Computation Theory
Solution Notes for Example Sheet 2

In this document you will find some solution notes for the problems of Example Sheet 2 of Computation Theory. If you find any mistake or any typos, please do let me know. Also, I am happy to hear (and include them in the notes (with credit) if you want) about alternative solutions to the problems or variations of a problem that you came up with.

Lecture 5

Further reading:

- Chapter 4 and 5 in M. Sipser's "Introduction to Computation Theory".
- The handout Computation Theory: Supplementary notes on decidability

Exercise 1 [Halting problem]
(a) Define what it means for an RM to decide the *halting problem.* (See [**2008P5Q10 (a)**] or [**2005P3Q7 (a)**] or [**2000P3Q9 (a)**])
(b) Prove that no such machine can exist. (See [**2017P6Q3**] or [**2005P3Q7 (c)**])
(c) Define what it means for a function to be *uncomputable.*

(a) See the referenced slides.

See **Lecture 5 slide 3**.

(b) See the referenced slides.

See **Lecture 5 slide 4**.

(c) A partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is uncomputable if it is not computable, i.e. there does not exist a register machine M with at least $n+1$ registers such that for all $(x_1, \ldots, x_n)$ and $y \in \mathbb{N}$, the computation of M starting with $\mathtt{R}_0 = 0, \mathtt{R}_1 = x_1, \ldots, \mathtt{R}_n = x_n$ and all other registers set to 0, halts with $\mathtt{R}_0 = y$ iff $f(x) = y$.

Exercise 2 [Unary functions] Does it make a significant difference that we are concentrating on *unary functions* (instead of n-ary)?

We can still represent all computable partial functions using RMs that "accept" a single argument. We can construct an RM that interprets $\mathtt{R}_1$ as a list of registers and then places the contents of the lists into the first n registers. If the list has more items than registers in the machine, then we can e.g. loop forever. The point is that there will exist some register machine that will compute the same partial function. The reverse direction is straightforward.

Exercise 3 [Characteristic function]
(a) Define the *characteristic function* χ_S of a set S. Give an example.
(b) Define what it means for a set to be *decidable/undecidable*? (See [**2014P6Q3 (d)**] or [**2005P4Q9 (a)**] or [**1995P3Q9 (c)**])

(a) Given a subset $S \subseteq \mathbb{N}$, its characteristic function is $\chi_S \in \mathbb{N} \to \mathbb{N}$, given by $\chi_S(x) \triangleq \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$.

A simple example is $S = \{1, 4, 6\}$, so $\chi_S = \{(0,0), (1,1), (2,0), (3,0), (4,1), (5,0), (6,1), (7,0), \ldots\}$.

See **Lecture 5 slide 15**.

(b) A set $S \subseteq N$ is call RM decidable if its characteristic function χ_S is a register machine computable function. Otherwise it is called undecidable.

See **Lecture 5 slide 16**.

Exercise 4 Describe two strategies for proving that a function is uncomputable. Describe a strategy for proving that a set is undecidable.

One strategy is to use a diagoninalisation argument, where the same program is applied to some modified program in order to reach a contradiction (as in the Halting problem). An alternative strategy (and more common) is to assume that a function is computable and the show that a partial function known to be uncomputable can be computed.
For sets, a similar set of strategies is used.

Exercise 5 [**Uncomputable/Undecidable**] Do as many of the following until you are confident with proving undecidability/uncomputability:

(a) Show that the set $\{x \mid \phi_x(0)\downarrow\}$ is undecidable.

(b) Show that the function $\{(x, 0) \mid \phi_x(x)\downarrow\}$ is uncomputable.

(c) Show that $\{e \mid \phi_e \text{a total function}\}$ is undecidable.

(d) Attempt [**2015P6Q3 (c)**] (you can take recursive to mean RM computable here).

(e) Attempt [**2014P6Q3 (e)**].

(f) Attempt [**2011P6Q3 (b)**].

(g) Attempt [**2009P6Q3 (b)**].

(h) Attempt [**2005P4Q9 (b)**].

(i) (optional) [**2008P5Q10 (c)**].

(a) We will prove a reduction to the halting problem. Assume we want to determine if program R terminates on input d, then we construct $R' = [\mathtt{R}_1 := d; A]$ (i.e. $[\mathtt{R}_1^+ \ldots \mathtt{R}_1^+; A]$) and see that R' terminates iff R terminates when run on d. Hence, deciding if a program terminates with 0 input is undecidable.

