CSC363S 2014 **Assignment A1**
Due: Monday January 25, 11 AM Total Score: 12.5 points

Please submit the assignment at the beginning of class on the due date. If you use *any* sources other than the course textbook or the lecture notes (e.g. other textbooks, online notes, friends) please cite them for your own safety.

Questions

1. Give a detailed description of a Turing Machine M which halts on every input and, when given a string $x \in \{0,1\}^*$, accepts if and only if there are strings $y, z \in \{0,1\}^*$ and a non-negative integer n such that $|y| = |z| = n$ and $x = y1^n z$. *Your Turing Machine must not alter any blank tape squares.* That is, your machine is free to write over the contents of the tape containing the input, but it is not allowed to write any symbol to a blank tape square other than $\sqcup$. In your description of M please give

   (a) A detailed description of the tape alphabet Γ (if you use any extra tape characters), giving the purpose of each extra character you introduce beyond $\Sigma \cup \{\sqcup\}$.

   (b) A detailed description of the movement of the read-write head on an arbitrary input.

   Note that

   $$1^n = \underbrace{11\cdots1}_{n \text{ times}},$$

   and 1^0 is the empty string.

   Solution. In our algorithm we set $\Gamma = \{0, 1, \sqcup, 0', 1', \underline{1}, 0'', \underline{1}''\}$, where $0', 0'', 1', 1'', \underline{1}, \underline{1}''$ are marked versions of the characters in Σ. The marked symbols $0', 1'$ are used in the first phase of the algorithm which locates the center of the string (and thus the substring 1^n, if it exists). The character $\underline{1}$ is used to denote the character in the middle of the input string — in order to accept the middle character must be a 1, so we do not need to add $\underline{0}$. The other marked symbols $0'', 1'', \underline{1}''$ are used in the second phase of the algorithm, which pairs up characters from each section of the string.

   At a high level, our algorithm proceeds across the tape in "sweeps", marking one character from the section containing y, the section containing z, and the section containing 1^n in turn. If we ever get a mismatch then we halt and reject. The main difficulty here is in finding the middle section containing the 1s. To find the middle section we first locate the middle of the string — this is done by pairing up the first and last character, then the second and second-last character, and so on. We mark the center of the string with $\underline{1}$ (technically, the character indexed by $\lfloor n/2 + 1 \rfloor$ will be marked). Then we can mark 1s from the center of the string growing "outwards" (e.g. first the middle character, then left of the middle, then right of the middle, then left, and so on).

   On to the algorithm. If the input tape is blank then immediately accept. Otherwise we begin the first phase to find the center character of the string:

   Phase 1.

   (a) Mark the first character of the string with $'$, proceed to the end of the string and mark it with a $'$. If there is only one character reject.

(b) Repeat the following. Proceed to the first unmarked character in the string and mark it; then proceed to the last unmarked character and mark it. If no unmarked characters remain, reverse the movement just made and check the character under the head. If it is $0'$, then reject. If it is $1'$, then replace it with $\underline{1}$.

It is not hard to see that the character marked with an underline is the $\lfloor n/2+1 \rfloor$st character. If n is odd, then $\lfloor n/2+1 \rfloor$ is the middle character in the input, and so it must be 1. If n is even, then either $n = 0$ or $n = 2$: if $n = 0$ then the tape is blank and it is accepted, and if $n = 2$ then $|x| = 6$, and so the $\lfloor n/2+1 \rfloor$st character must be 1. Now we begin the second phase.

Phase 2.

(a) Scan over the entire string, replacing each $1'$ with 1 and each $0'$ with 0. Leave $\underline{1}$ as it is.

(b) Move back to the beginning of the string, mark the character at the beginning of the string with $''$. Then scan to the end — mark the $\underline{1}$ character with $''$, and finally mark the last character with $''$.

(c) Repeat the following.

i. Move to the beginning of the tape and mark the first unmarked character with $''$ — if no such character exists goto (d).

ii. Move to the end of the tape and mark the last unmarked character with $''$ — if no unmarked character exists, reject.

iii. Move to $\underline{1}''$.

iv. If this is an odd iteration, scan to the **left**. If the first unmarked character is not 1, or if there is no unmarked character, reject. Otherwise, mark it with $''$ and go to (c) (i.e. repeat).

v. Else, if this is an even iteration, scan to the **right**. If the first unmarked character is not 1 or if there is no unmarked character, reject. Otherwise, mark it with $''$ and go to (c).

(d) Scan over the entire string. If all characters are marked with $''$, accept. Otherwise reject.

In each successful sweep of step (c), the algorithm marks one character from each section of the tape. The algorithm accepts if all sweeps succeed and every character in the string is marked, which is clearly equivalent to partitioning x into $y1^n z$.

2. In this question we consider another variant of Turing Machine that errors in a regular pattern when it writes a symbol to the tape. Define a *Faulty Turing Machine* to be a Turing Machine M which writes an incorrect symbol to the tape every ten steps of the computation. More precisely, the machine M operates like so. Initially, the input $x \in \Sigma^*$ is written on the tape. For each positive integer $i = 1, 2, \ldots$ when the machine M is about to perform the $10i^{\text{th}}$ transition and write the symbol γ to the tape, it instead chooses an arbitrary symbol $\gamma' \in \Gamma$ and writes γ' to the tape instead. The read-write head will still move to the left or right, as usual, without error. Acceptance of an input and the language computed by a Faulty Turing Machine are defined as usual.

Show that Faulty Turing Machines can simulate regular Turing Machines. That is, for every Turing Machine M show that there is a Faulty Turing Machine M' such that $\mathcal{L}(M') = \mathcal{L}(M)$ and for every input $x \in \Sigma^*$ the machine M halts on x if and only if M' halts on x. In your simulation, please give

(a) A detailed description of the tape alphabet Γ (if you use any extra tape characters), giving the purpose of each extra character you introduce beyond $\Sigma \cup \{\sqcup\}$.

(b) A detailed description of the movement of the read-write head on an arbitrary input.

Solution. There are many ways to approach this problem. We choose to simulate the machine M by a faulty machine M' which is "clocked": every ten steps when the machine M' errors on its write, M' breaks out of its simulation of M, reverse back to the cell it just wrote and repair it, and then resumes its simulation. We will not use any extra tape symbols other than those used by M.

