NATIONAL UNIVERSITY OF SINGAPORE

CS 5236: Advanced Automata Theory

Semester 1; AY 2022/2023; Final Exam

Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. Please write your Student Number. Do not write your name.
2. This assessment paper consists of TEN (10) questions and comprises TWENTY-ONE (21) printed pages.
3. Students are required to answer **ALL** questions.
4. Students should answer the questions in the space provided.
5. This is a **CLOSED BOOK** assessment with one helpsheet of A4 size.
6. You are not permitted to communicate with other people during the exam and you are not allowed to use additional material beyond the helpsheet.
7. Every question is worth SIX (6) marks. The maximum possible marks are 60.

STUDENT NO: ______________________________

This portion is for examiner's use only

Question	Marks	Remarks	Question	Marks	Remarks
Question 1:			Question 6:		
Question 2:			Question 7:		
Question 3:			Question 8:		
Question 4:			Question 9:		
Question 5:			Question 10:		
			Total:		

Each state of a Moore machine contains a, perhaps empty, word which is output whenever the machine is in that state. The transitions of a Moore machine are like those of a finite automaton, upon reading of a symbol, it transfers to some other state; a Moore machine is deterministic iff there is exactly one start state and for each pair of state and symbol there is exactly one transition. A run of the Moore machine is valid if it is at the end of the run in an accepting state.

Construct a Moore machine, which replaces in a ternary input word 0 by 1, 1 by 10 and 2 by 100. The empty word is mapped to the empty word. For example, 20120 is mapped to 1001101001.

Solution. The start state s has the empty word ε as output. Furthermore, there are states q_0, q_1, q_2 such that their outputs are 1, 10 and 100, respectively. From each of the states s, q_0, q_1, q_2, the Moore machine, upon reading symbol $d \in \{0, 1, 2\}$, transfers to state q_d.

Construct a dfa which for each binary number $a_n a_{n-1} \ldots a_0$ checks whether

$$\sum_{even\ i} a_i + 2 \sum_{odd\ j} a_j \equiv 2 \text{ (modulo 3)}.$$

Furthermore, provide a regular expression for this language which can use finite sets of binary words, concatenation, union, intersection, Kleene star, Kleene plus and brackets for making binding order clearer when needed.

The above sum only applies to $i, j \in \{0, 1, \ldots, n\}$ which are also even or odd, respectively. For example, the dfa should accept 1011 (eleven) as the above sum is 5 which is 2 modulo 3 and reject 101010 (twenty-one) as the above sum is 6 which is 0 modulo 3. The dfa reads the binary number from the front to the end, so when processing 1011 the digits are read in the sequence 1, 0, 1, 1.

Solution. The numbers $2, 8, 32$ and so on have remainder 2 by 3, the numbers $1, 4, 16$ and so on have remainder 1 by 3. Thus the algorithm counts, modulo 3 the value of all $2^m \cdot a_m$ and that is the value of the given the binary number modulo 3. This is equivalent to checking whether the binary number itself has the value 2 modulo 3.

This can be done by the following dfa which reads the number from the front: It has three states z, o, t which stand for zero, one and two of the part read so far modulo 3. z is the start state and t is the only accepting state. On 0 the remainder is doubled modulo 2, so z goes on 0 to z, o goes on 0 to t and t goes on 0 to o. On 1 the remainder is first doubled and then incremented, so z goes to o, o goes to z and t goes to itself.

The regular expression is given as follows: Let $I = \{00, 10\}^* \cdot \{01, 11\} \cdot \{00, 10\}^*$ and $J = \{00, 01\}^* \cdot \{10, 11\} \cdot \{00, 01\}^*$. Furthermore, in the following, $\cap$ binds more than $\cup$. Now the full regular expression is given by $(I \cdot I \cdot I)^* \cap ((J \cdot J \cdot J)^* \cdot J) \cup ((I \cdot I \cdot I)^* \cdot I) \cap ((J \cdot J \cdot J)^* \cdot J \cdot J) \cup ((I \cdot I \cdot I)^* \cdot I \cdot I) \cap (J \cdot J \cdot J)^*$.

Construct a deterministic Büchi automaton which recognises the set of all ω-words over $\{0, 1, 2, 3\}^\omega$ in which occur at least two different digits infinitely often. The automaton should not have more than 8 states; correct solutions with more states get only partial credit. Also write why the Büchi automaton does what it should do.

