CS3052 Practical 2

200007413

1 Computability

1.1 YesOnNone

Here, we will show that YESONSTRING $\leq_T$ YESONNONE, which will show that YESONNONE is 'harder' to compute than YESONSTRING, which is known to be uncomputable,[1] and so it must mean that YESONNONE is also uncomputable.

Proof. We will show how an arbitrary input to YESONSTRING can be computed using YESONNONE. YESONSTRING is a machine which takes inputs (P, I), and returns 'yes' if and only if $P(I)$ returns a string.

Let (P, I) be arbitrary inputs to YESONSTRING. We can easily construct P', an alternate version of P, taking input I, which computes and assigns $v = P(I)$. If v is a string, P' will return 'no'. Otherwise, P' will return 'yes'.

We can then construct a machine $i(P', I')$, which ignores input I', and instead simulates P' with input I. Therefore, for all inputs I', $i(P', I')$ will always return the value of $P'(I)$.

We can then solve YESONSTRING using the oracle function YESONNONE with input $(i(P', I'))$ (I' can be any string, as it is ignored). This will return 'yes' if and only if $P'(I)$ returns 'no', which in turn returns 'no' if and only if $P(I)$ returns a string. Determining this final step is exactly the computational problem of solving YESONSTRING, meaning the outputs of YESONNONE$(i(P', I')) =$ YESONSTRING(P, I). As YESONSTRING is uncomputable, YESONNONE must also then be uncomputable. □

1.2 AcceptsRegLang

Proof. Let S be the set of programs whose accepted language is regular. ACCEPTSREGLANG is then precisely the problem COMPUTESONEOF$_S$. Note that S includes at least one computable function: the constant function accepting a regular language; and excludes at least one computable function: the constant function accepting a non-regular language. Therefore, the conditions of Rice's theorem are met, and therefore we conclude that ACCEPTSREGLANG is undecidable. □

2 Finite Automata

2.1 DFA For RegEx

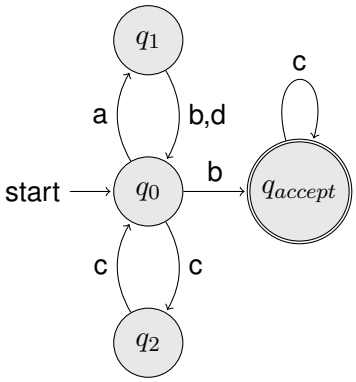

Figure 1: DFA accepting the language defined by the regex /(ab|cc|ad)*bc*/.
Reject state and it's transitions are omitted for clarity (i.e. inputs not shown for each state implicitly transition to the reject state)

2.2 Pumping Proof

Proof using the pumping lemma. Assume the language L, created from $0^a1^b0^c$ where $a + b = c$, is regular and argue for a contradiction. As there are an infinite set of numbers a, b, c such that $a + b = c$, then L is infinite, and so the pumping lemma applies. Therefore, a cutoff N exists, such that all $S \in L$ with $|S| \geq N$ can be pumped before N. Considering the example $S = 0^N1^N0^{2N}$. Note that $N + N = 2N$ andso S is a member of L. By our assumption, S can be pumped before N. This means there is a non-empty substring R in the first N characters of S, such that S can be pumped with R. We don't know exactly what this is, but we know that it must consist of one or more 0 characters. If this consists of k $0's$, where $k \geq 1$, then pumping S with one additional copy of R gives the new string $S' = 0^{N+k}1^N0^{2N}$. However, $N + k + N = 2N$ if and only if $k = 0$, which, by definition of k, is impossible. So, S' is not a member of L, which contradicts the fact that S can be pumped.

□

3 Polytime Reductions

3.1 01EQUALITY

To show that SAT-3 $\leq_P$ 01EQUALITY, we must first show that 01EQUALITY is in NP, and then show that we can reduce problems for SAT-3 to 01EQUALITY problems in polynomial time.

Proof that 01EQUALITY *is in NP.* For a decision problem to be in NP, the problem must (1) be a decision problem, and (2) have solutions which can be verified in polynomial time.

Here, (1) is self-evident, as the solutions are either 'yes' or 'no', for all inputs.

For (2), we can show that verifying solutions can be done in polynomial time. For each linear equation, we can calculate the sum of the left-hand side w in $\mathcal{O}(m)$ time, giving a total of $\mathcal{O}(mn)$ time for every equation. Each comparison to k can then becompleted in $\mathcal{O}(\log(k))$ time which totals to $\mathcal{O}(mn)$ time, which is in polynomial time. □

Proof that 3-SAT *reduces to* 01EQUALITY. As 3-SAT is in CNF, at least one of each of the variables in each clause is true, or the negation of a variable evaluates to true.

Therefore, we can convert the CNF C with clauses indexed j to their own linear equation in 01EQUALITY where values indexed i are set either as $x_j i = C_j i$ if $C_j i$ is positive, $x_j i = (1 - C_i j)$ when$C_j i$ is negated, and $x_j i = 0$ when $C_i j$ does not exist. The value of k can be calculated by counting the number of values in the clause where $C_j i$ is negated. Both of these can be done in a single pass through of each clause, giving a maximum of $n + 1$ time, which is just $\mathcal{O}(n)$ time, meaning the conversion is in polynomial time.

