11-711 Algorithms for NLP – Midterm Exam

October 23, 2014

- Before you go on, **write your name at the space provided on the bottom of this page and every page of the exam**.
- There are **8** pages in this exam (including this page). Verify that you have a complete copy.
- Write up your answers following each question in the exam. Adequate space has been provided.
- If you really feel that you you need more space you may continue on the reverse side, but you must clearly mark the page so that we know your answer continues on the reverse side.
- The exam is open book and open notes, and is worth a total of 100 points.
- You will be given 85 minutes to complete the exam. Budget your time accordingly.
- Keep answers short and to-the-point. Concise, direct answers will receive more credit than longer, essay-like answers.
- If you find a question ambiguous, state your assumptions precisely, and proceed.

Question	Points	Score
1	35	
2	35	
3	30	
Total	100	

Student Name: _____

Student Name: _____

Problem 1 – FSAs and Boolean Logic[1] (35 minutes/35 points)

In this problem we will consider the language of boolean expressions over the alphabet $\Sigma = \{p, \neg, \vee, \wedge\}$, where p is the name of a boolean variable, $\neg$ represents the unary negation operator ("not"), $\wedge$ represents the binary conjunction operator ("and"), and $\vee$ represents the binary disjunction operator ("or").

1. **Verifying well-formedness (10 points):** Draw an FSA that accepts the language consisting of all properly formed boolean expressions. Some examples of well formed boolean expressions are $\neg p$, $p \wedge \neg p \vee p$, and $\neg\neg\neg p$. Some examples of malformed expressions are $p \wedge \vee p$ and $\wedge p \vee p$.

   A formal definition of the language of well formed expressions is as follows:

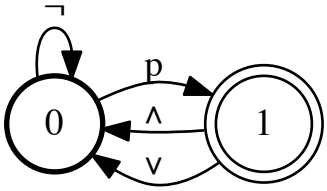
   - p is well formed.
   - if w is a well formed expression, then $\neg w$ is also well formed.
   - If w and v are well formed expressions, then $w \wedge v$ and $w \vee v$ are also well formed.

   Your FSA must contain as few states as possible. You may either accept or reject ϵ, whichever allows your FSA to use fewer states.

[1] Adapted from a problem by Sukhamay Kundu at UC Berkeley

Student Name: _____

SOLUTION: This is possible with only two states. Epsilons are not necessary:



2. **Determining satisfiability (25 points):** Consider the language of *satisfiable* boolean expressions using the same alphabet Σ. An expression $w(p)$ is satisfiable if it evaluates to true (T) for at least one of the cases p = true or p = false. For example $p \vee \neg p$ and $p \wedge \neg\neg p \vee \neg p$ are satisifiable, while $p \wedge \neg p$ and $\neg\neg p \wedge \neg p$ are not satisifable. You should assume that the expression is evaluated following standard order of operations. In particular, $\neg$ has the highest precedence (i.e. is evaluated first), $\vee$ has the lowest precedence (i.e. is evaluated last), and $\wedge$ is in the middle.

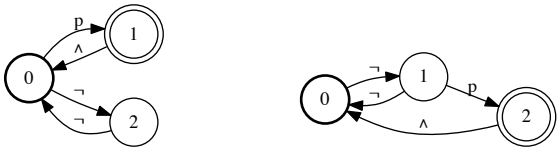
   Is the langauge of well formed, satisifiable boolean expressions regular?

   - If so, construct an FSA that accepts it. You may define the FSA formally or by drawing it, or you may build it up from smaller FSAs using union, intersection, negation, and/or concatenation operations.
   - If not, prove that it is not regular by using the pumping lemma.