Here the automaton should be deterministic and complete, that is, for each combination of state and symbol there is exactly one successor. Furthermore an ω-word is in L iff the automaton goes infinitely often through an accepting state when processing the ω-word.

Solution. The automaton has a start state s and for each digit $d \in \{0, 1, 2, 3\}$ the state q_d. If the automaton is in s and sees a digit d then it goes to q_d. If the automaton is in q_d and sees the digit d then it stays in q_d while on all other digits it goes to s. The state s is the only accepting state. One can see that whenever the automaton returns from q_d to s then it has seen besides d some other digit. Therefore if it goes to s at least $12k$ times then it has seen two digits at least k times; to see this, one picks the most frequently visited state q_d when coming from s. This state has been visited at least $3k$ times and thus the automaton has seen d at least k times. Furthermore, the automaton has returned from q_d to s at least $3k$ times, thus it has seen one of the digits different to d also at least k times. Therefore, if the automaton returns to s infinitely often then it has seen two digits at least infinitely often.

Consider the following game: First Anke puts a natural number $x > 1$. Then Boris can move as often as he wants and can always do one of the following mappings: Replace x by $2x$; if x is a multiple of 3 then replace x by $x/3$; if $x-1$ is multiple of 3 then replace x by $(x-1)/3$. Anke wins the game if the game never goes into 1. Boris wins the game if the game after finitely many moves reaches 1. Who has a winning strategy for this game?

☐ Anke; ☐ Boris.

Give a proof for your answer.

Solution. [✓ Boris]: Boris wins the game for all natural numbers $x > 1$.

If x or $x-1$ is a multiple of 3 and $x > 1$ then Boris can either go to $x/3$ or $(x-1)/3$ and both are smaller nonzero natural numbers, that is, they are at least 1. If $x-2$ is a multiple of 3 then Boris can move from x to $2x$ and $2x-4, 2x-1$ are both multiples of 3 and thus Boris can move on to $(2x-1)/3$ which is a nonzero natural number smaller than x. So in all situations Boris can go from x to a natural number y with $1 \leq y < x$. As it is always Boris' turn to move, he will eventually reach 1.

Three players, Anke, Boris and Claudia, play a game by adding numbers modulo 60. Anke can add 12 or 24 or 36 or 48, Boris can add either 20 or 40, Claudia can add 15 or 30 or 45. The game starts at 0 and the first player who moves back to 0 wins and the player to move after his player is second and the other player third. If the game runs forever, it becomes a draw. The first move which goes away from 0 has always to be made. The players move in sequence Anke - Boris - Claudia - Anke - Boris - Claudia - . . . until the game is back to 0.

(a) Are there players who can achieve that the game runs forever and becomes a draw, irrespective what the other two players do? If yes, which players are these?

(b) Are there coalition of players who can force a win? That is, they can make sure that the game terminates and one of them goes first and the other one goes second. If so, which pairs of players can form such a coalition?

(c) Assume that the game has gone on already for some while and all players agree on bringing it to an end in the fastest way possible, without caring who will be first, second or third. How many moves does this take in the worst case?

Solution. (a) Each player can block the game from terminating. If one looks at the remainders of the current number by $(4, 5, 6)$, respectively, Anke is the only player who can modify the remainder by 5, Boris is the only player who can modify the remainder by 3 and Claudia is the only player who can modify the remainder by 4. Each player has several choices and can therefore achieve that the remainder stays nonzero throughout the infinite duration of the game. Note that the first move in the game of each player has to move the own remainder from zero to some nonzero value.

(b) In the light of (a), even coalitions of two players cannot force a termination of the game and thus cannot enforce a win.

(c) For this question it is sufficient to distinguish the case that the own remainder of a player is zero or nonzero. If the own remainder is zero, the player has to make it nonzero, if it is nonzero, the player can either move to another number with a nonzero own remainder or to a number with a zero own remainder. Assume that it is Anke's turn to move. There are several cases depending on what the remainders of the players are. Here they are listed as Anke's remainder (by 5), Boris' remainder (by 3) and Claudia's remainder (by 4).

