NAME \_\_\_\_\_\_\_\_\_\_\_MODEL SOLUTIONS – MOSTLY CORRECT PROBABLY\_\_\_\_\_\_\_\_\_

CS 420/520 Theory of Computation, Spring 2019 at Indiana State University, taught by Jeff Kinne

Exam 3 - Final Exam

Points - each part is graded as 0, 1/3, 2/3, or 3/3.

Part I) Identify which complexity class, and prove it

We have studied the following complexity classes

(A) REG - regular languages, (B) CFL - context-free languages

(C) P - polynomial-time decidable, (D) NP - nondeterministic polynomial time) (E) decidable (but not necessarily in any of the above, (F) undecidable

Questions 1-6 are six languages. There is exactly one for each of the six complexity classes. For each language, indicate which complexity class it is in. For (A)-(E), give the appropriate type of machine/algorithm that shows the language is in the class. For (B) also give a proof that the language is not regular. For (C) also give a proof that the language is not CF. For (F) give a proof that the language is not decidable.

I.1. Language - {(ab)^n (ba)^n | n >= 0}

In the language: abba, ababbaba,abababbababa, empty string

Not in the language: abab, aaa, bbb, ab, ba, bababa, abbaba

Claim: language is CF and not regular

CFG:

S -> ab S ba | epsilon

PDA: see exam3\_sample\_pda\_I1.jff

Claim: language is not regular.

Proof: By contradiction. Assume language is regular, and use the pumping lemma to get a contradiction. Let p be the pumping length for this language. Let s = (ab)^p (ba)^p. Note that s is in the language and |s| >= p. Then the pumping lemma says that we can write   
 s = xyz where |xy| <= p, |y| >= 1, and xyiz is in L for all i >= 0.

Note that y must be in the (ab)^p portion of the string. All the possible ways of what y could look like…

Case 1 – y starts with a and ends with a – xyyz will have two a’s in row, and then xyyz would not be in the language.

Case 2 – y starts with a and ends with b – xyyz will have more ab’s in the first part of the string than ba’s in the second part, and then xyyz would not be in the language.

Case 3 – y starts with b and ends with a – xyyz will have more ab’s in the first part of the string than ba’s in the second, and then xyyz would not be in the language.

Case 4 – y starts with b and ends with b – xyyz will have two b’s in a row twice, and then xyyz would not be in the language.

In all cases, xyyz is not in the language. This is a contradiction to the pumping lemma, so it must be that the language is not regular.

I.2. Language - {(ab)^n (ba)^m | n >= 0 and m >= 0}

In the language: ab, ba, abba, ababba, ababbaba, baba, empty string

Not in the language: aa, abb, baab, bb, a, b

Regular.

RE: (ab)\* (ba)\*

See also: exam3\_sample\_dfa\_I2.jff

I.3. Language - {(ab)^n (ba)^n (ab)^n | n >= 0}

In the language: abbaab, ababbabaabab, empty string

Not in the language: abba, abab, baba, a, b, ababbabaab

Claim: in P and not CF

Proof that language is in P – we need to give an efficient algorithm.

* Loop through the input string as long as it is (ab)^n, record how many it was (what is n).
  + If we ever read two characters that is not ab or ba, reject.
* Continue looping through as long as it is (ba)^m, record how many it was (what is m).
  + If we ever read two characters that is not ab or ba, reject.
* Continue looping through as long as it is (ab)^k, record how many it was (what is k).
  + If we ever read two characters that is not ab or ba, reject.
* Accept if and only if n = m = k.

We do determine the correct answer, and the running is linear.

Proof that the language is not CF. Proof by contradiction using the pumping lemma for CFL. Assume the language is CF, and let p be the pumping length from the pumping lemma for CFL for this language. Now create a string s that is in the language and at least length p.

Let s = (ab)^p (ba)^p (ab)^p  
The pumping lemma guarantees that we can write s as uvwxy where |vwx|<=p, |vx|>=1, and uviwxiy is in the language for all i>=0. Let’s look at all the cases for where vwx could be in s.

