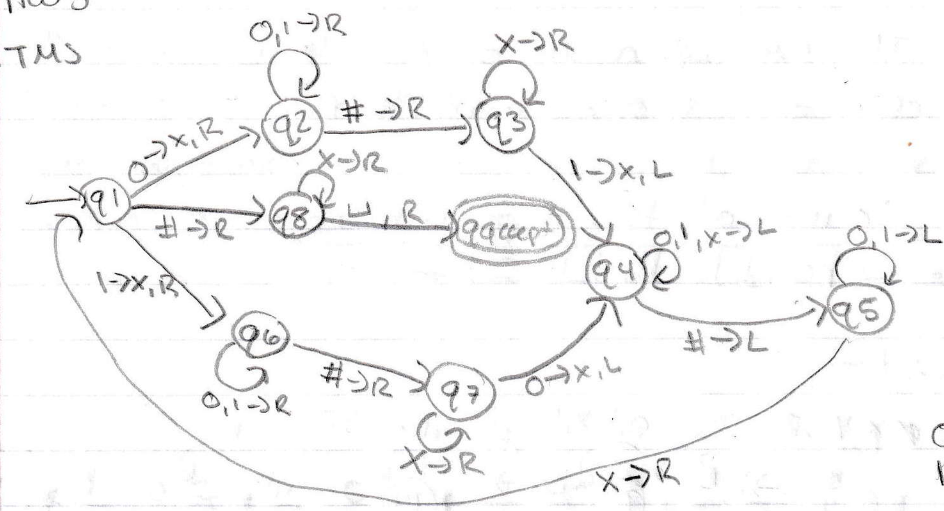
Braylee Cumro

Hw5

1. TMS



*negation: Change 0's to 1's + 1's to 0's

010
101

top see a 0 replace w/ x

X X X # X X X ⊔ 010 #101 ⊔ accepts

X X 1 # X 0 1 ⊔ does not accept

language:

$\{ w \# v \mid w \in \{0,1\}^*$ and v is the negation of $w \}$

1.2 Create TM

Create a TM that will decide the following language over the alphabet {0,1}. Solution should be a formal TM

$L(A) = \{ w \mid w$ contains twice as many 0s as 1s$\}$

X X X 1 X X ⊔   100100⊔   110000⊔   100001⊔

ideas: search for a 1

if one found

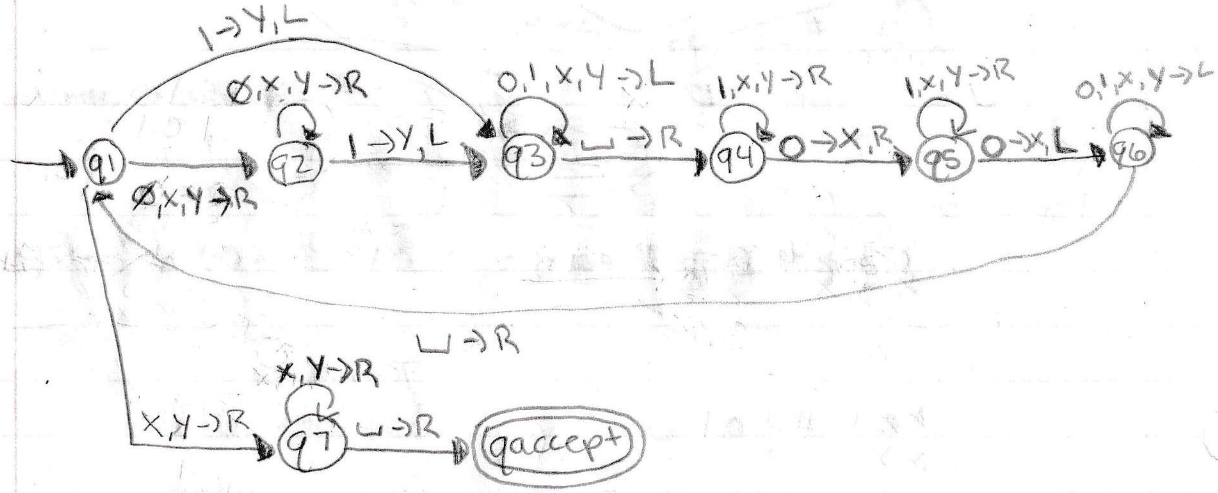
- return to leftmost element and search for 2 zeros if no zeros found reject

