NOTE: Please do not delete information from before the midterm. If you want your own copy of a certain size, please make your own file.
Contributors: Jason Berlinsky (with credits to study guide contributors)

Equivalence Relations

R is **reflexive** if for every x, xRx
R is **symmetric** if for every x and y, xRy implies yRx
R is **transitive** if for every x, y, z, $xRy \wedge yRz \rightarrow xRz$

Boolean Logic

$\neg$ - Not
$\wedge$ - And
$\vee$ - Or
$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
For any two sets A and B, $\bar{A \cup B} = \bar{A} \cap \bar{B}$

Finite Automata

DFA

Each state must have exactly one transition arrow for every item in the alphabet, and it may only occupy a single state at a time. Formally described by $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states (as $\{q_0, q_1, q_2\}$), Σ is the alphabet, δ is the transition function with domain $Q x \Sigma$ and range Q $(Q x \Sigma \rightarrow Q)$, q_0 is the start state $\in Q$, and F is the set of accept states, which may be the empty set, $\subset Q$.

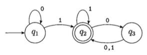


FIGURE 1.6
The finite automaton M_1

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and
5. $F = \{q_2\}$.

Complementation

Flip acceptance states to get the complement.

Combining

Let's say there are two DFAs, $A = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $B = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Let the intersection of the two, $C = (Q_1 \times Q_2, \Sigma, \delta_3, (q_1, q_2), F_1 \times F_2)$, where $\delta_3((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$.
The union of the two is given by closure properties: $A \cup B = \overline{\bar{A} \cap \bar{B}}. F_3 = \overline{\bar{F_1} \times \bar{F_2}}$

Regular Operations

Union ($\cup$): Returns a set containing all elements that appear in either set.
Concatenation (.): Returns a set containing all combinations of an element from set A and an element from set B
Star ($*$): Returns a set containing all permutations of each element in a given language. This returns an infinite set. Also all words over the alphabet.

Non-Deterministic

A more generalized form of a DFA. Each state does not need a transition arrow for each element in the alphabet. May have more than one active state, may also have more than one transition arrow for a given element in the alphabet. Have a special symbol ϵ which is followed when present as a transition and does not "eat" a character from the string.
Try all legal transitions in parallel. On choice, pick/guess best transition towards acceptance. Accept if there is some path from start to accept.

Let A be the language consisting of all strings over $\{0,1\}$ containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A.

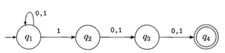


FIGURE 1.31
The NFA N_2 recognizing A

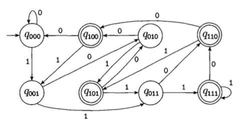


FIGURE 1.32
A DFA recognizing A

Closure and Projection

Closure: The idea that any of the regular operations performed on two regular languages will result in another regular language.
Projection: The intersection where all elements in both sets will be present in the new set.

Regular Expressions

Formal definition (R_n are regexps):

- α for some a in the alphabet Σ
- ϵ
- $\emptyset$
- $(R_1 \cup R_2)$
- $(R_1 . R_2)$
- (R_1^*)

α and ϵ represent the languages $\{\alpha\}$, $\{\epsilon\}$ respectively, and $\emptyset$ is the empty language. The remaining show what happens when a closure property is applied on the base languages.
Symbols:

- Σ: Any symbol in the alphabet
- $*$: Repeat the previous character 0 or more times
- $+$: Repeat the previous character 1 or more times.

Definitions:

- $R \cup \emptyset = R$ - Adding the empty language to any other language will not change it
- $R . \epsilon = R$ - Joining the empty string to any other string is the same string
- $A^* = \{\epsilon\} \cup A \cup AA \cup AAA \cup \ldots$
- $|A \cup B| <= |A| + |B|$
- $|AB| <= |A| . |A|$
- $\emptyset^* = \{\epsilon\}$. If $A \neq \emptyset$, then A^* is infinite
- $L(\emptyset) = \emptyset$
- $L(\epsilon) = \{\epsilon\}$
- $L((a \cup b)\epsilon) = \{a, b\}$
- $L((a \cup b)\emptyset) = \emptyset$

NFA $\rightarrow$ DFA

Given a set $R \subseteq Q$ of states in A_1, let $E(R)$ be the states reachable from R by following 0 or more ϵ-transitions. Given NFA $A_1 = (Q, \Sigma, \delta, q_0, F)$, construct DFA $A_2 = (Q', \Sigma', \delta', q_0', F')$. $Q' = P(Q)$, $q_0' = E(\{q_0\})$ (states reachable from q_0 by ϵ-archs. For $R \in Q'$ and $a \in \Sigma$, let $\delta'(R, a) = \{q \in Q | \exists r \in Rq \in E(\delta(r, a))\}$ (states reachable from some $r \in R$ by an a arc followed by any number of ϵ arcs.
$F' = \{R \in Q' | R$ contains an accept state of $A_1\}$

Regex $\rightarrow$ NFA

EXAMPLE 1.58
In Figure 1.59, we convert the regular expression $(a \cup b)^*aba$ to an NFA. A few of the minor steps are not shown.

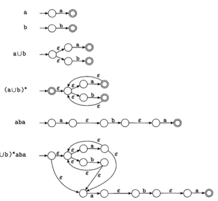


FIGURE 1.59
Building an NFA from the regular expression $(a \cup b)^*aba$

NFA $\rightarrow$ Regex

Remove states one at a time, maintain equivalence, until we get two states with one regex transition.

- Add new start state with an ϵ arc to old start state
- Add new single ccpet state, connected with ϵ arcs from old accept states
- No arcs into start state or out of accept state
- All other arcs present (with $\emptyset$ label if necessary)

DFA $\rightarrow$ Regex

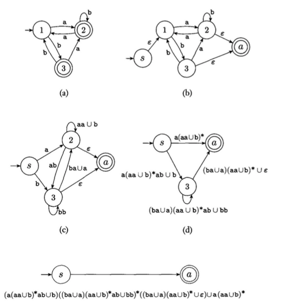
This is aided by a **generalized nondeterministic finite automaton** defined as the 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$. Q is the finite set of states, Σ is the input alphabet, $delta : (Q - \{q_{accept}\}) x (Q - \{q_{start}\}) \rightarrow R$ is the transition function.