$\boxed{\text{See } \textbf{Lecture 5 slide 19}.}$

(b) (**Solution 1**) Assume that there is an RM K that computes this function. Then $K(K)$ terminates iff $K(K)\uparrow$ iff $K(K)$ does not terminates (contradiction).

(**Solution 2**) Alternatively, we can decide the set of part (a) given RM R, then construct $R' = [\mathtt{R}_1 := 0; R]$, which halts on input R' (or any input) iff R halts on input R.

(c) (**Solution 1**) Again, we will prove a reduction to the halting problem. Assume we want to determine if program R terminates on input d, then we construct $R' = [(\textbf{if } \mathtt{R}_1 = d \textbf{ then } A \textbf{ else } 0]$. So, R' terminates on all inputs iff A terminates for the input d. Hence, deciding if a program terminates on all inputs is undecidable.

(**Solution 2**) Alternatively, we can defined $R' = [\mathtt{R}_1 := 0; A]$ and user part (a).

$\boxed{\text{See } \textbf{Lecture 5 slide 19}.}$

(d) The idea is that if we knew an upper bound for this we could emulate RM for $r(n)$ steps. If the RM halts during the emulation, then we know it halts. Otherwise, if the limit is reached before the RM halts, it means (by definition of r) that the machine will not terminate. Hence, we can decide the halting problem.

$\boxed{\text{See official solution notes}}$

(e) Assume we could decide the set, then we can also decide whether an RM program R terminates on input d. Consider the program $R' = [R_1 := d; R'']$ where R'' is R with `HALT` instructions are replaced by $[\mathtt{R}_1 := 1; HALT]$. Hence, R terminates on input d iff R'' is equivalent to *one*.

$\boxed{\text{See official solution notes}}$

(f) (b)(i) S is not computable. T is computable.

(b)(ii) Informally, T can be computed by a program that, given input P, runs the universal register machine with input P and all registers set to 0, while counting the number of steps. A full construction of the universal machine is not required, though a clear statement of its existence should be given.

To show that S is not computable, we rely on the undecidability of the halting problem. Again, a precise statement must be given. We can show that if S were computable, then the halting problem would be decidable by the algorithm that, given P first computes $S(P)$. If the result is non-zero, then return "yes". Otherwise, if P is the simple program `Halt`, return "yes". Otherwise, return "no".

See official solution notes

(g) Given an RM R construct the following program $R' = [\textbf{if } (R, d) \in S \textbf{ then } 0 \textbf{ else } (R'', d) \in S]$, where R'' is R where `HALT`s are replaced by simple loops. Hence, program R halts on input d iff R'' loops.

See official solution notes

(h) Let x be an input for which e_0 and e_1 disagree. Consider RM R and input d, then construct $R' = [\textbf{if } \texttt{R}_0 = d \textbf{ then } R'' \textbf{ else } e_1(x)]$, where R'' is R with `HALT`s (proper and improper) replaced by $[e_0(x); \texttt{HALT}]$. Hence, the R terminates for input d iff R' is equal to e_0.

Exercise 6 Show that there is a register machine computable partial function $f : \mathbb{N} \to \mathbb{N}$ such that both $\{x \in \mathbb{N} \mid f(x) \downarrow\}$ and $\{y \in \mathbb{N} \mid \exists x \in \mathbb{N}. f(x) = y\}$ are register machine undecidable.

[Exercise 4 in Lecturer's handout]

Consider the partial function $f(x) = \begin{cases} x & \text{if } \phi_x(x) \downarrow \\ \uparrow & \textbf{otherwise} \end{cases}$. Hence, if either set is decidable, then we can decide the halting problem.

Exercise 7 Suppose S_1 and S_2 are subsets of $\mathbb{N}$. Suppose $f : \mathbb{N} \to \mathbb{N}$ is register machine computable function satisfying: for all $x \in \mathbb{N}$, x is an element of S_1 if and only if $f(x)$ is an element of S_2. Show that if S_2 is RM decidable, then so is S_1.

