CS 3100 – Models of Computation – Fall 2011 – Notes for L21

1 Turing/Church Thesis

Two formalisms, namely Turing machines and Church's Lambda Calculus, both serve to define what is effectively (mechanically) computable. There are plenty of online resources that discuss the origins of these notions. Below are some historical perspectives from the Encyclopedia of Philosophy web site `stanford-ency-phil`. We now provide direct excerpts from this web site.

(Begin excerpts)

The Princeton logician Alonzo Church had slightly outpaced Turing in finding a satisfactory definition of what he called *effective calculability.* Church's definition required the logical formalism of the Lambda calculus. This meant that from the outset Turing's achievement merged with and superseded the formulation of Church's Thesis, namely the assertion that the Lambda calculus formalism correctly embodied the concept of effective process or method. Very rapidly it was shown that the mathematical scope of Turing's computability coincided with Church's definition (and also with the scope of the general recursive functions defined by Gödel). Turing wrote his own statement (Turing 1939, p. 166) of the conclusions that had been reached in 1938; it is *in the Ph.D. thesis that he wrote under Church's supervision*, and so this statement is the nearest we have to a joint statement of the Church Turing thesis:

> A function is said to be *effectively calculable* if its values can be found by some purely mechanical process. Although it is fairly easy to get an intuitive grasp of this idea, it is nevertheless desirable to have some more definite, mathematically expressible definition. Such a definition was first given by Gödel at Princeton in 1934. These functions were described as *general recursive* by Gödel. Another definition of effective calculability has been given by Church who identifies it with lambda-definability. The author [i.e. Turing] has recently suggested a definition corresponding more closely to the intuitive idea. It was stated above that a function is effectively calculable if its values can be found by a purely mechanical process. We may take this statement literally, understanding by a purely mechanical process one which could be carried out by a machine. It is possible to give a mathematical description, in a certain normal form, of the structures of these machines. The development of these ideas leads to the author's definition of a computable function, and to an identification of computability with effective calculability. It is not difficult, though somewhat laborious, to prove that these three definitions are equivalent.

Church accepted that Turing's definition gave a compelling, intuitive reason for why Church's thesis was true. The recent exposition by Davis (2000) emphasizes that Gödel also was convinced by Turing's argument that an absolute concept had been identified (Gödel 1946). The situation has not changed since 1937 (see how Prof. Geoff Draper viewed it - Figure 1). By the way, Professor Geof Draper (Univ of Utah alum) has a fabulous set of cartoons – check out at `http://draperg.cis.byuh.edu/cartoons/`.

2 Formal Definition of a Turing machine

A Turing machine M is a structure $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$. The finite state control of M ranges over the control states in Q, beginning at the initial control state $q_0 \in Q$. States $F \subseteq Q$ of M are called *final* states, and are used to define when a machine *accepts* a string. The input on which M operates is initially presented on the input tape. It is a string over the input alphabet Σ. Once started, it is possible that a Turing machine may

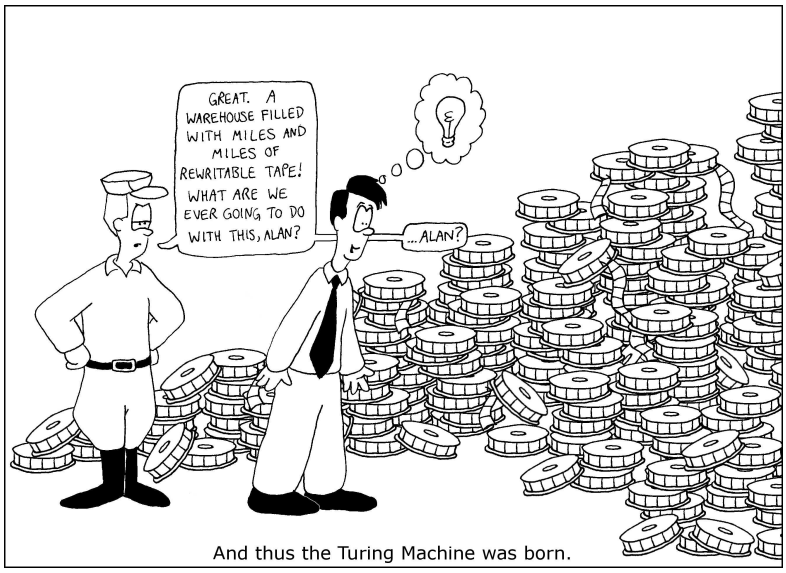

Figure 1: Alan Turing's "light bulb moment" – Prof. Geoff Draper drew this cartoon for my book

