Abrahim Ladha

Cook-Levin and Ladner's Theorem. 18

Whats the point of intractability, and NP-completeness. see the Garey-Johnson cartoon. Sometimes you are given some task and cannot find an efficient algorithm. Sometimes, you then cannot prove the problem is intractable or unsolvable. But sometimes, you can prove the problem was NP-complete. That elevates it to a special club, a class of problems all as hard as each other. A fast algorithm for one would imply a fast algorithm for all, and that $P = NP$. There are thousands of such problems across many domains. ~~Recall a language B~~ We say for two languages $A, B \subseteq \Sigma^*$ that A is polytime reducible to B (from A to B $A \leq_p B$) if $\exists$ a function f which is computable (halts on all inputs) in polytime with

$$w \in A \iff f(w) \in B.$$

If $A \leq_p B$, A lower bounds B, B upper bounds A. It is analogous to many-one reductions $\leq_m$, with this bounded restriction that f not only be computable, but be computable in polytime. $\leq_m$ can be used to prove problems are as solvable or unsolvable as each other. $\leq_p$ can be used to prove problems are as easy or hard as each other.

We say B is NP-complete if $B \in NP$ and $\forall L \in NP$ that $L \leq_p B$. You may also prove a language to be NP-complete much easier by doing transitivity. Choose a known NP-complete problem A, and prove $B \in NP$ and $A \leq_p B$.

Cook-Levin proved for us that SAT is NP-complete

$$\forall L \in NP \quad L \leq_p SAT.$$

By the web of reductions, we have thousands of other NP-complete problems.

$$\text{All of NP} \to \text{SAT} \to \text{3SAT} \to \text{Circuit SAT} \quad \text{etc}$$

$$\text{3SAT} \to \text{INDSET} \to \text{CLIQUE}$$

$$\text{INDSET} \to \text{VERTEXCOVER}$$

and so on. These reductions only work if there is a known NP complete problem, which we are going to prove. A reduction is just from one language to one language. The Cook-Levin theorem proves for <u>every</u> language in NP, there is a reduction to SAT. The way we will do it is generic for any language in NP.

- a variable is one of $x_1 \dots x_\ell$
- a literal is one of $x_1, \dots x_\ell, \neg x_1 \dots \neg x_\ell$
- a clause is ~~and the~~ OR of literals $(x_1 \vee \neg x_2 \vee x_3)$
- a CNF formula is an AND of clauses $(x_1 \vee x_2) \wedge (\neg x_3 \vee x_1)$ etc
- an assignment is a selection $x_1 \dots x_\ell \in \{0,1\}$. An assignment is satisfying if when you plug in $x_1 \dots x_\ell$ $\phi = 1$.

$$\text{SAT} = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable CNF} \}.$$

CNFs are surprisingly expressive. Usually real world constraint problems you have a set of constraints and must satisfy all of them, but there is more than one way to satisfy each.

$(x_1 \vee \neg y_1) \wedge (\neg x_1 \vee y_1) \wedge \dots \wedge (x_n \vee \neg y_n) \wedge (\neg x_n \vee y_n)$ is true if and only if $x_1 = y_1, \dots x_n = y_n$. $x = x_1 \dots x_n$, $y = y_1 \dots y_n \Rightarrow x = y$. This formula is satisfiable if and only if $x = y$. String equality..

We prove $\forall L \in NP$ that $L \leq_p \text{SAT}$. Obviously $\text{SAT} \in NP$. poly time verifier $V(\phi, c)$ checks if c is a satisfying assignment to ϕ.

let $L \in NP$. Then there exists a nondeterministic poly time machine N such that N accepts $w \iff w \in L$ consider the computation history, an accepting one of N on w. We use this to construct a CNF formula ϕ such that $w \in L \iff \phi \in SAT$. ϕ will be satisfiable if and only if N accepted w. the idea is conceptually simple but has a ton of hard little details. Since $L \in NP$, N uses at most poly time, say n^t, and also poly space, say n^s. Take each configuration $C_0, C_1, \ldots$ and line them up in a table, like so

#	q_0	1	1	1	␣	#
#	0	q_0	1	1	␣	#
#	0	0	q_0	1	␣	#
#	0	0	0	q_0	␣	#
#	0	0	0	␣	q_a	#

n^t

$n^s + 2$ish

note the tableau has dimension time × space $= n^t \times n^s$. It is polynomial sized. we will create a SAT CNF formula ϕ to loop over the table and check its correctness essentially.

$$\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

$\phi_{cell} = 1 \iff$ exactly one symbol is in each cell of the table

$\phi_{start} = 1 \iff$ first row is the initial configuration

$\phi_{move} = 1 \iff$ the $i+1$th row is the C_{i+1}th configuration, following the ith row

$\phi_{accept} = 1 \iff$ there is an accepting configuration in the table..