[Exercise 5 in Lecturer's handout]

Since S_2 is RM decidable, it means that that there exists a RM that decides S_2, i.e. χ_{S_2} is RM computable. Hence, construct a machine that given x, computes $f(x)$ (which is given to be computable and then returns the output of $\chi_{S_2}(f(x))$.

Exercise 8

(a) Show that the set of codes $\langle e, e' \rangle$ of pairs of numbers e and e' satisfying $\phi_e = \phi_{e'}$ is undecidable.

[Exercise 8 in Lecturer's handout]

(b) Show that for any fixed e, the set of codes of equivalent programs is undecidable.

(c) Show that checking if a program e changes the contents of register n is undecidable.

(d) Attempt [**1996P3Q9 (c)**].

(a) One way to prove this is that if this set were decidable, then we could check equivalence any program e and the program *one*, by simply checking $\chi_S(\langle e, \mathit{one}, \rangle) = 1$. But we showed in Exercise 5(e).

(b) Let e be an arbitrary code. Then $\phi_e(0)$ is either defined or undefined. Let e' be another arbitrary code, then we can create code e'' such that

$$\phi_{e''}(x) = \begin{cases} \phi_{e'}(0) & \text{if } x = 0 \\ \phi_e(x) & \text{otherwise} \end{cases}.$$

Hence, e'' is equivalent to e iff they agree at 0. Hence, we can determine if an arbitrary program terminates when given the input 0. But as we showed in Exercise 5(a), this problem is undecidable.

(c) Given an RM program e, we can obtain e' by remapping the registers, so that the program makes no reference to $\texttt{R}_n$ (A possible mapping being $0 \mapsto 0, 1 \mapsto 1, \ldots (n-1) \mapsto (n-1), n \mapsto (n+1), (n+1) \mapsto (n+2)$). Then we can obtain e'' from e' by converting all proper and improper `HALT`s to $[\texttt{R}_n := 1, \texttt{HALT}]$. So $\texttt{R}_n$ will be assigned iff the program is going to halt.

(d) Similarly to the previous exercise we can replace `HALT`s in the last step by setting all registers to 0.

Exercise 9 **[Recursive enumerability]** *Recursive enumerability* is a concept that does not appear in the lecture notes, but it appears in several past papers.

(a) Attempt [**2012P6Q4**] (or [**2020P6Q5**]).

(b) Attempt [**2009P6Q4**].

(c) Attempt [**2008P6Q10**].

(d) (optional) Attempt [**1996P4Q8**].

(e) (optional) Attempt [**2000P4Q8**].

(a) .

See official solution notes

(b) .

See official solution notes

(c) .

See official solution notes

Exercise 10 Are there any practical implications of undecidability?

Undecidability is useful is understanding the limits of computation and knowing them when trying to tackle problems. For example, with register machines (or any of the other equivalent formalisms) we cannot create a program that checks that every execution will terminate or that executions will terminate with a particular property being true (in the general case). This encourages (as it is the only option in these formalisms) the development of special purpose tools that allow to get provable guarantees in constrained environments.

Exercise 11 Define your own undecidable problem.

Lecture 6 / Lecture 7 (first part)

Further reading:

- Chapter 3.1 and 3.3 in M. Sipser's "Introduction to Computation Theory".

Exercise 12

(a) Define *Turing Machines (TMs).* (See [**2012P6Q3 (a)**] or [**2006P3Q7 (b)(i)**] or [**2004P3Q7 (b),(c)**])

(b) Define a *TM computation.*

(c) Define what it means for a partial function to be *TM computable.* (See [**2012P6Q3 (b)**])

(d) Explain briefly how to enumerate all possible TM computations, so that a given computation can be characterised by a single natural number code c. (See [**2001P3Q9 (c)**])

(a) A Turing Machine $M = (Q, \Sigma, s, \delta)$, where Q are the states, Σ is the set of tape symbols (including $\triangleright$ the start symbol and ␣ the black space, $s \in Q$ is the initial state, and $\delta \in (q \times \Sigma) \to (Q \cup \{\text{acc}, \text{rej}\}) \times \Sigma \times \{L, R, S\}$ is the transition function, which satisfies for every $q \in Q$, $\delta(q, \triangleright) = (q', \triangleright, R)$ for some q' (i.e. the left endmarker is not overwritable).

See **Lecture 6 slide 11**.

