OLLSCOIL NA hÉIREANN, MÁ NUAD

NATIONAL UNIVERSITY OF IRELAND, MAYNOOTH

M.Sc./H.Dip. SOFTWARE ENGINEERING EXAMINATION

SOLUTIONS TO SAMPLE 2001/2002

PAPER CS605

MATHEMATICS AND THEORY OF COMPUTER SCIENCE

Mr. T. Naughton

Answer four (4) questions from six (6). Time Allowed: 3 hours.

1.

2.

3. (a) $L = \{w : w \in \{a, b\}^*, w = a^n b^n, n \in \mathbb{N}\}$
$L = \{w : w, v \in \{a, b\}^*, w = vv^{\mathrm{R}}\}$
$L = \{w : w$ is a syntactically correct Java program$\}$

   (b)

   (c) Σ^* is not finite, but is countable.

4.

5.

6. (a)

   (b)

   (c)

   (d) 2^X will be countable exactly when X is finite. 2^X will be finite exactly when X is finite. (2^X will either be uncountable or finite.)

7.

8.

9.

10. (a) $L = \{w \in \{a, b\}^* : w = a^n b^n a^n, n \in \mathbb{N}\}$ is r.e. but not c.f.

    (b)

    (c) $L = \{w \in \{t : t$ is a Turing machine$\} \times \{s : s$ is a state$\}$: where TM t enters state $s\}$

    (d)

11.

12.

13.

14. *Note: this solution is quite long in order to make it understandable. A far more concise proof would be more acceptable in an exam.*

    To prove that *HALTS* and *CHANGE* are equally difficult problems we need to show two reductions: $HALTS \leq CHANGE$ proves that *CHANGE* is at least as hard as *HALTS* and $CHANGE \leq HALTS$ proves that *HALTS* is at least as hard as *CHANGE*.

    Assume we have a program *halts()* that takes a function *P()* and returns true if *P()* is a halting function and returns false otherwise. Assume also that we have a program *change()* that takes a function *Q()*, and a variable *a* declared in *Q()*, and returns true if *a* would be changed during the execution of *Q()*, and returns false otherwise.

    First, we show $HALTS \leq CHANGE$ (a reduction from *HALTS* to *CHANGE*). This reduction will prove that if we have a program to solve the latter, we can solve the former. Given a program *change()* we will solve the halting problem for a function *P()* as follows.

    Construct a function *Q()* of the following form:

```
void Q(void){
  int a;
  P(); // call function P()
  a = 5;
}
```

    If we have a program *change()*, we could execute it passing *Q()* and *a*. If *change(Q, a)* returns true then that means that *halts(P)* would have returned true and if *change(Q, a)* returns false that means that *halts(P)* would have returned false. More concisely we say that *change(Q, a)* returns true iff (if and only if) *halts(P)* would have returned true. Therefore we effectively have simulated the program *halts()*.

    Secondly, we show a reduction from *CHANGE* to *HALTS*. This will prove that problem *HALTS* is at least as hard as problem *CHANGE*. Given a program *halts()* that solves *HALTS* we can simulate a program *change()* that solves *CHANGE* for a function *Q()* as follows.

    Assume *Q()* contains N statements or function calls and is of the form:

```
void Q(void){
  statement0;
  statement1;
      :
  statementi;
  a = 5;
  statementi+2;
      :
  statementN;
}
```

    Let us extract the block of statements before the line where *a* is first changed (all statements from `statement0` to `statementi`). Let us put them into a function of their own and call it *P()*. *P()* would look like:

```
void P(void){
  statement0;
      :
  statementi;
}
```

Next we pass *P()* to *halts()*. *halts(P)* returns true iff *change(Q, a)* would have returned true. Therefore we have simulated *change(Q, a)*.

This method works even if "`a = 5;`" is part of a conditional statement or part of a function call. For example, imagine if the function looked like:

```
void Q(void){
  statement0;
      :
  statementi;
  if (condition){
    a = 5;
  } else {
    statementi+2;
  }
  statementi+3;
      :
  statementN;
}
```

Then we would pass the following function

```
void P(void){
  statement0;
      :
  statementi;
  if (condition){
    a = 5;
  } else {
    while (true){
      // infinite loop
    }
  }
}
```

to *halts()*. In this case, *halts(P)* also returns true iff *change(Q, a)* would have returned true.

