COT 4210 Reference Sheet

DFA Formal Description

Σ = input alphabet
Q = set of states
$\delta = Q \times \Sigma \rightarrow Q$, transition function ($\delta(q_i, a) = q_j$, is defined for each $q_i \in Q$ and each $a \in \Sigma$)
q_0 = start state
F = set of accept/final states (a subset of Q)

Regular Operations

If A and B are regular, then the following languages determine by these operations are also regular.

Union: $A \cup B = \{ z \mid x \in A \text{ or } x \in B \}$
Concatenation: $A \circ B = \{ xy \mid x \in A \text{ and } y \in B \}$
Star: $A^* = \{ x_1x_2\ldots x_k \mid k \geq 0 \text{ and each } x_i \in A \}$

NFA Formal Description

Same as DFA except for the transition "function", which has a set of outputs instead of a single state as output. In the text, it's denoted as follows:

$\delta: Q \times \Sigma_\varepsilon \rightarrow P(Q)$, where P(Q) represents the set of subsets of the set Q.

NFA to DFA conversion

If the NFA has states 1, 2, 3, … n, then create a DFA with 2^n states where each state represents a distinct subset of states from the NFA. For example, if the states of the NFA are 1, 2 and 3, then we can label the DFA states as 000, 001, 010, 011, 100, 101, 110, and 111. Each of the three bits represents whether or not a particular NFA state is included in the subset. For example, 101 represents the subset of {1, 3} from the NFA.

The start state of the DFA will correspond to the set of states that are reachable without reading any input into the NFA. (For example, if there are epsilon transitions from 0 to 1 and from 1 to 2, but none from 2, then the start state in the DFA will be the one corresponds to the subset {0,1,2}.)

For each transition, do as follows: If you are starting from a state {1,3,4} and reading an a, for example, calculate all the states you can reach in the NFA if you were in either state 1, 3 or 4 and read in a single a. (Account for epsilon transitions as well. This set of reachable states, denotes another state in the DFA. This is the output state when you read in an a from {1, 3, 4}. Do this for all states and input letters in the DFA.

The set of accept states are all states that include at least one accept state from the NFA. For example, if the NFA has states 1, 2, 3 and 4, and states 2 and 3 are accept states, then {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4}, {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4} and {1, 2, 3, 4} are all accept states in the DFA.

Proof of Closure for Regular Languages using NFAs

To create an NFA for $A \cup B$, where A and B are regular languages, create a new NFA with a new start state and connect it with two epsilon transitions to the two start states of the NFAs that accept A and B, respectively.

To create an NFA for $A \circ B$, put the NFA for A to the left of the NFA for B. Take each final state in the NFA for A and connect it with an epsilon transition to the start state for the NFA for B. Erase all final states in the NFA for A.

To create an NFA for A^*, Take the NFA that accepts A and add a new start state. Make his start state an accept state. Then, add an epsilon transition from this start state to the original start state for the NFA for A.

Regular Expression Definition

1) a for some letter in the alphabet Σ 2) ε 3) $\emptyset$
4) $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5) $R_1 \circ R_2$, where R_1 and R_2 are regular expressions
6) R^*, where R is a regular expression

Equivalence with NFAs

Simply use all the constructions given in the section for closure with NFAs to create an NFA that accepts the same language described by a regular expression. This shows that any language described by an RE can be expressed with a NFA.

Now, we must show the other side, that any language expressible by a DFA can also be expressed through a regular expression. We do this as follows:

1) To our DFA, add a start state and a new accept state. Create epsilon transitions from the new start state to the old one and from all the old final states to the new one. Make the old accept states not accept states. This is our GNFA.

2) Rip out each state, except for the start and accept state, one by one. For each rip go through the following steps:

For each pair of states i and j, NOT being ripped out (note that i and j can be the same), Consider the following four regular expression labels:

R_1: $q_i \rightarrow q_{rip}$ R_2: $q_{rip} \rightarrow q_{rip}$ R_3: $q_{rip} \rightarrow q_j$ R_4: $q_i \rightarrow q_j$

Now, replace these four labels with a single label for the transition $q_i \rightarrow q_j$ as follows: $(R_1)(R_2)^*(R_3) \cup R_4$

In the end, are left with two states with a single transition and a regular expression label. This label is a regular expression describing the same exact language as the original DFA.