(b) A TM computation is a countable sequence of configurations $c_0, c_1, \ldots$ where $c_0 = (s, \triangleright, u)$ is an initial configuration and $c_i \to_M c_{i+1}$ for each i. See the slides below for the definition of the configuration and the transition $\to_M$.

See **Lecture 6 slide 13**.

See **Lecture 6 slide 22**.

(c) Each computation corresponds to a single TM T and an input x. We can define an encoding for a TM $M = (Q, \Sigma, s, \delta)$, as follows :

- $\ulcorner Q \urcorner = |Q|$ and $\ulcorner \Sigma \urcorner = |\Sigma|$ and map each state to an natural number in $[|Q|] = \{0, 1, \ldots, |Q| - 1\}$ and $[|\Sigma|] = \{0, 1, \ldots, |\Sigma| - 1\}$. (This also defined the encoding for $s \in \Sigma$).
- δ can be defined as a collection Δ of quintuples $\Delta_i = (q_i, t_i, q_i', t_i', L_i)$. Naturally, we define $\ulcorner \Delta_i \urcorner = \ulcorner [\ulcorner q_i \urcorner, \ulcorner t_i \urcorner, \ulcorner q_i' \urcorner, \ulcorner t_i' \urcorner, \ulcorner L_i \urcorner] \urcorner$ and $\ulcorner \delta \urcorner = \ulcorner [\ulcorner \Delta_1 \urcorner, \ldots, \ulcorner \Delta_1 \urcorner] \urcorner$.

Hence, we can define the encoding of the TM as $\ulcorner M \urcorner = \ulcorner [\ulcorner Q \urcorner, \ulcorner \Sigma \urcorner, \ulcorner s \urcorner, \ulcorner \delta \urcorner] \urcorner$.

(Solution for terminating configurations) Each configuration consists of the state q, the non-empty string w on the left of the head and the non-empty string u on the right of the head. Hence, an encoding for this can be defined using the encoding function for lists, so that $\ulcorner [\ulcorner q \urcorner, \ulcorner w \urcorner, \ulcorner u \urcorner] \urcorner$, where $\ulcorner q \urcorner$ is just a numbering of the states and $\ulcorner w \urcorner$ (and $\ulcorner u \urcorner$) is an encoding of a list of symbols, i.e. $\ulcorner w \urcorner = \ulcorner [w_1, \ldots, w_{|w|}] \urcorner$. Finally a terminating computation is a finite sequence of configurations $c_0 \to c_1 \to \ldots \to c_N$. So we can encode it as $\ulcorner [c_0, c_1, \ldots, c_N] \urcorner$.

Exercise 13 For the example Turing machine given on **L6S12**, give the register machine program implementing $(S, T, D) := \delta(S, T)$, as described on **L6S30**. [**Note (by lecturer):** Tedious!—maybe just do a bit.] [**Note (by supervisor):** Implement this in Java if you prefer]

[Exercise 7 in Lecturer's handout]

Exercise 14 [RMs can simulate TMs]
(a) Describe the steps in simulating a TM using an RM. (See [**2004P3Q7**] or [**1998P3Q9**])
(b) Attempt [**2012P6Q3 (d)**].

(a) There are the following main steps in simulating a TM using an RM:

1. Fix a number encoding of M's states, tape, symbols and tape and configurations.
2. Implement M's transition function (finite table) using RM instruction codes.
3. Implement a RM program to repeatedly carry out $\to_M$.

See **Lecture 6 slide 27**.

(b) .

See official solution notes

Exercise 15 [TM computable]
(a) Explain how lists of naturals are represented on TM tapes.
(b) Define what it means for a partial function to be *TM computable*.

(a) A tape over $\Sigma = \{\rhd, \llcorner\!\lrcorner, 0, 1\}$ codes a list of naturals if precisely two entries are 0 (marking the beginning and ending of the list) and the only cells containing 1 occur between these. The number of 1s between two consecutive blacks, gives the natural in the list at that point.

See **Lecture 7 slide 4**.

(b) A partial function is TM computable if there is a TM M such that starting M from its initial state, the head on the left endmarker and the tape coding $[0, x_1, \ldots, x_n]$, M halts iff $f(x_1, \ldots, x_n) \downarrow$, and in the case of the final tape codes a list (of length ≥ 1) whose first element y where $f(x_1, \ldots, x_n) = y$.