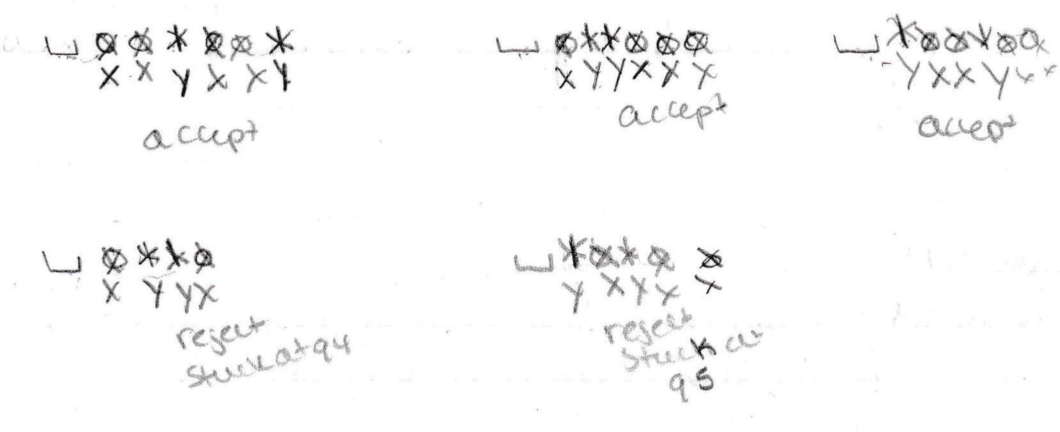
- if no one found

- reject if you see any 0

for this TM we will assume that there are "␣" to the left of our input string. This is one method we talked about using in class. For example if our input is '011010' the tape will appear as follows

...	␣	0	1	1	0	1	0	␣	...





The idea was to search left for a 1. if we found 1 go back and go through tape to find 2 zeros. fail to find two zeros reject.
If there remained ∅ or 1 on the tape we could not accept w/ q7 which only passes the marked "0" and "1"

~ The goal cross off as many a as b if you run out of a's and b still exist reject. When b's are gone now cross of a followed by c the # of a remaining should match # of c

1.3 Create a TM

alphabet = {a, b, c}

- accept + reject conditions

$L(A) = \{a^i b^J c^K \mid i, J, K \geq 0$ and $i - J = K$

$8 - 5 = 3$

~~aaaaaaaa~~ ~~bbbbb~~ ~~ccc~~

X X X X X X X X Y Y Y Y Y Z Z Z

1. Go right on the tape until an "a" is encountered replace it w/ an "x" - If you encounter "b" or "c" before ever encountering "a" reject
2. Continue right through any "a" or "x" until a "b" is encountered replace the "b" w/ a "y" - If no "b" encountered but hit "c" go to step 7
3. return the tape to the left end of the input string
4. repeat steps 1 and 2 until there are no "b" remaining on the tape ~> return the tape to left end of input string
6. Go right on the tape until an "a" is encountered replace it w/ an x - if no "a" encountered before a "b" or "c" reject
7. Go right on the tape through "a" + "y" + "x" until a "c" is encountered. replace the c w/ a "z" - if no "c" encountered before "␣" reject
8. return the tape to the left end of the input string
9. repeat steps 6 + 7 until all "a" are gone/replaced
10. return tape to left end of the input string
11. Go right on the tape through "x", "y" and "z" if you hit "␣" accept
    - if you hit "a", "b" or "c" remaining on tape reject

1.4 Describe a TM

alphabet $\{0,1\}$

$\{\langle G\rangle \mid G \text{ contains a cycle}\}$

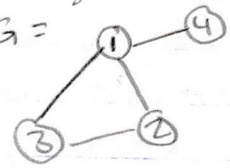
G undirected connected graph

(1) identify a method of encoding your graph as an input string

(2) Come up w/ a TM algorithm that will accept if and only if the input graph contains a cycle.