(nonzero, nonzero, nonzero): Three moves, each player makes the own remainder zero.
(nonzero, nonzero, zero): Two moves, the first two players make the remainders zero and then the game terminates.
(nonzero, zero, nonzero), (zero, zero, nonzero): Five moves: As Boris has to make his remainder nonzero, it is best to have all remainders nonzero after his move and Anke moves accordingly to make her remainder nonzero. Then Boris moves and afterwards the three players make in sequence their remainders zero.
(nonzero,zero, zero): One move, Anke makes her remainder zero.
(zero, nonzero, nonzero): Four moves: Anke must make her remainder nonzero and afterwards each of the players makes in one further move the own remainder zero.
(zero, nonzero, zero): Six moves, Anke has to make her remainder nonzero and then

five more moves are needed, as from Boris perspective, at the next move, the situation is of the form (nonzero, zero, nonzero), see above.
(zero,zero, nonzero): Five moves, Anke and Boris have to make their remainders nonzero and after that, all three players can make their remainders in sequence zero.
(zero,zero,zero): Zero moves, as by assumption there was already an initial move in the past and now the most recent move before the current one has brought the game to an end.

Assume that F is a finite field and let the string $a_0 a_1 \ldots a_n \in F^+$ represent the polynomial $a_0 + a_1 x + \ldots + a_n x^n$ where $a_0, a_1, \ldots, a_n$ are the coefficients with the polynomial.

Recall that adding of two polynomials is done by adding component wise the coefficients and omitting all a_m in the resulting polynomial if the coefficients from m onwards are all 0. Similarly for subtraction. Multiplying with x is done by mapping $a_0 a_1 \ldots a_n$ to $0 a_0 a_1 \ldots a_n$ and multiplying with x^2+1 is done by multiplying the given polynomial p with x^2 and then forming the sum $p + (x^2 \cdot p)$. Which of the following operations are automatic?

(a) Adding polynomials: ☐ Yes, ☐ No.

(b) Multiplying a polynomial with $x^3 + x + 1$: ☐ Yes, ☐ No.

(c) Multiplying a polynomial with itself: ☐ Yes, ☐ No.

Give brief reasons for the answers.

Solution. (a) Component wise addition of coordinates has no carry and can be easily carried out, for each pair of symbols one does table look up in an addition table of the finite field. Furthermore, all components in the addition which have a space symbol # can be interpreted as being zero. However, when verifying $p + q = r$, either none or one of the two inputs and the output have # and this has always to be the same. For each of p, q, r, when it has at one position a 0, there should later come a non-zero coordinate, as otherwise # has to be there.

(b) The multiplication with the fixed polynomial $x^3 + x + 1$ is automatic. When comparing p, q for $q = (x^3 + x + 1) \cdot p$, the automaton has to check that $q_{m+3} = p_m + p_{m+1} + p_{m+3}$ with respect to the addition in the finite field for all applicable m.

(c) Squaring (multiplying with itself) makes the output polynomial twice as long as the input. As the polynomials can be arbitrarily long, for every constant there is an input where the output is by more than c symbols longer than the input. An application of the Block pumping lemma shows that the graph of this operation cannot be automatic, thus the function is not automatic.

Which of the following monoids (= semigroups with neutral element) have an automatic or fully automatic representation? If these representations exist, explain how to construct them. If they do not exist, write in a few sentences why.

(a) $(\mathbb{N} \times \mathbb{N}, +, (0,0))$: Is this monoid automatic? ☐ Yes, ☐ No.
Is this monoid fully automatic? ☐ Yes, ☐ No.

(b) $(Finsubset(\mathbb{N}), \cup, \emptyset)$: Is this monoid automatic? ☐ Yes, ☐ No.
Is this monoid fully automatic? ☐ Yes, ☐ No.

(c) $(\mathbb{R} \times \mathbb{R}, \cdot, (0,0))$: Is this monoid automatic? ☐ Yes, ☐ No.
Is this monoid fully automatic? ☐ Yes, ☐ No.

Provide the constructions or reasons below. Here $Finsubset(\mathbb{N})$ is the set of all finite subsets of the natural numbers, $\mathbb{R}$ is the set of real numbers and $\cdot$ is, for the real numbers, the multiplication.