CONVERT(G):

1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. Return the expression R.
3. If $k > 2$, we select any state $q_{rip} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{start}, q_{accept})$, where
$$Q' = Q - \{q_{rip}\},$$
and for any $q_i \in Q' - \{q_{accept}\}$ and any $q_j \in Q' - \{q_{start}\}$ let
$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$
for $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$, and $R_4 = \delta(q_i, q_j)$.
4. Compute CONVERT(G') and return this value.

And of course the obligatory example problem:

EXAMPLE 1.68
In this example we begin with a three-state DFA. The steps in the conversion are shown in the following figure.

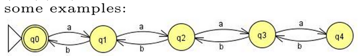


Nonregular Languages

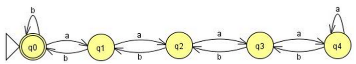
Any language that can not be determined in a finite amount of states on a DFA or NFA is nonregular. Good examples are languages that require the knowledge of a previous count to prove them, such as a language that has some number of 0 followed by the same number of 1s.

Pumping Lemma

The pumping lemma is a simple proof to show that a language is not regular, meaning a FSM can not be built for it. The canonical example is the language $(a^n)(b^n)$. It is trivial to build a FSM for some examples:



Such an FSM will work all the way up to $n = 4$, but the language does not constrain n. No matter how many states are added to the FSM, there will be an input where n is the number of states plus 1, and the machine will fail. Therefore, we must add a loop in the FSM to keep the number of states finite, as such:



This will work. However, after the first 4 as, the machine loses count of how many have bene input because it stays in the same state. Therefore, this erroneously accepts the string $aaaa(a^*)bbbb$. We can say that (a^*) can be pumped. The fact that the FSM is finite and n is not bounded guarantees that any machine which accepts all strings in the language must have this property – the machine must loop at some point, and when it loops, the language can be pumped. Therefore, no FSM can be built for this language, thus it is not regular.
Mathematically, let L be a regular language. Then there exists an integer $p >= 1$ depending only on L such that every string w in L of length $>= p$ can be written as $w = wyx$, i.e. w can be divided into 3 substrings, such that:

- $|y| >= 1$
- $xy <= p$
- $\forall i >= 0, xy^i z \in L$

y is the substring that can be **pumped** (removed or repeated any number of times, and the resulting string is always in L).
(1) means the loop y to be pumped must be of length at least 1, (2) means the loop must occur within the first p characters. $|x|$ must be smaller than p by (1) and (2); apart from that there is no restriction on x and z.

In other words, for any regular language L, any sufficiently long word $w \in L$ can be split into 3 parts, $w = xyz$, such that all the strings $xy^k z$ for $k >= 0$ are also in L.

Game

Consider $ADD = \{``a + b = c"\}$, e.g. $2 + 3 = 5 \in ADD$, but $2 + 3 = 6 \notin ADD$. Can a 17-state DFA M recognize this language? **No.** Consider $1, 2, 3, \ldots, 18$ as the value for a. Two different numbers, e.g. 2 and 5, get M to the same state. Then $2 + 1 = 3, 5 + 1 = 3$ take M to the same state. **CONTRADICTION.**
Paula wants to prove that a language is regular; skeptical Victor wants to verify.
Paula: "This 17-state M recognizes ADD
Victor: "Oh, really? What if we start with $1, 2, \ldots, 18$?
Victor wins; ADD **is not regular**
- - - - - - - - - - -
Show $A = \{ww : w \in \Sigma^*\}$ is not regular.
Paula: p
Victor: $w = 10^{p+1}$
Paula: $ww = xyz$ with $|y| > 0, |xy| <= p$
Victor: $i = 0$
$y \in 0^+$ has exactly one 1. $x = ``1"$, $y = ``00"$, $z = ``0010000" \rightarrow w = ``1000010000"$.
Alternatively, $w = 1000|010|000$. So xz has unequal 0 fields or an odd number of 1s. $xz \notin A \rightarrow$ contradiction.

General for Regular Languages

If a language L is regular, then there exists a number $p >= 1$ such that every string $uwv \in L$ with $|w| >= p$ can be written in the form $uwv = uxyzv$ with strings x, y, z such that $|xy| <= p$, $|y| >= 1$, and $uxy^i zv \in L \forall i >= 0$.

Converse

The converse of the pumping lemma is not true; a language that satisfies the conditions may still be non-regular. Both the original and general versions of the lemma necessary **but not sufficient condition** for the language to be regular.

Turing Machines

Hypothetical devices that manipulate symbols on a strop of tape according to some table of rules. Consists of:

- A **tape** divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains some special blank symbol and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and right. Cells that have no content are assumed to be filled with the blank symbol.
- A **head** that can read and write symbols on the tape and move the tape left and right one cell at a time.
- An **action table or transition function** of instructions that, given the state the machine is currently in and the current symbol being read from the tape, tells the machine to do the folllwing in sequence:
  - Either erase or write a symbol, and then
  - Move the head, and then
  - Assume the same or a new state as prescribed

Note that every part of the machine and its actions is finite, discrete and distinguishable.

Formal Definition

7-tuple $M = (Q, \Gamma, \beta, \Sigma, \delta, q_0, F)$, where:

- Q is a finite, non-empty set of states
- Γ is a finite, non-empty set of the tape alphabet/symbols
- $\beta \in \Gamma$ is the blank symbol (the only symbol allowed to occur on the table indefinitely often at any step during the computation)
- $\Sigma \subseteq (\Gamma - \{\beta\})$ is the set of input symbols
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final/accepting states
- $\delta : (Q - \{F x \Gamma\}) \rightarrow Q x \Gamma x \{L, R\}$ is a partial function called the transition function, where L is left shift, R is right shift, N is no shift.