See **Lecture 7 slide 6**.

Exercise 16 [TMs can simulate RMs] Describe the high-level steps for simulating an RM using a TM.

There are a few ways of doing this. One way is to keep a list of all register contents on the tape. Then we need to implement the operation to retrieve the i-th element on the list and then increment or decrement it by one and then proceed to the next instruction. The instruction jumping happens using the δ transition function of the TM.
An alternative solution is the following: since we know the number n of registers used by the program, we can define the i-th symbol on the TM tape to be the contents of the i-th element of the registers (so it will be a

n-tuple consisting of 0s and 1s). For example, if we had registers $\mathtt{R}_1 = 3, \mathtt{R}_2 = 6, \mathtt{R}_3 = 2, \mathtt{R}_4 = 0$, then these would be represented by:

(1,1,1,0)	(1,1,1,0)	(1,1,0,0)	(0,1,0,0)	(0,1,0,0)	(0,1,0,0)	(0,0,0,0)	(0,0,0,0)

We can do this by creating a symbol for each of the 2^n possible configurations. (This is not a problem since n is constant). If we want to increment register i then we start from the leftmost part of the tape and we move to the right as long as we encounter a 1 at the i-th position. In the first tape symbol that we encounter with a 0 in position i, we find change it to a 1. This changes should be implemented in the transition function so that given that the current state is $(\ldots, 0, \ldots)$, we should go to $(\ldots, 1, \ldots)$. A decrement operation can be implemented similarly with the only exception that we need to check if the register is already empty.

See **Lecture 7 slide 7**.

Exercise 17 [TM problems]

(a) Given a Turing machine, is it decidable whether or not for all possible initial configurations the machine will not halt after 100 steps of transition? Justify your answer. [**2006P3Q7 (b)(ii)**].

(b) Show that it is not possible to compute the maximum distance travelled by the Turing machine head from its initial position during halting computations as a function of the code c. Any results that you use should be stated clearly. [**2001P3Q9 (c)**]

(c) Show that it is not possible to compute a bound on the distance of the head from its starting position during HALTing Turing machine computations. [**1997P3Q9 (c)**]

(d) Show that there is no way of deciding by algorithm whether the blank character will be printed during the course of a general Turing machine computation. [**1993P5Q10 (b)**]

(e) Create your own TM undecidable program.

(a) Yes, it is decidable. In the first 100 steps of the computation, at most 100 symbols from the tape can be accessed. So, the number of possible initial configurations. This is a finite number, so we can try all the possible starting configurations emulate the Turing Machine and halt iff for all possible starting configurations, the TM halted.

(b) If we know that the maximum distance travelled by the head is d, then we can find an upper bound on the number of configurations. An upper bound to the number of possible configurations is $K = (\text{possible tape contents}) \cdot (\text{possible states}) \cdot (\text{possible head positions}) = |\Sigma|^d \cdot |Q| \cdot d$. We can simulate the TM for $K + 1$ steps. If it terminates then we are done. Otherwise, it must have visited $K + 1$ different configurations from the K possible. Hence, by the pigeon-hole principle the same state must have been visited twice. But then this means that the TM computation will loop indefinitely between these two configurations. So, we know that it cannot terminate.

So, being able to compute this code would mean that the machine is able to decide the halting problem, which we know to be undecidable. Hence, this problem must be uncomputable.