never halt; it may keep zigzagging on the tape, writing symbols all over, and running amok, much like many tricky programs do in real life. A Turing machine cannot manufacture new symbols ad infinitum - so all the symbols written by a Turing machine on its tape do belong to a *finite* tape alphabet, Γ. Notice that $\Sigma \subset \Gamma$, since Γ includes the *blank* symbol B that is not allowed within Σ. We assume that a Turing machine begins its operation scanning cell number 0 of a doubly-infinite tape (meaning that there are tape cells numbered $+x$ or $-x$ for any $x \in Nat$); more on this is in the following section. A fact to remember is this: in order to feed the string ε to a TM, one must present to the Turing machine a tape *filled* with blanks (B). However, some authors alter this convention slightly, allowing ε to be fed to a Turing machine by ensuring that, in the initial state, the symbol under the tape head is blank (B) (*i.e.,* the rest of the tape could contain non-blank symbols). In any case, a normal Turing machine input is such that for every $i \in length(w)$, $w[i] \neq B$ is presented on tape cell i, with all remaining tape cells containing B, and the head of the Turing machine faces $w[1]$ at the beginning of a computation.

A TM may be deterministic or nondeterministic. The signature of δ for a deterministic Turing machine (DTM) is

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}.$$

This signature captures the fact that a TM can be in a certain state $q \in Q$ and looking at $a \in \Gamma$. It can then write $a^{'}$ on the tape in lieu of a, move to a state $q^{'}$, and move its head left (L), or right (R), depending on whether $\delta(q, a) = \langle q^{'}, a^{'}, L\rangle$ or $\delta(q, a) = \langle q^{'}, a^{'}, R\rangle$, respectively.

For an NDTM, $\delta(q, a)$ returns a set of next control states, tape symbol replacements, and head move directions. The signature of δ for a nondeterministic Turing machine (NDTM) is

$$\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L,R\}}.$$

Think of a nondeterministic Turing machine as a C program where instead of the standard if-then-else construct, we have an `if/fi` construct of the following form:

```
if :: condition_1 -> action_1
   :: condition_2 -> action_2
   :: ...
   :: condition_n -> action_n
fi
```

The intended semantics is as follows. Each `condition` may be a Boolean expression such as `(x > 0)`. There may be more than one condition becoming true at any given time. In that case, one of the conditions is *nondeterministically* chosen. The `action`s can be `goto`, `assignment`, `while-loops`, etc., as in a normal C program. With *just* this change to the C syntax, we have a class of nondeterministic C programs that are equivalent to nondeterministic Turing machines. Nondeterministic Turing machines are powerful conceptual devices that play a fundamental role in the study of *complexity theory*.

> *In these notes, we shall treat Turing machines and programs synonymously. Therefore, "build a program" will also mean "build a Turing machine."*

2.1 Two stacks+control = Turing machine

A TM can be viewed as finite-state control coupled with *two* stacks, by modeling the infinite tape using two stacks:

- When a TM goes one step left from its current position, the tape segment to the left of the head shrinks by one cell while the segment to the right of the head grows by one segment. This can be viewed in terms of popping the left-hand stack and pushing the right-hand stack of the finite-state control.
- A TM taking one step to the right can be viewed in terms of popping the right-hand stack and pushing onto the left-hand stack.

3 Acceptance, Halting, Rejection

Given a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, the transition function δ is *partial* - it need not be defined for all inputs and states. When not defined, the machine becomes "stuck" - and is said to *halt*. Therefore, note that **a TM can halt in any state**. Also, by convention, no moves are defined out of the control states within F. Therefore, **a TM always halts when it reaches a state** $f \in F$. Here are further definitions:

- A TM **accepts** when halting at $f \in F$.
- A TM **rejects** when halting in any other state outside F.
- A TM **loops** when not halting.

