Alex Hubers
CSC:341 Spring 2024

Lab #12A

Due: April 27, 2024

#1

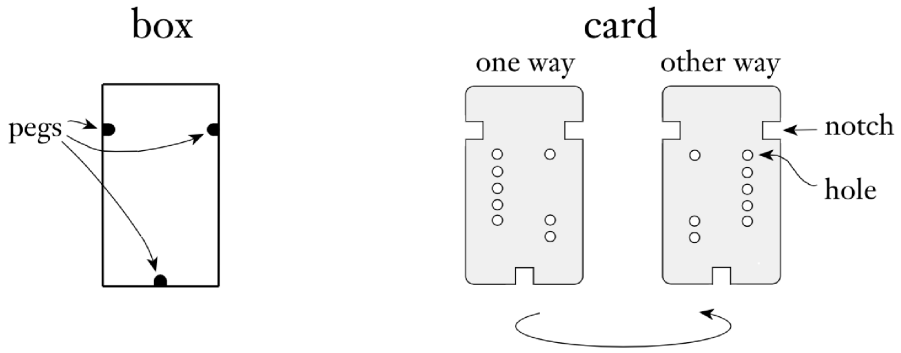
Answer the following key questions about the proof of the Cook-Levine theorem that tests our understanding of Turing machines and polynomial-time reductions.

1. What are the inputs and outputs of the mapping function of the reduction?
2. Why do the tableau under consideration have n^k rows?
3. Why can we bound the size of the tape, i.e., the number of columns of the tableau, to n^k?
4. What do the boolean variables $x_{i,j,s}$ represent in the reduction?
5. In a few sentences each, describe the structure and purpose of the following boolean formulae:
   - ϕ_{cell}
   - ϕ_{start}
   - ϕ_{move}
   - ϕ_{accept}
6. Does a 2×2 window, i.e., two rows and two columns, work? Why or why not?
7. How does the constructed boolean function account for the nondeterminism of the NTM it simulates?

1. The inputs to the mapping function is a Turing machine M and an input w. The output is a boolean formula ϕ.
2. The tableau have n^k rows because it is assumed that M runs in some polynomial amount of time n^k. Each row is a step of the computation.
3. We can bound the size of the tape to a polynomial because a Turing machine cannot access more memory than steps of time it takes to run.
4. $x_{i,j,s}$ is true if and only if cell (i, j) of the tableau is filled with symbol s.
5. Each of the boolean formulae of Cook-Levine:
   - ϕ_{cell}: checks that every cell of the tableau has exactly one symbol enabled.
   - ϕ_{start}: checks that the first row corresponds to the initial configuration of M on w.
   - ϕ_{move}: checks every 2×3 window of the tableau correctly encodes a potential step of execution of M.
   - ϕ_{accept}: checks that some row of the tableau ends on q_{accept}.
6. No it does not. There are many counterexamples that we could draw where one configuration to the next is not valid, however every 2 by 2 window is legal.
7. The constructed boolean function accounts for non-determinism by emitting sub-formulae in ϕ_{move} to account for every possible step of non-determinism. The assumed solver for SAT is then responsible for determine which of non-deterministic paths would result in acceptance, if any of them do.

#2

You are given a box and a collection of cards as indicated in the following figure. Because of the pegs in the box and the notches in the cards, each card will fit in the box in either of two ways. Each card contains two columns of holes, some of which may not be punched out. The puzzle is solved by placing all the cards in the box so as to completely cover the bottom of the box (every hole position is blocked by at least one card that has no hole there).



Define a problem, called PUZZLE, below.

$\mathsf{PUZZLE} = \{\langle c_1, \ldots, c_k \rangle \mid$ Each c_i represents a card and this collection of cards has a solution $\}$.

Ultimately, our goal will be to show that PUZZLE is NP-complete. The following steps lead us through the proof, however you may or may not find each step useful in your thought process. The only required parts (that you must turn in) are part b and part f.

1. First, let's explore this problem space a little bit. Design two sets of cards, C_1 and C_2. For C_1, ensure that there is a solution to the puzzle, i.e., an orientation of each card such that the bottom of the box is covered, and list this solution. For C_2, ensure that there is no such solution to the puzzle. Try to make these example cards non-degenerate, (for example, don't choose every card to have no holes) so that you can use your examples to generalize these results.

