Introduction to Algorithms **Problem Set 9 Solutions**
CS 4820 Fall 2018

Proving computability. For each of the following problems, prove whether it is decidable.

(a) *(6 points)* MODIFIES INPUT. Given a deterministic program M and an input x already in memory, does $M(x)$ ever modify x in place? (This is a relevant question for security: you might want to make sure someone can't write to a protected part of memory.)

Solution: The MODIFIES INPUT problem is not decidable.

To prove undecidability: We reduce to the Zero-Input Problem to our problem. Suppose we have a decider `modifiesInput(String M, String x)`. We can write `ZeroChecker(program)` as follows: first, make a new program `wrappedProgram(String` M`, String` x`)` in which `wrappedProgram` first simulates the execution of `program` on the `0` input, without using or modifying the input x at all (using a different space for computation). After this, if the program accepts `0` the program overwrites the first bit of x to change its value (either from 0 to 1, or 1 to 0) (and do not overwrite anything if the program rejects `0`). We set x to be just one bit, say a 0, and our `ZeroChecker` will return the result of `modifiesInput(wrappedProgram, 0)`.

- If `program` accepts on the `0` input, then `wrappedProgram` will reach the instruction where it modifies part of $x = 0$. This means `modifiesInput(wrappedProgram, 0)` must return true.
- If `program` never halts on the`0` input, or rejects this input, then `wrappedProgram` will never reach the instruction where it modifies part of $x = 0$, so `modifiesInput(wrappedProgram, 0)` must return false.

If `modifiesInput` were a program that decided MODIFIES INPUT, this routine above would decide $\mathcal{L}_0$, which is a contradiction. So no such decider can exist, and thus MODIFIES INPUT is undecidable.

Common Mistakes:

- Need to make sure M doesn't modify x while it's running, as the machine may naturally overwrite the input.
- Some students tried to do the reduction the wrong direction (to the known undecidable problem instead of from it). Just like showing a problem is NP hard, to show a problem is undecidable, you reduce from a known undecidable problem to an unknown problem by using a fictional program for the new problem to solve the known problem.
- Some students tried to use Rice's theorem. Modifying input in memory is **not** a semantic property, so Rice's theorem doesn't apply.

(b) *(6 points)* LIMITED MEMORY HALTING. Given a deterministic program M consisting of n instructions, an input x, and an integer c, does $M(x)$ halt having used less than c bits of memory total (where memory is rewriteable)?

Solution: LIMITED MEMORY HALTING is decidable. Consider a program M with n lines of code. There are only 2^c possible configurations of the c bits of memory, and only n lines of code that could be running for a particular memory state, so if $M(x)$ hasn't halted after $n2^c + 1$ time-steps, and didn't use more than c bits of memory, it must have repeated a combination of what line of code it was running and what the state of memory was, which means it is in some infinite loop.

A program that decides this would simulate the execution of M on x for $n2^c+1$ time-steps maximum, checking after each timestep that no more than c memory has been used.

Alternately, the more intensive route: take a program simulator and make it write out the up to c bits of memory used so far and what line of code is being executed (i.e. keep a list of the state of memory at each time-step). If the same line of code + state of memory is seen more than once or the memory limit is exceeded, return "false". This will happen in at most $2^c n + 1$ steps, as this is the total number of possible combinations of memory configurations and lines of code. If the simulated program returns before then without exceeding the memory limit, then return "true".

Common Mistakes:

- Many students had small errors on computing that the number of steps before it must loop is $n2^c + 1$.
- Some students check each instruction separately to find how much memory it allocates. This is not good enough: if the program has loops (or *goto* steps, than allocating 1 bit of memory can occur over and over again, resulting in using lots of memory).
- If we'd like to immediately halt if we ever make $n2^c + 1$ steps, the algorithm needs a time counter to keep track of when this occurs.
- Some students state that a deterministic program is guaranteed to halt, and use this fact to show the problem is decidable: run the problem, and check how much memory was used. This is incorrect - deterministic programs don't necessarily terminate, they just don't have randomness in them. If all deterministic programs terminated, the HALTING PROBLEM would also be decidable.
- Some claim the problem is undecidable by giving a backwards reduction. That is, showing that HALTING $\geq$ LIMITED MEMORY HALTING.

(c) *(6 points)* PROGRAM AGREEMENT. Given two deterministic programs M and M' and one input x, do M and M' agree on input x - i.e. do they both accept, both reject, or both never terminate?

Solution: PROGRAM AGREEMENT is not decidable.

Suppose by way of contradiction that we have a program `comparePrograms(String` M`, String` M'`, String` x`)` that *decides* whether M and M' have the same behavior on input x. Consider the program `neverHalts(input)`, which just loops forever on any input. And now consider running `comparePrograms(program, neverHalts, input)`. This program will decide the `Halting` problem: decide if `program` halts on `input`, which we know is not decidable. This contradiction shows that the `comparePrograms` is not decidable.

Another strategy that works is to pass two wrapped programs in for halting problem inputs M and x: one M' that runs $M(x)$ and then returns false, and another M'' that runs $M(x)$ and then returns true. These two will only agree if $M(x)$ never halts. This works too!

Common Mistakes:

- Attempting to use Rice's Theorem. Agreement between two programs on an input is **not** a semantic property of either program, so Rice's theorem doesn't apply. We could say agreement with M' `alwaysAccepts(input)` is a semantic property of the java program M, but properties applied to two programs at once aren't semantic properties of a single program.
- Like above, but using M' as the `alwaysAccept(input)` program (or the `NeverAccept`). Agreement with M' on x is effectively a correct reduction to the Accept problem with a wrong reference to Rice's theorem, as the problem is about a particular instance of the input x, while Rice's theorem input is just a program.
- Some people reduced from the new problem to an existing undecidable problem, which is backwards.

