

# 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 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<sup>1</sup> (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  $\epsilon$ , whichever allows your FSA to use fewer states.

---

<sup>1</sup>Adapted from a problem by Sukhamay Kundu at UC Berkeley

Student Name: \_\_\_\_\_

---

2. **Determining satisfiability (25 points):** Consider the language of *satisfiable* boolean expressions using the same alphabet  $\Sigma$ . An expression  $w(p)$  is satisfiable if it evaluates to true (T) for at least one of the cases  $p = \text{true}$  or  $p = \text{false}$ . For example  $p \vee \neg p$  and  $p \wedge \neg \neg p \vee \neg p$  are satisfiable, while  $p \wedge \neg p$  and  $\neg \neg p \wedge \neg p$  are not satisfiable. 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 language of well formed, satisfiable 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.

Student Name: \_\_\_\_\_

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

“Parachute Man” is a 1-player guessing game. You start with an unknown word  $\mathbf{y}$ , randomly chosen from a dictionary  $\mathcal{D} \subseteq \Sigma^*$ , where  $\Sigma = \{\mathbf{a}, \mathbf{b}, \dots, \mathbf{z}\}$ . Each letter of  $\mathbf{y}$  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  $\mathbf{y}$  contains the letter  $\sigma$ , then each occurrence of  $\sigma$  in  $\mathbf{y}$  is revealed. If not, you erase one string from your parachute. The goal of the game is to correctly guess every letter in  $\mathbf{y}$  before you lose all of the strings in your parachute.

A game state can be modeled as a triple  $(G, M, \mathbf{x})$ , 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  $\mathbf{x} \in (G \cup \{?\})^n$  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  $\mathbf{x}$  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  $\mathcal{D}_{(G, M, \mathbf{x})}$  of words in  $\mathcal{D}$  that are compatible with a game state  $(G, M, \mathbf{x})$ ?

Student Name: \_\_\_\_\_

---

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  $\mathcal{D}_{(G,M,x)}$ .

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

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  $D$ s.

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)**

Student Name: \_\_\_\_\_

---

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  $E_1E_2 \cdots E_k$ .  
**(10 points)**

3. Briefly argue informally why your conversion algorithm is correct.  
**(3 points)**

Student Name: \_\_\_\_\_

---

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

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)**