Let ~~X~~ X_{ijs} be all the variables with $1 \le i \le n^t$, $1 \le j \le n^s$, $s \in Q \cup \Gamma \cup \{\#\}$ where $X_{ijs} = 1 \Leftrightarrow$ cell $[i,j] = s$.
X_{ijs} ~~[illegible]~~ means symbol s is in cell $[i,j]$. Let $C = Q \cup \Gamma \cup \{\#\}$.

$\phi_{cell} = 1 \Leftrightarrow$ one symbol is in each ~~eell~~ cell, exactly one. $\phi_{ija} = 1 \Rightarrow \phi_{ijb} = 0$ etc

$$\phi_{cell} = \bigwedge_{\substack{1 \le i \le n^t \\ 1 \le j \le n^s}} \left[\left(\bigvee_{x \in C} X_{ijs} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \ne t}} \left(\overline{X_{ijs}} \vee \overline{X_{ijt}} \right) \right) \right]$$

double for loop over entire 2D table

guarantees atleast one symbol is in each cell

guarantees ~~at most [illegible]~~ no more than one symbol is in each cell

any satisfying assignment of ϕ_{cell} implies the corresponding table has exactly one symbol in each cell.

$\phi_{start} = 1 \Leftrightarrow$ first row is a start configuration of N on w and space.

$$\phi_{start} = X_{11\#} \wedge X_{12q_0} \wedge X_{13w_1} \wedge \ldots \wedge X_{1,n+2\,w_n} \wedge$$
$$X_{1,n+3,\sqcup} \wedge \ldots \wedge X_{1,n^s-1\,\sqcup} \wedge X_{1,n^s,\#}$$

note any satisfying assignment of $\phi_{start} \Rightarrow$ the corresponding table has the first row as our desired configuration

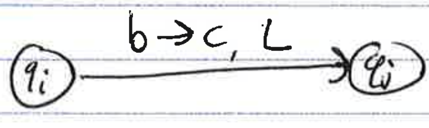
---

$\phi_{accept} = 1 \Leftrightarrow$ the table is accepting.

$$\phi_{accept} = \bigvee_{\substack{1 \le i \le n^t \\ 1 \le j \le n^s}} \phi_{i,j,q_a}$$

loop over entire table to make sure q_a symbol exists somewhere.
$\phi_a = 1$ ~~[illegible]~~, is satisfiable if "q_a" is in the table somewhere.

ϕ_{move} is the hardest one. We want it to enforce that each row follows the proceeding one by only legal moves. With the first initial configuration enforced, we want ϕ_{move} to enforce row 2 is the second Configuration and so on. The way ϕ_{move} will work is check every single 2×3 window of the table and determine if its a legal 2×3 window. For example if we had transitions like



a	q_i	b
q_j	a	c

would be a legal window There are other legal 2×3 windows like

a	b	c
a	b	c

a	a	q_i
a	a	b

#	q_0	1	1	1	␣	#
#	0	q_0	1	1	␣	#
#	0	0	q_0	1	␣	#
#	0	0	0	q_0	␣	#
#	0	0	0	␣	q_a	#

for this table, I have dotted a few windows near the head. If the whole table is legal, the windows near the state are the only ones which dont copy.

$$\phi_{move} = \bigwedge_{\substack{1 \le i < n^t \\ 1 \le j < n^s}} \left[\text{the } (i,j) \text{ window is legal} \right]$$

here again, by legal, we mean according to δ of N. like the PCP proof

$$\phi_{move} = \bigwedge_{\substack{1 \le i \le n^t \\ 1 \le j \le n^s}} \left[\underbrace{\bigvee_{\substack{a_1 \ldots a_6 \\ \text{is legal}}} \left(x_{i,j-1,a_1} \wedge \ldots \wedge x_{i+1,j+1,a_6} \right)}_{\text{checks if window } i,j \text{ is legal}} \right]$$

double for-loop over 2D table, checking all 2×3 windows constructing $\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$.