Case 1 – vwx completely contained in first (ab)^p – could end up with more (ab)’s in the first part than the length of the second and third parts of s (then uv2wx2y is not in the language), could end up with the first part not being (ab)^k). In either case, uv2wx2y is not in the language.

Case 2 – vwx completely contained in second or third parts – similar to case 1.

Case 3 – vwx is between first and second parts. If v is in first and x is in second part, we potentially have longer first and second parts than third part (or we could have v or x not following the rules – like v=aba or v=babab – when they are repeated). In all cases, uv2wx2y is not in the language

Case 4 – vwx is between second and third parts – similar to case 3

No matter where vwx is in s, uv2wx2y is not in the language. Contradicts the pumping lemma for CFL, therefore it must be that the language is not CF.

I.4. Language - {(M, x) | M is a TM that runs in at most n^2 time (n = |x|) and that takes two inputs and there exists at least two different y1 and y2 such that M(x,y1) and M(x, y2) both accept }

NP (with the modification of the language as noted in the video – need to put a limit on the running time for this to be in NP)

For example, x could be a graph, and y1/y2 could be a coloring assignment that M checks to see if it is valid for X and works as a coloring. In that case, M would be the “verifier” for the coloring problem.

For example, x could be a Boolean formula, and the second input is an assignment of T/F to the variables, with M checking that this makes the formula evaluate to T. Then M would be a verifier for the SAT problem.

Because we require M to run in n^2 time, this means for the second input is length at most n^2.

Claim – language is in NP.

Proof: Give a verifier for the language.

* Input to the verifier: (M,x)
* Certificate / Hint: y1 and y2 that cause M(x, \_\_) to accept
* Run M(x, y1) for at most n^2 steps
* Run M(x, y2) for at most n^2 steps
* Accept if and only if both simulations finish within n^2 steps and accepted, and y1 != y2
* Reject if any of – M(x, y1) did not finish, M(x, y2) did not finish, M(x, y1) rejected, M(x, y2) rejected.

Check that verifier has the properties –

* Efficient, polynomial time – running time is something like n^2 or some polynomial of n^2 (depending the precise details of how we simulate M). Yes.
* If (M,x) is in the language… Then there exists a y1 and y2 that make M(x, \_\_) accept within n^2 steps. These y1 and y2 are the certificate/hint to our verifier, and cause it to accept.
* If (M,x) is not in the language…. There are not two different y1 and y2 that cause M(x, \_\_) to accept within n^2 steps. This means no matter what y1 and y2 are given, at least one will cause our verifier to reject.

I.5. Language - {(M, w) | M is a TM and M(w) accepts using only the tape cells that w was originally written on}

Decidable.

Run M(w) and it stays within the tape cells that w starts on. Is there a limit on the running time? Maybe it can only run so long and still be using just the tape cells we started with? Could be decidable but take long time (exponential)?

Algorithm

* Simulate M(w)
  + Keep track of the configuration (tape contents, internal control state, tape head location) at each point in our simulation.
  + If we ever repeat a configuration, we are in an infinite loop, and reject.
  + Configurations would look like -   
    \_, q0, 1, 0, 1, 1, 0, 0, 0, 1, \_  
    \_, 1, q1, 0, 1, 1, 0, 0, 0, 1, \_  
    \_, q2, 1, x, 1, 1, 0, 0, 0, 1, \_  
    …  
    \_, q2, 1, x, 1, 1, 0, 0, 0, 1, \_
* If M(w) ever causes the tape head to move over a \_ then reject
* If M(w) halts and rejects, then reject
* If M(w) halts and accepts, then accept

Properties of our algorithm

* If (M,w) is in the language – that means that M(w) accepts and does not use more than the tape it started with – our algorithm does accept
* If (M,w) is not in the language –
  + M(w) goes onto a \_ cell at some point – we reject
  + M(w) never goes onto a \_ cell but rejects – we reject
  + M(w) never goes onto a \_ cell and does not reject or accept because M(w) is in an infinite loop
    - By storing all of the different configurations as we simulate, we detect an infinite loop if there is one and reject.
    - How many different possible configurations?
      * Tape head location: n
      * Internal control state: |Q|
      * Tape contents: n tape cells, |Sigma| many choices for each, |Sigma|^n total possible tape contents
      * Total: n \* |Q| \* |Sigma|^n

Claim: our algorithm is correct.

