Some Notes on Problems and Complexity

Paul Cull

Department of Computer Science
Oregon State University

May 11, 2011

Chapter 1

Nondeterminism

One of the most enduring problems of philosophy is the problem of *free will*:

> *If God can see the outcome, is Man free to make choices?*

We don't plan to or hope to solve this problem, but instead we raise a similar problem for computing machines and programs:

> Do we increase a program's ability to compute things if we allow the program to make "free" choices?

More specifically, if we give programs the freedom to make choices, are there things that these programs can compute which cannot be computed by programs without free choices? Or slightly differently, will the ability to make correct choices allow a program to compute things using **less resources** than those used by a program without free choices?

To make matters worse, we will provide the answers that sometimes free choice doesn't help, sometimes free choice helps, and in the most interesting case, we don't know if free choice helps.

1.1 Nondeterministic Acceptors (and Rejectors)

As in Chapter 1, we have three senses in which we can solve **YES/NO** problems – by recognizers, by acceptors, and by rejectors. There, these machines were embodied as deterministic programs, that is, as programs in our usual programming languages, where which instruction to execute next is strictly determined by the computation that has already occurred. In the simplest situation, the program executes the instruction which is sequentially next after the current instruction. A **GO TO** or **Jump** tells the program which instruction to execute next. A **Branch** statement computes some value, and on the basis of this value the program computes which instruction to execute next. Similarly, for **Loops** or **Recursion**, a computed value tells the program to stay in the loop or continue recursing or to jump out of the loop or bottom out the recursion.

Here we want to generalize our normal programming languages to allow for choices. So, for example, we want to add **GO TO** and **Jump** instructions which allow for several possibilities. We could have

GO TO($L9$, $L917$, $L24$)

which would mean:
execute next the instruction with label $L9$,
or execute next the instruction with label $L917$,
or execute next the instruction with label $L24$.
Similarly, we could allow a **Branch** instruction to branch to one of several alternatives with no value to tell it which of these possibilities to take. For **Loop** or **Recursion** we could allow the program the freedom to stop the loop or the recursion without a specific condition having to be satisfied.

First, let us consider acceptors. A deterministic acceptor, M, accepts an input x exactly when M eventually halts when given x as input. But since a nondeterministic acceptor has choices, how can we say that it halts? For example, the deterministic acceptor:

```
BEGIN
    STOP
END
```

halts on each and every input. On the other hand the nondeterministic acceptor, M_{NON}:

```
BEGIN
    L1: GO TO (L1, L2)
    L2: STOP
END
```

could consistently choose to go to $L1$ and never halt, but if it ever decided to go to $L2$ it would halt. To deal with these choices, we think of the idea of *fork* which is familiar from most operating systems. If we executing M_{NON} and we reach a choice, we *fork* copies of M_{NON}, one copy for each choice alternative. In the example, if we are executing M_{NON} and come to the instruction $L1$, we have two choices, so we *fork* one copy of M_{NON} which takes the $L1$ branch, and we *fork* another copy of M_{NON} which takes the $L2$ branch. The first forked copy reaches $L1$ and because there is a choice, two copies are forked. Instead of immediately following these new copies, we look at the other copy from the first fork. We see that this copy is at $L2$ and therefore it halts. Although we could try to follow the secondary forked copies, it is more reasonable to stop at this point and to say that M_{NON} has halted. What we are saying is that if **any** sequence of choices leads to halt then our nondeterministic acceptor halts and, by our usual definition of acceptor, this acceptor accepts the input. So according to this covention, M_{NON} accepts every input, because for every input there is a sequence of choices which takes M_{NON} to halt.

Following the above discussion, we may say that a nondeterministic program has a set of **computation paths** which consists of the sequences of instructions the program could follow. In our example, "L1: GO TO (L1, L2) ; L2: STOP " would be one computation path and "L1: GO TO (L1, L2) ; L1: GO TO (L1, L2) ; L2: STOP " would be another computation path. The program also has the infinite computation path "L1: GO TO (L1, L2) ; L1: GO TO (L1, L2) ; . . . L1: GO TO (L1, L2) ; . . . ".