3-State Busy Beaver

A turing machine that attains the maximum number of steps performed or number of nonblank symbols finally on the tape among all Turing machines of a certain class.

- $Q = \{A, B, C, HALT\}$
- $\Gamma = \{0, 1\}$
- $\beta = 0$ ("blank")
- $\Sigma = \{1\}$
- $q_0 = A$ (initial state)
- $F = \{HALT\}$
- δ is given on attached page

Decidable Languages

A decidable language is:

- A recursive subset in the set of all possible words over the alphabet of the language
- A formal language for which there exists a Turing machine which will, when presented with any finite input string, halt and accept if the string is in the language, and halt and reject otherwise. The Turing machine always halts; it is known as a decider and is said to decide the language.

Decider

A decider is a machine that halts for every input. Also known as a total turing machine. $L(M)$ is decidable.

Closure Properties

Decidable languages are closed under the following operations (if L and P are two decidable languages, then the following are also decidable):

- Kleene star L^*
- Concatenation $L.P$
- Union $L \cup P$
- Intersection $L \cap P$
- Complement $\bar{L}$
- Set difference $L - P$ (can be expressed in terms of intersection and complement)

Decidability

Let $A_{DFA} = \{< B, w > | B$ is a DFA that accepts input string $w\}$

- Need to present a Turing machine M that decides A_{DFA}
- $M =$"On input $< B, w >$, where B is a DFA and w is a string:
  - Simulate B on input w
  - If the simulation ends in an accept state, accept. Otherwise, reject
- When M receives its input, it checks whether the input is actually a DFA and string. If not, reject.
- M then carries out the simulation, keeping track of B's state and position on its tape
- When M finishes processing the last symbol of w, it accepts if B is in an accepting state; rejects otherwise.

Undecidable Languages

Assume that $A_{TM} = \{< M, w >$: $M(w)$ enters "accept"$\}$ is undecidable. The following are undecidable:

- $\{< M, w >$: $M(w)$ enters state $q_5\}$
- $\{< M, w >$: $M(w)$ prints 5$\}$
- $\{< M, w >$: $M(w)$ enumerates $0^n 1^n\}$
- $\{< M >$: $M()$ enumerates $0^n 1^n\}$
- $\{< M >$: $M()$ enumerates a regular language$\}$
- $\{< M >$: For M as an enumerator, $L(M)$ is regular$\}$
- $\{< M >$: For M as a recognizer, $L(M)$ is regular$\}$
- $\{< M >$: For M as a recognizer, $L(M)$ is ...$\}$

Halting Problem

$A_{TM} = \{< M, w > | M$ is a TM and M accepts $w\}$
Theorem: A_{TM} is undecidable.
Proof:

- Assume A_{TM} is decidable. Suppose H is a decider for A_{TM}: $H(< M, w >) = \{$accept if M accepts w; reject otherwise$\}$
- Construct a new TM D that calls H to determine what M does when M is its own input. $D =$"On input $< M >$:
  - Run H on input $< M, < M >>$
  - Output the opposite of what H outputs
- Example of D: $D(< M >) = \{$accept if M does not accept $< M >$, reject if M accepts $< M >\}$
- Run D with itself as input: $D(< D >) = \{$accept if D does not accept $< D >$; reject if D accepts $< D >\}$
- No matter what D does, it is forced to do the opposite $\rightarrow$ contradiction. Thus, neither D nor H can exist. Therefore, A_{TM} is undecidable.

Relationships

- If L is decidable, then L is recognizabl. **Given decider D, let R simulate D. If D is about to enter REJECT, then R instead loops**
- The decidable languages are closed under complement. **Exchange REJECT and ACCEPT states.**
- The recognizable languages are closed under projection. If $\{(x, y) : P(x, y)\}$ is some decidable language of pairs, then $\{x : \exists y P(x, y)\}$ is recognizable. **Given R for the pairs language, build R' to try all y, breadth-first, and simulate** $R(x, y)$
- The recognizable languages are not closed under complement, so some recognizable language is not decidable
- L has a recognizer iff L has an enumerator
- L has a decider iff L has an enumerator that outputs **in lexicographic order**

Equivalencies

- If L has a recognizer R, then a simulating enumerator E tries all strings $w = \epsilon, 0, 1, 00, \ldots$ breadth-first and prints w if $R(w)$ accepts. Then $L(R) \subseteq L(E)$, $L(E) \subseteq L(R)$
- If L has an enumerator E, then a simulating recognizer R on input w runs E. R accepts if R sees that E prints w
- If L has a decider D, then a simulating emulator E tries all strings $w = \epsilon, 0, 1, 00, \ldots$ depth-first
- If L is finite, L has a decider. Otherwise, if L has a sorted-order enumerator E, then a simulating decider D on winput w runs E. D accepts w if D sees that E pritns w, and D rejects w if D sees that E prints anything after w
- If L is recognizable and $\bar{L}$ is recognizable, then L is decidable
- Some language is neither recognizable nor co-recognizable
- If A_{TM} were co-recognizable, A_{TM} would be decidable. On input $< M, w >$, interleave recognizers for A_{TM} and $\overline{A_{TM}}$. When one of the recognizers halts, decide $< M, w > \in A_{TM}$. So $\overline{A_{TM}}$ is not recognizable.
- Similarly, if L is recognizable and co-recognizable, then L is decidable.

Recognizers

Recognizers either reach **accept** or loop forever. $L(M) = \{w : M(w)$ reaches ACCEPT$\}$ is recognizable.

Enumerators

An enumerator is a variant of a TM with an attached printer; used as an output device to print strings. Every time the TM wants to add a string to the list of recognized strings it sends it to the printer. An enumerator starts with a blank input tape. If it does not halt, it may print an infinite list of strings.
The language recognized by the enumerator is the collection of strings that it eventually prints out. These strings may be generated in any order, possibly with repetitions.
Theorem: A language A is Turing-recognizable iff some enumerator enumerates it.
if: If A is recognizable it means that there is some TM M that recognizes A. Then we can construct an enumerator E for A. For this, consider $s_1, s_2, \ldots$, the list of all possible strings in Σ^*, where Σ is the alphabet of M.
$E =$ "Ignore the input.

