ECE374: Algorithms

Composite of Lecture and At-Home Study

Pradyun Narkadamilli

Contents

1 General Reminders and Gotchas

- Substrings and Subsequences - not the same thing!

2 General Overview

- Algorithm will have a *runtime* complexity (EXPSPACE, PSPACE, NP, co–NP)
- Problems have a *complexity class* since they are never actually run
  - Recursively enumerable (Turing Machines)
  - Context-Sensitive (Linear bounded automata)
  - Context-free (Push-down Automata)
  - Regular (DFAs, NFAs, RegEx)
- *Algorithm* - step-by-step problem solving method
- *Problem* - some question we'd like answered given input (does input fulfill a property X)

3 Regular Languages

- *Language* - set of strings. Given an alphabet Σ, we get a language that subsets Σ^* - the set of all strings of all lengths including an empty string ϵ (contains no symbols).
- A *string* over an alphabet is just a finite sequence of symbols
- $\emptyset$ is the nullset, or empty set. Contains nothing (not even ϵ)
- String concatenation expressed as adjacent symbols (either without an operator or with $\cdot$)
  - Associative, not commutative operator. ϵ is the identity operand for this
- *Subsequence* - select set of characters from a string still in the same order. Do not need to be contiguous, but need to be ordered.
  - Ex: *EE37* is a substring of *ECE374*
- *Substring* - similar to subsequence, but contiguity required.
- *String exponent*: For a string w we define it as follows:
  - $w^0 = \epsilon$
  - $w^n = ww^{n-1}$
- The *complement* of a language L on alphabet Σ is written as $\overline{L} = \Sigma^* \setminus L$, or $\overline{L} = \Sigma^* - L$ (set difference from the full string set)
- *Set Concatenation* - forms a set with strings formed by every permutation of concatenation of the sets' elements
  - Mathematically: $XY = \{xy | x \in X, y \in Y\}$
- Set exponentiations have 3 forms:
  - To the $n \in \mathbb{Z}$ implies all strings of some fixed length
  - To the $*$ implies all strings of any finite length (Kleene star)
  - To the $+$ implies all non-ϵ strings

3.1 Grammars

- *Grammar* - set of rules defining the strings in a language
- Defining Grammars requires a quadruple $G = (V, T, P, S)$
  - V is a finite set of non-terminal (variable) symbols - need to be substituted with a string composed of symbols in T via a production $p \in P$
  - p is of the form $A \to \alpha$ for $A \in V$ and $\alpha \in (V \cup T)^*$
  - We have a start symbol S, which is the starting symbol of the grammar

3.2 Properties of Regular Languages

- *Kleene's Theorem* - A language is considered regular if it can be obtained from finite languages applying the union, concatenation, and repetition operators a finite number of times.
  - By extension: DFAs, NFAs, and RegExes all encompass the same class of languages
- Regular Languages can also be combined into more regular languages:
  - The union and intersection of two regular languages is also regular
    * Intersection can be proven via De Morgan's Theorem
  - The concatenation of two regular languages is regular
  - For a regular language L, the language obtained with the Kleene star is also regular
  - The complement of a regular language is regular
- **Lemma**: Every finite language L is regular
  - Infinite languages can still be regular, like a Kleene star language
- Any language generated by a finite sequence of operations will be regular
  - The Kleene star is considered a single operation
- **Lemma**: If you have many regular languages over an alphabet Σ, their union $(\cup_{i=1}^{\infty} L_i)$ is not necessarily regular

3.3 Regular Expressions

- Simple patterns used to describe related strings
- Regular expressions also have inductive cases
  - Regular expressions can be union'd to represent a language
  - Regular expressions can be concatenated to concatenate languages
  - regular expressions can also have a Kleene star to represented that on a language
- Regular expressions denote regular languages showing the operations used to form the language
- Regular expressions are *equivalent* if they represent the same language