In using a program as an acceptor we are interested in **terminating computation paths**, that is, finite length computation paths whose last instruction is a STOP (or HALT or FINISH) instruction. The first two examples are terminating computation paths because they are finite and their last instructions are STOP instruction. The third example is not a terminating computation because it is not finite. There is still one more possibility for a non-terminating computation, a finite computation whose last instruction is not a STOP instruction. For example, the program

```
BEGIN
   L0: GO TO (L3)
   L1: GO TO (L1, L2)
   L2: STOP
END
```

has the non-terminating computation

" L0: GO TO (L3) ".

Since there is no instruction with label "L3", the program has nothing to do next. We could consider this last program to be a syntatically valid program which simply has no terminating computations. Of course, we could define our programming language so that every "GO TO" or "JUMP" must have at least one one valid next instruction, and in this new programming language the above program would not be syntatically valid.

If we're using Turing machines as our model of computation, then we can define **deterministic** Turing machines as those machines that have **exactly one** instruction for each state/symbol pair, and define **nondeterministic** Turing machines as those machines that have a **set** of instructions for each state/symbol pair. In particular, we could have a nondeterministic Turing machine which has no instruction for some state/symbol pair, say $q_{17}\, s_4$. In this case, the set of instructions for the pair $q_{17}\, s_4$ would be the null (empty) set. If this machine ever reached state q_{17} and was looking at symbol s_4, the machine would not know what to do next. As with programming languages, we can talk about the terminating computations of Turing machines, and declare that a Turing machine accepts the input x **iff** there is some terminating computation of this Turing machine when started on the input tape x in standard initial configuration. Again there may be non-terminating computations which simply continue and are therefore infinite, and there may be non-terminating computations which simply run out of instructions to follow.

In summary, a nondeterministic program (or machine) accepts the set Set exactly when the program has a terminating computation on input x **iff** $x \in Set$.

We can symmetrically define a **nondeterministic rejector** as a nondeterministic program (or machine) which rejects an input x **iff** the program has a terminating computation when given the input x, and the program rejects Set

iff it rejects each $x \in Set$.

The syntax for a nondeterministic acceptor or rejector is a simple generalization of the syntax for a deterministic program. This generalization allows a set of next instructions instead of a single next instruction. For a Turing machines, this means that a nondeterministic acceptor or rejector has a set of instructions for each state/symbol pair. So checking for syntatic validity of a nondeterministic acceptor or rejector is as easy as checking for syntatic validity of a deterministic program. In fact, it may be simpler to check for syntatic validity of a nondeterministic Turing acceptors or rejectors since one does not have to check that there is *exactly* one instruction for each state/symbol pair.

1.2 Nondeterministic Recognizers

A recognizer can be formed from an acceptor and a rejector. Such a construction is easy for deterministic machines because exactly one of the acceptor/rejector pair will halt. The same construction works for nondeterministic machines. The only difference is that one has to worry about sets of computations rather than single computations. In spite of this, the nondeterministic acceptor/rejector pair stills forms a recognizer. If $x \in$ Set, the acceptor has at least one halting (and accepting) computation, and the rejector has NO halting (and rejecting) computations. Conversely, if $x \notin$ Set, the rejector has at least one halting (and rejecting) computation, and the acceptor has NO halting (and accepting) computations.

So, a nondeterministic recognizer recognizes a set, Set, if and only if, for all $x \in$ Set, the recognizer has at least one halting (and accepting) computation, and has NO halting (and rejecting) computations, and for all $x \notin$ Set, the recognizer has at least one halting (and rejecting) computation, and has NO halting (and accepting) computations.

The big difficulty is in verifying these conditions. To start with, even for a deterministic Turing machine it is unsolvable to determine if the machine halts on a specific input. It certainly looks more difficult to decide if the machine is a recognizer because then we would have to verify that the machine halted on all inputs. For a deterministic acceptor/rejector pair it looks very difficult to determine whether for each input exactly one of the two machines halts. Adding nondeterminism cannot make these problems easier, and it may be more difficult to determine when a nondeterministic machine is a recognizer, or whether a nondeterministic acceptor/rejector pair do consistently recognize a set.

1.3 Nondeterministic Functions

Since a nondeterministic program (or machine) has, in general, many computation paths, how can we say that such a program computes a function which is required to have exactly one output value for each input value?

A simple slimy "out" is to say that nondeterministic programs compute *multi-functions*. For each input value a **multi-function** outputs a set of values rather than the single value output by a function. In other words a multi-function can output zero or more output values for each input value. A function is a multi-function which outputs exactly one output value for each input value.