The algorithm for M' is very simple. Let M be any regular Turing Machine. Our faulty machine M' simulates M but on a ten-step clock — since ten is a fixed constant, we can encode this clock into the state machine of M'. On every tenth transition of M' the machine writes an incorrect symbol to the tape and moves the head in the correct direction. Suppose that the machine was supposed to write the symbol γ to the tape in this transition. At this point, M' will break out of its simulation of M and perform the following algorithm:

Local Correction

(a) If the head just moved to the left, move the head back to the right, write γ to the tape square and move left again. Otherwise, if the head just moved right, move to the left, write γ to the tape square, and move right again.

(b) Resume the simulation of M.

The machine M' will continue to clock the moves it takes during the local error correction since it makes an error every ten transitions, regardless of its actions. Since the local fix only requires two transitions the machine M' will not make any errors during the local correction step. Clearly all local errors will be fixed, so the simulation of M will proceed correctly and it follows that $\mathcal{L}(M')$ is well defined and equal to $\mathcal{L}(M)$. Furthermore, the machine M' halts if and only if M halts since it is essentially simulating M.

3. Now define an *Extra Faulty Turing Machine* to be a Turing Machine M which writes an incorrect symbol to the tape every *two* steps of the computation. More precisely, the machine M operates like so. Initially, the input $x \in \Sigma^*$ is written on the tape. For each positive integer $i = 1, 2, \ldots$ when the machine M is about to perform the $2i^{\text{th}}$ transition and write the symbol γ to the tape, it instead chooses an arbitrary symbol $\gamma' \in \Gamma$ and writes γ' to the tape instead. The read-write head will still move to the left or right, as usual, without error. Acceptance of an input and the language computed by a Extra Faulty Turing Machine are defined as usual.

Let M be any Turing Machine which halts on every input and has input alphabet Σ. Using M, show that there is an Extra Faulty Turing Machine which accepts the language

$$L = \{\sigma_1 b_1 \sigma_2 b_2 \cdots \sigma_{n-1} b_{n-1} \sigma_n \mid n \in \mathbb{N}, \sigma_1 \sigma_2 \cdots \sigma_n \in \mathcal{L}(M), b_1, b_2, \ldots, b_n \in \Sigma\}.$$

In your simulation, please give

(a) A detailed description of the tape alphabet Γ (if you use any extra tape characters), giving the purpose of each extra character you introduce beyond $\Sigma \cup \{\sqcup\}$.

(b) A detailed description of the movement of the read-write head on an arbitrary input.

Solution. Let M be any Turing Machine. Our Extra Faulty Turing Machine M' is very simple: it operates exactly as M does, except it repeats every transition of M twice, ignoring the character under the read-write head on the second transition. For example, if M was to write γ to the tape and move the head to the left, then M' will write γ to the tape, move to the left, ignore the character under the head and write γ again, and then move to the left again. Intuitively M' is simulating M on every other square of the tape — this is why it accepts any string in $\mathcal{L}(M)$ which has arbitrary characters interleaved into the string.

Now we formally prove that M' accepts the language L described above. Suppose that $x = \sigma_1\sigma_2\cdots\sigma_n \in \mathcal{L}(M)$, and let $b_1, b_2, \ldots, b_n \in \Sigma$ be arbitrary characters. We show that $y = \sigma_1 b_1 \sigma_2 b_2 \cdots \sigma_{n-1} b_{n-1} \sigma_n \in \mathcal{L}(M')$.

As described above, the machine M' simulates M with two transitions for every single transition. Since M' makes an error every second transition, it follows that M' makes a correct transition every second transition. Thus, it is correctly simulating M on every other square of the tape. The string $x = \sigma_1\sigma_2\cdots\sigma_n$ is encoded on every other square of the input tape of M', and so M' accepts the input y if and only if M accepts the input x. This holds for every possible choice of characters $b_1, b_2, \ldots, b_n$ interleaved into the string y, and so it follows that $\mathcal{L}(M') = L$.

4. Let $M_0, M_1, M_2, \ldots, M_i, \ldots$ be an enumeration of all Turing Machines in some fixed computable order — that is, there is a computable function $g : \mathbb{N} \to \{0,1\}^*$ such that $g(i) = \langle M_i \rangle$ for all i. Let $f : \mathbb{N} \to \mathbb{N}$ be any one-to-one, computable function. Give a careful proof that the following language *is* semi-decidable:

$$L = \{\langle i \rangle \mid i \in \mathbb{N} \text{ and for all } j \leq f(i), \langle M_j \rangle \in \mathcal{L}(M_i)\}.$$

You may use the Church-Turing thesis when defining your algorithm, but be sure to use results given in class where necessary when justifying the correctness of your algorithm. Then prove that L is not decidable.

Solution.

Here is a Turing Machine M accepting the language L, described using the Church-Turing thesis.

Algorithm for L

(a) On input $\langle i \rangle$, simulate the algorithm for g to obtain the description of the machine $\langle M_i \rangle$.

(b) For each $j \leq f(i)$ (note that $f(i)$ can be computed from $\langle i \rangle$)

  i. Run the algorithm for g to compute the TM description $\langle M_j \rangle$

  ii. Simulate the machine M_i on $\langle M_j \rangle$ using the Universal Turing Machine simulation.

  iii. If any simulation rejects, halt and reject.

(c) Halt and accept.

This algorithm clearly halts and accepts if and only if $\langle M_j \rangle \in \mathcal{L}(M_i)$ for all $j \leq f(i)$; thus, L is semi-decidable.

Proving that L is not decidable is much more tricky. We do this by contradiction. Assume that L is decidable and let $\hat{M}$ be the Turing Machine deciding L. If this is true, then the following algorithm decides A_{TM}. Let $(\langle M \rangle, x)$ be any well-formed input to A_{TM}, and consider the Turing Machine M' specified by the following algorithm.

Special TM M'

(a) On input y, check if $y = \langle M_j \rangle$ is an encoding of a Turing Machine. If not, reject.

(b) Write the encoding $\langle M' \rangle$ of M' on the tape.

(c) Let $i \in \mathbb{N}$ be chosen so that $\langle M' \rangle = g(i)$. This can be computed by enumerating $g(i)$ for each i, and checking if $g(i) = \langle M' \rangle$. Since g enumerates all Turing Machines, this step eventually halts.

(d) Similarly, compute $j \in \mathbb{N}$ such that $g(j) = \langle M_j \rangle = y$. If $j < f(i)$, accept.

(e) Otherwise, if $j = f(i)$, simulate $\langle M \rangle$ on x and accept or reject if the simulation accepts or rejects, respectively.

For each pair $(\langle M \rangle, x)$ there is a machine M' above. Using M' we can decide A_{TM}.