2. To show NP-completeness, we must show both conditions of Definition 7.34. First, show that $\mathsf{PUZZLE} \in \mathsf{NP}$.

   (Hint: what is a potential solution to PUZZLE? How do we verify that the solution is correct?)

3. Now, let's consider the second condition of Definition 7.34. (Note: If a language satisfies the second condition, we call the language NP-hard.) Here, we will show NP-hardness by reducing SAT to PUZZLE where we know via the reading that SAT is NP-complete. First, let's identify the "inputs" to each of the problems. What are the inputs to (predicates that decide) SAT and PUZZLE?

4. With the inputs, let's constrain our reduction function f. Write down the type of the reduction function f, i.e., what does f take as input and produce as output?

5. Next we must design a the function f. Let's experiment with designs that link together these parts and see if we can strike gold! To do this productively, first design a toy SAT example of 3–4 variables and 3–4 clauses with a satisfying assignment. Give at least two potential designs for reductions from SAT to PUZZLE showing how your reductions operate on your toy example, even if the designs do not work! If the designs do not work, describe what was broken about the reduction and how you could then address it in a subsequent attempt.

6. Repeat the previous step until you find a working reduction. Once you have this reduction, formally show that SAT $\leq_p$ PUZZLE, demonstrating that PUZZLE is NP-hard. Make sure to prove both sides of the correctness condition of your reduction!

1. Answers will vary.

2. PUZZLE $\in$ NP.

   Proof. The verifier for PUZZLE takes a stack of cards as input and a recommended orientation of the cards as a certificate. The verifier can then pile the cards, orienting them according to the certificate, and verify whether the bottom of the box is visible. It takes a linear amount of work to orient and pile the cards and a constant amount of work to sight-verify the holes. □

3. Inputs to SAT: A boolean formula, potential assignments for variables.

   Inputs to PUZZLE: A collection of cards with punched out holes, potential orientations for the cards.

4. f takes in a boolean formula, and produces a collection of cards.

5. Answers will vary.

6. PUZZLE is NP-hard

   Proof. We reduce SAT to PUZZLE, i.e., SAT $\leq_p$ PUZZLE:
   Let ϕ be a boolean formula in 3-cnf form. For each variable $(x_1, x_2, \ldots, x_k)$, create a card, and for each card, create a row on that card for each clause of ϕ. For each row on each card, punch a hole out of the left-hand column if the variable corresponding to the card does not appear in the clause corresponding to the row. Punch a hole out of the right-hand column if the variable's complement does not appear in the clause. Finally, include an extra card in which the left-hand column is entirely punched out and the right-hand column untouched.

   Now we show that this reduction preserves correctness of inputs:
   Before we begin, we introduce some terms to simplify the presentation. Call the left-hand column of each card the positive column and the right-hand column the negative column. Call the orientation of a card positive when the card is untouched, i.e., the positive column is on the left and the negative column is on the right. Call the orientation of a card negative when the card is flipped, i.e., the positive column is on the right and the negative column is on the left.

   First, we show that if ϕ is satisfiable, then the resulting cards have a solution. Orient the cards so that if variable x is true in the satisfying assignment, then the card is oriented positively. If x is false in the satisfying assignment, then the card is oriented negatively. Finally, the extra card is included in the deck as-is.

   Now consider each row of the resulting card deck and its corresponding clause in ϕ. Since the assignment in question is satisfying, one of the literals of these clause must be true, call it x. Observe that by our construction, the card corresponding to x must not be punched out in this position (the left-hand column of the row). This is true for each such row since each clause is true and thus the left-hand column is covered. Finally, by construction of the extra card, the right-hand column is also covered as it has no holes on the right-hand side.

   Finally, we show if the card deck has a solution then ϕ is satisfiable. Construct the satisfying assignment for ϕ by assigning each variable x of ϕ to true if its corresponding card is positively oriented. Otherwise, assign x to false if its corresponding card is negatively oriented.

   To see how the resulting assignment is indeed satisfying for ϕ, consider each clause of ϕ and its corresponding row among the cards. For the deck to have a solution, some card of the deck must cover the left-hand column of this row. By construction, this card corresponds to some variable x of this clause. If the card is oriented positively, then x itself makes the clause true. If the card is oriented negatively, then $\overline{x}$ makes the clause true. □