A nondeterministic program has a set of terminating computational paths, and the output values of the corresponding multi-function are the values output by these terminating computational paths. The slimy part of this definition is that when we find some output values for the program, we don't know whether or not there are other values which could be output following other computational paths.

It seems that we could "fix" a multi-function by saying that instead of outputting values in some set R, the multi-function outputs a subset of R, and thus, the multi-function becomes a function which has outputs in $\mathcal{P}(R)$ the power set (set of all subsets) of R.

If we want nondeterministic programs to compute functions, we have to add some stipulations about the values output by the various computation paths. A **nondeterministic program computes a function** when for each input there is at least one terminating computation path and every terminating computation path outputs the same value. As we'll see, it may be difficult to verify that a nondeterministic program computes a function. On the other hand, it is also difficult to show that a deterministic program computes a function, because even for a deterministic program it is difficult to show that the program always terminates.

1.4 Simple Nondeterministic Programs (Dijkstra)

The following examples are from the recently deceased computer scientist Edsger Wybe Dijkstra.

Consider a bag which contains black and white balls. If you stick your hand in the bag you can feel the balls but you can't determine their colors until you remove them from the bag. Picking balls will be the nondeterministic step in the following programs. We could view the choice of which balls you pick to be determined by a deamon or a genie. The genie might want to help you get a result, but the deamon might want to prevent you from getting a answer. We are **not** considering the choices to be probabilistic. Which color balls you pick does not depend on the relative frequency of black and white balls in the bag.

First we'll say what to do if there are zero or one balls in the bag, i.e. $n = 0$ or $n = 1$. For n small, we use the following **OUTPUT** program. This program should be considered to be a part of each of the subsequent programs.

DIJKSTRA OUTPUT

```
IF   $n = 0$   THEN Return( 0 )
IF   $n = 1$   THEN
                 IF  the ball is Black    THEN  Return( 0 )
                 IF  the ball is White    THEN  Return( 1 )
```

Next, we have our first program which deals with $n \geq 2$.

DIJKSTRA ONE

```
WHILE   there are at least 2 balls in the bag
                nondeterministically take 2 balls from the bag
      IF   they are the same color
              THEN discard both
              ELSE put the White ball back in the bag
                     and discard the Black ball
ENDWHILE
```

To understand this program, we use the two ideas of **termination** and **invariant**. For termination, we'd like to find a natural number variable which is always decreasing. For this program, n, the number of balls in the bag, is an appropriate decreasing natural number variable. If we have n balls in the bag, then on one iteration of the WHILE loop, the number of balls will decrease to either $n - 1$ or $n - 2$. So, if we start with a finite number of balls, n, eventually there will be either 1 or 0 balls in the bag, and the WHILE loop will terminate, and the output routine will output either 0 or 1. At this stage, we have shown that the program returns a value, but we don't yet know if it computes a function – with some choices of balls the program may return 1, but with other choices the program might return 0.

Next, we want an **invariant**, a statement which is true before an iteration of the WHILE loop and remains true after the iteration. Clearly,

"The number of balls $=$ n "

is not an invariant. While, it is true before the iteration, it is clearly false after the iteration. How about

"The number of black balls $\geq$ the number of white balls "?

This statement may or may not be true before the iteration, and even if it is true before the iteration, it may be false after the iteration. A true but useless invariant is

“Black number $=0 \mod 2$ or Black number $=1 \mod 2$ ”.

Certainly one of these two disjuncts is true before the iteration and one of these two disjuncts is true after the iteration, although the same disjunct may not be true both before and after the iteration.

Let PAR(BAG) be the number of WHITE balls taken mod 2, that is, the *parity* of the number of WHITE balls. A useful invariant is

“PAR(BAG) $=$ PAR(INITIAL BAG)”.

Certainly before we start, the parity is just the parity of the initial bag. If we let a = the number of WHITE balls and b = the number of BLACK balls then on an iteration of the loop one of several things can happen

(a) if we choose two black balls, then $a \to a$ and $b \to b-2$

(b) if we choose two white balls, then $a \to a-2$ and $b \to b$

(c) if we choose one white ball and one black ball, then $a \to a$ and $b \to b-1$.

In all three cases, a changes by 0 or 2, and so the parity of a does not change. With this invariant, we can state that if this program terminates, then it outputs the parity of the white balls in the bag.