Algorithm for A_{TM}

(a) On input $(\langle M \rangle, x)$, reject if $\langle M \rangle$ is not a well-formed Turing Machine.

(b) Construct the Turing Machine M' from $(\langle M \rangle, x)$ above.

(c) Calculate i such that $g(i) = \langle M' \rangle$, and simulate $\hat{M}$ on $\langle i \rangle$.

By the construction of M', if $\hat{M}$ accepts $\langle i \rangle$ then for all $j \leq f(i)$, $\langle M_j \rangle \in \mathcal{L}(M_i) = \mathcal{L}(M')$. But by the definition of M', this implies that $\langle M \rangle$ accepts x, and thus $(\langle M \rangle, x) \in A_{\mathrm{TM}}$.

On the other hand, if $\hat{M}$ rejects $\langle i \rangle$, then there exists a $j \leq f(i)$ such that $\langle M_j \rangle \notin \mathcal{L}(M')$. But by construction, M' must accept $\langle M_j \rangle$ for all $j < f(i)$. It follows that M' does not accept $\langle M_{f(i)} \rangle$, which implies by the definition of M' that $\langle M \rangle$ does not accept x. Thus $(\langle M \rangle, x) \notin A_{\mathrm{TM}}$.

CSC363S 2014 **Assignment A2**
Due: Monday February 8, 11 AM Total Score: 12.5 points

Please submit the assignment at the beginning of class on the due date. If you use *any* sources other than the course textbook or the lecture notes (e.g. other textbooks, online notes, friends) please cite them for your own safety.

Questions

1. In this question we give an alternative proof that *nondeterministic* Turing Machines compute exactly the languages in SD. (For the definition of a nondeterministic Turing Machine, check the Sipser book or the notes from Tutorial 2).

   Fix an alphabet Σ. Give a **direct** proof that for any language L, there is a nondeterministic Turing Machine M such that $L = \mathcal{L}(M)$ if and only if there is a computable relation $R \subseteq \Sigma^* \times \Sigma^*$ such that

   $$x \in L \Leftrightarrow \exists y \in \Sigma^* : R(x, y).$$

   Solution. Let M be a nondeterministic Turing Machine computing L, and we show that there exists a decidable relation $R \subseteq \Sigma^* \times \Sigma^*$ such that

   $$x \in L \Leftrightarrow \exists y \in \Sigma^* : R(x, y).$$

   Our relation R mimics the definition of R in the version of the theorem where M is deterministic. For each $x, y \in \Sigma^* \times \Sigma^*$ let $R(x, y)$ hold if and only if y encodes a consistent sequence of configurations in any of the possible accepting computations of M on x. Equivalently, we can think of the certificate y as encoding an accepting path in the computation tree produced by M on input x. Given such a certificate y, it is easy to check if it is of the prescribed form: we just cross-reference each configuration in the sequence encoded by y with the transition relation δ of M, checking that each one in the sequence is yielded from the previous configuration in the sequence by a legal transition included in δ. For any $x \in \Sigma^*$, if x is in L then there is an accepting computation of M on x, and so we take y to be the sequence of accepting configurations in any of the possible accepting paths. Conversely, if $R(x, y)$ holds, then there must exist an accepting computation of M on x just by taking the computation encoded by y.

   Now, suppose that such a decidable relation R exists, and we construct a nondeterministic Turing Machine M. The definition of M is even easier than in the deterministic case: just non-deterministically guess a string $y \in \Sigma^*$ and check if $R(x, y)$ holds. By definition of R, $x \in L$ if and only if there is a $y \in \Sigma^*$ which will make this non-deterministic procedure accept.

2. Let $\Sigma = \{0, 1\}$. For each of the following languages $L \subseteq \Sigma^*$, classify L with respect to $\mathsf{D}, \mathsf{SD}, \mathsf{coSD}$. That is, for each language L and for each class $\mathsf{C} \in \{\mathsf{D}, \mathsf{SD}, \mathsf{coSD}\}$, prove that L is in C or that L is not in C. **You may not use Rice's Theorem.**

   (a) $L_1 = \{(\langle M \rangle, \langle i \rangle) \mid M \text{ is a TM}, i \in \mathbb{N}, \text{ M accepts all strings of length } i\}$

   (b) $L_2 = \{(\langle M \rangle, x) \mid M \text{ is a TM and } M \text{ halts on } x \text{ with } 11111 \text{ written on the tape}\}$

   (c) $L_3 = \{\langle M \rangle \mid M \text{ is a TM and } \exists i \in \mathbb{N} : \text{M accepts all strings of length } i\}$

(d) $L_4 = \{\langle M\rangle \mid M$ is a TM and there is an $x \in \Sigma^*$ beginning with 0 such that M does not accept $x\}$

Solution.

The language L_1 is in SD, but it is not in D or coSD.

It is easy to see that L_1 is in SD by the following algorithm: on input $(\langle M\rangle, \langle i\rangle)$ just simulate M on each of the 2^i strings of length i. If all such simulations accept, then accept, otherwise if any simulation rejects, then reject. It is important to note that some of the simulations may not halt (and we exploit exactly this fact to show that it is not in D or coSD).

We reduce A_{TM} to L_1. Let $(\langle M\rangle, x)$ be any TM-string pair, and consider the algorithm $M' = M'(M, x)$ defined as follows:

Algorithm for M'

(a) On input $y \in \Sigma^*$

(b) If $|y| = 1$, then simulate M on x and accept if and only if it accepts.

(c) Else if $|y| \neq 1$ then reject.

The reduction maps the pair $(\langle M\rangle, x) \mapsto (\langle M'(M, x)\rangle, \langle 1\rangle)$. The encoding of $M'(M, x)$ is easily computable from $\langle M\rangle$ and x, and so we focus on showing

$$(\langle M\rangle, x) \in A_{\text{TM}} \Leftrightarrow (\langle M'(M, x)\rangle, \langle 1\rangle) \in L_1.$$

If $(\langle M\rangle, x) \in A_{\text{TM}}$ then by the definition of M' we have that M' accepts all strings of length one. Thus $(\langle M'\rangle, \langle i\rangle) \in L_1$. Conversely, if $(\langle M'\rangle, \langle i\rangle) \in L_1$, then M' accepts all strings of length one, which immediately implies that M accepts x by the definition of M'. Thus $(\langle M\rangle, x) \in A_{\text{TM}}$, and $A_{\text{TM}} \leq_m L_1$. Since A_{TM} is not in D or coSD we immediately get that L_1 is not in D or coSD.

The language L_2 is in SD, but it is not in D or coSD.