4 Regular Automata

4.1 DFA

- Discrete Finite Automata (DFA) also called an FSM
  - We say that each state has transitions coming out of it associated with a symbol Σ
  - DFA has only one transition per state per symbol
- A DFA *accepts a string* if the walk represented by the string produces a valid walk within the DFA that ends on an "accepting" (or final) state
  - The set of valid walks on a DFA M is represented as a language $L(M) = \{w | M accepts w\}$
- DFA is formally defined with a 5-component tuple
  - Q - set of states
  - Σ - input alphabet
  - δ - transition function defined on $Q \times \Sigma \rightarrow Q$
  - An initial state $s \in Q$
  - A set of accepting/final states $A \subseteq Q$
- We define a shorthand function $\delta^*(q, w)$ that evaluates the walk given by string w by recursively evaluating δ
- **Theorem**: Languages accepted by DFAs are *closed under complement*
- We can take the "union" of two DFAs by creating a *cross-product* machine
  - Each state is the concatenation of the old states, and transition on a symbol will be to the correct concatenation of old states
  - Effectively evaluates 2 DFAs in parallel
- DFAs effectively express the same set of languages as regular expressions

4.2 NFA

- *NFA* - Non-deterministic Finite Automata.Theoretical device for having more than one output for the same machine.
  - Capable of taking multiple states *concurrently* - when a decision is given, the NFA takes both paths and continues evaluating both branches concurrently
- An NFA is capable of having multiple outgoing transitions on the same state for a single symbol
  - Furthemore, we introduce ϵ transitions, which do not require any symbol to be taken (always branched out into)
- Due to the concurrency of an NFA, it is easier to show that the string is *accepted* than to show that it is *not* accepted
- **Formal Definition**: an *NFA* is defined as a 5-tuple
  - Q is the finite set of states
  - Σ is the set of symbols this NFA accepts (input alphabet)

  - δ is the transition function - this just got more complicated
    * Defined on ϵ, 0, and 1 inputs. Each output is now a set of states instead of a single state
  - s is the start state
  - A is the set of accepting, or "end" states

4.3 DFA/NFA/RegEx Equivalence

- In the DFA $\rightarrow$ NFA direction, it is trivial - an NFA by default supports the same constraints that a DFA does
  - Simply convert the delta function to a set notation, and add ϵ to the supported alphabet
- To encompass any possible concurrency of state in an NFA, for an NFA with $\|Q\| = n$, we can create a DFA with at most 2^n states and brute-force transitions into concurrent NFA meta-states (so to speak)
- Regular Expressions can be constructed from a DFA by employing the state removal strategy
  - Convert symbol-level transitions into string-level transitions, thereby removing intermediate states
  - Attempt to condense the DFA until you have an accepting state with an expression for its self-loop
- Regular expressions can also be directly constructed from an NFA
  - We first normalize the NFA by adding epsilon-transitions from all accepting states to a singular q_f, then collapse all the epsilon transitions
  - Use same analysis strategies as for a DFA to create a single transition from start state to end state- this transition is the NFA regex
- For mathematical equality, we require the inverse as well (RegEx $\rightarrow$ NFA/DFA)
- RegEx can be converted to an NFA via *Thompson's Algorithm*
  - General idea is to correspond every operation in a RegEx to an NFA structure
  - Concatenation $\rightarrow$ series connection
  - Union (+) $\rightarrow$ branching in NFA on ϵ-transitions
  - Kleene Star $\rightarrow$ branch from start to next state - one branch is ϵ, other branch is a DFA-esque loop representing content of the repeated expression
- RegEx can be converted to a DFA via *Brzozowski's Algorithm* (wtf kind of last name is that)
  - Incrementally convert prefix operation of the RegEx to a sub-DFA, then merge them via serial connections

5 Non-Regularity

- Until now, regular languages only encompass the *regular* class of Chomsky's Computability Hierarchy
  - Want to now expand to the *context-free* computability class
- Class of regular languages is *countably infinite* - set of all languages should be *uncountably infinite*
  - Ex: $L_1 = \{0^n 1^n | n \geq 0\}$ is non-regular - seems easy to construct, but you can't come up with a concatenation or union to form it!
  - Presents an interesting precedent - non-regular languages can be a *subset* of a regular language