Putting both termination and invariant together, we can state that this nondeterministic program computes the parity function on the number of white balls.

Now, let’s consider a variant of the above program.

DIJKSTRA TWO

```
WHILE   there are at least 2 balls in the bag
              nondeterministically take 2 balls from the bag
        IF   they are the same color
             THEN discard both
             ELSE put both balls back in the bag
                       and add a Black ball
ENDWHILE
```

In this new program the invariant

“PAR(BAG) $=$ PAR(INITIAL BAG)”.

still holds, but the termination variable is no longer decreasing because the ELSE branch allows n to increase to $n + 1$. Luckily, **DIJKSTRA TWO** still computes a function because all *terminating* computations output the same value. (This value is still the parity of the number of white balls.) Of course, there are now many computation paths which do not lead to termination.

A third variant is even stranger.

DIJKSTRA THREE

```
WHILE    there are at least 2 balls in the bag
                nondeterministically take 2 balls from the bag
      IF    they are the same color
              THEN put both balls back in the bag
              ELSE put both balls back in the bag
                     and add a Black ball
ENDWHILE
```

Here, the invariant still holds, but the number of balls is never decreasing. When $n = 0$ or $n = 1$, this program still outputs the parity of the number of white balls, but for $n \geq 2$, there are **no** terminating computations. I doubt that we would want to say that this program computes the parity of the number of white balls. This points out that in our definition of a nondeterministic program computing a function, the stipulation that

"for each input there is at least one terminating computation"

is essential.

We conclude that **DIJKSTRA THREE** does not compute a function.

A fourth variant makes the point that non-deterministic time is *NOT* the same as probabilistic time.

DIJKSTRA FOUR

```
WHILE   there are at least 2 balls in the bag
              nondeterministically take 2 balls from the bag
      IF   they are both White
              THEN don't put them back in the bag
      IF   they are both Black
              THEN put 4 Black's into the bag
      IF   they are one White and one Black
              THEN put only the White back into the bag
ENDWHILE
```

We claim that this program still computes the parity, but only if there is at least one white initially in the bag.

It is easy to check that the invariant

"PAR(BAG) = PAR(INITIAL BAG)"

holds, so every terminating computation computes PAR(white). Next, we see that there are terminating computations, and so **DIJKSTRA FOUR** computes a function. Of course, this is a nondeterministic computation.

A specific sequence of choices is:

(a) while there are black balls pick a black and a white and eliminate a black,

(b) always pick pairs of whites and eliminate them 2 at a time,

(c) output the final number (0 or 1) of white balls.

How long does this take?
Since blacks are eliminated one at a time, and whites are eliminated two at a time, the above sequence will have about $|B| + |W|/2$ steps, and we can reasonably say that this is a *polynomial time nondeterministic* computation. As in previous examples, in addition to this terminating computation, there will be nonterminating computations (at least, for every input containing at least 2 blacks).

Now, let's consider this as a *probabilistic* computation. Clearly, if there are at least two black balls, then one could pick two black balls and increase the number of black balls, and also the probability of picking two black balls at the next choice. So, one could get a sequence of choices which always chose black balls, which of course would not terminate. Since, there is a non-zero probability of nontermination, then if we count nontermination as infinite time, the expected time for this probabilistic computation (in which pairs of balls are chosen at random) will be infinite. Even if we try to average time only over terminating computations, the expected run-time of this probabilistic algorithm will certainly be much, much greater than the linear time for the nondeterministic computation.

1.5 A Generic Program for $\mathcal{NP}$ Problems

As we've seen, nondeterministic programs should give the same output regardless of which nondeterministic choices are made, but the running time may depend on the choices.

$\mathcal{NP}$, nondeterministic polynomial time programs seem to give us some difficulties because the programs are able to "guess" **YES** or **NO**. The **YES** computations are less problematic because there are a series of guesses which will allow a quick computation of **YES**. The **NO** computations cause some problems because, if the program guesses **NO** and a **YES** answer was possible, then the program would not compute a function, i.e. the program would have multiple output values for a single input value.