- Repeat for $i = 1, 2, 3, \ldots$
- Run M for i steps on each input $s_1, s_2, \ldots, s_i$

- If any computation accepts, print the corresponding s_j

Note that if M accepts s, eventually it will appear on the list generated by E. It will appear infinitely many times because M runs from the beginning on each string for each repetition of step 1 – it appears that it runs in parallel on all possible input strings.
only if: If we have an enumerator E that enumerates a language A then a TM M recognizes A. M operates on input w:

- Run E on w. Every time E outputs a string, compare it with w
- If W ever appears in the output of E, accept.

Clearly, M accepts those strings that appear on E's list.

Double Indefinite Tape

A Turing machine with double indefinite tape has indefinite tape to the left and right. Assume that the tape is initially filled with blanks except for the portio that contains the input. Show that a TM D with double infinite tape can simulate an ordinary TM M and M can simulate a TM D with double infinite tape.

Simulating M by D

A TM D with double infinite tape can simulate M by marking the left-hand side of the input to detect and prevent the head from moving off that end.

Simulating D by M

First, we show how to simulate D with a 2-tape TM M

- The first tape of M is written with the input string; second tape is blank
- Cut the tape pf the double infinite tape TM into two parts at the starting cell of the input string
- The portion with the input string and all blank spaces to its right appears on the first tape of the 2-tape TM. The other portion appears on the second tape in reverse order.

Multi-Tape Turing Machines

Theorem: Every multitape Turing machine has an equivalent single tape Turing machine.
Proof: We show how to convert a multitape TM M into a single tape TM S. Assume that M has k tapes. S will simulate the effect of k tapes by storing their information on its single tape. S uses a symbol "#" as a delimeter to separate the contents of the tapes, and keeps track of the locations of the heads by marking the symbols where the heads should be with a dot.
S ="On input $w = w_1 w_2 \ldots w_n$

- Put $S(tape)$ in the format that represents $M(tapes)$, described above
- Scan the tape from the first "#" (representing the left-hand end) to the $(k+1)$st "#" (right-hand end) to determine the symbols under the virtual heads. Then S makes a second pass to update it according to M's δ function
- If S moves one of the virtual heads to the right of a "#", M has moved on the corresponding tape to unread blank contents. So S shifts the tape contents from this cell until the rightmost "#" and writes a blank value on the freed tape cell. It continues the simualtion.

Rice's Theory

A property of recognizable languages is itself a recognizable language. Because of this, "regularity" is a property. If language L has a nontrivial property P, such that $\{< M >: L(M) \in P\}$, L is not decidable. This indicates that **if a language has some nontrivial property, it is not decidable**.
Note that the theorem does not stipulate whether a language is **recognizable**

Non-Trivial Property

A **property** is a set of recognizable languages, e.g. regularity, "two as", etc. A **non-trivial property** P is a language which contains and avoids at least one recognizable language. Simply, P if any property which requires more computing capability than can effectively be used, thus can not be decided.

Countability

A set is considered countable if it either has a finite number of elements or it has the same size as the set of natural numbers ($\{0, 1, 2, 3, \ldots\}$). **The real numbers are uncountable**

Misc Theorems

- The class of regular languages is closed under the union operation
- The class of regular languages is closed under the concatenation operation
- The class of regular languages is closed under the star operation
- Every nondeterministic finite automaton has an equivalent deterministic finite automaton
- A language is regular iff some nondeterministic finite automaton recognizes it
- A language is regular iff some regular expression describes it
- If a language is described by a regular expression, then it is regular
- Every multitape Turing machine has an equivalent single tape turing machine
- A language is Turing-recognizable iff some multitape Turing machine recognizes it
- A language is Turing-recognizable iff some enumerator enumerates it
- Regular expressions, NFAs, and DFAs are decidable
- Every context-free language is decidable
- The set of real numbers is uncountable
- Some languages are not Turing-recognizable
- A language is decidable iff it is Turing-recognizable and co-Turing recognizable
- Regular languages are undecidable
- If M is a linear bounded automaton (Turing machine where tape head states inside of the input) where $L(M) = \emptyset$, then the machine M is undecidable
- If G is a context-free grammar and $L(G) = \Sigma^*$, then the machine G is undecidable

Mapping Reducability

Mapping a language A onto language B is easily solvable. Since we mapped A onto it, we can use the solver for B to show that A is solvable. Let's say you can check if a string is in B or not. You want to determine whether a string $x \in A$. **Map** x by f, yielding $f(x)$, then check if it is in B. f is a mapping reduction if $(x \in A) \leftrightarrow (f(x) \in B)$.

Comparable Function

A function $f : \Sigma^* \to \Sigma^*$ is a computable function if $\exists$ some TM M that, on every input w, halts with just $f(w)$ on the tape.

Reduction

Language A is **mapping reducable** to language B, written as $A <=_m B$, if there is a computable function $f : \Sigma^* \to \Sigma^*$, where $\forall w, w \in A \leftrightarrow f(w) \in B$. The function f is called the **reduction** of A to B.

Turing Reducability

Turing reductions can **NOT** be used to show that a problem is $NP-$complete.

Oracle

A device for language B that tells whether any string w is a member of B. It doesn't matter how it does this ("magic" is valid); can be applied to languages that aren't decidable

Oracle Turing Machine

A Turing Machine that has the added capability of querying an oracle, denoted M^B

Turing Reducible

Language A is **Turing reducible** to language B, written $A <=_T B$, if A is decidable relative to B. Turing reductions are stronger than mapping reductions, in the sense that they are more flexible. However, this flexibility comes at a cost – for example, one can not use a Turing reduction to show that a problem is **NP-hard**.

Complexity

Time Complexity

For deterministic TMs, the time complexity $f(n)$ is the maximum number of steps it takes on an input of length n.