Pumping Lemma

For all regular languages, A, there exists a positive integer p, known as the pumping length, such that all strings, s, in A of length p (or greater) satisfy the following criteria:

There exists a way to split up s into three components x, y and z, where s = xyz such that

1) for each $i \geq 0$, $xy^iz \in A$
2) $|y| > 0$
3) $|xy| \leq p$

Context-Free Grammar

Formal Defintion:

V is a set of finite variables.
Σ is a set of finite terminals, which are disjoint from V.
R is a set of finite rules. Each rule maps a single variable to a string containing variables and/or terminals.
S is the start symbol.

The language of a grammar is simply the set of strings (of terminals only) that can be derived from the start symbol, through 1 or more applications of the rules.

Chomsky Normal Form

Every rule is of one of the following two forms:

$A \rightarrow BC$ (two variables)
$A \rightarrow a$ (one terminal)

The transition $S \rightarrow \varepsilon$ is also allowed, but no other epsilon rules are allowed.

Context-Free Grammar to Chomsky Normal Form

1) Add a new start state, S_0, with the single rule $S_0 \rightarrow S$, where S was the original start state.

2) Remove epsilon rules. For each rule of the form $A \rightarrow \varepsilon$, look through each right side production. In any string that contains an A, consider what it might look like without the A, and add this in that rule. For example, if there was a rule:

$B \rightarrow ABbA$ with the first A missing, the production would be BbA. With the second rule missing it would be ABb. With both A's missing, it would be Bb. Simply add all three rules:

$B \rightarrow ABbA \mid BbA \mid ABb \mid Bb$

Do this for each production for which the variable that goes to epsilon appears. After adding all of these rules, you can remove the epsilon rule.

What might occur is that the removal of one epsilon rule might create another one. For example, if we had $B \rightarrow A$ and $A \rightarrow \varepsilon$ to remove, then we'd create $B \rightarrow \varepsilon$ as we remove $A \rightarrow \varepsilon$. Then,

we'd have to remove the $B \rightarrow \varepsilon$ rule. We continue in this fashion until no epsilon rules are left, except for possibly, $S \rightarrow \varepsilon$.

3) Remove unit rules. For each rule of the form $A \rightarrow B$, remove it by adding in all possible productions for B into the list for A. For example, if we have:

$A \rightarrow B \mid a \mid bC \mid BC$ $\qquad$ $B \rightarrow bb \mid CB \mid b \mid AbCa$

Then we would add ALL of the productions for B in the line for A, and then remove B:

$A \rightarrow a \mid bC \mid BC \mid bb \mid CB \mid b \mid AbCa$ $\qquad$ $B \rightarrow bb \mid CB \mid b \mid AbCa$

4) Convert all rules into proper form
- a) remove terminals from right side strings of length 2 or more
- b) rewrite rules of length greater than 2.

If we use the example above, we can achieve (a), but creating new variables for a and b as follows:

$D \rightarrow a$ $\qquad$ $E \rightarrow b$
$A \rightarrow a \mid EC \mid BC \mid EE \mid CB \mid b \mid AECD$
$B \rightarrow EE \mid CB \mid b \mid AECD$

Now, to remove rules of length greater than 2, create all the necessary "bridge" variables. Namely, for a string such as AECD, create a new variable F so that this string can be rewritten as AF, where F will expand out to ECD. But, since F can only be two letters, let F simply go to EG, where G expands out to CD. Here is this portion of the example converted:

$D \rightarrow a$ $\qquad$ $E \rightarrow b$ $\qquad$ $A \rightarrow a \mid EC \mid BC \mid EE \mid CB \mid b \mid AF$ $\qquad$ $B \rightarrow EE \mid CB \mid b \mid AF$
$F \rightarrow EG$ $\qquad$ $G \rightarrow CD$

<u>Pushdown Automata</u>
Formal Definition

Q = set of states $\qquad$ Σ = input alphabet $\qquad$ Γ = stack alphabet $\qquad$ q_0 = start state
$\delta = Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$, transition function $\qquad$ F = set of accept/final states (a subset of Q)

A string is accepted by a PDA if and only if there exists a path to read the string into the PDA so that it's in an accept state after reading in the whole string. The state of the stack does not affect whether or not a string is accepted after the string has been processed.