(d) *(6 points)* ACCEPTS 1 IN INPUT. Given a deterministic program M, does M accept at least one input x that has a 1 in the input string?

Solution: This not decidable.

The easiest proof that this is undecidable is just Rice's Theorem: acceptance of any input x with a 1 as one of its symbols is a *nontrivial property* of a program (e.g., the program that accepts all inputs is in ACCEPTS 1 IN INPUT, and the program that rejects all inputs is not). They may also prove this by reduction from the HALTING PROBLEM or from the ACCEPT PROBLEM or the ZEROINPUT problem, in which case it will look almost exactly like the reduction used for ZEROINPUT.

Common Mistakes:

- Reduction in the wrong direction: people attempted to solve ACCEPTS 1 IN INPUT using the HALTING PROBLEM.
- Forgetting to state the property is semantic or nontrivial: the proof with Rice's theorem is short, but it should still briefly state why this is a nontrivial semantic property.

(e) *(6 points)* QUANTIFIED SAT. Deciding who has a winning strategy in games offer an interesting class of hard problems. Here we consider a very CS-ish game: two players play setting variables in a formula: Player 1 wants to make the formula true, player 2 wants to make it false, and they get to set variables in turn. Here is how the game is played: given a formula Φ with variables $x_1, \ldots, x_n$; player 1 sets variable x_1, then player 2 sets variable x_2, etc. In this game, player 1 has a winning strategy if he can set variable x_1 in a way, that no matter how player 2 sets variable x_2, he then can then set variable x_3, and so on $\ldots$, so that the formula becomes true. The problem of deciding if player 1 has a winning strategy is formalized by the following QUANTIFIED SAT problem.

Consider a SAT formula Φ (in conjugative normal form) with variables $x_1, \ldots, x_n$ with an even n. The QUANTIFIED SAT problem is to decide if the following quantified version of the formula is true

$$\exists x_1 \, \forall x_2 \ldots \exists x_{n-1} \, \forall x_n \; \Phi$$

meaning that there is a setting of x_1 such that for all settings of x_2, there is a setting of x_3, etc. that makes the formula true. For an example, for the formula

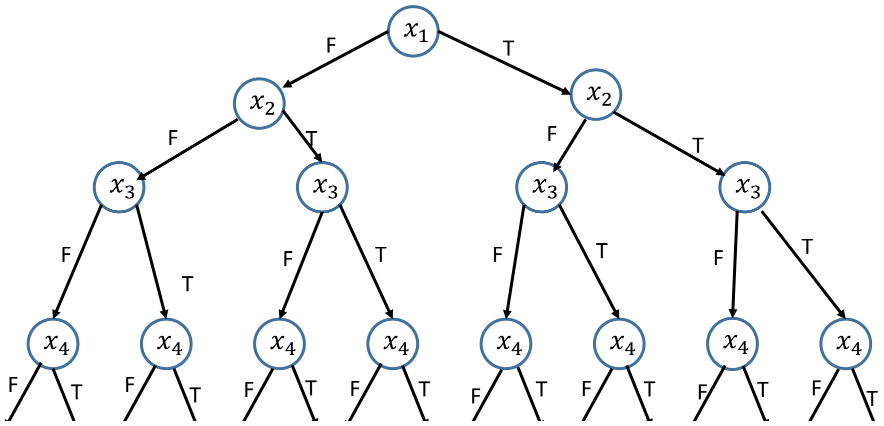
$$\Phi_1 = (x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (x_4 \vee \bar{x}_4)$$

the quantified version is true: by setting $x_1 = T$, no matter what x_2 is set, there is a way to satisfy the formula. In contrast for the formula

$$\Phi_2 = x_1 \wedge (x_2 \vee x_3) \wedge (\bar{x}_3 \vee x_4)$$

is not true: we need to set $x_1 = T$, now if $x_2 = F$, then we must set $x_3 = T$, and now setting $x_4 = F$ gets the formula false.

Solution: This is decidable. There is an exponential size binary decision tree of all the options: the top level is the decision about variable x_1, at level i the decisions about level x_i, as shown on the figure below



At the bottom level of the tree all variables are set, and with each setting we can decide if the formula is true or false. We can then propagate the decision up the tree: at an even level the formula is true if both decisions at one level down lead to true formulas, while at odd levels (when player 1 gets to choose), the formula is true if one of the two decisions at one level down lead to true formulas.

Common Mistakes:

- Many solutions effectively solved the SAT problem in general, instead of the 2-player game version. A satisfying assignment of all variables does not necessarily guarantee a winning strategy for player 1.
- Some students assumed that we can get player 1 to pick values for all x_i variables with odd index i, and then player 2 picks all values for the even variables. Effectively considering the game
$$\exists x_1 \, \exists x_3 \, \ldots \, \exists x_{n-1} \, \forall x_2 \, \forall x_4 \ldots \, \forall x_n \, \Phi$$
This is not equivalent to the alternating move game considered above. For example, if the formula Φ_1, player 1 has a winning strategy in the alternating game as explained above, but would loose in the version of picking all odd variables as the first move.
- Some students only check all possible assignment of True/False, and either forget to explain or incorrectly explain how to combine them (like in the tree explanation above) to determine when player 1 has a winning strategy.