We can use the following generic $\mathcal{NP}$ nondeterministic program. In this program, S(I) is the set of possible "proofs" for input I, and VERIFY(I, s) is a deterministic routine which outputs **TRUE** if s is actually a proof of I, and outputs **FALSE** if s does not prove I. For example, in the Hamiltonian path problem, I, an input, would be a graph, and S(I) would be the set of all permutations of the vertices of the graph I. In this example, VERIFY(I, s) outputs **TRUE** if the permutation s corresponds to a Hamiltonian path of I, i.e. if for each j there is an edge of I between s_j and s_{j+1}, where s_j is the j^{th} vertex in the permutation. (Since s is a permutation each vertex appears exactly once.) Notice that VERIFY works quickly. Even a very bad implementation of VERIFY would run in $O(\,n^3\,)$ where n is the number of vertices in I. So VERIFY is a polynomial time deterministic recognizer which given a graph and a permutation of the graph's vertices, accepts those pairs for which the permutation is a Hamiltonian path, and rejects those pairs for which the permutation is not a Hamiltonian path.

$\mathcal{NP}$ PROGRAM

```
FUNCTION YN( I )
          {Let S(I) be the set of possible "proofs" of I }

     WHILE   there are unused elements of S(I) DO
          Pick s, an unused element of S(I)
          IF   VERIFY( I, s )
               THEN RETURN ( YES )
     ENDWHILE
     RETURN ( NO )
```

There are two ways to get out of the WHILE loop. One is the RETURN jump which terminates this function routine. This exit can only be taken when a verifiable proof is discovered. The second exit is the "normal" exit, i.e. when the WHILE loop's condition is false. But the only way to reach this exit is when **ALL** of S(I) has been tested. In our Hamiltonian path example, S(I) is all permutations of n vertices. Since there are $n!$ such permutations, in this example, the program must take at least $\Omega(n!)$ steps to get to **NO**.

1.6 $\mathcal{NP}$ and Languages

The crucial and essential idea:

Each $\mathcal{NP}$-Complete problem defines a programming language.

These languages are only sufficient to represent problems which are in $\mathcal{NP}$. Rather than being procedural, these are declarative languages. That is, instead of telling what to do next at each step, these languages only state (or declare) what problem is to be solved.

1.7 Cook's Theorem

Here we want to give (at least in outline) a proof of Cook's theorem.

Theorem 1. *(Cook)* **SAT** *is $\mathcal{NP}$-Complete.*

The main idea is to use the idea of decision tree, and to argue that the only nondeterministic part is deciding which branch to take in the tree.
So the *guessing* gives a binary string which determines which of the branches to

take. (Binary is used for simplicity. We are assuming that all branches are two-way branches. Clearly higher branching factors can be represented by repeated binary branches.)

So I want to define a nondeterministic program as a program together with these nondeterministic two-way branches, and then (at least in principle) show that such a program leading to an **ACCEPT** can be represented by a clause form BOOLEAN formula. **THE PROBLEM HERE IS TO SHOW THAT ALL COFIGS DO NOT HAVE TO BE LOOKED AT**

1.8 Comparing Deterministic and Nondeterministic Classes

Is nondeterminism powerful? This is one of the central questions of computer science. We know the answer in some contexts, but in other contexs answers have been elusive. Here, we want to investigate some classes in which nondeterminism gives us nothing extra. That is, we'll look at some deterministic classes, which we'll call $\mathcal{C}$, and the corresponding nondeterministic classes, which we'll call $\mathcal{NC}$, and show that $\mathcal{C} = \mathcal{NC}$.

1.8.1 Savitch's Theorem

Savitch's Theorem gives the very powerful result that nondeterminism can be simulated in only slightly more space. Specifically,

$$\mathcal{NSPACE}(S) \subseteq \mathcal{SPACE}(S^2).$$

This result can be applied immediately to certain classes. For example, if **PRIM** is the class of sets that can be recognized by bounded deterministic Turing machines, then

$$\mathcal{N}\mathbf{PRIM} = \mathbf{PRIM}$$

because if the space for the nondeterministic computation is bounded by B, then the space for a deterministic simulation of this nondeterministic computation is bounded by $\mathcal{O}(B^2)$ and so the problem has a bounded deterministic Turing machines recognizer with just a bigger bound.