5.1 Proving Non-Regularity

- *Distinguishable States*: Two states in a DFA are considered *distinguishable* if there is at least one string $w \in \Sigma^*$ that will form a path to only one of the two states
  - Can extend definition to strings: *distinguishable strings* when $x, y \in \Sigma^*$ and $\exists w \in \Sigma^*$ where only one of xw, yw is in $L(M)$
- Two strings equivalent on language L are denoted $x \;_L y$
  - The relation $\;_L$ can partition a language L into equivalence classes
- *173 Review:* An equivalence relation on some set A constructs an equivalence class $[a] := \{x \in A | x \; A\}$
  - These relations must be reflexive, symetric, and transitive
- There are 3 big methods we can use to prove non-regularlity
  - Fooling sets
  - Closure properties
  - Pumping lemma (not discussed in 374)

5.2 Fooling Sets

- *Fooling Set*: also called a distinguishing set, this is a set for a language L where every two strings $x, y \in F$ where $x \neq y$ are distinguishable
  - **Theorem**: Given a finite fooling set $F \subseteq L$, there exists no DFA M accepting L with less than $\|F\|$ states
  - **Corrollary**: If there is an infinite fooling set $F \subseteq L$, then L is non-regular

5.3 Closure Properties

- As discussed prior , there are some properties that regular languages will have when interacted with other regular languages (concat, complement, etc.) - specifically that regularity is preserved
- The general strategy here is to try to take known regular languages and combine them with some unproven language L
- *Myhill-Nerode Theorem*: A language is regular if *and only if* there is a finite number of equivalence classes
  - This is an equivalent condition to requiring a finite fooling set - each element of the fooling set represents an equivalence class

6 Context-Free Languages

- Like regular languages, context-free languages can be defined by a *context-free grammar (CFG)*
  - Uses the same four-tuple as regular languages
- *Derives relation*: Given $\alpha_1, \alpha_2 \in (V \cup T)^*$ for a CFG G, we say $\alpha_1 \rightsquigarrow \alpha_2$ if there are intermediate strings $\beta, \gamma, \delta \in (V \cup T)^*$ such that:
  - $\alpha_1 = \beta A \delta$

  - $\alpha_2 = \beta\gamma\delta$ where $A \to \gamma \in P$

- We describe a single-step derives above, where α_2 directly derives from $alpha_1$. We can also define this relation inductively
  - $\alpha_1 \rightsquigarrow^0 \alpha_2$ if $\alpha_1 = \alpha_2$
  - $\alpha_1 \rightsquigarrow^k \alpha_2$ if $\alpha_1 \rightsquigarrow \beta \in G$ and $\beta \rightsquigarrow^{k-1} \alpha_2$

- *Context Free Languages*: Given a CFG G, we construct the language $L(G) := \{w \in T^* | S \rightsquigarrow^* w\}$
  - Interpreting the expression: Any w made of concatenated terminal symbols that can derive from the start symbol

- In regular languages, terminals can only appear on *one side* of the production string and only a *single variable* is allowed in the result of a production - this is not true for a CFL

- Much like RLs, CFLs are also closed under union, concatenation, and the Kleene star

7 Pushdown Automata

- The key idea behind our CFGs and CFLs is that we want *recursive definitions* - to do so, we need stack

- *Pushdown Automata (PDA)*: The machine for CFGs - acts as an expansion on NFAs that can incorporate a stack
  - Defined on a 6-tuple $P = (Q, \Sigma, \Gamma, \delta, s, A)$
    - Q, Σ, s, A retain their traditional definitions
    - Γ is a finite set called *stack alphabet*
    - To incorporate the stack, the transition function $delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \cup \{\epsilon\} \to \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\}))$

- For PDA, transition edges now denoted as $a, b \to c$ where $a \in \Sigma, b \in \Gamma, c \in \Gamma$
  - a is the input symbol
  - b is the stack item that we pop (e.g we only take this transition if b is ϵ or stack top is b)
  - c is the stack symbol we push
  - Direction of the arrow denotes our destination state