3.1 "Acceptance" of a TM closely examined

Compared to DFAs and PDAs, the notion of acceptance for a TM is unusual in the following sense: a TM can accept an input without fully reading the input—or, in an extreme situation, *not even reading one cell of the input* (e.g., if $q_0 \in F$)! Hence, curiously, any TM with $q_0 \in F$ has language Σ^*. On the other hand, if a TM never accepts, its language is $\emptyset$. This can be the result of the TM looping on every input, or the TM getting stuck in a reject state for every input. As a final illustration, given an arbitrary language L_1, a TM that accepts any string within the set L_1 and does not accept (loops or rejects) strings outside L_1, has L_1 as its language.

3.2 Instantaneous descriptions

While capturing the 'snapshot' of a TM in operation, we need to record the control state of the machine, the contents of the tape, and what the head is scanning. All these are elegantly captured using an *instantaneous description* containing the tape contents, w, with a single state letter q placed somewhere within it. Specifically, suppose that the string lqr represents that the tape contents is lr, the finite-state control is in state q, and the head is scanning $r[0]$. The initial ID is q_0w, for initial input string w. We define $\vdash$, the 'step' function that takes IDs to IDS, as follows:

> $l_1par_1 \vdash l_1bqr_1$ if and only if $\delta(p, a) = (q, b, R)$. The TM changes the tape cell contents a to b and moves right one step, facing $r_1[0]$, the first character of r_1.
>
> Similarly, $l_1cpar_1 \vdash l_1qcbr_1$ if and only if $\delta(p, a) = (q, b, L)$. The TM changes an a to a b and moves left one step to now face c, the character that was to the left of the tape cell prior to the move.

We define the language of a TM M using $\vdash^*$:

$$L(M) = \{w \mid q_0w \vdash^* lq_fr, \ for\ q_f \in F, \ and\ l, r \in \Sigma^*\}.$$

In other words, a TM that starts from the initial ID q_0w and attains an ID containing q_f (namely lq_fr, for some l and r in Σ^*) ends up accepting w.

4 Examples

We now present short examples that illustrate various concepts about Turing machines (Section 4.1). This is followed by a deterministic Turing machine that accepts strings of the form $w\#w$ for $w \in \Sigma^*$ (Section 4.2). Section 5 introduces NDTMs, and Section 5.2 presents an NDTM that accepts strings of the form ww.

Turing machines are specified fully at the *implementation level* by completely specifying their δ function (or δ relation for NDTMs) in a tabular form as in Figure 2 or diagrammatic form as in Figure 4. After gaining sufficient expertise with Turing machines, we will allow them to be specified at the *high level* through pseudo-code or precise English narratives. When resorting to high-level descriptions, the specification writer must strive to ensure sufficient clarity so that a reader can, in principle, reconstruct an implementation level description if necessary.

4.1 Examples illustrating TM concepts and conventions

Illustration 4.1 Consider the Turing machine with a doubly-infinite tape $M = (Q, \Sigma, \Gamma, \delta, q0, B, F)$, where $Q = \{q0, qa\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, B\}$, $F = \{qa\}$, and δ is as below:

$\delta(q0, 0) = (q0, 0, L)$

$\delta(q0, 1) = (q0, 1, L)$

$\delta(q0, B) = (q0, B, L)$.

This TM will not stop running - it will keep moving left (even this is called "looping"). Its language is $\emptyset$.

Illustration 4.2 Now consider changing δ to the following:

$\delta(q0, 0) = (qa, 0, L)$

$\delta(q0, 1) = (qa, 1, L)$

$\delta(q0, B) = (qa, B, L)$.

Now the language of this Turing machine is Σ^*. Notice that we need not specify moves for state `qa` for input `B`. In other words, this Turing machine will move one step to the left and get "stuck" in state q_a which is accepting.