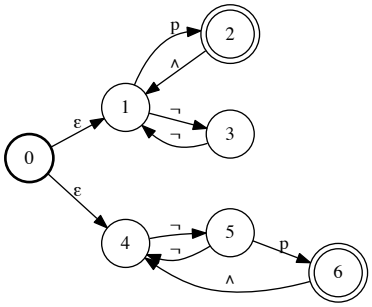
SOLUTION:
Note that since $\vee$ has the lowest precedence, any valid expression has the form $w_1 \vee w_2 \vee w_3 \ldots$, where w_i is a well-formed expression consisting of only p, $\neg$, and $\wedge$. Further note that the whole expression is fulfillable if any one w_i is fulfillable. Thus we may express the solution in regex form as $(q\vee)^* v (\vee q)^*$ where q is a regex matching any valid (non-ϵ) expression, and v is a regex matching a satisfiable expression.
Furthermore, note that v can be broken down as $v_T + v_F$ where v_T is a regex matching expressions satisfied when $p = True$, and v_F is a regex matching expressions satisfied when $p = False$.
Thus the desired FST can be expressed as simple FSA operations (namely concatenation, union, and closure) on three basic FSAs, each of which contains at most three states.

In FSA form we have v_T and v_F:

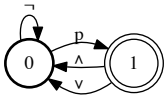
v_T v_F



Then we have $v_T \cup v_F = v$:

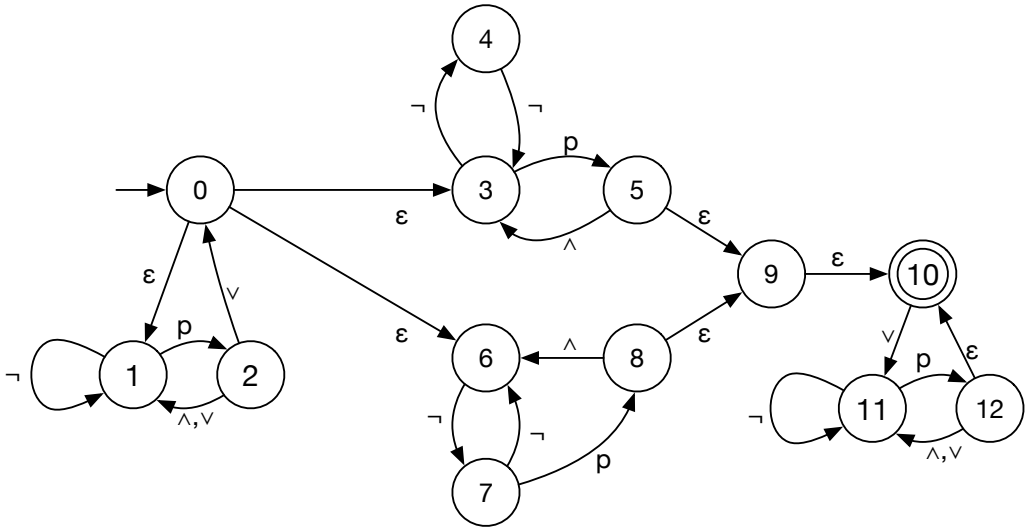


q is the FSA from part 1:



Putting all the pieces together, as per the regex results in this:

Student Name: _____



Student Name: _____

Problem 2 – Parachute Man (20 minutes/35 points)

"Parachute Man" is a 1-player guessing game. You start with an unknown word $\boldsymbol{y}$, randomly chosen from a dictionary $\mathcal{D} \subseteq \Sigma^*$, where $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. Each letter of $\boldsymbol{y}$ is initially obscured by a "?" symbol, so at the beginning of the game, you only know its length. There is a stick figure drawn attached to a parachute by 10 strings.

In each turn, you guess a letter $\sigma \in \Sigma$ that you haven't already guessed. If $\boldsymbol{y}$ contains the letter σ, then each occurence of σ in $\boldsymbol{y}$ is revealed. If not, you erase one string from your parachute. The goal of the game is to correctly guess every letter in $\boldsymbol{y}$ before you lose all of the strings in your parachute.

A game state can be modeled as a triple $(G, M, \boldsymbol{x})$, where $G \subseteq \Sigma$ are the letters you've guessed correctly, $M \subseteq \Sigma$ are the letters you've guessed incorrectly (so $G \cap M = \emptyset$), and $\boldsymbol{x} \in (G \cup \{?\})^n$ represents the partially revealed solution.

Your best strategy to win at Parachute Man is to pick the letters that are mostly likely to be used in the solution, given everything you know about the current game state. One impractically slow algorithm for finding your best guess would be to loop over each word $\boldsymbol{x}$ in the dictionary $\mathcal{D}$, checking that it is compatible with the current game state (same length, does not contain any letters in M, etc.).