- The PDA is considered "complete" when we are in an accepting state *and* the stack is empty
  - We can append a "$" character to the stack at the very beginning of the PDA to denote the bottom of the stack (e.g not ready for exit until we see this char again)
  - The above acts as an explicit condition for enforcing stack emptiness

8 Algorithms

- *Algorithm*: a method to solve a specific problem
  - We define a "problem" to simply be a function f going from one string to another on a finite alphabet
  - Its steps and instructions are primitive, and can be mechanically executed
  - Must be finitely and universally describable (cannot have an infinite number of unpredictable instructions)

  - Is allowed to have state/memory (how else do you recurse bozo)
- We consider a computer a mechanism that implements the primitive instructions for an algorithm
  - It automates the execution of the algorithm, and keeps track of state
- *Model of Computation*: an idealized mathematical construct that describes primitive instructions and other details
  - The computer implements one of many possible models of computation
  - Examples: stochastic computing, standard programming model, Turing Machine model
- *Unit-Cost RAM Model*: A simplified version of the standard programming model
  - Basic data type is an integer number
  - Numbers fit in a "word" of memory, and operating on words take constant time
  - Arrays allow constant time random access to any word, and pointers fit in a word as well
  - Assume bitwise functions, floor functions, and bounded word sizes are all restricted or disallowed
- When analyzing algorithms, some big things we look out for
  - Asymptotic worst-case runtime
  - Asymptotic worst-case space usage
- *Reduction*: map a problem A onto another problem B
  - Positive direction of this is that an algorithm for B implies the existence of an algorithm for A
  - Negative direction is that no good algorithm for A implies no good algorithm for B
- *Recursion* acts a subcase of reduction, where a problem A can be mapped onto a smaller version of itself
  - *Ex*: the fibonacci sequence for a number N can be mapped onto the fibonacci sequence for $N-1$
  - Recursion terminates when the instance gets to a point where it can be trivially solved (base case)
  - *Ex Runtimes*: Hanoi has a recursive solution in exponential time, Mergesort is $n\log(n)$
- *Backtracking*: Traverse a search tree in a DFS-esque reucrsion pattern, then "backtrack" if an invalid permutation is reached
  - Keep recursing if there is another valid permutation reachable from the current point of recursion

8.1 Divide and Conquer

- Consider QuickSort as an initial example
  - Instead of binary split of Mergesort, we pick a "pivot" element (typically last or first element)
  - Split array into 3 sub arrays - less than pivot, greater than pivot, and pivot
  - Quicksort on each of the non-pivot subarrays, then concat (no iterated sort on coalesce)

8.2 Dynamic Programming

- As opposed to recursive algorithms, you can potentially *cache* results from past recursions to reuse them in other parts of the recursion tree
- Two methods of memorizing values
  - *Explicit*: Initialize a fixed-memory hash table to store intermediate results
    * Requires knowledge on number of potential sub-problems
    * Can potentially reduce space complexity by saving results *relevant* to higher order computation, not all recursions
  - *Automatic*: Use a hashmap to store results after computation, check if present before a potential recursive call
- Finding recursions that can be efficiently memorized is called *Dynamic Programming*
  - Summarized as a combination of *smart recursion* and *explicit memorization*
  - Can lead to potentially polynomial time algorithms
  - Does not necessarily need to be an iterative algorithm, but we prefer to remove recursion
- **General Method for Dynamic Programming**
  - Find a recursive backtracking solution for some problem
  - Identify structure of subproblems, estimate number of subproblems
  - Rewrite subproblems more compactly
  - Rewrite rescursive algorithm in terms of subproblem notation
  - Solve subproblems bottom-up to convert recursion to iterative
  - Optimize with additional data structuers or ideas