Big-O

The asymptotic upper bound of a function – formally $f(n) = O(g(n)) \to g(n)$ is the upper bound. Big-O can appear in the exponent and behaves similarly, as in $f(n) = 2^{O(n)}$, which is like saying that the function is bound by some power of 2 (e.g. $((2^c)^n)$). $\exists k : f(x) <= k \times g(x)$. If $\lim_{x\to\infty} |\frac{f(x)}{g(x)}| < \infty$, then $f(x) = O(g(x))$.

Polynomial Bounds

n^c, where $c > 0$

Exponential Bounds

c^n

Small-O

Similar to Big-O, but a strict upper bound ("it WILL take less time than this"). $\forall k : f(x) < k \times g(x)$. If $lim_{x\to\infty} |\frac{f(x)}{g(x)}| = 0, f(x) = o(g(x))$.
$TIME(t(n))$ is the collection of all languages that are decidable by an $O(t(n))$ TM.

P, NP

P

The class of problems that are solvable in a polynomial time regardless of computation model ($O(n^{O(1)})$. If the runtime is polynomial, then the size of input, output and sapce must also be polynomial with the lenght of the input.
P is not closed under projection, e.g. the verification of $HAMPATH \in P$, but $HAMPATH$ itself (the projection of $HAMPATH$ verification) is in NP.

NP

P is a subset of NP, which is the class of languages that have a polynomial-time verifier.

P vs NP

P is the class of languages for which membership can be *decided* quickly. NP is the class of languages for which membership can be *verified* quickly.
$NTIME(t(n))$ is a collection of languages decided by an $O(t(n))$ time non-deterministic Turing machine. NP is the union of all languages in $NTIME(n^k)$.

TQBF

True Quantified Boolean Formula – a fully qualified formula. Written as $\Phi \forall x \exists y[(x \vee y) \wedge (\neg x \vee \neg y)]$. TODO: More from Wikipedia

FORMULA-GAME

A game with two players A and E. Each player selects a value of a variable quantified either the $\forall$s if the player is A, or $\exists$s if player E, based on the order of the quantifiers. In the end, if the formula is $TRUE$ player E wins, else player A wins. Consider this as Peggy vs. Victor. Peggy takes existential qualifiers, Victor takes universal ones. They don't have to alternate turns – the order is specified by the order of quantifiers in the expressions.

Problems

SAT

TODO

HAMPATH

A directed graph in which all vertices can be hit just once. Polynomially verifiable, but is not solvable in polynomial time.

UNHAMPATH

An undirected graph in which all vertices can be hit just once. Polynomially verifiable, but not solvable in polynomial time.

CLIQUE

A clique is a subgraph wherein every two nodes are conected by an edge; a $k-$clique is a clique that contains k nodes.

MAXCLIQUE

TODO
MAXCLIQUE is $NP-$hard.

Decision Version

TODO
Is there a k-clique in a given graph G?
NP-Complete

SUBSET-SUM

A problem concerning integer arithmetic, where we have numbers $x_1, \ldots, x_k$ and a target number t. We want to determine whether the collection contains a subcollection that adds up to t. This problem is $NP-$complete.

3SAT

A special instance of the satisfiability problem, written in **3CNF**-form:
$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6) \ldots$.
TODO

VERTEX-COVER

For an undirected graph G, a vertex cover is a subset of nodes where every edge of G touches one of those nodes.

3COL

Graph 3-colorability. NP-complete.
TODO: More

TODO: MORE

NP-Completeness

There are certain problems $\in NP$ whose complexity is related to the entire class of NP, so if one polynomial time algorithm is found, then all of NP is solvable in polynomial time.

Formal Definition

A language B is NP-complete if both $B \in NP$ and every A in NP is polynomial-time reducible to B. **The second condition indicates that B is $NP-$hard**

Cook-Levin

$SAT \in PiffP = NP - SAT$ is $NP-$complete.

Polynomial-Time Reduction

Language A is **polynomial time mapping reducible** to language B, written $a <=_p B$, if there is a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ where $\forall w, w \in A \leftrightarrow f(w) \in B$. The function f is called the **polynomial time reduction** of A to B.

Space Complexity

Whereas in the last section we were dealing with time, now we consider space as the number of cells the TM uses.
TODO: FIGURE 8.7

SPACE(f(n))

A language decided by an $O(f(n))$-space deterministic TM.

NSPACE(f(n))

A language decided by an $O(f(n))$-space nondeterministic TM

PSPACE

The class of languages that are decidable in polynomial space on a deterministic TM. $PSPACE = NPSPACE$ **via Savitch's Theorem.**

PSPACE Completeness

A language B is PSPACE-complete if B is in PSPACE and $\forall A \in PSPACE$, A is polynomial time reducible to B. If only the second is satisfied, then it is $PSPACE - hard$

Approximation Algorithms

While we usually look for a best **optimal** solution, an approximately optimal solution will usually suffice.
Approximation algorithms are designed to find this.

2-approx for minVC

Let G be a graph with the set of edges E.
C = $\emptyset$ (vertex cover approximate graph).
While there is some edge $e = \{u, v\} \in E$,
$C \leftarrow C \cup \{u, v\}$
$G \leftarrow G - \{u, v\}$
Let C^* be the optimal (minimal) vertex cover. Then $C^* <= C <= 2C^*$.
The first indented step above adds vertices u and v to graph C, and the second removes u, v and all adjacent edges from G.

Zero Knowledge Proofs

A zero knowledge proof is an interaction between Peggy (prover) and Victor (verifier) where P tries to convince V that a given statement is true without conveying any information besides the truth value of the statement.
Such a proof must have:

- **Completeness** - if P can solve the problem, P can convince V to accept with a high probability.
- **Soundness** - if P cannot solve the problem there is a very low probabiliyt of P convincing V to accept
- **Zero knowledge transfer** - The prover does not give any new information to the verifier

Such proofs can be simulated, but cannot be relayed to outsiders. They also are not deterministic and are probabilistic proofs, not mathematical.