Illustration 4.3 Suppose the entries for (qa,1,L) as well as (qa,B,L) are removed from the transition table given in Illustration 4.2. The language of this Turing machine will then be $0(0+1)^*$. This is because:

- Neither $\delta(q0, 1)$ nor $\delta(q0, B)$ is specified—hence, the input has to begin with a 0.
- State qa accepts everything. Hence, $(0+1)^*$ can follow.

	q0	q1	q2	q3	q4	q5	q6
q0	-	a; X, R	-	-	Y; Y, R	-	B; B, R
q1	-	a; a, R Y; Y, R	b; Y, R	-	-	-	-
q2	-	-	Z; Z, R b; b, R	c; Z, L	-	-	-
q3	X; X, R	-	-	b; b, L Y; Y, L a; a, L Z; Z, L c; c, L	-	-	-
q4	-	-	-	-	Y; Y, R	Z; Z, R	-
q5	-	-	-	-	-	Z; Z, R	B; B, R
q6	-	-	-	-	-	-	-

Figure 2: A TM for $\{a^n b^n c^n \mid n \geq 0\}$, with start state `q0`, final state `q6`, and moves occurring from the row-states to column-states

4.1.1 A Turing machine for $a^n b^n c^n$

A Turing machine that recognizes $L_{anbncn} = \{a^n b^n c^n \mid n \geq 0\}$ is given in Figure 2 in the JFLAP tool's saved-table syntax. The algorithm implemented is one of turning `a`'s into `X`'s, `b`'s into `Y`'s, and `c`'s into `Z`'s. Here is an execution on `aabbcc`:

- We start in state `q0` where we seek an `a`, changing it to an `X` when we see one, and at the same time entering state `q1`.
- In `q1`, we skip over `a`'s going right, while staying in `q1`. Upon encountering a `b`, we change it to a `Y`, and move over to state `q2`.

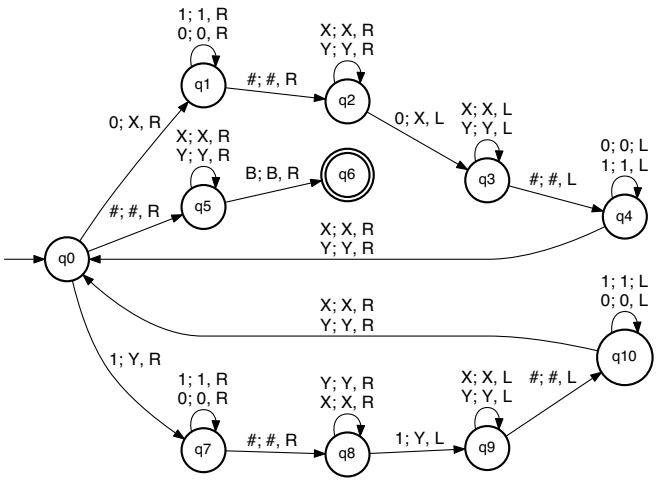

Figure 3: A Turing machine for $w\#w$

- In `q2`, we move over b's going right, until we encounter a `c`, turning it into a `Z`, and then move over to state `q3`, and start moving in the left (`L`) direction.
- Notice that in `q3`, we keep moving left upon seeing any one of `b, Y, a, Z, or c`, and stop only when we see an `X`. Then we sweep over the input once again.

In `q0`, encountering an `X` is impossible (why?). Also notice that we explicitly decode the B (blank) input case in the `q0` state. While in `q0`, encountering a `Z` is impossible. We may, however, encounter a `Y` (e.g., consider the input 'abc' which would be changed to XYZ; and then in state `q3`, we would skip over the `X` and be in state `q0` facing a `Y`). The reader is invited to argue that this DTM is correct with respect to the advertised language $L_{a^n b^n c^n}$.

4.2 A DTM for $w\#w$

In Figure 3, we provide a deterministic Turing machine for the language of strings of the form $w\#w$, where $w \in \Sigma^*$. Notice how the presence of `#` allows the midpoint to be deterministically located. The Turing machine basically hovers to either side of `#`, scoring off matching characters.