Note ϕ is satisfiable only if:
1 each cell of the table ~~tbl~~ contains exactly one symbol
2 the first row is a start configuration
3 the $(i+1)$th row is the c_{i+1}th configuration following the ith row as the c_ith config
4 one of the configurations is accepting.

so ϕ is satisfiable only if an accepting computation history of N on w exists, which can only happen if there is a computation branch of N on w
so $w \in L \iff \phi$ SAT

note that for a polynomial sized table, each of the subformulas also took polynomial time to construct so the computation to build ϕ takes polynomial time. We observe $L \leq_p SAT$. Since $SAT \in NP$ and this is $\forall L \in NP$ we conclude SAT is NP-complete.

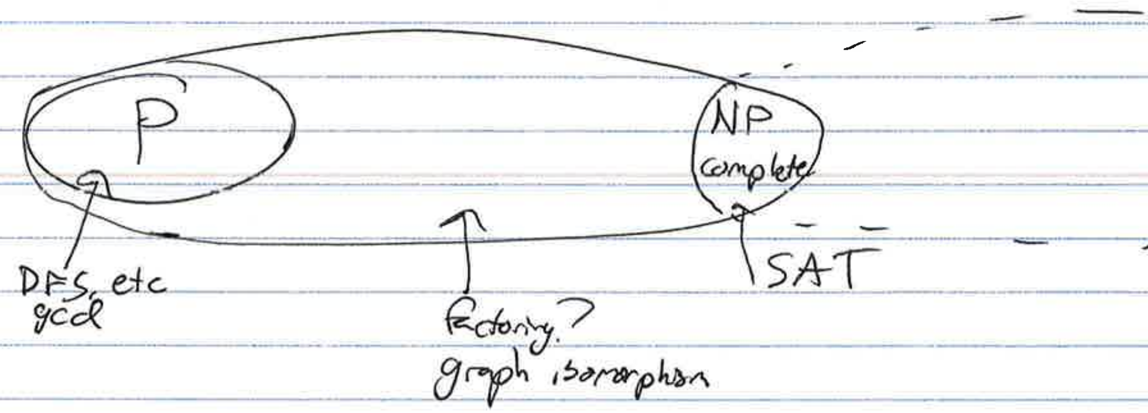
Now that we have proved SAT is NP-complete, we may prove many other languages are NP-complete, not by repeating the proof, but by a simple reduction.

for example if you prove $3SAT \in NP$ and $SAT \leq_p 3SAT$, then since we proved $\forall L \in NP$ that $L \leq_p SAT$, we can use transitivity.
$L \leq_p SAT \leq_p 3SAT \implies L \leq_p 3SAT$.
the reduction reuses and transforms the proof, rather than redoing it.

SAT is not the only language ~~as~~ that could be proved as an NP-complete problem. Sipser and CRLS both include a proof by a similar construction that circuit SAT is NP-complete. Levin originally proved a kind of tiling problem. Cook proved not SAT necessarily, but tautologies are NP-complete.

If $SAT \in P \Rightarrow P = NP$. To prove, recall $\forall L \in NP$ that $L \leq_p SAT$. So if $SAT \in P$, then there is a polytime alg for SAT. Since every $L \in NP$ is polytime reducible to SAT, combining this reduction plus the polytime algorithm for SAT is a polytime alg for L. So $L \in P$. But since L is any language in NP, we see $NP \subseteq P$ since we know $P \subseteq NP$, $\Rightarrow P = NP$.