Solution. (a) The monoid is both, automatic and fully automatic. These are based on representations of $(\mathbb{N}, +)$. For the fully automatic representation, this is, in suitable order of bits, just the addition on binary numbers and detailed out in the lecture notes. One takes the convolution of two such representatives (i, j) and then adds (h, k) component wise to (i, j) giving $(i + h, j + k)$, this pair is the represented as $conv(i+h, j+k)$. The automatic representation of $(\mathbb{N}, +)$ has the unary domain $\{0\}^*$ together with string concatenation. Now one combines two such representations to get the domain $\{0\}^*\{1\}^*$ with the addition $0^i1^j + 0^h1^k = 0^{i+h}1^{j+k}$. When (j, k) are fixed to constants, then this operation is an automatic function. The fully automatic addition, however, works only on representations which use digits in some form (like binary and decimal).

(b) As the semigroup is not finitely generated, it is not automatic. For fully automatic, consider the representation by strings $a_0a_1 \ldots a_n$ with $a_n = 1$; the empty set is represented by the empty string ε. Now m is in the set represented by $a_0a_1 \ldots a_n$ iff $m \leq n$ and $a_m = 1$. Furthermore, let # be the symbol used to make the strings into the same length for processing in the automaton. The symbols are now triples of the form (a, b, c) with $a, b, c \in \{\#, 0, 1\}$; on these symbols one defines the order $\# < 0 < 1$. Let A, B, C be finite subsets and $str(A)$ represent the string representation of the finite set A, analogously for B and C. A dfa recognising

$$\{conv(str(A), str(B), str(C)) : A \cup B = C\}$$

has two states, an accepting start state s and an always rejecting state r. Once in r, the dfa transfers on all triples to r again. In state s upon receiving a triple with components a, b, c, if $\max\{a, b\} = c$ then the dfa stays in s else it goes to r.

(c) The structure is neither automatic nor fully automatic. The simplest reason is that the domain is not countable, what is required for representing the domain as finite words over a finite alphabet Σ. It is also known that one can not find an ω-automatic representation of the reals and multiplication, but knowing this is not required for this question.

A ω-transducer is a deterministic transducer which translates infinite words as inputs into outputs. A run with outputs is valid iff the transducer goes infinitely often through an accepting state while producing the output.

Say an ω-word represents a function f iff the ω-word is of the form $0^{f(0)}10^{f(1)}10^{f(2)}1\ldots$ and an ω-word with only finitely many 1s does not represent a valid input.

The goal is to construct a transducer which computes for ω-words representing f, g an ω-word representing $f+g$; that is, the run should be valid iff the input ω-words represent some functions f, g. Now the output should be $0^{f(0)+g(0)}10^{f(1)+g(1)}10^{f(2)+g(2)}1\ldots$ and while writing this output, the ω-transducer should go infinitely often through an accepting state. However, the ω-transducer goes only finitely often through accepting states in the case that the output was not correctly guessed (due to nondeterminism) or in the case that one of the inputs is not representing a function.

If there is a deterministic ω-transducer doing the addition, please provide correspondingly a deterministic ω-transducer; however, if there is no deterministic ω-transducer, provide a nondeterministic ω-transducer and explain why it cannot be made deterministic.

Solution. The ω-transducer can be made deterministic. It has states s, t, u, s is the start state and u is the accepting state. The transition relation and production of output is the following:

In state s, on input $(0, \varepsilon)$ the transducer produces the output 0 and goes to s. On input $(1, 0)$ the transducer produces the output 0 and goes to state t. On input $(1, 1)$ the transducer goes to u and produces the output 1.

In t, on input $(\varepsilon, 0)$, the transducer produces output 0 and goes to t. On input $(\varepsilon, 1)$ the transducer produces output 1 and goes to u.

In u, on input $(0, \varepsilon)$ the transducer goes to state s and produces the output 0. On input On input $(1, 0)$ the transducer produces the output 0 and goes to state t. On input $(1, 1)$ the transducer goes to u and produces the output 1.

The accepting state u is visited each time after completely reading and processing $(0^{f(k)}1, 0^{g(k)}1$ for some k where then $0^{f(k)+g(k)}1$ is output. If the transducer gets stuck in s or t due to reading an infinite sequence of zeroes on one of the inputs then the transducer goes only finitely often through the state u and thus the computation is not valid. One can easily verify that the ω-transducer is deterministic.

ONLINE LOGSPACE Algorithms. An algorithm to process words is called an ONLINE LOGSPACE algorithm iff the following conditions are satisfied:

The algorithm is carried out by a register machine which can add, subtract and compare ($<, =, >, \leq, \geq, \neq$). Furthermore, assignments with additive expressions and constants are possible and conditional and nonconditional branchings and gotos. Calls without recursion to functions are allowed. The input word is read symbol by symbol with the special symbol # used to separate inputs and to mark the end of the last input. Symbols can be compared with symbol constants ($=, \neq$) and the outcome can be used for conditional branching.