Claim: running time is exponential.

Note – what about if we required M(x) use at most n^2 tape cells? Similar. If we enforce any reasonable maximum amount of tape cells to use, then it is decidable. <= 2^n tape cells -> decidable (2^(2^n)).

I.6. Language - { Y | Y is a Java program that never goes into an infinite loop, so it halts for all inputs}

Claim: language is undecidable.

Proof: reduction from A\_TM. Have an instance of A\_TM that we want to solve, let’s say it’s (M,x). Does M(x) accept?

Build an instance Y of our language that will give us the solution of (M,x). Let Y be the following Java program.

* Doesn’t care about its input.
* Simulate M(x) by hard-coding M and x into the Java program. We can make a Java program to simulate TM because Java is Turing complete.
* If simulation of M(x) halts and accepts, make our Java program Y accept.
* If simulation of M(x) halts and rejects, make our Java program Y go into an infinite loop.
* Note that if M(x) has an infinite loop then our Java program Y also has an infinite loop.

Note that if Y does not have an infinite loop on any input, then this must be because M(x) halted an accepted. Either way, Y is in our language if and only if (M,x) is in A\_TM.

We have given a reduction from A\_TM to our language. A\_TM is undecidable. Therefore our language is undecidable.

Note – our reduction is efficient (given a simulator in Java of TM’s, we could create this Y efficiently).

Part II) Regular Languages

II.1. Give a Python3 regular expression for the given language.

Note - you should know what the following symbols do/mean in Python3 RE's

. ^ $ \* | + ? {} [] () \d \s \w

and you should remember the \\ issue and using r''

Language: valid C identifier string - sequence of at least one character from letters, numbers, underscore, and begins with non-digit.

s = input('')

re.match(r'[a-zA-Z\_](\w|\_)\*', s)

(More practice - valid phone #, email address, name, sentence.)

II.2. Prove the following statements. Statements I will choose from -

1. regular languages are closed under intersection – two dfa’s, create a new dfa that keeps track of what would have been happening in both (# states is multiplied)
2. regular languages are closed under union - easy
3. regular languages are closed under concatenation – easy (RE)
4. regular languages are closed under complement – given dfa, flip which states are final and which are not
5. given an NFA, there exists a DFA that accepts the same language – create a dfa that keeps track of all possible states the nfa could be in at this step (# states in the dfa is 2 to the power # of states in the nfa)
6. given a RE, there exists an NFA that accepts the same language – need to be able to handle the regular operations (\* | . grouping ()). Suppose you have NFAs for RE’s R1, R2, create a NFA for   
   R1 | R2. Same for \* . To handle | see below about union. To handle R1 . R2 – take nfa for R1, add epsilon transitions from each final state of nfa R1 to the start state of nfa for R2. To handle \*, have an nfa for R, want to do R\*.

Level of detail looking for. Something like the following –

ii) If L1 and L2 are regular languages, then so is L3 = L1 union L2.  
x is in L3 precisely if x is in either L1 or L2 or both

If L1 and L2 are regular, then we can assume a RE RE1 for L1 and RE2 for L2. We can also assume dfa’s D1 for L1 and D2 for L2. We need to create either an RE for L3 or a nfa/dfa for L3.

We create a RE for L3: (RE1) | (RE2)

Verify that if x is in …

We can also create an nfa for L3: a copy of D1, a copy of D2, a new start state with epsilon transitions to the start states of D1 and D2.

Note – that is a little more than the level of detail in the “proof idea” in Sipser, but not as much as the full proof.

