3SAT expression true

Two Related Problems

- The *clique problem*:
  - Given a graph G and a number k, is there any set of at least k vertices that form a clique
  - A clique is a set of vertices such that all edges between them exist
- The *node cover problem*:
  - Given a graph G and a number k, is there a set of at most k vertices that form a node cover
  - A node cover is a set of vertices such that it contains at least one endpoint of every edge
- There are simple reductions from the independent set problem to each of these problems (I'll leave that as an exercise!)

A Few Other NP-complete Problems

- The *max cut problem*:
  - The optimization version of this problem asks us to find the maximum cut in a graph
  - The decision version asks if a cut exists with a capacity of at least some given number k
  - Recall that the minimum cut, related to the maximum flow, can be easily determined
- The *graph coloring problem*:
  - This problem asks us to determine whether the vertices of a graph, G, be colored with k or less colors, such that no adjacent vertices will have the same color
  - Even if you simplify the problem to 3-coloring or 4-coloring, and you restrict the graph to planar graphs, the problem is still NP-complete
- The *Hamiltonian cycle problem*:
  - Given a graph, G, determine if the graph contains any Hamiltonian cycle
  - A Hamiltonian cycle is a simple cycle that starts and ends at the same node and visits every other node exactly once
  - The optimization problem asks you to find the shortest Hamiltonian cycle in a graph; this is also known as the Traveling Salesman Problem

And a Few More NP-complete Problems

- The *knapsack problem*:
  - This problem asks you to select a subset of items from n items; each item i has value v_i and weight w_i, and both are positive integers
  - There is a limit, W, to the total weight of items you can choose (a weight constraint)
  - The decision version of the knapsack problem asks whether it is possible to choose items with a total value of at least some given value k
  - The optimization version asks you to maximize the value of your items
- The *bin packing problem*:
  - This problem asks if it is possible to place N items into B bins
  - Each item has a specified integer size, and all bins share the same capacity, C
  - The optimization problem asks you to find the minimum number of bins that can fit the items
- The *partition problem*:
  - Given a set of N integers, this problem asks you to determine if it be divided into two independent subsets such that the sums of the numbers in each subset are the same
  - To me, this is most surprising NP-complete problem

Why is this important?

- There are many known NP-complete problems
- If any of these problems can be solved in polynomial time, then they all can, as can all NP problems in general
- If this turns out to be the case, then P = NP
- However, it is generally believed by most computer scientists that P ≠ NP
- That would mean that the NP-complete problems can only be solved by algorithms that require exponential time
- This is a problem, because many NP-complete problems, or variations of them, show up often in real-world applications
- Often, it is the optimization version of the problem that comes up in practice, but obviously, these are at least as hard as the decision version of the problem
- If we have learned anything in the two Data Structures and Algorithms courses, we have learned that exponential solutions are not feasible in practice

Approximation Algorithms

- Fortunately, for real many real-world applications that deal with *optimization problems*, you don't need an algorithm that is guaranteed to find the best solution
- An **approximation algorithm** is a polynomial-time algorithm for a problem that is guaranteed to find a solution whose cost is "close" to the best possible solution
- Consider an algorithm, M, that can be applied to an instance, x, of some optimization problem
  - Let c(M(x)) be the cost of the solution that M finds
  - Let OPT(x) be the cost of optimal solution for x
- We say that M is an **ε-approximation** algorithm (one type of approximation algorithm) if it is guaranteed that:

- For *maximization problems*, the cost of the solution returned by an ε-approximation algorithm is never less than 1 - ε times the optimal solution
- For *minimization problems*, the solution returned by an ε-approximation algorithm is never more than 1 / (1 - ε) times the optimal solution

Example of an ε-Approximation Algorithm

- For example, consider the optimization version of the node cover problem
- Given a graph, G, we want to create a node cover, C, such that each edge has at least one of its endpoints in C; the smaller the node cover, the better
- It might seem that a good idea would be to use a greedy algorithm that selects the node in the graph with the largest degree, then adds it to the node cover and removes it from the graph, etc.
- It turns out that this is not an ε-approximation algorithm for any ε less than one, and the error ratio grows as log n, where n is the number of nodes in the graph
- However, a less sophisticated greedy algorithm leads to an ε of 1/2
- Start with an empty set of vertices, C
- While there are still edges in G, choose any edge; add both vertices that the edge connects to C; remove both vertices (and incident edges) from the graph; continue
- Since C was composed of vertices from 1/2 * |C| edges of G, no two of which share a node, any node cover must contain at least one node from each of these edges
- Therefore, OPT(G) ≥ 1/2 * |C| and |C| ≤ 2 * OPT(G)
- Surprisingly, this is the best-known ε-approximation algorithm for this problem

More about ε-Approximation Algorithms

- The best-known ε-approximation algorithms for the optimization versions of different NP-complete problems differ wildly in quality
- For the knapsack problem, for any given $\varepsilon > 0$, you can construct a polynomial-time ε-approximation to solve the problem
- For the general traveling salesman problem, on the other hand, it is provable that there is no polynomial ε-approximation algorithm with $\varepsilon < 1$ (unless P = NP)
- However, for certain types of graphs (e.g., graphs that obey the triangle inequality), there are known ε-approximation algorithms
- There are even better ε-approximation algorithms for Euclidean graphs
- For complex graphs that do not allow ε-approximation algorithms, there still may be *heuristics* that are likely to find close-to-optimal solutions
- Unlike an approximation algorithm, however, a heuristic is not guaranteed to find a solution close to optimal