The LOGSPACE condition means that there is a polynomial p such that for inputs with combined length of n the numerical value of each register is never smaller than $-p(n)$ or larger than $p(n)$.

For example, when x, y, z are registers and s is a symbol variable, instructions like "if $x < y$ then let $z = x + y$ else begin let $z = x - y - 5$; goto line 8 end" and "if $s = a$ then goto line 12" are possible. If there are two input words aab and cc then the input read is $aab\#cc\#$ and $n = 7$ (5 generators plus two symbols for input-end).

The Semigroup Considered. The semigroup is generated by words a, b, c, d, its neutral element is ε (empty word). The semigroup has the rules $abc = ba$, $cd = dc = \varepsilon$, $ac = ca$, $ad = da$, $bc = cb$, $bd = db$. Furthermore, if one uses the above rules to order the generators in the word alphabetically, then $a^i b^j c^k d^h = a^{i'} b^{j'} c^{k'} d^{h'}$ iff $i = i'$, $j' = j'$ and $k - h = k' - h'$.

The Word Problem. Let $el_G(u)$ assign to a word $u \in \{a, b, c, d\}^*$ the group element represented by the word. The word-problem of a semigroup is to decide for given words v, w over the generators whether $el_G(v) = el_G(w)$.

Task. Construct an ONLINE LOGSPACE algorithm to decide the word problem of this semigroup. Note that the words are not necessarily given in alphabetically ordered form.

Solution. The idea of the algorithm is to transform an input word into the equivalent word of the form $a^i b^j c^k$ where $i, j \geq 0$ while k can be any value in $\mathbb{Z}$. A d counts as c^{-1} due to d being the inverse of c. So the following subfunction initialises variables i, j, k, given as parameters to the subfunction and assigns to them the number of a, b, c of the current input word, when translated into the normal form $a^i b^j c^k$. Here when reading an a, it must move over the currently j b which causes the creation of j symbols c. Thus the counting algorithm is as follows, where s is a symbol variable.

```
1. Function Readword($i, j, k$);
2. $i = 0$; $j = 0$; $k = 0$;
3. Read $s$;
4. If $s = a$ then begin $i = i + 1$; $k = k + j$ end;
5. If $s = b$ then $j = j + 1$;
6. If $s = c$ then $k = k + 1$;
```

```
7. If $s = d$ then $k = k - 1$;
8. If $s \neq \#$ then goto 3;
9. Return.
```

The main function now just compares the two inputs.

```
1. Function Wordcompare;
2. Read($i_v, j_v, k_v$);
3. Read($i_w, j_w, k_w$);
4. If $i_v = i_w$ and $j_v = j_w$ and $k_v = k_w$ Then write “Words are equal” Else write
   “Words are different”;
5. Return.
```

Note that in Readword, i, j can never be larger than n, as there are n symbols read at most. Furthermore, k can in each step only change by at most n, thus it is never larger than n^2 or smaller than $-n^2$. Thus the algorithm is an ONLINE LOGSPACE algorithm.

Provide a semigroup for which the word problem cannot be decided by an ONLINE LOGSPACE algorithm as defined in the previous question. Give a proof that this semigroup has no such algorithm.

Solution. A possible semigroup looked for is the semigroup of binary words with the concatenation as semigroup operation. Then $0, 1$ are generators and cannot be expressed as products of other generators; hence they have to be part of any set F of generators. There are 2^n binary words of length n. Now an ONLINE LOGSPACE algorithm can have after processing a word u of length n followed by # only $p(n)$ many different configurations (memory content and position in the program). Thus there are, for all sufficiently large n, two distinct binary words $v, w \in \{0, 1\}^n$ such that the algorithm after processing the input $v\#$ and $w\#$ is in the same configuration. Therefore the algorithm gives the same output on the inputs $v\#v\#$ and $w\#v\#$. However, one time the algorithm has to say "equal" and one time it has to say "different", thus the algorithm cannot be correct. Therefore this semigroup does not have an ONLINE LOGSPACE algorithm. The same applies to the free group with two generators.

END OF QUESTION PAPER.