5 NDTMs

An NDTM is a Turing machine with nondeterminism in its finite-state control, much like we have seen for earlier machine types such as PDAs. To motivate the incorporation of nondeterminism into Turing machines, consider a problem such as determining whether an undirected graph G has a clique (a completely connected

subgraph) of k nodes.[1] No efficient algorithm for this problem is known: all known algorithms have a worst-case *exponential* time complexity. Computer scientists have found that there exist thousands of problems such as this that arise in many practical contexts. They have not yet found a method to construct a *polynomial* algorithm for these problems. However, they have discovered another promising approach:

> They have found a way to formally define a class called "NP-complete" such that finding a polynomial algorithm for even *one* of the problems in the NP-complete class will allow one to find a polynomial algorithm for *all* of the problems in the NP-complete class. Furthermore, most of these thousands of problems that have confounded scientists have been shown to belong to the NP-complete class.

In short, scientists now have the strong hope of resorting to the maxim "solving one implies solving all," meaning solving even one NP-complete problem using a polynomial-time algorithm will provide a polynomial-time algorithm for the thousands of known NP-complete problems.

The aforesaid techniques rely on measuring the runtime of nondeterministic algorithms in a certain way, briefly described as follows:

> If an NDTM can solve a certain problem X in polynomial-time, then the problem belongs to the class NP. If, in addition, problem X belongs to the NP-hard class, then this combination (being in NP and NP-hard) ensures that X is in the NP-complete class. X belonging to the NP-hard class means that ever problem in NP has a polynomial transformation (mapping reduction) to X.

5.1 Guess and check

While all this may sound bizarre, the fundamental ideas are quite simple. The crux of the matter is that many problems can be solved by the "guess and check" approach. For instance, finding the prime factors of very large numbers is hard. In Pratt's 1976 paper, an interesting observation is provided: it was conjectured by Mersenne[2] that $2^{67} - 1$ is prime. This conjecture remained open for two centuries until Frank Cole showed, in 1903, that it wasn't; in fact, $2^{67} - 1 = 193707721 \times 761838257287$. Therefore, if we could somehow have guessed that 193707721 and 761838257287 are the factors of $2^{67} - 1$, then checking the result would have been extremely easy! The surprising thing is that there are two gradations of "difficulty" among problems for which only exponential algorithms seem to exist:

- those for which short guesses exist, *and* checking the guesses is easy, and
- those for which the existence of short guesses is, as yet, *unknown*.

To sum up:

- If, for a problem p, we can generate a "short guess" and check the guess efficiently, then p belongs to the class NP. *Clique* is in NP because a "guess" will be short (simply write out k of the graph nodes) and the "check" is easy (see if these nodes include a k-clique).
- If a problem is NP-complete, it is believed to be unlikely that it will have a polynomial algorithm, although this issue is open.

[1] If you walk into a room full of people, and imagine drawing a graph of *who knows each other mutually*, then a k-clique exists wherever all pairs within a group of k people know each other. Whether there exists a group of such "tight-knit" people is, essentially, the clique problem.

[2] Prime numbers of the form $2^p - 1$ are known as Mersenne primes. The 42nd known Mersenne prime was discovered on February 18, 2005 and is $2^{25964951} - 1$, having 7,816,230 decimal digits !!

- Problems for which the guesses are not short, and also checking guesses is not easy, do not belong to NP. Hence, these problems are thought to be much harder to solve.

For instance, $\overline{Clique}$ is the problem: "the given graph does not have a k-clique." There is no known way to produce a succinct guess of a solution for this problem, let alone check the guess efficiently. Every purported solution that a graph *does not* have a k-clique seems to warrant providing a guess of a solution of the form, "this set of k nodes does not span a clique; neither does this other set of k nodes; etc. etc." This may, however, end up enumerating all k-node combinations, which are exponential in number.

NDTMs are machines that make the study of complexity theory in the above-listed manner possible. Their use in defining complexity-classes such as NP-hard and NP-complete forms the main hope for finding efficient algorithms for thousands of naturally occurring NP-complete problems—or to prove that such algorithms cannot exist.