<u>Pumping Lemma for CFG</u>
If A is a context-free language, then there is a number p, called the pumping length, where, if s is any string in A of length at least p, then s may be divided into five pieces s = uvxyz satisfying the conditions:

1) for each $i \geq 0$, $uv^ixy^iz \in A$ $\qquad$ 2) $|vy| > 0$ $\qquad$ 3) $|vxy| \leq p$

Turing Machines

Formal Definition

Q = set of states Σ = input alphabet Γ = tape alphabet, with blank q_0 = start state
$\delta = Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, q_{accept} , and q_{reject}

Input placed on tape. Computation follows rules. If machine ever hits q_{accept} , then string is in the language. A string not in the language may either hit q_{reject} or loop.

A Turing machine is a decider if it correctly halts on all inputs and determines whether or not the input is in the desired language.

A Turing machine is a recognizer if it correctly halts on all inputs IN THE LANGUAGE and doesn't erroneously accept any inputs not in the language.

A language is called Turing decidable (recursive) if there exists a Turing machine that decides it.
A language is called Turing recognizable (recursively enumerable) if there exists a Turing machine that recognizes it.

Turing Machine Variants
Multitape Machine with alternative transition function:

$\delta = Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k$

This is equivalent to a regular TM. To simulate a multitape on a regular TM, use the "dot" trick and hash marks to separate out each of the separate tapes being simulated. The dot keeps track of where each tape head is and you sweep left to right to perform one move. Whatever the multitape does in one step, the regular does in O(f(n)) steps, where f(n) is the time taken by the multitape machine.

Nondeterminstic Turing Machine transition function:

$\delta = Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$

This is also equivalent to a regular TM. We simulate this on a multitape. One tape is the input tape, another the simulation tape and the third is the "decision counter", telling us which choices to make when we have non-determininstic branches. The decision counter goes in lexicographical order from shortest to longest. Iteratively try each possibility of the decision counter by copying the input to the simulation tape and just running those steps. If any of these accept, accept.

PROBLEM TIPS: To show that a machine variant is equivalent to a standard TM, you have to do two directions of the proof: 1) show that any standard TM can be transformed into an equivalent machine of the new model, 2) show that any machine of the new model can be transformed into an equivalent standard TM. Key in both directions is the manipulation of the transition function, often times requiring new states and extra steps to substitute for a single step.

An enumerator is a machine that takes no input and prints out strings. An enumerator of a language is guaranteed to print out all strings in the language eventually and guaranteed not to print out any string not in the language. For any language that we can create an enumerator, we can create a Turing machine that recognizes the language and vice versa.

Decidable Langauges
Here is a list from section 4.1:

A_{DFA}, A_{NFA}, A_{REX}, E_{DFA}, EQ_{DFA}, A_{CFG}, E_{CFG}

A means "this machine accepts this string"
E means "this machine accepts no strings"
EQ means "these two machines accept the same set of strings"

We prove a language is decidable by creating an algorithm to decide membership in the language. We must prove the correctness of the algorithm and we must prove it always halts. Often times, either algorithms from COT 4210 or COP3503 are useful in these proofs.

Countability
We prove that sets are countable by showing a one-to-one mapping from the natural numbers to the set. We showed this with fractions and ordered pairs of integers by weaving the diagonal. If we go down one row or column, we never make it to the next row or column, which is why we must do successive diagonals, all of which end.

We prove that the real numbers and the number of subsets of N are uncountable by using diagonalization. We assume the opposite, that the sets are countable. We take the "given list" of complete items in the set, and use the "flip item i in location i" trick to create a new item not in the given list, concluding the contradiction.

Undecidable Problems

A_{TM} = { <A, w> | A is a TM that accepts w }

We prove that this language is undecidable via contradiction, assuming that a machine H exists that decides A_{TM}. Based on the design of H, we create another machine D that "does the opposite of what D should do on its own input". Unfortunately, this creates a contradiction between what D says will happen on its own input vs what actually happens on its own input when run. Another proof of this shows how the number of TMs is countable but the number of languages isn't, so we run into a problem trying to map each TM to a language. (There are languages that have no corresponding TMs.)

Reducibility

Other undecidable problems: $HALT_{TM}$, E_{TM}, $REGULAR_{TM}$, EQ_{TM}

In general, to prove a new language is undecidable, we do the following:

1) Assume the language, L, is decidable by some TM H.