To show that L_2 is in SD we give an algorithm.

Algorithm for L_2

(a) On input $y \in \Sigma^*$

(b) If y does not encode a pair $(\langle M\rangle, x)$ where $\langle M\rangle$ is a TM, then reject.

(c) Simulate M on x. If M halts, and if the tape has 11111 written when the simulation has halted, then accept. Otherwise reject.

The algorithm for L_2 clearly accepts an input $(\langle M\rangle, x)$ if and only if M halts on x with 11111 written on the tape, and so $L_2 \in$ SD. We show that L_2 is not in D or coSD by a reduction from A_{TM}.

Let $(\langle M\rangle, x)$ be any pair, and consider the algorithm $M' = M'(M, x)$ defined as follows.

Algorithm for M'

(a) On input $y \in \Sigma^*$.

(b) If $y = 0$, simulate M on x. If M accepts x, blank the tape, write 11111, and halt. Otherwise if M rejects x, loop forever.

The reduction maps $(\langle M \rangle, x) \mapsto (\langle M'(M, x) \rangle, 0)$. We focus on proving

$$(\langle M \rangle, x) \in A_{\mathrm{TM}} \Leftrightarrow (\langle M'(M, x) \rangle, 0) \in L_2.$$

If M accepts x, then by definition the machine M' on input 0 will halt with 11111 written on the tape. Thus $(\langle M \rangle, x) \in A_{\mathrm{TM}}$ implies that $(\langle M' \rangle, 0) \in L_2$. On the other hand, if $(\langle M' \rangle, 0) \in L_2$ it follows that M' must halt on 0 with 11111 written on the tape. If so, by the definition of M' we have that M accepts x, and so $(\langle M \rangle, x) \in A_{\mathrm{TM}}$. This shows $A_{\mathrm{TM}} \leq_m L_2$ and so since A_{TM} is not in D or coSD, neither is L_2.

The language L_3 is in SD, but it is not in D or coSD.

To show that L_3 is in SD we give an algorithm using dovetailing.

Algorithm for L_3

(a) On input $y \in \Sigma^*$

(b) If y does not encode a Turing Machine $\langle M \rangle$ then halt and reject.

(c) For each $j \in \mathbb{N}$.

  i. For each $i \in \mathbb{N}$.

    A. Simulate M over all inputs of length i for j steps. If all simulations of M halt and accept after j steps, then accept. Otherwise reject.

By definition of the algorithm above, the algorithm accepts an input $\langle M \rangle$ if and only if for some $i \in \mathbb{N}$ the machine M accepts all inputs of length i. Thus $L_3 \in \mathsf{SD}$.

To show that L_3 is not in D or coSD, we give a reduction from (you guessed it) A_{TM}. Let $(\langle M \rangle, x)$ be a Turing Machine-input pair, and consider the machine $M' = M'(M, x)$ defined by the following algorithm.

Algorithm for M'

(a) On input $y \in \Sigma^*$.

(b) Simulate M on x, and accept if and only if M accepts x.

The reduction maps $(\langle M \rangle, x) \mapsto \langle M'(M, x) \rangle$. If M accepts x then by definition M' accepts all inputs y, and so $\langle M' \rangle \in L_3$. Conversely, by the definition of M', if there exists an i such that M' accepts all inputs of length i then M' actually accepts *all* inputs, and moreover M must accept x. Thus $(\langle M \rangle, x) \in A_{\mathrm{TM}}$. This means that $A_{\mathrm{TM}} \leq_m L_3$, and so L_3 is not in coSD or D.

The language L_4 is not in SD or coSD

We first show that L_4 is not in coSD by a reduction from A_{TM}. Let $(\langle M\rangle, x)$ be any TM/string pair, and consider the machine $M' = M'(M, x)$ defined by the following algorithm.

Algorithm for M'

(a) On input $y \in \Sigma^*$.

(b) If y does not begin with 0, reject.

(c) If y begins with 0, simulate M on x for $|y|$ steps. If M accepts x after $|y|$ steps of simulation, halt and reject. Otherwise halt accept.

The reduction maps $(\langle M\rangle, x) \mapsto \langle M'(M, x)\rangle$, and is easily seen to be computable. If M accepts x then M' rejects y for some y beginning with 0 by the definition of M'. Thus if $(\langle M\rangle, x) \in A_{\mathrm{TM}}$ then $\langle M'\rangle \in L_4$. Conversely, if there is a string y beginning with 0 such that M' does not accept y, then it follows that M must halt and accept x after $|y|$ steps of simulation. This means that $(\langle M\rangle, x) \in A_{\mathrm{TM}}$, and so $A_{\mathrm{TM}} \leq_m L_4$. In particular L_4 is not in coSD.

Now we show that L_4 is not in SD by a reduction from

$$\overline{A}_{\mathrm{TM}} = \{(\langle M\rangle, x) \mid \langle M\rangle \text{ does not encode a TM or } x \notin \mathcal{L}(M)\}\,.$$

Let $(\langle M\rangle, x)$ be any TM/string pair, and consider the following machine $M^* = M^*(M, x)$ defined by the next algorithm.

Algorithm for M^*

(a) On input $y \in \Sigma^*$.

(b) If $\langle M\rangle$ is not an encoding of a TM and y begins with 0, reject.

(c) Otherwise, if y begins with 0 simulate M on x for $|y|$ steps. If M accepts x after $|y|$ steps of simulation, accept. Otherwise reject.

The reduction maps $(\langle M\rangle, x) \mapsto \langle M^*(M, x)\rangle$, and is easily seen to be computable. If $(\langle M\rangle, x) \in \overline{A}_{\mathrm{TM}}$ and $\langle M\rangle$ is not a well-defined Turing Machine, then by definition $\langle M^*\rangle \in L_4$. So assume that $\langle M\rangle$ is well-defined. If M does not accept x then M^* accepts all y beginning with 0, and so $\langle M^*\rangle \in L_4$. Thus, in either case $(\langle M\rangle, x) \in \overline{A}_{\mathrm{TM}}$ implies $\langle M^*\rangle \in L_4$.

On the other hand, suppose that $\langle M^*\rangle \in L_4$. It follows that M^* either rejects a y beginning with 0 at step (b) or step (c), by definition of M^*. In the first case $\langle M\rangle$ is not a well-defined TM, and so $(\langle M\rangle, x) \in \overline{A}_{\mathrm{TM}}$. In the second case, by the definition of M^* we have that M must never accept the input x, and so we have $(\langle M\rangle, x) \in \overline{A}_{\mathrm{TM}}$. We now have $\overline{A}_{\mathrm{TM}} \leq_m L_4$, and so in particular L_4 is not SD.

