yColDefn

Definition: 1

2

CSCC63 – Week 2 Tutorials

Exercise courtesy of Mustafa Quraish and Prof. Cheng

1. This exercise makes use of the Turing Machine Simulator https://mustafaquraish.github.io/TMSim/ created by former C63 student Mustafa Quraish.

   Let M be the TM whose description for the simulator is given below. (You can cut and paste the TM description into the TM simulator.)

   The input alphabet is 0,1.

	-	q0	qA	qR	TM from week 2 tutorial
q0	1	q0	0	R	
q0	-	q1	1	R	
q0	0	q4	.	R	
q1	-	q2	.	L	
q2	1	q3	.	L	
q3	0	q3	.	L	
q4	0,1	q4	.	R	
q4	-	q5	.	L	
q5	1	q5	.	L	
q5	0	qA	1	L	

   (a) Draw the state diagram for M.

   (b) i. Find a string that M accepts.
   ii. Find a string that M rejects.
   iii. Find a string on which M loops.

   (c) Describe $L(M)$.

   (d) So far we've only looked at TMs as automata that produce a yes/no answer. But sometimes we want to work with programs that have other types of outputs.

   The output of a TM is whatever is on its tape when it halts. So, for example, if we wanted to write a TM that computes x^2 given an input of $\langle x \rangle$, we'd use the TM to write x^2 on some section of the tape, then erase everything else on the tape, and then halt.

   This means that the output of a TM is *undefined* if the TM loops — in which case we use a special symbol $\perp$ to indicate that the output function is undefined. A function that may not be defined for some inputs is called a *partial function*.

   We can write these functions using the notation you already use for piecewise functions, e.g.:

$$f(w) = \begin{cases} \text{description 1}, & \text{case 1} \\ \text{description 2}, & \text{case 2} \\ \vdots & \\ \perp & \text{case } n \end{cases}$$

Let $f : \{0,1\}^* \to \{0,1\}^*$ be the partial function computed by M.

i. Describe f.

ii. What is $f(f(f(f(11010))))$?

2. Let our input alphabet be $\{0,1\}$. We consider our input to be a number in its binary representation. Design a TM that adds 2 to the input, then accepts with the head at the first tape cell.

   For example, if the initial configuration is

   $$q_0 1011 \sqcup$$

   then the final configuration should be

   $$q_A 1101 \sqcup$$

   Also, if the initial configuration is

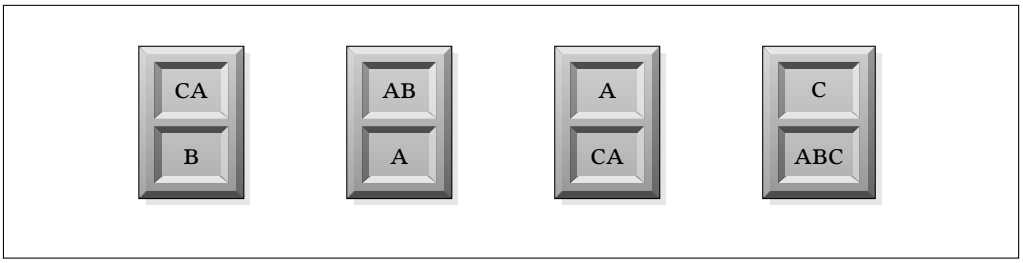
   $$q_0 110 \sqcup$$

   then the final configuration should be

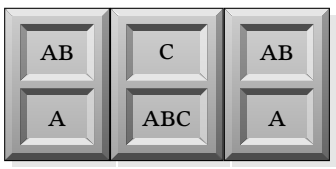
   $$q_A 1000 \sqcup$$

3. Let's play a bit with the **Post Correspondence Problem**.

   The Post Correspondence Problem (PCP) is a decision problem whose instances are made up of sets of tiles:



   We can take copies of these tiles and line them up:



   Notice how we use tile number two twice in this sequence. We can use each tile type as often as we want.

When we take a sequence of tiles like the one above, we can read off two strings: one from the top and one from the bottom:

Top string: AABCA

Bottom String: ABCAB

In this case the top and the bottom string are different.

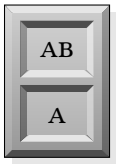
The question is this: If I give you a set of tiles such as the one above, can you find a non-empty sequence of tiles such that the top and bottom strings are the same?

> **The Post Correspondence Problem (the PCP)**:
>
> **Input**: A finite set of tiles.
> **Question**: Can we find a non-empty sequence of tiles such that the top and bottom strings read from that sequence are equal?

As always, the string encodings of yes-instances for this problem form a language. We'll refer to this language as the PCP, as well.

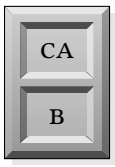
Now, suppose that I was trying to determine whether the set of tiles above was a *yes*-instance of the PCP. Then I should be able to find a matching set of tiles.

In fact, the only tile such a matching could start with would be the second tile:



Can you see why?

You can also see that the next tile is forced — it has to have a bottom string that starts with a B, so the only choice is:



And we can go on from there — the bottom string of the next tile has to start with C, and so on. The idea is that if a PCP instance is built right, a search for a matching set of tiles boils down to extending the possible starting sequences one time at a time. Every time we add a tile, then next tile follows from the last.

We're going to try to use this idea to "code" using the PCP. The problem is this: your input is a directed graph $G = (V, E)$, and two nodes s and t in V. You want to tell whether there's a path in G from s to t (feel free to ask questions if you need a refresher on graphs from CSCB63).

Now, we know from CSCB63 how to answer this question. But the question here is: can you, without directly solving the path problem, create a PCP instance that is a *yes*-instance iff your input graph G has an s-to-t path?