2) Utilize H as a subroutine in TM D to decide another known undecidable language L'.
3) Conclude that since we know L' is undecidable, the initial assumption in the proof must have been incorrect, thus L is not decidable and is undecidable, as desired.

The key to these proofs is treating that TM H as a black box and figuring out creatively, how you can use H to solve membership in another known undecidable problem L'. We often pick L' = A_{TM}, but other choices may work better for some problems. It's critical to get your input parameters to both H and D correct in terms of number of parameters and type.

Proof of A_{LBA} decidability hinges on a finite number of possible configurations. This is infinite for regular TMs.

Both E_{LBA} and ALL_{CFG} are undecidable via computation histories. For the first proof, we assume we have a decider for E_{LBA}. We then use that decider to decide A_{TM} as follows. Our input is a TM M and a string w. We use this to create an LBA that accepts all valid computation histories of M on w. (There is either 0 or 1 of these.) When we find out that there are no such valid computation histories we can say that M doesn't accept w. Alternatively, if we find out there is at least one, we know M does accept w.

PCP Proof
Problem is given a set of tiles, can we arrange some sequence of the tiles (with repeats allowed) so the top and bottom read the same.

We decide A_{TM} using a PCP solver as follows:

First tile: top = #, bottom = #$q_0w_1 \ldots w_n$# (so bottom is ahead by one step)

For each right move, put in the tile: top = qa, bottom = br, where q is the input state, a the input char, b is the output char and r is the output state.

For each left move, put in the tile: top = cqa, bottom = rcb, where input state is q, input char a, b the output char and r the output state. Place all possible chars for c.

Place each char top = a, bottom = a for each input char (not for the #)

Place dominoes #/# and #/blank#, these are to add blanks.

Place dominoes [aq_{accept}/q_{accept}] and [$q_{accept}a/q_{accept}$] to help swallow up characters from the bottom so that the top can start catching up, one char per "turn".

Mapping Reducibility
A language A is mapping reducible to a language B iff there exists a computable function f such that for all inputs w, iff $w \in A$, then $f(w) \in B$.

If A is mapping reducible to B and A is undeciable, B is undecidable.
If A is mapping reducible to B and B is decidable, then A is decidable.
If A is mapping reducible to B and B is Turing recognizable, A is Turing recognizable.

Note: If L is not decidable, but is Turing recognizable, then the complement of L is co-Turing recognizable.

Class P

A language is in TIME(O(f(n)) if a TM exists that correctly decides membership on any input of size n in O(f(n)) steps. The class P is simply the set of languages for which there exist a Turing machine correctly decides membership in the language in $O(n^k)$ time for some constant k.

Examples of problems in P: PATH, RELPRIME, A_{CFG}, and many of the problems taught in CS1 and CS2.

We prove a problem is in P by coming up with an algorithm to decide membership in the language. We must prove the algorithm to be (a) correct, and (b) take a polynomial number of steps with respect to the input size.

Class NP

A language is in NTIME(O(f(n)) if a non-deterministic TM exists that correctly decides membership on any input of size n in O(f(n)) steps. This means that **ALL** computation branches have depth at most O(f(n)). The class NP is simply the set of languages for which there exist a non-deterministic Turing machine that correctly decides membership in the language in $O(n^k)$ time for some constant k.

Examples of problems in NP: 3-SAT, CLIQUE, VC, SS (subset sum), HAM-PATH

We prove a language, L, is NP-Complete as follows:

1) Show that L is in NP. We do this by showing that a certificate for a positive answer can be verified in polynomial time of the input.

2) Show than a known NP-Complete problem can be reduced in polynomial time to L.

Proofs of NP-Completeness

3-SAT - the most detailed proof. We encode a Turing machine M's directions and a string w into a large boolean formula that is true iff there exists a computation history of M on w that results in an accept. First part of the boolean formula is forcing each cell of the tableau to be occupied by exactly one valid symbol. This loops through each cell, forcing each one to have at least one thing, but never to have more than one thing. The next part of the formula is forcing the starting configuration on the first row of the tableau with a bunch of ands, for which symbols must be there. The third part of the formula is just saying that q_{accept} must appear somewhere on the tableau. The final portion of the formula is the most complicated, requiring that all 2 x 3 windows adhere to the valid rules of the machine.