Similarly, let $\mathcal{PSPACE}$ be the class of sets which can be recognized by deterministic Turing machines which use $\mathcal{O}(n^K)$ space,
(Here, of course, n is the size of the input and K will be fixed for each set but will differ for different sets.)
and let $\mathcal{N} - \mathcal{PSPACE}$ be the corresponding nondeterministic class, that is, the class of sets which can be recognized by nondeterministic Turing machines which use $\mathcal{O}(n^K)$ space, then by Savitch's Theorem this computation can be simulated by a deterministic Turing machine using only $\mathcal{O}(n^{2K})$ space, and thus

$$\mathcal{N} - \mathcal{PSPACE} = \mathcal{PSPACE}$$

because Savitch gives the inclusion of $\mathcal{N} - \mathcal{PSPACE}$ within $\mathcal{PSPACE}$ and clearly $\mathcal{N} - \mathcal{PSPACE}$ includes $\mathcal{PSPACE}$.

The proof of Savitch's Theorem uses the following recursive program. The idea is to decide if the nondeterministic program has a computation path from the initial configuration Q_0 to the accepting configuration Q_f.
(For simplicity, we'll assume that there is only one accepting configuration. See Exercise 1.2.)

Savitch's Simulation of Nondeterminism

```
PROCEDURE  Reach( $Q_0$, $Q_f$, $2^k$ )
          IF $k = 0$
          THEN
                    IF $Q_0 = Q_f$
                           or $Q_f$ is an immediate successor of $Q_0$
                           THEN Return (TRUE)
                           ELSE Return (False)
          ELSE FOR  All $Q_i$
                    IF Reach( $Q_0$, $Q_i$, $2^{k-1}$ )
                           and Reach($Q_i$, $Q_f$, $2^{k-1}$ )
                           THEN Return (TRUE)
                           ELSE Continue
                ENDFOR
                Return (FALSE)
```

Instead of actually finding the path (because it might require too much space to remember the path), the program recursively decides if a path exists by breaking the supposed path in half. That is, to go from Q_0 to Q_f in 2^k steps means that there was some intermediate configuration Q_i, so that the program can go from Q_0 to Q_i in 2^{k-1} steps and then the program can go from Q_i to Q_f in 2^{k-1} steps.

To complete the proof, we need to find an upper bound, say K_0 so that if there is a path between any two configurations, then there is a path of length at most 2^{K_0}, and we need calculate the amount of space used by Savitch's Simulation program.

For space usage, notice that the simulation program has only one call of size 2^{k-1} going at a time. The second size 2^{k-1} call will only be started after the first size 2^{k-1} call is completed. So, on the recursion stack, the simulation program can only have a sequence of smaller and smalled sized calls. Specifically, it can have only one call of each of the sizes $2^k,\ 2^{k-1},\ \ldots,\ 2,\ 1$. Hence, there are at most k calls on the stack at any one time. So the space used on the stack will be at most k times the space needed to represent a configuration of the program being simulated. But, each configuration can be represented in $\mathcal{O}(\,S(n)\,)$ space and so Savitch's Simulation program uses at most $K_0 \;\cdot\; \mathcal{O}(\,S(n)\,)$ space.

To find K_0 we want an upper bound on the number of configurations. If there are at most C configurations, then if there is a path between two configurations, there will be a path of length $\leq\; C$. Since we're looking at paths of length 2^k, we can take K_0 to be $\log C$. Specifically for a Turing machine, a configuration consists of *tape contents*, *state*, and *positions of read/write heads*. We can upper bound the number of configurations by

$$|\Sigma|^{S(n)} \;\cdot\; \#\text{ of states } \;\cdot\; h\,S(n)$$

where h is the number of heads. We don't need to take K_0 any bigger than the log of this number. So we can take $K_0 \;=\; \mathcal{O}(\,S(n)\,)$, and conclude that Savitch's Simulation program uses at most

$$\mathcal{O}(\,S(n)^2\,) \quad \text{space.}$$

There is one minor exception, the result may not hold if we have too little space. A configuration has to include the position of the read head on the input tape, and this requires at least $\mathcal{O}(\,\log\, n\,)$. So we have only proved Savitch's Theorem if $S(n)$ is at least $\Omega(\,\log\, n\,)$.

We have stated Savitch's result in terms of recognizers, but we can see that it also applies to acceptors. If a deterministic space bounded acceptor uses $S(n)$ space, then there is a recognizer which simply behaves like the acceptor but *rejects* if the acceptor tries to use too much space. This recognizer will use only $\mathcal{O}(\,S(\,n\,))$ space. Savitch's result tells us that a nondeterministic acceptor's outcome can be computed by a deterministic program. Notice that the Savitch simulation program always outputs **TRUE** or **FALSE**. So, for space bounded computation, acceptors (or rejectors) and recognizers are essentially equivalent, but nondeterminism may convert $S(\,n\,)$ space to $\sqrt{S(\,n\,)}$ space.