By solving these linear equations, we are showing whether or not there exists values which result in each clause having at least one satisfied literal, meaning either a negated literal is true, or a positive literal is

Therefore, solving 01EQUALITY here is directly solving 3-SAT, and so the reduction is complete. □

3.2 TSPD

NP-complete problems are decision problems which are the 'hardest' in NP. To show that TSPD is NP-complete, we must therefore show that (1) TSPD is a decision problem, (2) TSPD is in NP, and (3) $C \leq_P$ TSPD for all problems C in NP.

For there to be multiple NP-complete problems, all NP-complete problems must necessarily be equally hard. Therefore, we can prove (3) by showing that $C' \leq_P$ TSPD for some known NP-complete problem C'.

Proof that TSPD *is a decision problem.* A decision problem is a problem which has exactly two possible return values: 'yes' for positive instances, or 'no' for negative instances. All instances of TSPD will either have a Hamilton cycle of length L or not, meaning there are exactly two possible solutions. □

Proof that TSPD *is in* NP. Problems in NP are problems whose solutions can be verified in NP time, as NP is equivalent to PolyCheck.

Therefore, given a positive solution, along with the input, we must construct an algorithm which can verify it in polynomial time.

Given that we know that the Undirected Hamilton Circuit problem is NP-complete,[1] it must be in NP. This problem is identical to TSPD, except no limit on the length of solutions is in place. Therefore, we know that we can validate that a Hamilton cycle exists for a given input in polynomial time, and verifying the solution has a length at most L can be done in atmost n time, which is also in polynomial time. □

Proof there exists an NP-complete C such that $C \leq_P$ TSPD. Here, we will show a poly-reduction of the NP-complete problem Undirected Hamilton Circuit to TSPD.

UNDIRECTED HAMITON CIRCUIT $\leq_P$ TSPD

Given a Undirected Hamilton Circuit problem, we can convert it to a TSPD problem trivially, with the same setup input format describing the node, edges, and weights. The value of L needs to be great enough to not reject any answers based upon the circuit length. This can be prevented by summing all of the weights, as no edge can be passed over more than once. This takes a maximum of $n - 1$ time, thus the conversion can be done in polynomial time. □

4 Complexity Analysis

4.1 Array Rotation Verifier

4.1.a Big O Estimation

Algorithm 1 has an estimated time complexity of $\mathcal{O}(n^2)$.

First, there is a for loop which runs n times. Each of these calls `rot([1,...,n],i)` whichconcatenates the substrings `a[i,...,n-1]` and `a[0,...,i-1]` which takes a total of n time, as it is a concatenation of $n - i$ and i elements. A nested for-loop within the first then loops over a maximum of n elements if the given array is equal to the rotation calculated in the previous step.

As the rotation, and nested for-loop both occur within the same for loop, both taking n time each, this reduces to just $\mathcal{O}(n)$. Therefore, there are n loops which each take n time, giving anestimated time complexity of $\mathcal{O}(n^2)$.

4.1.b Ascending Sequence Algorithm

Algorithm 1 Determine ascending array rotation point for array a of length n in time $\mathcal{O}(\log n)$

```
1:  start = 0
2:  end = n
3:  x = 0
4:  while end − start > 1 do
5:      x = randomInt(start, end-1) // exclusive of max value
6:      if a[x] > a[x+1] then
7:          break
8:      end if
9:      if a[x] < a[0] then
10:         end = x + 1
11:     else
12:         start = x
13:     end if
14: end while
15: return n − (x + 1)
```

Algorithm 1 takes $\mathcal{O}(\log n)$ time.

A sorted array, once rotated, consists of two sub-arrays which are each sorted, where every value of the first sub-array is greater than all values in the other sub-array (as the smallest value in the first sub-array was adjacent in the original sorted array to the greatest value in the second sub-array).

The pivot point to convert the value back to b can be found by finding the index p of the first value where the array stops ascending, as this will be the index at which we go from the first to the second sub-array. As a and b have the same length, finding p gives x, by calculating $n - p$.

Searching through the list in order runs in n time, as we don't know where p is. We can do this more efficiently by starting at a random point i, and checking whether the value on the right is smaller than it. If it is, then we have found the pivot point. If not, determining if p is to the left or right of i by comparing the value at the start of a to the value at i. If $a[i] > a[0]$, then i must be in the first sub-array, and p must be further right. Otherwise, i is in the second sub-array, and the pivot point is farther left.

We can repeat this process, instead getting a random point in this new region, either the left or right half of a, and comparing the value to $a[0]$. This will eventually find p (thus finding x). Due to the nature of the random searching, this continual splitting of the array into smaller and smaller chunks runs in $\mathcal{O}(\log n)$ time.

4.2 Vertices Index Algorithm Analysis

The worst case running time for this algorithm for a given input length n would have each vertex in V connected as a single line, meaning two vertexes are included in exactly one edge in E, and all other vertexes in V are included in exactly two edges in E. The worst-case value for x would be either one of the vertexes included in just one edge.

This would give a total of $n - 1$ edge pairs in E which would be looped over by the first while loop. No values would be changed, except for the distance value for the vertex next to x which would be set to one. In the next while loop, all vertexes would again be checked, and only the next connected distance value would change. This would repeat for all $n - 1$ edges.

Therefore, there would be $n - 1$ while loops where values are changed, each checking over $n - 1$ edges. A final loop would check all $n - 1$ edges, giving a total of n loops of $n - 1$ checks. This simplifies to a run time of $\mathcal{O}(n^2)$.

References

[1] John MacCormick. *What can be computed?: A practical guide to the theory of computation*. Princeton University Press, 2018. DOI: 10.1515/9781400889846.