ZKP for Graph Non-Isomorphism

TODO: Define isomorphism
P is trying to convince V that graphs G_1, G_2 are not isomorphic.
On input $G_1 = (V, E_1), G_2 = (V, E_2)$

- V picks random $b \in \{1, 2\}$ and permutation $\pi : V \to V$; sends $G = \pi(G_b)$ to P
- P finds $a \in \{1, 2\}$ such that G_a and G are isomorphic; sends a to V.
- V checks that $a == b$, and accepts if so

If G_1, G_2 are not isomorphic, P can always determine which one V started from, so V accepts with probability 1.
If G_1, G_2 are isomorphic, P cannot determine which one V started from, so V accepts with probability w^{-k}, with k being the number of repetitions.
Because V already knows the answer, no transfer of knowledge occurs.
TODO: What happens if someone behaves predictably instead of randomly?

Color-Blindness

Imagine your friend is color-blind. You have two balls: one red, one green, but otherwise identical. You want to prove to him that they are differently colored, without having him learn which is red and which is green.
Give the two balls to him so one is in each hand. Don't tell him which is which. Have him put both hands behind his back. He will either switch the balls between his hands or leave them, with $P = 0.5$ of either event. He brings them in front of him, showing them. You now have to "guess" whether or not they were switched. By identifying whether or not the balls were switched, using the colors, you can prove that your friend is colorblind after a number of repetitions.

Graph-3-Colorability

The public input is a graph $G(V, E)$ of n vertices and m edges, with $m <= n^2$. P gets as private input a function $c : V \to \{R, G, B\}$ such that $\forall (u, v) \in E, c(u) \neq c(v)$.
P chooses a random 1-to-1 function $\Gamma : \{R, G, B\} \to \{1, 2, 3\}$. P defines $c' : V \to \{1, 2, 3\}$ to be such that $\forall v \in V, c'(v) = \Gamma(c(v))$. P computes $y_1, \ldots, y_n$ such that y_i is a commitment to $c'(v_i)$, where v_i is the i^{th} vertex. P then sends $y_1, \ldots, y_n$ to V.
V chooses a random edge $(v_i, v_j) \leftarrow_R E$ and sends (v_i, v_j) to P.
P sends $r_i, r_j \in \{0, 1\}^n$ and $x_i, x_j \in \{1, 2, 3\}$ such that $y_i = C(x_i, r_i); y_j = C(x_j, r_j)$. These two are considered the openings.
V accepts iff the openings are valid, $x_i, x_j \in \{1, 2, 3\}$, and $x_i \neq x_j$.
To show soundness, show that if G is not 3−colorable, then V will reject with probability of at least $1 - \frac{1}{m}$, where m is the number of edges.

Sudoku

TODO

Theorems

TODO

NP-Hard Problems

TODO: Enumerate...
CLIQUE, IS, VS

Definitions

Soundness: A resolution is sound if it never declares satisfiable formulas unsatisfiable ("no" means no).
Completeness: A resolution is complete if all unsatisfiable formulas are declared unsatisfiable.
TODO: List of reductions!

Homework Problems

HW6

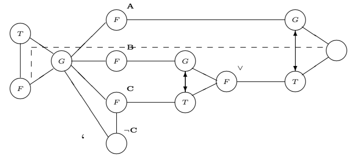
Show how to compute $a^b mod p$, where p is prime, in polynomial time in the length of the input (a, b, p)
A straightforward approach is to multiply a by itself b times, for a total of $b - 1$ multiplications, then take the result mod p. This fails on two counts: The number of multiplications is approximately b, which is exponential in the *length* of b. And, no matter how we ended up with a^b, the length of a^b is $b \cdot |a|$, which is too long to handle (read, write, etc.)
The solution is to do repeated squaring and reduce mod p as we go. As for the hint, I meant to write "first try b a power of 2." If $b = 32$, then a^b is $((((a^2)^2)^2)^2)^2$. This involves just 5 multiplications. We need to reduce mod p as we go, so we get $(\cdots(a^2 \bmod p)^2 \bmod p \cdots)$. We end up doing the transformation $x \to x^2 \bmod p$ 5 times, and each transformation takes time polynomial in the length of the operands and produces output that is of length at most $|p|$. In general, if $b = 2^\beta$, we perform the transformation $\beta = |b|$ times.
Now consider general b. Henceforth, all arithmetic is considered to be modulo p and we reduce modulo p where ever possible; we focus on the repeated squaring. First compute $a, a^2, a^4, a^8, a^{16}, \ldots$ by repeated squaring, as above. Then multiply together a^{2^k} if $1 \cdot 2^k$ appears in the binary representation of b. As an example, if $b = 13 = 1101_2$, then we compute $a^8 \cdot a^4 \cdot a$. The number of a^{2^k} that we compute is $O(\log b)$, as desired, and the number of multiplications we need to combine the factors of a^{2^k} is also $O(\log(b))$.
One could also state this inductively in b. As a base case, $a^0 = 1$. If $b > 0$ and even, then $a^b = \left(a^{\frac{b}{2}}\right)^2$. If $b > 0$ and odd, then $a^b = a \cdot a^{b-1}$. To analyze this, let $M(b)$ denote the number of multiplications needed to form a^b. We have

$$M(b) = \begin{cases} 1 + M(b-1); & b > 0 \text{ and } b \text{ odd} \\ 1 + M(b/2); & b > 0 \text{ and } b \text{ even} \\ 0; & b = 0 \end{cases}$$