1.8.2 DCFL $\neq$ CFL

Here, we want to show that nondeterminism is actually helpful. As you probably remember **CFL** is the class of Context Free Languages. These are the languages which are often used to describe the syntax of programming languages. The acceptor which corresponds to this langauge class is the *nondeterministic* pushdown automaton. These acceptors have a **limited** form of indefinite memory. This indefinite memory is a *stack*. That is, the acceptor can

(a) put an item into memory, but only on the top of the *stack*

(b) remove an item from memory, but only from the top of the *stack*.

The following language, the union of two sets, is clearly in **CFL** :

$$\{a^n b^n \mid n \geq 0\} \;\cup\; \{a^n b^{2n} \mid n \geq 0\}$$

The *nondeterministic* acceptor for this language puts a's on the stack while it is reading a's, then when it starts reading b's, the acceptor makes one *nondeterministic* guess: to either pop single a's or pairs of a's. The acceptor then continues popping until the end of the input. (Of course, if an a or other symbol appears, the acceptor can detect this, and reject the input.) If the stack is empty at the end of the input, the acceptor says **YES**.

We claim that *no* deterministic pushdown machine can accept this language. Consider the deterministic computation:

$$q_0, \emptyset \xrightarrow{a^n} \hat{q}, Z_n \xrightarrow{b^n} q_{YES}, \emptyset \xrightarrow{b^n} q_{YES}, \emptyset$$

where the q's represent states, and $\emptyset$ and Z_n indicate contents of the stack. Here, the first two arrows indicate that $a^n b^n$ will be accepted, and the third arrow indicates that $a^n b^{2n}$ will also be accepted. Now consider what this *deterministic* stack machine does on input $a^n b^{3n}$,

$$q_0, \emptyset \xrightarrow{a^n} \hat{q}, Z_n \xrightarrow{b^n} q_{YES}, \emptyset \xrightarrow{b^n} q_{YES}, \emptyset \xrightarrow{b^n} q_{YES}, \emptyset$$

since it must first behave in the same way as when it looks at $a^n b^{2n}$, and then it looks at b^n and as the fourth arrow indicates, the acceptor must also accept $a^n b^{3n}$. So, the *deterministic* acceptor accepts a string not in the language. Thus **any** *deterministic* acceptor must fail to accept something in the language or accept something not in the language. We conclude the **no** *deterministic* acceptor can accept exactly this language.

This example shows that *nondeterminism* does give us some extra computational power.

1.8.3 Regular = N - Regular

For Regular languages, nondeterminism gives us nothing extra, that is, a language recognized by a nondeterministic finite state machine is also recognized by a deterministic finite state machine. But, as we'll see, the deterministic machine may be much larger, i.e. have many more states.

If Q is the state set of a nondeterministic finite state machine, let 2^Q be the set of all subsets of Q. Notice that 2^Q contains the null set as well as the unit sets which contain one state of Q and, many other sets which contain several states from Q.

Now, if a nondeterministic finite state machine **M** is in state q and it receives the input symbol a then **M** may go to any one of several states, say $q_1, q_1, \cdots, q_r$. For the deterministic machine **DM**, we declare that if **DM** is in state $\{ q \}$ and it receives the input symbol a then **DM** goes to $\{ q_1, q_1, \cdots, q_r \}$. What we are doing, is making 2^Q the state set of **DM**. In general, if **DM** is in a "state" consisting of several states of **M**, then on receiving a, **DM** transitions to a "state" which is the union of the sets of states **M** could transition to, and this union is taken over **all** states of **M** which are in the "state" of **DM**.

To finish the description of **DM**, we need to specify its initial "state" and its set of accepting (recognizing) "states". The initial state of **DM** is $\{ q_0 \}$ where

q_0 is the initial state of **M**. The accepting states of **DM** are each "state" which contains at least one accepting state of **M**.

Now, it's easy to show that **DM** accepts x **iff** **M** accepts x, (see Exercise 1.9.) and we have:

Theorem 2. **N - Regular** $=$ **Regular**.