5.2 An NDTM for ww

In Figure 4, we provide a nondeterministic Turing machine for the language of strings of the form ww, where $w \in \Sigma^*$. Letter 'S' in the edge from `q0` to `q2` means that the head stays where it is (can be simulated by an R followed by an L). This TM has to "guess" the midpoint; this happens in the initial nondeterministic loop situated at state `q2`. Notice that the Turing machine can stay in `q2`, skipping over the 0s and 1s or exit to state `q3`, replacing a 0 with an X or a 1 with a Y. This is how the Turing machine decides the midpoint; after this step, the Turing machine zigzags and tries to match and score off around the assumed midpoint. Any wrong guess causes this check phase to fail. One guess is guaranteed to win if, indeed, the input is of the form ww.

The animation of this NDTM in action using JFLAP would be highly intuitive, and the reader is strongly urged to do so.

6 Simulations

We show that having multiple tapes or having nondeterminism does not change the inherent power of a Turing machine.

6.1 Multi-tape vs. single-tape Turing machines

A k-tape Turing machine simply maintains k instantaneous descriptions. In each step, its δ function specifies how each ID evolves. One can simulate this behavior on a single tape Turing machine by placing the k logical tapes as segments, end-to-end, on a single actual tape, and also remembering where the k tape heads are through "dots" kept on each segment. One step of a k-tape Turing machine now becomes a series of k activities conducted one after the other on the k tape segments.

6.2 Nondeterministic Turing machines

A nondeterministic Turing machine can be conveniently simulated on a single tape deterministic Turing machine. However, it is much more convenient to explain how a nondeterministic Turing machine can be simulated on a multi-tape deterministic Turing machine. For the ease of exposition, we will carry this explanation out with respect to the example given in Figure 4.

We will employ a 3-tape deterministic Turing machine to simulate the NDTM in Figure 4 (hereafter called *ww_ndtm*). The first tape maintains a read-only copy of the initial input string given to *ww_ndtm*. We call it