CSC363S 2016 **Assignment A3**

Due: ~~Monday, March 7th~~ Monday, March 14th, 11 AM Total Score: 12.5 points

Please submit the assignment at the beginning of class on the due date. If you use *any* sources other than the course textbook or the lecture notes (e.g. other textbooks, online notes, friends) please cite them for your own safety.

Questions

1. In complexity theory we think of P as the polynomial-time analogue of D, NP as the polynomial-time analogue of SD, and coNP as the polynomial-time analogue of coSD. In computability theory, it is not too hard to prove $\mathsf{D} = \mathsf{SD} \cap \mathsf{coSD}$, however, the analogical statement in complexity theory

$$\mathsf{P} = \mathsf{NP} \cap \mathsf{coNP}$$

is an open research problem! Let's study this a bit more closely.

Let PR be the class of *polynomial-time recognizable languages*. That is, a language L is in PR if and only if there is a Turing Machine M such that for all $x \in L$, M halts and accepts x in polynomial time, but for $x \notin L$ the machine M either rejects (with no bound on the running time) or does not halt. A language L is in coPR if and only if its complement $\overline{L} \in \mathsf{PR}$.

(a) Show that $\mathsf{P} = \mathsf{PR} \cap \mathsf{coPR}$.

(b) Show $\mathsf{PR} \subseteq \mathsf{NP}$ and $\mathsf{coPR} \subseteq \mathsf{coNP}$.

(c) Discuss whether or not $\mathsf{NP} \subseteq \mathsf{PR}$ and $\mathsf{coNP} \subseteq \mathsf{coPR}$ (proofs are not necessary — just give your intuition!)

Solution. We begin by showing that $\mathsf{P} = \mathsf{PR} \cap \mathsf{coPR}$. Note that $\mathsf{P} \subseteq \mathsf{PR}$ and $\mathsf{P} \subseteq \mathsf{coPR}$, so we focus on showing $\mathsf{PR} \cap \mathsf{coPR} \subseteq \mathsf{P}$. The most obvious way to proceed is to mimic the proof that $\mathsf{D} = \mathsf{SD} \cap \mathsf{coSD}$. Let $L \in \mathsf{PR} \cap \mathsf{coPR}$, let M_1 be the polynomial-time recognizer for L, and let M_0 be the polynomial-time recognizer for $\overline{L}$. Consider the following algorithm M for L:

Algorithm for M.

(a) On input y.

(b) Simulate M_0 and M_1 on y in parallel. If M_1 halts and accepts y then halt and accept. If M_0 halts and accepts y then halt and reject.

Since every string y is in either L or $\overline{L}$ it follows that the above algorithm halts on all inputs; moreover, since both machines are polynomial time it follows that the time complexity is at most twice the maximum time complexity of both M_0 and M_1, which is polynomial. It is clear that the machine M computes the language L exactly, and so $L \in \mathsf{P}$.

To show $\mathsf{PR} \subseteq \mathsf{NP}$, we follow the proof of Theorem 2.10 in the lecture notes (i.e. the certificate definition of SD). Let $L \in \mathsf{PR}$, let M be the polynomial-time recognizer for L, and we construct a verifier V for L. Intuitively, on input (x, y), the verifier V checks that y encodes an accepting computation of M on x; if so, V accepts, and otherwise V rejects.

Verifier V for L

(a) On input (x, y).

(b) Check that y encodes an accepting computation of M on input x. If so, accept. Otherwise reject.

This algorithm clearly runs in polynomial time, since we have efficient encodings of Turing Machine configurations and since every accepting computation of M will have length polynomial in $|x|$. Moreover, the string x is in the language L if and only if the machine M accepts x, and thus if and only if there is a string y encoding an accepting computation of M on x. It follows that V is a polynomial-time verifier for L, and thus $L \in \mathsf{NP}$.

Now we show that $\mathsf{coPR} \subseteq \mathsf{coNP}$. We can reduce this proof to our argument above:

$$L \in \mathsf{coPR} \Rightarrow \overline{L} \in \mathsf{PR} \Rightarrow \overline{L} \in \mathsf{NP} \Rightarrow L \in \mathsf{coPR}.$$

Now consider the question of whether or not $\mathsf{NP} \subseteq \mathsf{PR}$. In the proof of Theorem 2.10, the analogous statement (showing that a language L has an unbounded verifier implies that L has a recognizer) is performed by dovetailing over all possible certificates y. But clearly it is impossible to dovetail over all polynomial-length certificates y in polynomial time since there are exponentially many of them!

2. Let $G = (V, E)$ be a *directed* graph. A *directed clique* in G is a subset $V' \subseteq V$ such that for all $u, v \in V'$ there are two directed edges $uv \in E$ and $vu \in E$. The *Directed Clique* problem is the following: given a directed graph G and a positive integer k, decide if G contains a directed clique of size at least k.

   Show that $\mathrm{Clique} \leq_p \mathrm{DirectedClique}$ and $\mathrm{DirectedClique} \leq_p \mathrm{Clique}$.

   Solution. First let us reduce Clique to $\mathrm{DirectedClique}$. Let $G = (V, E)$ be a graph and let k be a positive integer. Let G' be the directed graph on the same set of vertices V which is obtained by replacing each edge $uv \in E$ with two directed edges (u, v) and (v, u), and let $k' = k$. Clearly G' can be constructed in polynomial time from the definition of G since we just need to examine each edge in G once.

   Now we prove correctness. If $V' \subseteq V$ is a clique in G with k vertices, then the same set of vertices is a directed clique in G' by construction. The converse follows symmetrically. Thus $\mathrm{Clique} \leq_p \mathrm{DirectedClique}$.

   Now, let $G = (V, E)$ be a directed graph and let k be a positive integer. Let G' be the graph on the same set of vertices, where for each pair of vertices u, v in V we add an edge uv if both directed edges (u, v) and (v, u) are present in G. Similarly set $k' = k$. This is again efficient, and as before, it is clear that each directed clique in G maps in a one-to-one way with each clique in G', and thus $\mathrm{DirectedClique} \leq_p \mathrm{Clique}$.

3. Consider the following *Clique-Independent Set* problem. Given graph $G = (V, E)$ with n vertices and a positive integer $k \leq n/2$, decide if G contains both a clique of size at least k and an independent set of size at least k.

   Show that $\mathrm{CliqueIndependentSet}$ is NP-Complete.

   Solution. First we show that Clique-Independent Set is in NP by giving a polynomial-time verifier.