Finally, if there had been many statements in *Q()* that assign some value to *a*, then we just repeat the procedure for each one. As long as the piece of code immediately before any one statement assigning a value to *a* halts, then *a* will be changed.

By showing that $HALTS \leq CHANGE$ and that $CHANGE \leq HALTS$ we prove that *HALTS* and *CHANGE* are equally difficult problems.

15.

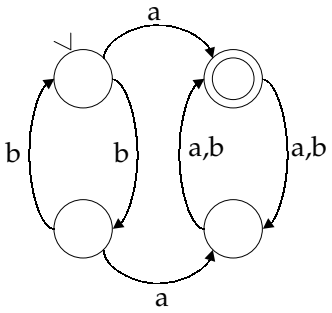
16. (a) i. (1,1,1,1,1,1) : because a FA exists to accept such unary numbers
    ii. (1,1,1,1,1,1) : because a FA exists to accept such a language

iii. (0,0,1,0,0,0) : (0,0,1,0,... because the halting problem can be reduced to it, and ...,0,0) because at least one EXPTIME-complete language is accepted by a TM that goes into a state 01.

iv. (0,0,1,0,0,0) : (0,0,1,0,... because the halting problem can be reduced to it, and ...,0,0) because at least one EXPTIME-complete language is accepted by a Java program that executes a `print` statement.

v. (1,1,1,1,1,1) : because a FA exists to accept such a language

vi. (1,1,1,1,1,1) : because the following FA accepts such a language:



(b)

17.

18. (a)

(b) To prove that the class $\mathcal{P}$ is closed under complement we must show that if a language is in $\mathcal{P}$ then its complement must also be in $\mathcal{P}$. Consider a language L that is in $\mathcal{P}$. Since L is in $\mathcal{P}$, some polynomial-time function `boolean f(String x)` must exist that returns true if $x \in L$ and returns false otherwise. Consider the following function:

```
boolean g(String x){
  if (!f(x)) {
    return true;
  } else {
    return false;
  }
}
```

This function decides exactly the language $\overline{L}$, the complement of L. Moreover this function is also polynomial so $\overline{L}$ must be in $\mathcal{P}$.

(c) To prove that SAT is in $\mathcal{NP}$ we must show that each instance of SAT has a witness (solution) that can be verified in polynomial time. We do this by outlining a polynomial-time algorithm to verify any satisfying truth assignment for any instance of SAT.

Given a conjunctive normal form with a list of n clauses, each with at most m literals (boolean variables), from an alphabet of a literals, and a truth assignment for each of the a literals (this is our witness), the algorithm could be defined as follows.

Stage 1. Take each clause, and iterate through the literals in the clause, replacing each literal with a T or an F as appropriate from the list of truth assignments. If the negation of a literal is encountered, replace it with the negation of the value from the truth assignment. This substitution stage will take at most $(n \times m \times a)$ copy and comparison operations (assuming the list of truth assignments is not sorted).

Stage 2. Iterate through the clauses, ensuring that each clause has at least one T in it. If a single clause does not have at least one T in it then the truth assignment was not valid. If the end of the list of clauses is reached then the truth assignment was a valid one. This checking stage requires at most $n \times m$ comparison operations.

The total time complexity is $nma + nm$, which can be approximated as a polynomial function $c^3 + c^2$ for some c where $c \geq \max(m, n, a)$.

19. (a)

(b)

(c) $\mathcal{P} = \mathcal{NP}$ implies that if a language L is in $\mathcal{P}$ then L is also in $\mathcal{NP}$. We also know that $\mathcal{P}$ is closed under complement so that if a language L is in $\mathcal{P}$ then $\overline{L}$ is also in $\mathcal{P}$. Therefore, if L is in $\mathcal{NP}$ then $\overline{L}$ is in $\mathcal{NP}$. This is equivalent to saying that $\mathcal{NP}$ is also closed under complement, or that $\mathcal{NP} = \text{co}\mathcal{NP}$.