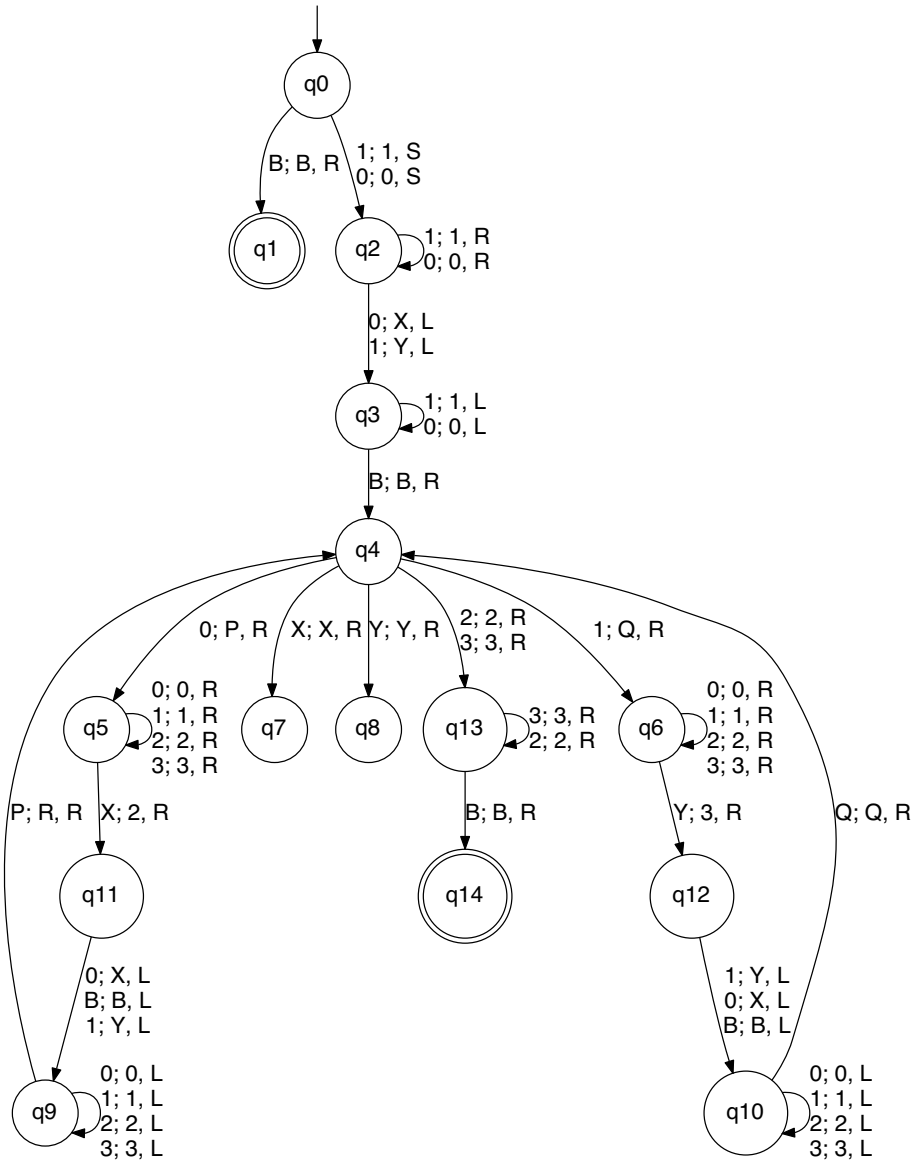

Figure 4: A Nondeterministic Turing machine for ww. Letter 'S' means that the head stays where it is

the input tape. The second tape maintains a tree path in the nondeterministic computation tree of *ww_ndtm*. We call it the tree path tape. The third tape is the "working tape."

6.2.1 Tree path conventions

Notice that state `q2` is the only nondeterministic state in *ww_ndtm*; and hence, the computation tree of *ww_ndtm* will have a binary nondeterministic split every so often—whenever the nondeterminism in `q2` is invoked. The tree paths in *ww_ndtm* can be specified as follows, with the associated computations shown next to it:

ε—the empty computation beginning at $q0$,

0—the computation $q0 \rightarrow q1$ of length 1,

1—the computation $q0 \rightarrow q2$,

$1, 0$—the computation $q0 \rightarrow q2 \rightarrow q2$, and

$1, 1$—the computation $q0 \rightarrow q2 \rightarrow q3$.

We will uniformly use the 0 path for a "self" loop (if any), and 1 for an exit path; example: $1, 0$ and $1, 1$ above. The only exception to this convention is at state $q4$ where there are three exits, and we can number them 0, 1, and 2, going left to right. Therefore, note that the tree path 1,1,1,4,0 refers to the march q0, q2, q3, q4, q13, and back to q13. Notice that we do not have a path $0, 0$ or $0, 1$, as state $q1$ has no successor. When we require the next path in numeric order to be generated below, we skip over such paths which do not exist in the computation tree, and instead go to the next one in numeric order.

6.3 The Simulation itself

Here is how the simulation proceeds, with the second tape (tree path tape) containing ε:

- If the machine has accepted, accept and halt.
- Copy the first (input) tape to the third (working) tape.
- Generate the next tree path in numeric order on the tree path tape.
- Pursue the δ function of the NDTM according to the tree path specified in the tree path tape, making the required changes to the working tape. If, at any particular point, the tree path does not exist or the symbol under the TM tape head does not match what the δ function is forced to look, move on to the next option in the tree path enumeration.
- Repeat the above steps.

For example, if the input $0, 1, 0, 1$ is given on the input tape, the simulation will take several choices, all of which will fail except the one that picks the correct midpoint and checks around it. In particular, note that in state $q2$, for input string $0, 1, 0, 1$, the self-loop option can be exercised at most three times; after four self-loop turns, the Turing machine is faced with a blank on the tape and gets stuck (rejects). The outcome of this simulation is that

> the given NDTM accepts *ww* if and only if the DTM that simulates the NDTM as described above accepts.

The astute reader will, of course, have noticed that we are walking the exponential computation tree of the nondeterministic Turing machine *repeatedly.* In fact, we are not even performing a breadth-first search (which would have been the first algorithm thought of by the reader) because

- Performing breadth-first search (BFS) requires maintaining the frontier
- Even after the extra effort towards maintaining a BFS frontier, the overall gain is not worth it: we essentially might achieve an $O(2^n)$ computation instead of an $O(2^{n+1})$ computation.