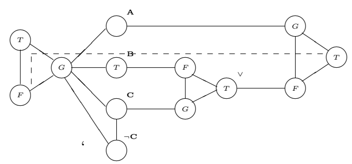
Note that, if b is odd, we are not reducing b by much immediately, though b is reduced on the *next* step. That is, if b is odd and $b > 1$, we get $M(b) = 1 + M(b-1) = 2 + M((b-1)/2) \leq 2 + M(b/2)$. Also, if b is even, we get $M(b) = 1 + M(b/2) \leq 2 + M(b/2)$. Here we should be able to see that $M(b) \leq 2 \lg b$, but we proceed formally a little further. We have $M(1) \leq 2$ and let's avoid $b = 0$ to avoid taking a log of zero. Then we can solve exactly. The inductive hypothesis is that if $2^{k-1} < b \leq 2^k$, then $M(b) \leq 2 + k$. True if $k = 0$ (and $b = 1$).
If it's true for all $b \leq 2^k$ and we have $2^k < b \leq 2^{k+1}$, then

$$M(b) \leq 2 + M(b/2) \leq 2 + (2 + \lg(b/2)) \leq 2 + (k+$$

The instructions accidentally asked you to consider p a power of 2, say, $p = 2^\pi$. Taking x mod p amounts to taking the π least significant bits of x. This can be done in time linear in $|x|$ and is straightforward. For general p, you can do a long division with remainder of x divided by p. Assuming for simplicity that $|x| = |p| = n$, there will be $O(n^2)$ symbols written down in the long division and each symbol will take $O(1)$ time to compute.
Give a reduction from $3SAT$ to $G3C$ (graph 3-colorability), thereby showing that $G3C$ is $NP-$hard
Consider the following graph:



If A, B, C are all colored F, then all the other indicated colorings are forced. The dotted edge shows that there is no possible coloring for the rightmost vertex. On the other hand, if at least one of A, B, C is colored T, then a 3-coloring is possible:



In general, the reduction is as follows. Make a pair of vertices for each variable labeled with the variable and its negation, and form a triangle with G. Each clause results in 6 more nodes and 11 more edges, as shown. It is straightforward to build the graph from a 3CNF Φ by making a few triangles and extra edges. Now suppose Φ has a satisfying argument v. Color all the literal vertices T or F according to v. In each clause gadget, pick one literal colored T and color its neighbor F. Color the rest of the clause gadget as indicated above. Thus, if Φ is satisfiable, then the graph has a 3-coloring.
Conversely, suppose we have a 3-coloring Γ for the graph. We can read a truth assignment from the coloring of the literal vertices. This is sensible because exactly one of $\{x, \neg x\}$ is colored F and the other is T.

HW7

Sudoku to adhocSAT

TODO

Sudoku to 3SAT

TODO

Sudoku to Graph Coloring

TODO

Farmer, Fox, Goat and Cabbage

TODO

HW8

Specializing TQFB

In class, we showed how to reduce a generic PSPACE computation to an instance of TQBF in which all the quantifiers are in front, but the matrix (the part after the quantifiers) is arbitrary. We also showed that the restriction (special case) of TQBF in which the matrix is a 3-CNF reduces to Generalized Geography. To show that Generalized Geography is PSPACE-hard, we need to close this gap:
Show that the restriction (special case) of TQBF in which the matrix is a 3-CNF is PSPACE-hard, by reducing a general TQBF instance to the special case.
Recall that our generic reduction of PSPACE to adhocTQBF went something like this. Given a generic PSPACE machine M, we want to define a formula $\phi(c_1, c_2, t)$ that is true iff there is a path of exactly t steps from M-configuration c_1 to c_2, where we may assume that t is a power of 2. We gave a "middle-out" recursive definition:

$$\phi(c_1, c_2, t) = \begin{cases} \exists m \forall c_3 \forall c_4\ (c_3, c_4) = (c_1, m) \vee (c_3, c_4) = (m, c_2) \rightarrow \phi(c_3, c_4, t/2), & t > 1 \\ c_2 \text{ equals } c_1 \text{ except for small changes}, & t = 1. \end{cases}$$

Note that, in the recursive constuction of $\phi(c_3, c_4, t/2)$, at first there will be quantifiers embedded:

$$\phi(c_1, c_2, t) = \begin{cases} \exists m \forall c_3 \forall c_4\ (c_3, c_4) = (c_1, m) \vee (c_3, c_4) = (m, c_2) \rightarrow [\exists m_1 \forall c_5 \forall c_6 \ldots], & t > 1 \\ c_2 \text{ equals } c_1 \text{ except for small changes}, & t = 1, \end{cases}$$

but, since m_1, c_5, and c_6 don't involve m, c_3, or c_4, the quantifiers can be passed out:

$$\phi(c_1, c_2, t) = \begin{cases} \exists m \forall c_3 \forall c_4\ \exists m_1 \forall c_5 \forall c_6\ (c_3, c_4) = (c_1, m) \vee (c_3, c_4) = (m, c_2) \rightarrow [\ldots], & t > 1 \\ c_2 \text{ equals } c_1 \text{ except for small changes}, & t = 1, \end{cases}$$

That is, $\alpha \rightarrow \exists x \forall y\ \beta$ is the same thing as $\exists x \forall y\ \beta$ in this case.
So we are left with a formula with all the quantifiers in front and some quantifier-free matrix following that.
The details of the matrix depend on the precise model of computation, which we are *not* specifying. In any case, we need to express equality of configurations, (as in $(c_3, c_4) = (c_1, m)$, above) and *near* equality of configurations (as in the base case, when we express that c_2 is like c_1 except for a small number of changes). For our purposes, we assume that the matrix is some boolean formula of size polynomial in n, the size of input to our generic PSPACE computation. Our goal is to convert this to a 3-CNF, possibly with additional quantifiers in front.
That is, we start with a formula like $\alpha = \exists w \forall x \psi(w, x)$, where ψ is an arbitrary boolean formula, and we want to get a new formula $\beta = \exists w \forall x \exists y \forall z\ v(w, x, y, z)$, such that

- v is a 3-CNF
- *beta* is constructed from α in time polynomial in n (in particular, v is of polynomial size), and
- β is true iff α is true.

To do this, we regard ψ as a circuit and put new boolean variables on the wires of ψ. These are all existentially quantified. We then express that

- Gate-by-gate and input-by-input, ψ works correctly
- The input wires of ψ agree with the original variables $w, x, \ldots$
- ψ's output is true.