9 Graphs

9.1 Intro

- Graphs represented as a two-tuple (V, E)
  - V is the set of nodes/vertices, E is the set of edges
  - Common representation between directed and undirected graphs, but the E is slightly different
- Edge between two nodes usually noted as a set $\{i, j\}$
  - The tuple notation (i, j) is reserved for *directed* edges
  - For simple graphs, $u \neq v$ for every $\{u, v\} \in E$
- Each node has some *degree*, which is the number of nodes adjacent to it
  - A node is *adjacent* to another if there is an edge connecting the two
  - The set of nodes adjacent to some node a is called the *neighborhood* of a ($N_G(a)$)
  - Minimum degree and maximum degree of a graph are denoted $\delta(G)$ and $\Delta(G)$
- Various data structures can be used to encode information about graphs
  - *Adjacency Matrix*: high space complexity (n^2) but constant look up time for adjacency

- *Adjacency List*: store the neighborhood of each node. Low space complexity, but adjacency check may not be constant time
    - Based on outgoing edges only for directed graphs
- In this class, assume graphs are usually represented as unsorted adjacency lists
- Two nodes are considered connected if a path can be formed from one to the other
    - A *cycle* is formed if a node can form a path back to itself with a sequence of distict vertices and edges
    - The connectivity relation is reflexive, symmetric, and transitive
    - Based on the above properties, connected components of a graph are equivalence classes of the connectivity relation
    - A connected graph will only have one connected component

- Connectivity criteria slightly more complex for directed graphs
    - We define rch(u) to be all the nodes reachable from u via directed paths
    - A node is *strongly connected* to another node if directed paths can be formed in both directions
    - The *strong* connectivity relation is reflexive, symmetric, and transitive - normal connectivity is not
    - We find strongly connected components for directed graphs to be the equivalence classes

9.2 Directed Graphs

- *Source*: No incoming edges
- *Sink*: No outgoing edges
- *Directed Acyclic Graph*: A directed graph is a DAG if there is no *directed cycle*
    - Every DAG has at least one source and at least one sink
    - Any directed graph with a topological ordering is a DAG
- Want to be able to ”order” the nodes in a directed graph
    - If nodes were ordered left-to right in *topological order*, all edges would point to the right
- Can implement topological ordering (top sort) in $O(m + n)$
    1. Count in-degree of each node
    2. For all sources, add node to out array and lower degree of connected nodes
    3. Repeat step 2 until no more nodes left to order
- Note that topsort is a partial order, not strict
    - Cannot topsort cyclical graphs
- You know what a DFS is, here’s some more info
    - Runtime for a DFS is always $O(m + n)$
    - Output will be dependent on vertex ordering
    - The set of edges and nodes forming the search path is called the ”forest” T
    - You will only have one incoming edge per node that is in T

- Can tag each node with pre/post time (start and end time of its recursive call)

- Note that for any two nodes u, v the intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ either have a containment relation or are disjoint
- Can classify any edge of the graph G w.r.t the DFS tree
  - *Tree edges* are in T
  - *Forward edges* are not in the DFS tree, but go to a node with a containment relation on times
  - *Backward edges* are not in DFS tree, but go to node with inverse containment relation on times
  - *Cross edges* are not in DFS tree, but go to a node with disjoint pre/post times
- Can use DFS to topsort and to do cycle detection on a directed graph
  - While computing a topsort, if the sort fails we assume a cycle is found, and return it
  - When computing DFS, any back edge indicates a cycle
  - If there is a cycle, return the path from u to v in T and then the back-edge
  - A DFS will inhrently compute the topological sorts if you linearize the search tree
  - If $post(v) > \text{post}(u)$, then no edge $(u \to v)$ exists
- We can create a meta-graph of the strongly connected components in G by collapsing cycles
  - Effectively, for a graph G, G^{SCC} will have no cycles
  - Each node in G^{SCC} is a strongly connected component
  - This meta graph can be computed in $O(m+n)$