Verifier for Clique-Independent Set

(a) On input $(G, k), y$.

(b) Check that y encodes two disjoint subsets of vertices V_0, V_1 in G with $|V_0|, |V_1| \geq k$. If not, reject.

(c) For each pair of vertices u, v in V_0, check that $uv \in E$; if not, reject.

(d) For each pair of vertices u, v in V_1 check that $uv \notin E$; if not, reject.

(e) Accept.

This algorithm clearly runs in polynomial time: checking that y encodes two subsets of vertices can be done efficiently, and checking that V_0 is a clique and V_1 is an independent set requires $O(|E|^2)$ time in the worst case. Moreover, it is clear that the algorithm accepts (G, k) and y if and only if y encodes both an independent set and a clique in G with at least k vertices each. Therefore G contains both a clique of size at least k and an independent set of size at least k if and only if there is a certificate y making the verifier accept, and so $\mathrm{CliqueIndependentSet} \in \mathsf{NP}$.

Now we show Clique-Independent Set is NP-Hard by a reduction from the Clique problem. Let (G, k) be a tuple containing a graph G and a positive integer k, and we construct a pair (G', k') such that G contains a clique with at least k vertices if and only if G' contains a clique and an independent set with at least k' vertices each. The reduction is quite straightforward: set $k' = k$, and let G' be the graph obtained by adding k isolated vertices to G. If G contains a clique V' with at least k vertices, then V' and the k isolated vertices form a clique-independent set pair in G' with k vertices apiece. Similarly, if G' contains a clique and an independent set of size at least k, it is clear that G contains a clique of size at least k. Moreover the graph G' can be easily constructed in polynomial time given the description of G, thus the reduction is complete.

4. Let m and n be positive integers and consider the following game – called Pebble Up – played on an $n \times n$ grid. We will denote the square in the ith row of the grid and the jth column of the grid by the pair (i, j). For each $1 \leq i, j \leq n$, the square (i, j) will be given a *colour* (either black or white) and a *number* (some positive integer $k \leq m$). The grid, along with a given colour and integer for each square, will be referred to as a *game board*.

   Pebble Up is played on the game board as follows. You have a bag of black and white pebbles, and for each integer k between 1 and m, you must choose to place either a black or a white pebble on all of the squares labelled with the integer k (so, squares with different numbers can have different coloured pebbles, but if two squares have the same number then they must have the same coloured pebble). After choosing a coloured pebble for each integer k, the game moves on to the next phase.

   For each square (i, j), with $1 \leq i, j \leq n - 1$, examine the set of four squares

$$S_{ij} = \{(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)\}.$$

   This set of four squares is said to be *winning* if at least one of squares has the same colour as the pebble lying on it. You *win* the game if every set S_{ij} is winning.

   Figure 1 depicts a game board (with $n = 3, m = 5$) along with two configurations of pebbles. The middle configuration is a winning configuration: any set S_{ij} of four squares has at least one pebble which has the same colour as its square. If we switch the colour of the "5" pebble from

white to black, we get the configuration on the right. This is not a winning configuration since the set S_{12}, consisting of the four squares in the top-right corner of the game board, has each square and pebble differing in colour.

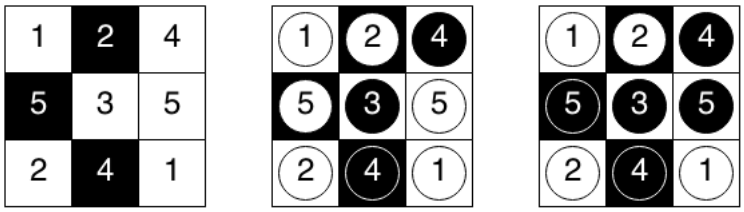

Figure 1: A game board and two possible configurations of pebbles

Associated with Pebble Up is the following decision problem: As input, the algorithm receives two positive integers m, n, and for every (i, j) a colour b_{ij} (either black or white) and a positive integer $k_{ij} \leq m$. The goal is to decide if it is possible to win the Pebble Up game on the given game board (that is, to decide if there exists an assignment of pebbles to each of the numbers in a winning configuration).

Prove that (the decision problem associated with) Pebble Up is NP-Complete.

First we show that Pebble-Up is in NP by giving a polynomial-time verifier for it. The certificate for the Pebble-Up problem will be interpreted by the verifier as an assignment of pebbles to squares of the board. The verifier then iteratively checks all sets of 4 squares of the board (which takes $O(n^2)$ time) to see if this assignment of pebbles causes each set to be winning. If so, the verifier returns 1, and otherwise it returns 0.

Now we prove that Pebble-Up is NP-Hard by a reduction from 4-SAT. Let $C_1, C_2, \ldots, C_m$ be a collection of clauses, each containing at most four literals, defined over the variables $x_1, x_2, \ldots, x_n$. The first step is to pad each of the clauses in the input which have less than 4 variables. This is easy: introduce 6 boolean variables $h_1, t_1, t_2, o_1, o_2, o_3$. We use h_1 to pad the clauses with 3 variables, t_1 and t_2 to pad the clauses with 2 variables, and o_1, o_2, o_3 to pad the clauses with 1 variable. If $C = z_1 \vee z_2 \vee z_3$ is a clause with three variables, we replace it with the two clauses

$$(z_1 \vee z_2 \vee z_3 \vee h_1) \wedge (z_1 \vee z_2 \vee z_3 \vee \neg h_1),$$

and both of these clauses are satisfiable if and only if the original clause is. Similarly, if $C = z_1 \vee z_2$ then we replace it with four clauses

$$(z_1 \vee z_2 \vee t_1 \vee t_2) \wedge (z_1 \vee z_2 \vee \neg t_1 \vee t_2) \wedge (z_1 \vee z_2 \vee t_1 \vee \neg t_2) \wedge (z_1 \vee z_2 \vee \neg t_1 \vee \neg t_2).$$

These four clauses are satisfiable if and only if the original clause is. If C only has one variable, we replace it with 8 clauses in the same way (by adding every combination of the three variables o_1, o_2, o_3 to C). The resulting 4-SAT instance will have at most $8m$ clauses (each with 4 literals) and 6 more variables, which is certainly within a polynomial of the size of the original instance.