This is similar to what we did to convert a general SAT instance into a 3SAT instance.

Traveling Salesperson

The Traveling Salesperson Problem, MinTSP, is the following optimization problem. Consider a complete undirected graph on n vertices, with a cost on each edge. A *tour* is a cycle that visits each vertex exactly once. The cost of a tour is the sum of the weights over edges in the tour. The goal is to find a tour of minimum cost. (Formally, the input is the number n of vertices and the edge costs, but you can think of the input as the graph with edge weights.)
In MinMetricTSP, we are promised that the edge weights satisfy the triangle inequality. That is, if $C(u, v)$ denotes the cost associated with edge u, v, then we are guaranteed that $C(u, w) \leq C(u, v) + C(v, w)$.
In the Minimum Spanning Tree problem, MST, we are given an edge-weighted undirected connected graph G, as above, and the goal is to find a tree T on G that is induced (edges of T are in G), spans (each vertex of G is in T), and the cost is minimum, where the cost of T is the sum of the costs of the edges in T.
In this homework, we will walk through a 2-approximation for MinMetricTSP, using an algorithm for Minimum Spanning Tree.
Polynomial-time algorithms for MST exist. You're welcome to look this up or ask about it on Piazza, but, for this homework, you may simply assume that the algorithms are available.
Consider the following approximation algorithm for MinMetricTSP.

- **Given G that satisfies the triangle inequality, find a minimum spanning tree, T.**
- **Duplicate each edge in T, getting T'.**
- **Find an Euler tour C' of T'. (An Euler tour visits each *edge* exactly once. There is an obvious polynomial-time algorithm for this—try to find one.)**
- **Adjust C' to a tour C (that visits each vertex exactly once), by starting at some vertex s and repeatedly going to the next *unvisited* vertex in C', returning to s in one hop immediately after each vertex has been visited.**

Show:

1. **This approximation algorithm runs in polynomial time (assuming a polynomial-time algorithm for MST).**
2. **The algorithm produces a feasible solution, *i.e.*, a cycle that visits each vertex exactly once.**
3. **The approximation ratio is at most 2, *i.e.*, the cost of the cycle C produced is at most twice the cost of an optimal tour.**
4. **In any optimal tour on the *plane* (i.e., vertices are on the plane and edge costs are given by Euclidean distance), no pair of edges cross.**
5. **Assuming the undirected Hamiltonian Cycle problem is NP-hard, show that there is no 1000-factor approximation to the general MinTSP problem (where we do *not* assume that the triangle inequality holds for edge costs.)**

Note: The factor of 2 for this approximation algorithm for MinMetricTSP can be improved to the factor 3/2, by what is known as Christofides's algorithm. Many variations are studied.

1. This approximation algorithm runs in polynomial time (assuming a polynomial-time algorithm for MST). Duplicating each edge takes time linear in the number of edges. To find an Euler tour of a doubled tree, trace around the tree. The vertices will be visited in the order of a depth-first search of the tree:

   There are faster things to do in practice, but, to show polynomial time for adjusting C' to C, it suffices to trace the Eulerian tour and to try all vertices at every stage and (easily) see whether they are marked as having been visited. Then mark newly visited nodes.
2. The algorithm produces a feasible solution, *i.e.*, a cycle that visits each vertex exactly once.
   Solution.
   This is true by construction. The Eulerian tour T visits every vertex and so T', C', and C visit every vertex. (Note that C visits every node because we only skip a vertex v in C' if v had already appeared in C.) And C skips already-visited vertices, so C visits each vertex exactly once.
3. The approximation ratio is at most 2, *i.e.*, the cost of the cycle C produced is at most twice the cost of an optimal tour.
   Solution.
   If we take an optimal TSP tour and remove one edge, we get a spanning tree. So the cost of the tour must be at least as great as the cost of the best possible spanning tree, T. The cost of T' is at most twice the cost of T; the cost of T' is exactly the cost of C'. Finally, the cost of C is at most the cost of C'. That is,
   $$c(C) \leq c(C') = c(T') \leq 2c(T) \leq 2c(C_*),$$
   where C_* is the optimal TSP tour and $c(\cdot)$ is the cost.
4. In any optimal tour on the *plane* (i.e., vertices are on the plane and edge costs are given by Euclidean distance), no pair of edges cross.
   Solution.
   If a tentative solution had crossed edges, we can uncross the edges and get a better tour, as follows:
5. Assuming the undirected Hamiltonian Cycle problem is NP-hard, show that there is no 1000-factor approximation to the general MinTSP problem (where we do *not* assume that the triangle inequality holds for edge costs.)
   Solution.
   Given an instance $G = (V, E)$ of the undirected Hamiltonian Cycle problem, we reduce to TSP as follows. Let n be the number of vertices in G. Consider the complete graph K on n vertices (the same vertices as G), with edge weight 0 on (u, v) if $(u, v) \in E$ and edge weight 1 if $(u, v) \notin E$. Then there is a TSP tour with total cost zero iff there is a Hamiltonian Cycle on the original graph. If there is no TSP with total cost 0, then the best TSP has cost at least 1, which is at least 1000 times optimal. It follows that, if we can approximate the cost of the best TSP tour to within the factor 1000, we could distinguish a Hamiltonian graph from a non-Hamiltonian graph. Finally, we note that the reduction clearly takes polynomial time.
   Note: Instead of edge weights 0 and 1, one could use edge weights 1 and $1000n$. Then a Hamiltonian graph leads to a TSP tour of n edges, each weight 1, for a total cost of n, whereas a non-Hamiltonian graph leads to a TSP of total cost at least $1000n$, which is bigger by the factor 1000.

3-State Busy Beaver State Table

Tape Symbol	Current State A			Current State B			Current State C		
	Write Symbol	Move Tape	Next State	Write Symbol	Move Tape	Next State	Write Symbol	Move Tape	Next State
0	1	R	B	1	L	A	1	L	B
1	1	L	C	1	R	B	1	R	**HALT**