(1) identify a method of encoding graph as input string

G =



$\langle G\rangle$ =

(1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))

To encode a graph as an input string we want it to look like the example above on the tape it would appear like

(	1	,	2	,	3	,	4	)	(	(	1	,	2	)	. .	. .	⊔

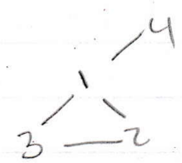
In this format, the vertices or nodes are listed 1st in the format $(v_1, v_2, v_3, \ldots v_n)$ w/ parenthesis denoting the group of vertices and commas seperating them out.

This is followed by another set of parenthesis these ones holding the set of edges in the graph formatted as (v_1, v_2), (v_1, v_2) w/ parenthesis encapsulating individual edges seperated by commas. The start and end vertices of the edges are seperated by a comma.

(3) come up with an algorithm that will accept iff the input graph contains a cycle

M = "on input ⟨G⟩, the encoding of graph G

1. For all nodes in G:
   2. mark node as visited ("v")
   3. write node as immediate character after "⊔" to keep track of parent
   4. for each node connected to the current node by an edge in graph G not marked used
      - if node is marked "v" and not the character written after "⊔" on the tape
        - accept cycle found
      - else, mark node as visited "v"
        - mark edge w/ current node and character after "⊔" as used, "u"
        - write current node after "⊔" go to 4
5. if you get out of loop for step 1 reject

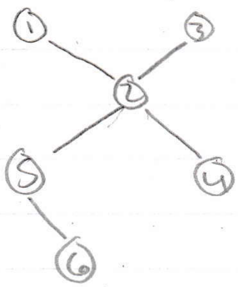


(1, 2, 3, 4) ((1,2) (2,3) (3,1) (1,4))

v v v   u u

accept

⊔ ~~1~~ ~~2~~ 3



(1, 2, 3, 4, 5, 6) ((1,2) (3,2) (4,2) (5,2) (5,6))

v v v v v v   u u u u

⊔ ~~1~~ ~~2~~ ~~3~~ ~~2~~ ~~4~~ ~~2~~ ~~5~~ 5

reject

My goal was to get a depth first search going. I was struggling w/ keeping track of the parent node this is the best idea I have.

Decidability

2.1 Consider the problem of determining wether a DFA and a regular expression are equivalent

(1) Express this problem as a language

$$EQ_{DFA\ REX} = \{\langle A, B\rangle \mid A \text{ is a DFA and } B \text{ is a regular expressions and } L(A) = L(B)\}$$

(2) provide a formal description of the language a TM

Formal description:

This language aims to test whether a given DFA (A) recognizes the same language as a given regular expression (B). The TM should accept when the language L(A) (the language recognized by the DFA) is equal to language L(B) (the language recognized by the regular expression).

Idea convert regular expression to NFA to DFA so we can use EQ_{DFA}

claim: $EQ_{DFA\ REX}$ is decidable

proof idea: create a TM N that accepts if a DFA and regular expression recognize the same language

N = on input $\langle A, B\rangle$, where A is a DFA and B is a regular expression

① convert the regular expression B to NFA C using the ideas in lemma 1.55 page 67 of Sipser

② convert NFA C to DFA D using procedure from theorem 1.39 in Sipser

③ Run TM from theorem 4.5 in Sipser (EQ_{DFA}) on $\langle A, D\rangle$

④ If the TM accepts, accept; otherwise reject

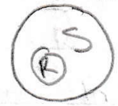
(3) why does TM work properly

This TM works because it is based on EQ_{DFA} which we proved to be decidable in class. EQ_{DFA} is defined as $\{\langle A, B\rangle \mid A$ and B are DFAs and $L(A) = L(B)\}$. We know that regular expressions $\equiv$ NFA $\equiv$ DFA and we can convert a regular expression to a DFA because of this. Thus we can change the regular expression to a DFA and plug this DFA in with the other from the problem to see if a DFA and a regular expression recognizes the same language. Therefore we can convert B (the regular expression) to a DFA using a finite set of steps and compare the converted B w/ A (DFA), using EQ_{DFA} and when that TM accepts $L(B) = L(A)$

2.2 $A = \{\langle R, S\rangle \mid R, S$ are regular expressions and $L(R) \subseteq L(S)\}$
show A is decidable