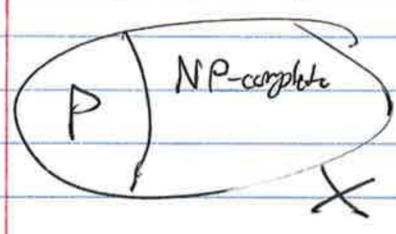
We don't believe SAT has a polytime algorithm. We don't even believe SAT has a quasi-polytime algorithm



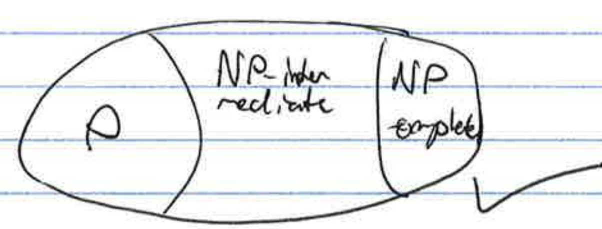
Not all languages in $NP \setminus P$ are NP-complete, we will prove it shortly. Factoring is a candidate for an NP-intermediate problem. It is (believed) not to be in P, but has sub exponential time algorithms. Factoring $\in Time(o((1+\varepsilon)^n))$ $\forall \varepsilon > 0$.

The hardness of SAT can be formalized as an assumption ETH = Exponential Time Hypothesis. Essentially, SAT cannot be solved in subexponential time. It is a stronger assumption than $P \neq NP$ since it implies $P \neq NP$ among many other things.

This proof disproves this



proves this



We prove Ladner's theorem: If $P \neq NP$ then $\exists$ languages that

- are not in P
- are in NP
- are not NP-complete.

essentially, that the NP-intermediate languages exist. If you could prove these exist unconditionally then of course you found a language in $NP \setminus P$ so $P \neq NP$. We will proceed by diagonalization, proving a weaker result. We will assume ETH instead of $P \neq NP$ to get an easier proof of the same ideas.

Assume ETH. There is no subexponential time algorithm for SAT. $\forall \epsilon$, $SAT \notin TIME(2^{\epsilon n})$. Consider the following language we essentially weaken SAT to not be NP-complete, but no so weak that $SAT \in P$. We pad SAT.

$$L = \{ \langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle \mid \phi \in SAT \}$$

Note $L \in NP$. Our witness is the assignment. V on input $\langle w, c \rangle$ checks if w is of the form $\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle$, doing some math to count the padding. Then it checks if c is a satisfying assignment for ϕ.

Note that $L \notin P$. Suppose it was, there exists an algorithm A to decide L in time polynomial in the size of the input. A runs in time $O((n + 2^{\sqrt{n}})^k)$ for some k. We give a subexponential time algorithm A' for SAT

A' on input ϕ
1. build $\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle = y$
2. run $A(y)$

$$A \text{ runs in time } O(2^{\sqrt{n}}) + O((n + 2^{\sqrt{n}})^k) = 2^{O(\sqrt{n})} = 2^{o(n)}$$

violating ETH by assumption, so $L \notin P$.

Now we show L is not NP-complete. The proof idea is that if it was, there is a reduction for it, such a reduction could be used to solve SAT too fast, violation of ETH.

Assume to the contrary L is NP-complete. Then there exists a polynomial time computable reduction such that $SAT \leq_p L$. This reduction function, say f works such that $f: \phi' \to \phi, 1^{2^{\sqrt{|\phi|}}}$

$$\phi' \in SAT \iff \langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle \in L$$

were ϕ, ϕ' may be different. Since our f is polytime, there exists some k such that $|f(\phi')| = n^k$. Any polynomial time algorithm, (here, a reduction) can only produce a polynomial sized output. It takes time to write that output down. Here $|\phi'| = n$, the size of the input. Since the output is $\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle$, ϕ must be small enough such that $2^{\sqrt{|\phi|}} \leq |f(\phi')| = n^k$

so

$$2^{\sqrt{|\phi|}} \leq n^k \Rightarrow \sqrt{|\phi|} \leq k \log n \Rightarrow$$

$$|\phi| \leq (k \log n)^2. \text{ Or that } |\phi| << n, \ o(n).$$

ϕ is much much smaller than n. This gives us a fast algorithm for ϕ'. To see if $\phi' \in SAT$, perform the polytime transform reduction, and try all assignments of ϕ. This will take time

$$2^{(k \log n)^2} = 2^{o(n)} \quad \text{violating ETH.}$$

Therefore we conclude $L \notin P$, $L \in NP$, but L is not NP-complete. So the class NP-intermediate exists

:)