CLIQUE: Proof is reduction from 3-SAT. We take a Boolean formula and output a graph and an integer k. For each clause, create three vertices in a group not connected to each other. For each pair of vertices not in the same group, connect them so long as their labels aren't opposing labels.

(Connect x to everything but $\bar{x}$ in the other groups, etc.) Output this graph and the number of groups as k.

VERTEX COVER: Proof is reduction from 3-SAT. We take a boolean formula and output a graph and an integer k. Each clause converts to three vertices connected in a triangle. Each variable converts to two vertices labeled with x and $\bar{x}$ connected by an edge. From each of these "bars" draw edges to the triangles for identical variable labels only. Output this graph and the number of variables plus 2 times the number of clauses as your k.

SUBSET SUM: Proof is reduction from 3-SAT. We take a boolean formula and output a set of numbers and a target. Create 2 rows for each variable and two rows for each clause. The number of columns will be the number of variables plus the number of clauses. For each variable, there are two rows, one for setting that variable true, the other for false. For both rows, set the column with its variable number to 1. For the columns for the clauses, for the first row set the clause to true if the variable appears in the clause. For the second row set the clause to true if the complement of the variable appears in the clause. For the last set of rows, for each clause have two ones each in the corresponding column as slack variables so that each clause can have three things true: 1 from the expression and up to two from the slack variables. The target is a bunch of 1s (# of vars) followed by a bunch of 3s (# of clauses).

3SAT to Independent Set: Input is a boolean formula, output a graph and an integer k. For each clause, output a triangle (3 vertices all connected). Connect each variable in a triangle with the variables in other clauses that oppose it. Output this graph and the number of clauses as k.

Independent Set to Vertex Cover: Use the same graph and make the new k equal to the number of vertices in the old graph minus its k. All the vertices NOT in the independent set form a vertex cover.

Independent Set to CLIQUE: Output the complement graph and k. An independent set is a set of vertices where none are connected, if we "flip our connections" this becomes a clique.

Techniques to prove new problems NP-Complete:

First, pick the problem to map from. 3-SAT is popular, but some items are harder to create from boolean formulas than others. If it's a problem dealing with graphs, see if another graph problem will work. (Note that these last couple reductions above are much easier and shorter than the others.) If a problem has numbers, see if Subset Sum will work, and so forth. There's some creativity involved. Key is to make an input to the new problem that is true if and only if the old input was true for the old problem. If you look at all the old proofs, we have "devices" in the new input to force its answer to go a particular direction, so to speak. The idea of slack variables in the two subset sum proofs is important, as is the idea of connecting variables to themselves or their opposite, and between triangles or other groups.

PSPACE

SPACE(O(f(n)) is the set of languages that can be decided by a regular TM using O(f(n)) tape squares for any input of size n.

Savitch's Theorem: NSPACE(O(f(n)) $\subseteq$ SPACE(O(f^2(n)) where f(n) $\geq$ n.

It follows that PSPACE = SPACE(O(n^k)) for a constant k and NPSPACE = NSPACE(O(n^k)) are equal classes.

We prove Savitch's Theorem using a divide and conquer algorithm that limits the stack space to a polynomial size. For each possible intermediate configuration c_m, we look to see, if in time t/2 we can go from c_1 to c_m and if we can go from c_m to c_2.. This allows us to see if we can move from c_1 to c_2 in t steps. Then we just try to see if we can go from the starting configuration to an accepting one in an exponential number of steps.

TQBF = the set of true quantified boolean formulas. You have a formula with there exists and for all clauses with variables. If the formula is true, it's in the language. This language is PSPACE complete.

Formula Game: Given a Fully Quantified Boolean Formula, player one is the there exists player and player two is the for all player. In order of the quantifiers, the players play, picking their variable. Does there exist a strategy for the there exists player to make the formula true? If so, this is an instance of Formula Game. Due to the lack of alternating players, this problem is precisely TQBF.

Generalized Geography: Given a directed acyclic graph and a starting vertex, player 1 chooses any outgoing edge. Player 2 can respond with any outgoing edge that visits a new vertex. Play alternates until one of the players has no new outgoing edges to take that lead to an unvisited vertex. If player 1 has a winning strategy, then the input graph is in GENERALIZED GEOGRAPHY.