The idea of the reduction is as follows. We construct the game board so that for each clause $C = z_1 \vee z_2 \vee z_3 \vee z_4$, there will exist a set of four squares $S_{ij} = \{(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)\}$ such that the clause C can be satisfied if and only if there is a winning assignment of pebbles to S_{ij}. For each z_i, let ℓ_i be the label of z_is variable (so, if $z_i = \neg x_3$, then $\ell_i = 3$) and let c_i be black

if z_i is negated and white otherwise. We define

$$\begin{aligned} b_{ij} &= c_1, & k_{ij} &= \ell_1 \\ b_{i,j+1} &= c_2, & k_{i,j+1} &= \ell_2 \\ b_{i+1,j} &= c_3, & k_{i+1,j} &= \ell_3 \\ b_{i+1,j+1} &= c_4, & k_{i+1,j+1} &= \ell_4 \end{aligned}$$

Figure 2 presents this reduction. In this way, there is a natural correspondence between assignments to the variables in the clause and pebbles placed on squares: black pebbles correspond to a variable being assigned 0, and white pebbles correspond to a variable being assigned a 1.

Figure 2: Reducing a clause to a set of 4 squares

To prevent neighbouring clauses from conflicting with one another, we further pad the game board with extra rows and columns that separate the clauses. We fix the same number in each square in all separating rows and columns, and alternate colours. Such a padding is shown in Figure 3 (where C_1, C_2, C_3, and C_4 are separated clauses). With the separating rows and columns labelled

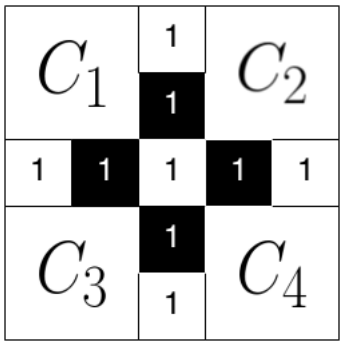

Figure 3: Padded clauses

in this way, all of the intermediate sets of 4 squares which do not correspond to any fixed clause must overlap a separating row or column in at least 2 squares, and so it will be satisfied for any choice of pebble on the number.

We have shown how to construct each individual set of 4 squares from each clause, and also how to separate the clauses. Constructing the board is now simple: we are embedding m clauses into a square grid. Let $k = 2\lceil\sqrt{m}\rceil - 1$, and the game board will be a k by k grid. We can embed each of the clauses into the grid in any order, placing a separating row and column in between the clauses as described above. If there is leftover "room" in the grid, we pad it with more squares having a fixed number and alternating colour so that they will always be winning. This reduction can be performed in $O(k^2) = O(m^2)$ time.

Proving the reduction is correct is easy. Each number on the game board corresponds to a variable in the 4-SAT instance, and the colour of a pebble placed on all squares with that number corresponds exactly to an assignment to that variable. Assume that if a number receives a black pebble, the corresponding variable is set to 0 and vice-versa. Otherwise the variable is set to 1. If there is a satisfying assignment to the 4-SAT instance, we can construct a pebbling of the game board in exactly the way described above. Each set of 4-squares is either vacuously satisfied (as they are the result of padding), or correspond exactly to one of the 4-SAT clauses and thus must be satisfied. Conversely, if there is a winning configuration of pebbles on the game board, then we can construct a satisfying assignment using the method above and the 4-SAT instance must also be satisfied.

CSC363S 2016 **Assignment A4**
Due: ~~Monday, March 21th~~ Monday, March 28th, 11 AM Total Score: 12.5 points

Please submit the assignment at the beginning of class on the due date. If you use *any* sources other than the course textbook or the lecture notes (e.g. other textbooks, online notes, friends) please cite them for your own safety.

Questions

1. Let k be any fixed positive integer, and consider the following restriction of the Clique problem.

Problem. k-Clique **Input**: A graph $G = (V, E)$. **Problem**: Decide if G contains a clique with **exactly** k vertices.

   For any positive integer k, give a polynomial-time algorithm for the k-Clique problem. Briefly discuss why this problem is in P, while the regular Clique problem is NP-Complete.

   Solution. The following algorithm solves k-Clique in polynomial time.

   Algorithm for k-Clique

   (a) On input $G = (V, E)$.

   (b) For each subset of vertices $V' \subseteq V$ with $|V'| = k$.

      i. Check if V' is a clique (i.e. check if for all pairs of distinct vertices $u, v \in V'$ we have $uv \in E$). If so, accept.

   (c) Reject.

   Clearly the above algorithm accepts if and only if G contains a k-clique, since we explicitly check all subsets of vertices of size k. There are

   $$\binom{n}{k} = O(n^k)$$

   subsets of vertices of size k, and checking each potential clique takes $O(k^2)$ time, so the run-time of the algorithm is $O(k^2 n^k)$. Since k is a fixed positive integer independent of the input graph G this is polynomial.

   The reason that this algorithm does not apply to the regular Clique problem is because the size k of the potential clique is now part of the input. It follows that the run-time of the algorithm $O(k^2 n^k)$ is exponential.

2. Let $G = (V, E)$ be a graph. A *three colouring* of G is a labelling of each vertex $v \in V$ with a colour $c(v) \in \{\text{red}, \text{blue}, \text{green}\}$ such that for each edge $uv \in E$ we have $c(u) \neq c(v)$. That is, a three colouring is any assignment of colours to the vertices of G such that every pair of vertices connected by an edge have different colours.

   We have the natural three-colouring problem and its search variant:

Problem. Three Colouring
Input: A graph $G = (V, E)$.
Problem: Decide if G has a three colouring.

Problem. Three Colouring Search
Input: A graph $G = (V, E)$.
Problem: Output a three-colouring of G or reject if no such colouring exists.

Give a search-to-decision reduction from Three Colouring Search to Three Colouring. (**Hint:** Observe that a triangle must be coloured with three different colours — try adding a triangle connecting it to the vertices of G somehow!)

Solution. Consider the following algorithm for Three Colouring Search which uses an oracle for Three Colouring.

Three Colouring Search

(a) On input $G = (V, E)$.

(b) Query the three-colouring oracle to check if G is three-colourable. If not, reject.

(c) Let $T = (\{r, g, b\}, \{rg, rb, gb\})$ be a triangle on three vertices r, g, b not appearing in G.

(d) Let G' be the graph obtained from G by adding T.

(e) For each vertex $v \in G'$.

  i. Add the edges rv, gv to G' and query the three colouring oracle on the resulting graph. If it accepts, continue to the next iteration of the loop. Otherwise remove the new edges.

  ii. Add the edges rv, bv to G' and query the three colouring oracle on the resulting graph. If it accepts, continue to the next iteration of the loop. Otherwise remove the new edges.

  iii. Add the edges gv, bv to G' and query the three colouring oracle on the resulting graph. If it accepts, continue to the next iteration of the loop. Otherwise remove the new edges.