1. **FSA for compatible words (10 points):** Assume that you have a FSA A encoding $\mathcal{D}$ (in other words, $L(A) = \mathcal{D}$). How can you construct an FSA B which accepts exactly the set $\mathcal{D}_{(G,M,\boldsymbol{x})}$ of words in $\mathcal{D}$ that are compatible with a game state $(G, M, \boldsymbol{x})$?

   SOLUTION: Let n be the length of x. Build a linear chain FSA with $n+1$ states. Let state 0 be the start state and state n be the lone final state. For each $i \in [1, n]$, if $x_i \neq ?$, add a transition from state $i-1$ to state i with arc label x_i. Otherwise $x_i = ?$, so add an arc from state $i-1$ to state i $\forall \sigma \in \Sigma - G - M$, representing the fact that any letter in σ, other than those that we have already guessed, could fill that slot. This FSA represents all sequences in Σ^* that are compatible with $\mathbf{x}$, regardless of whether they are in $\mathcal{D}$.
   Now intersect this linear chain FSA with A. Since we know (as per HW1) that FSAs are closed under intersection, the result will be an FSA that only accepts words that are both compatible with $\mathbf{x}$ and in $\mathcal{D}$.

2. **Count compatible words (10 points):** Explain how you can use B, the FSA you constructed in the previous answer, along with the PathSum (i.e., Forward) algorithm to count the number of words in $\mathcal{D}_{(G,M,\boldsymbol{x})}$.

   SOLUTION: Each path in B corresponds to a unique word that is consistent with (G, M, x). Thus if we give each arc a weight of 1 and run the PathSum algorithm with the standard path counting semiring (shown below), the result will be the total number of words consistent with the game state.
   $\mathbb{K} = \mathbb{Z}^*$
   $\oplus = +$
   $\otimes = \times$
   $\bar{0} = 0$
   $\bar{1} = 1$

3. **Letter counts (15 points):** Give an algorithm Count(σ) that returns the number of times the letter $\sigma \in \Sigma$ occurs in $\mathcal{D}_{(G,M,\boldsymbol{x})}$. You may return either the number of **words** that contain σ or the number of **times** σ occurs in all words (i.e., you can count "o" once or twice in "food"); just declare which problem you are solving.

   SOLUTION: To count the number of **words** that contain σ give each arc in B a weight of 0 if its label is σ and a label of 1 otherwise. Now, simply run the PathSum algorithm with the same semiring as in Part 2. The result will be the number of words in B that do **not** contain σ. Simply subtract this number from the result from Part 2 to get the total number of words that do contain σ.

Student Name: _____

Problem 3 – Formal Language Theory (30 minutes/30 points)

Several parsers in recent years have found it useful to use the following extended version of Context-Free Grammars, which allows specifying limited forms of regular expressions on the right-hand side (RHS) of context-free production rules. Formally, an extended CFG G is defined as $G = (T, V, P, S)$, where T, V and S are respectively a set of terminal symbols, a set of non-terminal symbols, and a starting non-terminal. This is the same as a standard CFG. Productions P however consist of a left-hand side (LHS) non-terminal $A \in V$, and a RHS, that is a finite string of zero or more elements $E_1 E_2 \cdots E_k$, where each E_i is one of the following:

1. a terminal symbol $a \in T$
2. a non-terminal $B \in V$
3. $(B|C)$, denoting a disjunction of B or C, where B and C are nonterminals from V
4. $[B]$, denoting an optional $B \in V$
5. B^*, denoting zero or more occurrences of $B \in V$

For example, the rule $A \rightarrow (B|C)\ D^*$ denotes that the RHS starts with either B or C, followed by zero or more Ds.
We wish to show that such extended-CFGs can accept only CFLs. Show that given an extended-CFG G, it can be converted into a standard CFG G' such that $L(G') = L(G)$. To do so, describe how to convert any extended production in G into one or more standard CFG grammar rules, in the following way:

1. For each of the above possible cases of a RHS element E_i, describe how to convert a rule of the form $A \rightarrow E_i$ into one or more standard CFG rules.
   (10 points)

   SOLUTION: For cases 1 and 2, no modification is necessary. Rules of these types, $A \rightarrow a$ and $A \rightarrow B$ are already standard CFG rules.
   For case 3, convert the rule $A \rightarrow (B|C)$ into three rules: $A \rightarrow A'$, $A' \rightarrow B$, and $A' \rightarrow C$.
   For case 4, convert the rule $A \rightarrow [B]$ into three rules: $A \rightarrow A'$, $A' \rightarrow B$, and $A' \rightarrow \epsilon$.
   For case 5, convert the rule $A \rightarrow B^*$ into three rules: $A \rightarrow A'$, $A' \rightarrow B\ A'$, and $A' \rightarrow \epsilon$.