How big is **DM**? From our construction, **DM** has $2^{|Q|}$ states if **M** has $|Q|$ states. So, we have an exponential blow-up of the state set. Actually, for the construction to work, we only need the "states" reachable from $\{ q_0 \}$. (see Exercise 1.10.) Still, we want to show that for *some* machines the exponential blow-up is unavoidable. In the following theorem, we're talking about an infinite sequence of machines, **M**'s, so that the corresponding sequence of **DM**'s has a sequence of state sets, and the sizes of **DM**'s state sets must grow much more rapidly than the sizes of **M**'s state sets.

Theorem 3. *For some nondeterministic machines* **M***, any deterministic machine which recognizes the same set as* **M***, must have exponentially more states than* **M**.

Proof. Let **DM** be the deterministic machine corresponding to **M**, where **M** recognizes the binary strings which have a 1 in the K^{th} from the end of the string. Let q_0 be the start state of **DM**. Assume that there are two strings x and z of length $K - 1$ which take q_0 to the same state q. Since x and z are distinct, assume that x has a 1 in some position in which z has a 0. There is a string w, so that xw is in the set, and zw is not in the set. But both xw and zw must take **DM** to the same state, and we have a contradiction. Hence, **DM** must have distinct states for each binary string of length $K - 1$. Since there are 2^{K-1} such sequences, **DM** must have at least 2^{K-1} distinct states. To complete this proof, one has to show that there is a nondeterministic machine with $O(K)$ states which accepts this set.(see Exercise 1.11.) □

1.9 Exercises

Ex 1.1. Let $\mathcal{ESPACE}$ be the class of sets which can be recognized by deterministic Turing machines which use $\mathcal{O}(c^n)$ space.
(Here, of course, n is the size of the input, and c will be fixed for each set but will differ for different sets.)
Let $\mathcal{N} - \mathcal{ESPACE}$ be the corresponding nondeterministic class, that is, the class of sets which can be recognized by nondeterministic Turing machines which use $\mathcal{O}(c^n)$ space,
Use Savitch's Theorem to show that

$$\mathcal{N} - \mathcal{ESPACE} \ = \ \mathcal{ESPACE}$$

Ex 1.2. Show that Savitch's Theorem holds even if there are several accepting configurations.
HINT: Modify the program to handle each of the accepting configurations.

Ex 1.3. If a nondeterministic program is $\mathcal{O}(S(n))$ space bounded, how many configurations can the program have?

Ex 1.4. If a nondeterministic program is $\mathcal{O}(S(n))$ space bounded, what is the program's nondeterministic running time?

Ex 1.5. If a nondeterministic program is $\mathcal{O}(S(n))$ space bounded, what is the deterministic running time of **Savitch's Simulation**?

Ex 1.6. Consider computing 36 divided by 12.

(a) Compute this by using long division.

(b) Compute this by factoring 36 and factoring 12 and canceling common factors.

(c) Explain how this pair of methods gives you a nondeterministic program for computing a function.

Ex 1.7. Consider computing $\int x^3\, dx$.

(a) Compute this integral using the rule for integrating powers.

(b) Compute this integral by using the substitution $u = x^2$.

(c) Try to compute this integral by using the substitution $x = \cos u$.

(d) Explain how these methods give you a nondeterministic program for computing a function.

Ex 1.8. Use the idea of *fork* to show that:

(a) **N - RE** $=$ **RE**

(b) **N - coRE** $=$ **coRE**

(c) **N - Recursive** $=$ **Recursive**.

Ex 1.9. Complete the proof that **N - Regular** $=$ **Regular** by showing that:

(a) if **DM** accepts x then **DM** accepts x, and

(b) if **M** accepts x then **DM** accepts x.

(c) Show that the word "accepts" in the previous 2 statements can be replaced by the word "recognizes".

Ex 1.10. From **M** given below, construct **DM** with only the states reachable from $\{ q_0 \}$. How many states does this **DM** have? Describe in words the set recognized by **M** (and **DM**) if q_3 is the only recognizing state.

M	0	1
q_0	$q_1,\ q_2$	q_1
q_1	q_1	q_3
q_2	q_2	q_3
q_3	q_3	q_3

Ex 1.11. Show that the set of all binary strings with a 1 in the k^{th} position from the end can be recognized by a nondeterministic machine with $O(\,K\,)$ states.

(a) For $K = 2$, give both **M** and **DM** and compare the number of states in these two machines.

(b) For $K = 3$, give both **M** and **DM** and compare the number of states in these two machines.