$\boxed{\text{30 Apr 2021}}$

Rice's Theorem (TM Notes, §7)
Approximation Algorithms (K.&T., §11.1)

Announcements

1. Prelim 2 solutions posted on CMS. Regrade requests due a week from now.
2. Problem Set 9 will be released Tuesday, 5/4, due Thursday, 5/13. This will be the last 4820 homework this spring.

Theorem. Let M_0, M_1 be two Turing machines such $L(M_0) \subsetneq L(M_1)$.
Then there is no Turing machine that
a. Accepts every x such that
$$L(M_x) = L(M_0)$$
b. Doesn't accept any x such that
$$(M_x) = L(M_1)$$

Proof. By contradiction. Suppose M accomplishes a and b. We need to show how to use M to solve coHP.

Accordingly, given coHP input $x\#y$ we want to write down a description of a TM z such that

- if M_x halts on y then $L(M_z) = L(M_1)$.
- if M_x doesn't halt on y then $L(M_z) = L(M_0)$.

Idea: M_z processes its input w while simultaneously, on an extra set of working tapes, running a universal TM on $x\#y$.

As long as the $UTM(x\#y)$ has not halted, M_z simulates M_0 processing w.

The moment $UTM(x\#y)$ halts, M_z switches to simulating M_1 processing w.

Pseudocode.

1. On Tape 2, write the string $x\#y$.
2. Set status = false ("status" indicates if $UTM(x\#y)$ has halted)
3. Copy input from Tape 1 to Tape 3.
4. while status = false
   run one transition of M_0 on Tape 3
   run one transition of UTM on Tape 2.

if UTM halted set status = true
if M_0 accepted on Tape 3
Enter accept state

5. Clear Tape 3. Make fresh copy of Tape 1.
6. repeat forever:
   run one transition of M_1 on Tape 3.
   if M_1 accepts, enter accept state.

Some observations

1. If M_x halts on y, then $UTM(x\#y)$ halts, so status is eventually set to "true".
   That means M_z accepts its input if and only if one of the following happens:
   - M_0 accepts input before UTM halts.
   - M_1 accepts input

   That means $L(M_z)$ is the union of $L(M_1)$ with a subset of $L(M_0)$. But $L(M_0) \subsetneq L(M_1)$ so $L(M_z) = L(M_1)$.

2. If M_x doesn't halt on y, then $UTM(x\#y)$ loops, so the while-loop in step 4 never exits.

That means M_z accepts its input if and only if M_0 accepts it.

$\Rightarrow \quad L(M_z) = L(M_0)$.

Now assume $\exists\, M$ that accepts all z s.t. $L(M_z) = L(M_0)$ and doesn't accept any z s.t. $L(M_z) = L(M_1)$.

Then coHP would be r.e. because the following Turing machine M' would have $L(M') = $ coHP...

M', on input $x\#y$, constructs a description of the Turing machine M_z defined above, and then runs machine M on this description, z.

We reached a contradiction, hence $\nexists\, M$ that satisfies (a) and (b) above.

Recall, a <u>language</u> is a set of finite strings.
A <u>property of languages</u> is just a function from languages to $\{TRUE, FALSE\}$.
A property is <u>non-trivial</u> if it is true for at least one r.e. language and false for at least one r.e. language.

<u>Rice's Theorem.</u> For any non-trivial property of r.e. languages, it is undecidable to test, given x, whether $L(M_x)$ has that property.

<u>Proof.</u> Assume WLOG $\emptyset$ has the property. (Otherwise replace the property with its negation.)
Let $L(M_0) = \emptyset$
Let $L(M_1)$ be any r.e. language that doesn't satisfy the property.
If the property is decidable then $\exists$ a Turing machine M that accepts x if and only if $L(M_x)$ has the property.
In particular M accepts all x s.t. $L(M_x) = L(M_0) = \emptyset$

and M doesn't accept any x s.t. $L(M_x) = L(M_1)$.
Since $L(M_0) = \emptyset \underset{\neq}{\subset} L(M_1)$, the theorem above tells us there is no such M.

Examples. The following properties of a Turing machine M_x with description x are undecidable.

1. M_x accepts the string "11".
2. M_x accepts at least one string.
3. M_x accepts more strings of length 4820 than of length 3110.
4. Every string accepted by M_x has at most 4820 1's in it.

Some of these are r.e., others are not.
"Undecidable" just means
$$\{x \mid L(M_x) \text{ has the property}\}$$
is not recursive.
Rice's Theorem alone can't tell you if that set is r.e.
The first theorem in this lecture often can tell you about r.e. as well.

Approx Algorithms: coping with NP-Hard optimization problems (like minimum vertex cover) by designing algorithms that may produce suboptimal solutions, but they are provably not too much worse than optimal.

Example. Minimum Vertex Cover is the problem: given undirected $G=(V,E)$ find a vertex set $C \subseteq V$ that contains an endpt of each edge and has as few vertices as possible.

This is NP-Hard.

Algorithm for approximate the min VC.

1. Initialize $\tilde{E} = E$ (edge set of G).
2. Initialize $C = \emptyset$
3. while $\tilde{E}$ contains an edge (u,v).
   $C = C \cup \{u,v\}$
   Delete from $\tilde{E}$ every edge having

u or v as an endpoint

4. output C.

<u>Theorem.</u> This outputs a vertex cover, C, s.t.

$$|C| \leq 2 \cdot |C^*|$$

where C^* is a minimum vertex cover.

<u>Proof.</u> Let $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$, ..., $e_k = (u_k, v_k)$ be the edges chosen in the k iterations of line 3.

$$|C| = 2k.$$

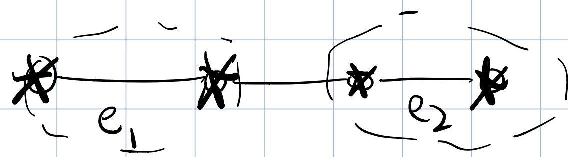
Since C^* is a vertex cover, it contains an endpoint of e_i $\forall i = 1, \ldots, k$. The endpoints of these edges are all distinct. So

$$|C^*| \geq k.$$

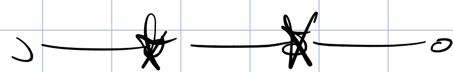
Therefore

$$|C| \leq 2 \cdot |C^*|.$$

C is a vertex cover because an edge isn't deleted from $\tilde{E}$ until it has an endpt in C, and the algo doesn't until $\tilde{E}$ is empty.



C



C^*