2. Outline (in words or pseudo-code) an algorithm for converting an extended grammar rule where the RHS consists of a string of zero or more such elements $E_1 E_2 \cdots E_k$. **(10 points)**

   SOLUTION: Let us assume each rule in our extended grammar is of the form $A \rightarrow E_1 E_2 \cdots E_k$, where each E_i is of one of the five forms given in the question. Note that if $k = 0$, then by convention we write $A \rightarrow \epsilon$ rather than simply leaving the RHS blank.

   We desire to convert a set of grammar rules of this form into a set of rules where each E_i is constrained to be only of form 1 or form 2. We will do this by iteratively choosing a rule r that contains an E_i of form 3, 4, or 5 and replacing r with three rules as follows.

   If E_i is of the form $(B|C)$ (i.e. r is of the form $A \rightarrow E_1 E_2 \cdots E_{i-1}(B|C)E_{i+1}E_{i+2} \cdots E_k$) then remove r from the grammar and add the following three rules:
   $A \rightarrow E_1 E_2 \cdots E_{i-1} A' E_{i+1} E_{i+2} \cdots E_k$, $A' \rightarrow B$ and $A' \rightarrow C$. Note that the latter two rules do not contain any extended symbols in their right hand sides.

   If E_i is of the form $[B]$, remove r from the grammar and replace it with two rules: $A \rightarrow E_1 E_2 \cdots S_{i-1} A' E_{i+1} E_{i+2} \cdots E_k$, $A' \rightarrow B$, and $A' \rightarrow \epsilon$. Again the latter two rules do not contain any extended RHS symbols.

   If E_i is of the form B^*, remove r from the grammar and replace it with three rules: $A \rightarrow E_1 E_2 \cdots E_{i-1} A' E_{i+1} E_{i+2} \cdots E_k$, $A' \rightarrow B\ A'$, and $A' \rightarrow \epsilon$. Yet again, note that the latter two rules do not have any non-standard RHS elements.

   We can iteratively apply this algorithm, removing extended symbols (those of forms 3, 4, or 5) on the RHS until the grammar contains no more of them at all.

3. Briefly argue informally why your conversion algorithm is correct.
   (3 points) SOLUTION: Each iteration of the above algorithm chooses a single extended RHS symbol to remove. We then construct a grammar that produces the same language, but contains exactly one fewer extended symbols on the RHS of its rules. Thus, if we apply this iteratively, eventually we will arrive at a grammar that contains no extended RHS symbols, yet still produces the same language as the original grammar.

4. Execute your conversion procedure on the rule $A \rightarrow (B|C)\ D^*$ and show the resulting standard CFG rules.
(2 points) SOLUTION: First convert the disjunction:
$A \rightarrow A'D^*$
$A' \rightarrow B$
$A' \rightarrow C$

Next convert the closure element: $A \rightarrow A'D'$
$A' \rightarrow B$
$A' \rightarrow C$
$D' \rightarrow DD'$
$D' \rightarrow \epsilon$

The resulting grammar contains 5 standard CFG rules.

5. Assume you have an extended-CFG G of n rules. How many rules would the converted CFG G' have *in the worst case*?
(5 points) SOLUTION: Assume we have an extended-CFG G of n rules, each of which has at most k elements on its RHS. Each time we convert an extended CFG element, we produce three new rules, and remove the original one. In the worst case, all k of the RHS elements of all n rules are extended elements, and thus we will need to run the algorithm detailed in part 3 a total of nk times. Each time our rule count increases by 2. Thus our final rule count will be the original n rules plus an additional $2nk$ new rules, for a total of $(2k + 1)n$ rules.