(f) Let c be an "empty" colouring.

(g) For each vertex $v \in G$.

  i. If v has edges rv, gv in G, then set $c(v) = b$.

  ii. Else, if v has edges rv, bv in G then set $c(v) = g$.

  iii. Else set $c(v) = r$.

(h) Output c.

We prove efficiency and correctness. For efficiency, observe that each single operation in the algorithm is $O(1)$, and the algorithm consists of two $|V|$ loops. Thus the run time is $O(|V|)$, which is certainly polynomial in the input G.

Now correctness. If the graph G does not have a three colouring, then the first operation of the algorithm correctly rejects G by query the oracle. Thus, assume G has a three colouring. The main observation here is that in any three colouring of the triangle T the three vertices must have different colours. It follows that for each vertex $v \in G$, if we add two edges to v connecting it to two different vertices in T (say, r and g), then any colouring of the new graph G' must colour v the colour of the third vertex. This is because we are only allowed three colours to colour the

entire graph, and the two new edges prevent v from being coloured the same colour as the vertices r or g. In the algorithm, we check all possible ways of connecting v to the triangle T and use the oracle to verify that the vertex can be coloured consistently with the new added edges. Since G is three-colourable, it follows that in each iteration of the first loop *some* pair of edges connecting v to the triangle T will be added. From this, it follows that the colouring we construct in the final loop of the algorithm must indeed be a three colouring of the graph G.

3. Here we will show that *every* NP-Complete problem[1] admits a search-to-decision reduction, and so at least for these problems finding a solution is no harder than determining if a solution exists. First, a definition: if $y, z \in \{0,1\}^*$ are strings then we say z is an *extension* of y if $z = yw$ for some string $w \in \{0,1\}^*$ (i.e. z extends y if we can get z from y by appending characters to the end of y).

   Let $L \subseteq \{0,1\}^*$ be any NP-Complete language. Let V be a polynomial-time verifier for L, and let c, d be positive integers such that V runs in time $c|x|^d$ for every input (x, y) to V.

   Associated with L and V are two problems: the *search* problem and the *extension* problem. First, the search problem:

   Problem. L_{SEARCH}
   Input: A string $x \in \{0,1\}^*$.
   Problem: Output a string y of length at most $c|x|^d$ such that V accepts (x, y), or reject if no such string exists.

   Now, the extension problem:

   Problem. L_{EXT}
   Input: Two strings $x, y \in \{0,1\}^*$.
   Problem: Decide if there is an extension z of y such that V accepts (x, z).

   In this problem we give a search-to-decision reduction from L_{SEARCH} to L, using L_{EXT} as an intermediate step.

   (a) Show that L_{EXT} is in NP.

   (b) Give a polynomial-time algorithm computing L_{SEARCH} that uses an oracle for L_{EXT}.

   (c) Using (a), and the fact that L is NP-Complete, modify your algorithm from (b) to give a search-to-decision reduction from L_{SEARCH} to L.

   Solution. We begin by showing that L_{EXT} is in NP. Consider the following verifier.

   Verifier for L_{EXT}.

   (a) On input (x, y), z.

   (b) Check that z is an extension of y. If not, reject.

[1] Surprisingly, under some very believable complexity-theoretic assumptions, there are some NP problems where search is *harder* than decision [1]. Thus the NP-Complete assumption is essentially necessary.

(c) Simulate V on (x, z) and accept if and only if it accepts.

Clearly this verifier runs in polynomial time: checking if a string z is an extension of y is easy, and V runs in polynomial time in $|x|$ by assumption. It is similarly obvious that this is a correct verifier for L_{EXT} by the definition of the problem L_{EXT} — the above verifier accepts if and only if z is an extension of y and V accepts (x, z).

Now we give a polynomial-time search-to-decision reduction from L_{SEARCH} to L_{EXT}. Consider the following algorithm.

Algorithm for L_{SEARCH}

(a) On input x.

(b) Query the oracle for L_{EXT} on (x, ε), where ε is the empty string. If the oracle rejects, reject.

(c) Let $y = \varepsilon$.

(d) Repeat the following loop.

  i. Simulate V on (x, y). If it accepts, halt and output y.

  ii. Query the L_{EXT} oracle on $(x, y0)$. If it accepts, set $y = y0$ and continue to the next iteration of the loop. Otherwise set $y = y1$ and continue to the next iteration of the loop.

We argue efficiency and correctness. Let $x \in \{0, 1\}^*$. The algorithm immediately rejects if there is no certificate $y \in \{0, 1\}^*$ such that V accepts (x, y), so assume that there is such a string y. Intuitively the algorithm builds up y iteratively — we begin with the empty string ε, and then build y one bit at a time, using the oracle for L_{EXT} to verify that the current partial certificate y can be extended. Formally, it should be clear that at the end of every iteration of the loop, there is a string z extending y such that V accepts (x, z). In the case that $z = y$, the algorithm will halt and output y correctly. To see efficiency: since the length of the partial certificate y increases at each iteration, it is also clear that the algorithm will *eventually* halt. To show that halts in polynomial time, observe that the verifier V runs in time at most $c|x|^d$ for all x. It follows that if there is a certificate z making the verifier accept (x, z), then there is a z of length at most $c|x|^d$ making V accept. This means that the loop iterates at most $c|x|^d$ times, and since V is a polynomial time algorithm this implies that the algorithm for L_{SEARCH} is a polynomial time algorithm.

Finally we discuss how to modify the above algorithm to a search-to-decision reduction from L_{SEARCH} to L (such a reduction is called typically called a *self reduction*). Since L is NP-Complete by assumption, every language in NP reduces to L. In particular, L_{EXT} reduces to L since we showed L_{EXT} is in NP. Thus, there is a polynomial time computable function $f : \{0, 1\}^* \to \{0, 1\}^*$ such that for all (x, y),

$$(x, y) \in L_{\text{EXT}} \Leftrightarrow f(x, y) \in L.$$

Modifying the previous algorithm is now easy: whenever the previous algorithm is about to call the oracle for L_{EXT} on a string (x, y), instead call the L oracle on the string $f(x, y)$. By the definition of a polynomial-time reduction the oracle will return that $f(x, y) \in L$ if and only if $(x, y) \in L_{\text{EXT}}$, and thus correctness and efficiency follows from the algorithm above.

References

[1] Mihir Bellaire and Shafi Goldwasser. *The complexity of decision versus search*. SIAM Journal of Computing, 23:1 (1994).