(1) provide a TM (high level description)

If we intersect R + S



$R \cap S = R$ iff $R \subseteq S$

b/c if R is a subset of S their intersection / shared values should be equal to the subset (R)

- plan go from regular expressions to DFA's. DFA_R + DFA_S.
- create a DFA that is the intersection of these two DFAs I
- Use EQ_{DFA} on $\langle I, DFA_R\rangle$ if it accepts R is a subset of S

claim: A is decidable

proof idea: Create a TM M that decides A

M= "on input <R,S> where R and S are regular expressions

1. convert regular expression R to NFA_R and the regular expression S to NFA_S using the ideas found in lemma 1.55 on page 67 of Sipser
2. convert NFA_R to DFA_R and NFA_S to DFA_S using the procedure from thearm 1.39 in Sipser
3. Construct a DFA ~> DFA_{inter} that accepts $D_R \cap D_S$
   - $L(DFA_{inter}) = DFA_R \cup DFA_S$
   - page 46 of sipser talks about how ton construct a DFA for intersection in the footnote
4. Run EQ_{DFA} on <DFA_{inter}, DFA_R>
5. if EQ_{DFA} accepts accept; otherwise reject

(2) why this works

a subset diagram for this problem would look like the following

Showing that for R to be a subset of S all of its elements must also exist in S.

The idea behind this Turing machine is to use EQ_{DFA} to test if the intersection (pieces S + R share) is equivalent to R which would mean all elements of $R \in S$. To do this we must 1st convert the regular languages of R + S to NFA and then DFAs using algorithms that exist since NFA ≡ DFA ≡ regular expressions. Then we create another DFA which is the intersection of these two DFAs. Finally we check if the intersection

and the DFA for R are equal using EQ DFA which we already showed was decidable.
-Note that we know DFAs are closed under intersection from the footnote on page 46 of sipser.
-This TM will only accept when the elements of R are all also elements of S making R a subset of S

2.3 A useless state in a PDA is never entered on any input string. Show that determining wether a PDA has any useless states is decidable
(1) define the language
$L(u) = \{ \langle P \rangle \mid \text{where P is a PDA and P has a useless state} \}$

idea: for each state in PDA create a CFG w/ that node as accept state. Use E_{CFG} and if E_{CFG} rejects move to next node if E_{CFG} accepts accept. If you make it through all nodes in PDA as start state reject.

proof: create TM R on $\langle P \rangle$:
(1) for each state of PDA P
(2) mark the state as the only accept state in the PDA
(3) convert the PDA w/ this node as the start state to a CFG C using lemma 2.27 in sipser that starts on page 121
(4) Run E_{CFG} on $\langle C \rangle$.
(5) if E_{CFG} accepts accept
(6) at end of loop reject

(3) Why this works
ECFG accepts when a CFG has an empty language. If this CFG has one accept state which is a singular state in the original PDA, then if ECFG accepts that node is useless in the PDA. There is no words in the language of the PDA that can reach that node and we know this sense it doesnt accept any strings into its language
We can convert the PDA to a CFG b/c CFG ≡ PDA and we have a lemma for it
- It accepts the correct language b/c it only rejects when it found no nodes that have an empty language when they are the start state
- There is a finite number of states so the machine cannot loop through states forever (Definition of a PDA).

Bonus Problem

The introduction of the Turing Machine helped formalize our modern day notion of an "algorithm" by helping us understand what was possible with algorithms or rather helping show that some things were impossible. Turing machines were one way to show that algorithms could have three outcomes accept, reject or loop. As with most models some problems are easier to understand on one model over another. With the introduction of the Turing machine came the church-turing thesis making algorithms equivalent to a turing machine. This helped us find some problems Some problems are easier to see loops on on a turing machine rather then trying to convince yourself that you have tried all combinations of an algorithm. If the church-Turing thesis was never introduced we might be behind in our understanding of what is possible on our modern machines. The Church-Turing thesis shows that algorithms written in pseudocode have an equivalent turing machine (representation on a modern computer). Sometimes writing pseudocode is much simpler then creating a program/TM so the Church-Turing thesis helps show what is possible (even if that algorithm is inefficient or not yet formally written out).