9.3 Shortest Path Algorithms

- BFS is also $O(m+n)$ - prefer this for distance exploration, DFS for graph structure exploration
  - DFS uses stack (recursion has this implicitly), BFS uses a queue (cannot be done recursively)
  - BFS search has same completeness as DFS
  - Is u reachable from s and $(u \to v)$ is an edge, then $\text{dist}(v) \leq 1 + \text{dist}(u)$
- BFS search tree can be represented as "layers", where each layer represents a distance class
  - Forward/backward edges would cause a jump between layers
  - Tree edges will be in parallel to other forward edges
  - Cross edges would be within the same layer
- *Path*: sequence of distinct vertices where any two subsequent vertices have an edge $v_i \to v_{i+1}$
  - The shortest path is determined by the smallest sum of edge weights
  - BFS looks for fewest number of hops, but does not guarantee weight-sum optimality
- *Walk*: simlar to path, but no constraint on *distinct* vertices
- **Djikstra's**: Max Verstappen CS edition made it up because he was board, and now you have to learn it
  - Source node takes a distance of 0, all others assumed to be ∞ until explored

- At each iteration, take the "unsettled" node with the smallest distance estimate, and explore its neighbors
  - For each explored neighbor, update distance estimate and log the "previous node" associated with estimate
  - Add the iterated node into the settled list
  - Once all nodes are settled, we have shortest distance (and path) from s to any $v \in V$

- Runs in $O(m + n^2)$ - n iterations of n to select min-cost node, and m to explore every possible edge
- Runtime can be reduced to $O(m+n\log(n))$ or $O((m+n)\log(n)$ via priority queues or Fibonacci heaps
- Djikstra's should be run on G^{rev} if we want closest distance from all V to s

9.4 Graph DP

- Djikstra's assumes that we can ignore a path completely if the partial's cost exceeds the true length of another partial
  - This assumption becomes false if we have negative edge lengths
  - Normalized addition is bad because the additive correction is multiplicative over edge count
- **Bellman-Ford**: Finds the minimum cost path
  - Maintain a counter for number of edges used - can be at most $n - 1$ on a path
  - Recursive formulation will brute force potential edges and take the minimum, or just burn an edge in the counter
  - DP solution brute forces the discrete possibilities: minimizing cost at each node with at most k edges to use up
    * $O(mn)$ DP solution possible with $O(n)$ memory complexity
    * Iterate over all edges $n - 1$ times to generate the minimum cost of any node in the graph to s in under n edge path
  - Check if there is any extra minimization on an n-th iteration to see if there is a negative cycle
- Can use a topsort and then a simple iteration over edges to pull out the minimum distance from s to every other node in $O(m + n)$ if the graph is a DAG
- **Floyd-Warshall**: Generate all-pairs shortest paths
  - Djikstra's only accounts for a single start node, so pulling all-pairs would be $O(nm + n^2\log(n))$
  - Floyd-Warshall iterates over every pair and gradually allows more and more intermediate nodes
  - Runs in $O(n^3)$ with space $O(n^3)$

10 Reductions

- Reductions used for two big reasons
  - Determining if a problem has a more efficient algorithm
  - Determining if a problem has *no* algorithm
- Can map down most problems onto another fundamental problem that someone smarter than us has already established the computational hardness of

- If the core problem is unsolveable, then we end up having conditional results on our new problem
  - Usually limit attention to *decision* problems when proving hardness (boolean functions on some Σ^*)
- We form **reductions** as an algorithm mapping one problem's instance onto another as to form a bijection
- Classic example: An algorithm to find a "clique" of size k in a graph can trivially be reduced down to the algorithm to find an independent set of size k
  - Your reduction step is inverting each element in the adjacency matrix, effectively
  - Reduction is additive to the other algorithm's runtime, and change in input size needs to be accounted
- Note that not every reduction will be efficient by default - example is NFA onto DFA reduction
  - An algorithm known to be PSPACE on the DFA can suddenly turn into an exponential NFA algo
  - As a result, we are mainly interested in *polynomial-time* reduction steps (e.g *Karp reductions*)
  - On Karp reductions, we know that if Y is polynomial and $X \leq_P Y$, then X is polynomial
- *Conjuctive Normal Form*: POS-form formula built on literals (boolean variable or its complement)
  - A formula φ is a CNF where each sum clause has exactly 3 different literals
- We construct the SAT problem where we inputs to make an arbitrary CNF hold true
  - We construct the 3SAT problem for φ compliant CNFs in particular
  - SAT is short for *satisfaction* or *satisfiability*