Part III) Context Free Languages

III.1. Given the following CFG -

<real> --> <digit> <digit\*> <decimal part> <exp>

<digit\*> --> <digit> <digit\*> | epsilon

<decimal part> --> '.' <digit> <digit\*> | epsilon

<exp> --> 'E' <sign> <digit> <digit\*> | epsilon

<sign> --> + | - | epsilon

<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note – this is an CFG for decimal #’s, possibly with ., possibly with exp.

Give a string that is in the language: 0, 1234.123, 123.21E-23

Give a string that is not in the language: ., 0., E, E23

Is this language itself closed under star? That is, if a string w is in the language, are ww, www, wwww, etc. also in the language?

12.23, 12.23 are in the language, is 12.2312.23? No.

III.2. Write out the transition table for the following PDA:

http://cs.indstate.edu/~jkinne/cs420-s2019/code/exam3\_sample\_pda.jpg

Transition table –

a,- a,b b,- b,b -,- // input,popped\_symbol

q0 q1,a reject // state,pushed\_symbol

q1 q1,b q2,-

q2 q3,-

q3 q0,-

Give a string that is accepted by the PDA: abbab, abbbbab abbab

Give a string that is not accepted by the PDA: a, b, ab, abb, abab

What is the shortest string accepted by the PDA? Empty string, abbab

If the set of accept states was swapped (so 1, 2 and 3 are accept states) - what would the language be? (Note – typo fixed – flipping accepting states would make q1, q2, q3 the accepting states)

In this case, we would be part way through the last “time around”   
 (abb b\* ab)\*

So, it would be   
 ((abb b\* ab)\* a) | ((abb b\* ab)\* ab) | ((abb b\* ab)\* abb) |   
 ((abb b\* ab)\* abb b\*) ((abb b\* ab)\* abb b\*a)

Part IV) TM's and Other

IV.1. Configuration history of a TM. Write down the configuration of the following TM running on the following input, and write the configurations until the machine enters the accept state, enters the reject state, or rejects due to no-transition-defined.

TM: https://bytesoftheday.files.wordpress.com/2014/10/turing\_machine.png

Input: abbc

Note: let’s call the states in the TM from top-left to bottom-right q0, q1, q2, q3, and then qY and qN. Let’s say qY is accepting, qN rejecting.

\_,q0,a,b,b,c,\_

\_,x,q1,b,b,c,\_

\_,x,x,q2,b,c,\_

\_,x,x,b,q2,c,\_

\_,x,x,b,x,q3,\_,\_

\_,x,x,b,x,\_,q0,\_,\_

\_,x,x,b,x,\_,\_,qY,\_

Note – seems to be accepting the language a^n b^n c^n ? Try out abbc, aabc, abcc, abc, aabbcc. Okay, not quite.

IV.2. If the language is decidable, describe an algorithm to decide the language and explain why it is correct. If the language is undecidable, prove it is undecidable using one of the following techniques - reduction of known undecidable language, application of Rice's Theorem, or direct proof by diagonalization.

Language - {<Y,x> | Y is a Java program that contains at least one variable that is initialized but then never used after initialization}

If we run Y(x), is there a variable that is initialized but never used after initialization? Consider a program something like …

Int i = 3;

Int k = 2;

While (condition()) {  
 k ++; // maybe this is easy to figure that this happens

…

If (…)

Goto SOMEWHERE;

RecursiveCall(…);

If (…) {

i = 0;  
 }

}

Is i ever going to be set to 0?

Intuition tells you this may be the halting problem. We may have to run the program to see what happens (because it’s too complicated to figure out just be looking at the code), and then we won’t know if the program is going to halt or not.

So, claim: the language undecidable.

Proof outline:

* Reduction from A\_TM
* Input (M,w) from A\_TM, determine if M(x) halts and accepts
* Create program Y
  + Ignore its input x
  + Y initializes a variable int x=0;
  + Y simulates M(w). If M(w) halts and accepts then Y sets x=1 and halts.
* Note – can be shown that (M,w) is in A\_TM if and only if (Y,x) is NOT in our language. If we could solve our language we could solve A\_TM. Since A\_TM is undecidable, so is our language.