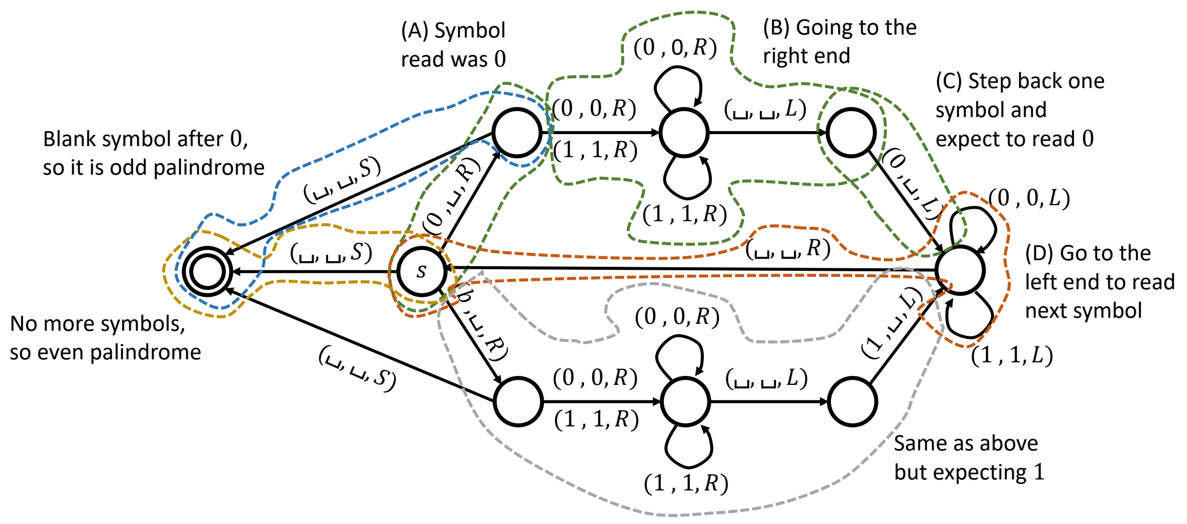
(c) Similarly to the previous question, if we could compute an upper bound, then we would simulate for the upper bound instead of the exact value.

(d) This is similar to deciding if an RM program uses a register $\mathtt{R}_n$. Given a TM T we replace all usages of the black symbol with a new symbol σ to obtain TM T'. Then we replace every halt with a step that writes the black character on the tape and then undoes this, to obtain TM T''. Hence, T halts iff T'' write the empty symbol at some point on the tape.

Note: It is also possible to argue that in some problems, the TM cannot see only a finite amount of memory. For example, when checking if a string is palindrome. (But then one can correct this and say that the maximum distance is in terms of the code and the input).

Exercise 18 Design a TM that checks if the string $s \in \{0, 1\}^*$ on the input is a palindrome.

All arrows not appearing in the following diagram are considered to be rejecting.


(A) Symbol read was 0
(B) Going to the right end
(C) Step back one symbol and expect to read 0
(D) Go to the left end to read next symbol
Blank symbol after 0, so it is odd palindrome
No more symbols, so even palindrome
Same as above but expecting 1
s
(0 , 0, R)
(1 , 1, R)
(␣, ␣, L)
(0, ␣, L)
(0 , 0, L)
(1 , 1, L)
(␣, ␣, R)
(␣, ␣, S)
(0, ␣, R)
(1, ␣, R)
(1, ␣, L)

Exercise 19 [Multi-tape TM] (optional)

(a) Show that a TM with multiple tapes and heads can be simulated by a classical TM.

(b) Does this machine have any advantage over classical TMs?

(a) The multi-tape and multi-head TM works by having a head on a constant number of k tapes and in each step moving the heads by considering all symbols under the k heads. A Turing Machine can simulate the multi-tape/multi-head TM by keeping a tape where each symbol on the tape will encode the symbols at that position for the k tapes in addition to whether the i-th head is there. In the simulation the TM will (i) collect the symbol under each head and (ii) apply the change for each of the heads. To achieve (i), the TM will first loop through the entire tape searching for the marker of the 0-th head, then for the marker of the 1-th head and so on. Once all of these have been collected, the TM can determine (by "looking" at the transition tape of the multi-tape/multi-head machine) what the new states should be. Finally, similarly to (i), it will loop through all the head markers to update the symbols.

This would be quite tedious to explain rigorously. But it might be easier to write a Java program which emulates this (and is "equivalent" to an RM which is equivalent a TM program).

(b) There is an advantage in terms of computational complexity. We can have a TM that decides the language of palindromes in linear time. The idea is that we copy the palindrome in the second tape and we set the head in the second tape at the of the string. Then in each step we compare the characters under the head in the first and the head in the second tape. If we find a mismatch, then it means that the string is not palindrome. Otherwise, we move the first head one position to the right and the second head one position to the left. When the first head reaches the end of the string, then we check if over and the string can be accepted. This takes only a linear (to the length of the string) number of operations.

Exercise 20 [Non-deterministic TM] (optional - connection to Complexity Theory) Show that a *non-deterministic TM*, i.e. one that can perform more than one operations at a single step (similar analogy between DFAs and NFAs) is equivalent to a TM. Does this machine have any advantage over classical TMs?