10.1 Complexity

- We can partition all problems into a couple of fundamental complexity classes
  - P problems are polynomial time
  - P is encapsulated by PSPACE, which spans all problems solveable by a Turing Machine in polynomial space
  - EXPTIME encapsulates PSPACE, and denotes all problems solveable by a Turing Machine in exponential time
  - EXPSPACE is solvable with exponential space by a Turing Machine, encapsulates EXPTIME
- Within the space of PSPACE, we define two new complexity classes
  - NP encapsulates P but is within the bounds of PSPACE
    - It is solved by a non-det turing machine in $O(n)$ to return a YES (SAT, 3SAT, factorization)
  - coNP overlaps NP partiall and also encapsulates P within PSPACE - it
    - Solved by an NTM in $O(n)$ to check NO instances (inverse SAT, clique/independent set)
- NP-hard problems encapsulate NP and coNP problems while potentially being undecidable
  - An prroblem is undecideable if there is no algorithm to solve it
  - These problems are *at least* as hard as the hardest problems in NP
  - The problems overlapping NP and NP-hard are *NP-complete* - all NP problems can reduce to these

- ***What is NP?***: NP is a set of decision problems with *nondeterministic* polynomial time algorithms
  - They are guaranteed to have exponential time algorithms, and are a superset of P
  - Nondeterministic computers can take both paths for any decision in a decision tree (NFA but computer)
- A problem is considered *NP-Complete* if every other NP problem can be reduced onto it
  - It's generally believed that $P \neq NP$, but solving an NP-complete problem would imply equality
  - Thus NP problems are usually unlikely to be solved efficiently (need to be brute forced)
- **Classic NP-Complete Problems**
  - Hamiltonian Path
  - 3-Coloring
  - 3Sat
    - Can be mapped onto both 3-Coloring and Hamiltonian Path
    - 3-Coloring map basically forms "gates" with graph color induction

11 Decidability

- *Cantor's Diagonalization Argument*: Shows countability of a set
  - Should be able to *systematically* list out the elements of a set, even if it's infinite
  - $\mathbb{R}$ is famously not countable
- Set of all possible languages is *uncountable*
- Set of all programs is... *countable*???
  - Some languages... cannot be represented by a Turing Machine
  - These languages are **undecidable**
- A *recursively enumerable* language (RE) is the language representation of some Turing machine
  - shitty - undecidable, may not halt on negative
- A *decidable* language is the language representation of a Turing machine that halts on all inputs
  - not shitty - decidable, always gives an accept/rejection
- *Halting Problem*: Given a program Q, will it stop?
- *Halting Theorem*: No program can deterministically stop while solving the halting problem
- *Decider*: A program (TM) for a language that always stops, and outputs acceptance/rejection for any possible input string
  - Turing machine *on top* of a TM!
  - A language with a decider is *decidable*
- *Recognizable Languages*: There exists a TM such that it stops on enough inputs such that $L(M) = L$ for the recognizable language L

- If a language and its complement are both recognizable, then both languages are decidable (rejection and acceptance are both halting)

- *Oracle*: yes/no function returning whether $w \in L$ for some language L, with w as the problem instance
  - A language X reduces to another language Y if we can form a decider given an oracle for Y
  - e.g $X \Rightarrow Y$ - if Y decidable, then X decidable (contrapositive is also true)
  - Can prove language undecidability by reducing a known undecidable problem w/ a decider for Y
- **Undecidable Languages To Remember**
  - These languages are of *pairs* of a machine and an input
  - $A_{TM} = \{\langle M, w\rangle | M$ is a TM and M accepts $w\}$
  - $A_{\text{HALT}} = \{\langle M, w\rangle | M$ is a TM and M stops on $w\}$
- The language of empty DFAs is *decidable* - can be determined via a BFS/DFS effectively
  - DFA equivalency is *also* decidable!
- Most properties *defining a TM language* will end up being undecidable
- *Rice's Theorem*: If L is a language consisting of Turing machines:
  - *If* membership is solely dependent on $L(M)$ for a machine M
  - And *if* the set $L \neq \emptyset$ and L does not contain every TM
  - L *must* be undecidable