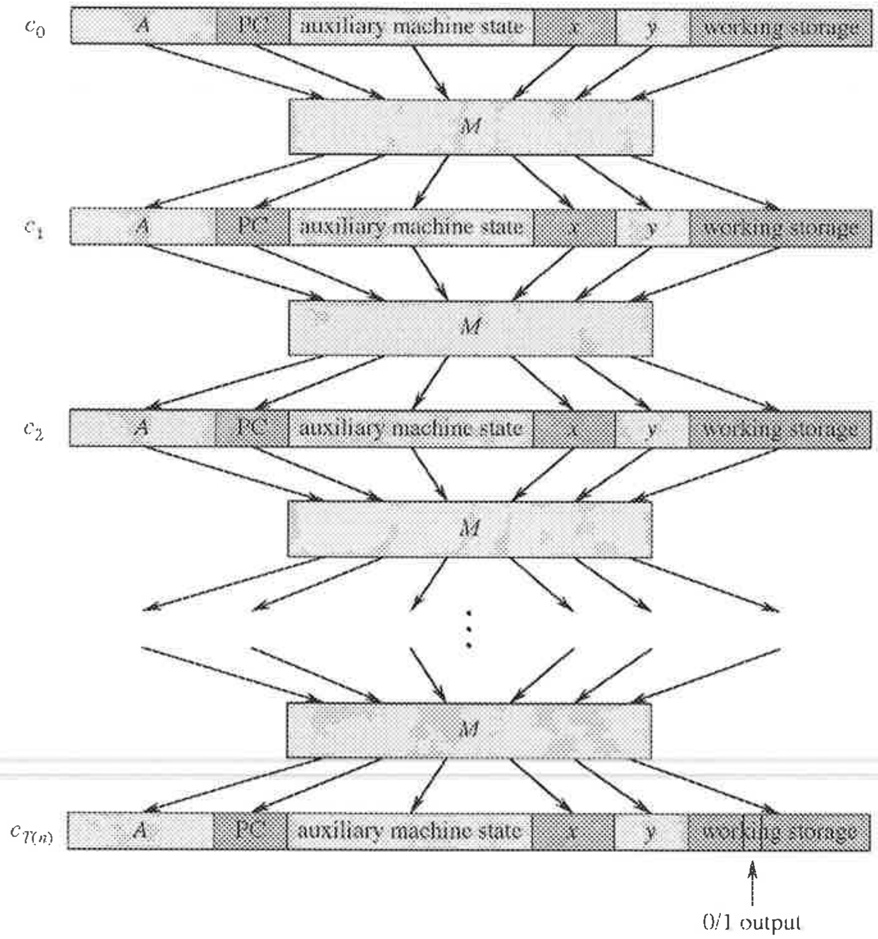
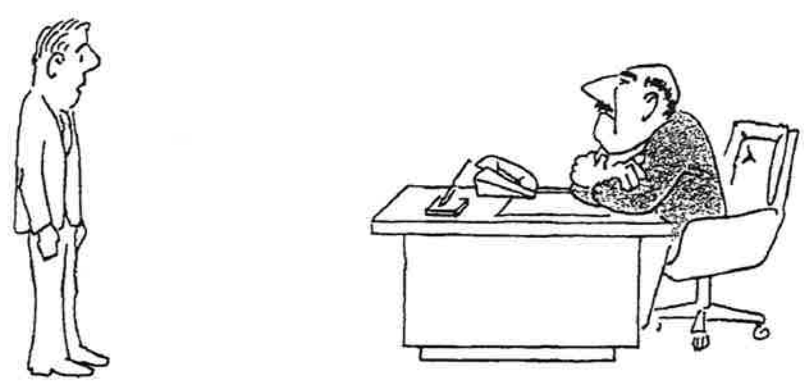


Figure 34.9 The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate y, the initial configuration c_0 is constant. A boolean combinational circuit M maps each configuration to the next configuration. The output is a distinguished bit in the working storage.

certificate is $O(n^k)$. (The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate, but since the length of the certificate is polynomial in the length n of the input string, the running time is polynomial in n.)

The basic idea of the proof is to represent the computation of A as a sequence of configurations. As Figure 34.9 illustrates, consider each configuration as com-

:ifications, and the bandersnatch department is already 13 components ind schedule. You certainly don't want to return to his office and re-t:



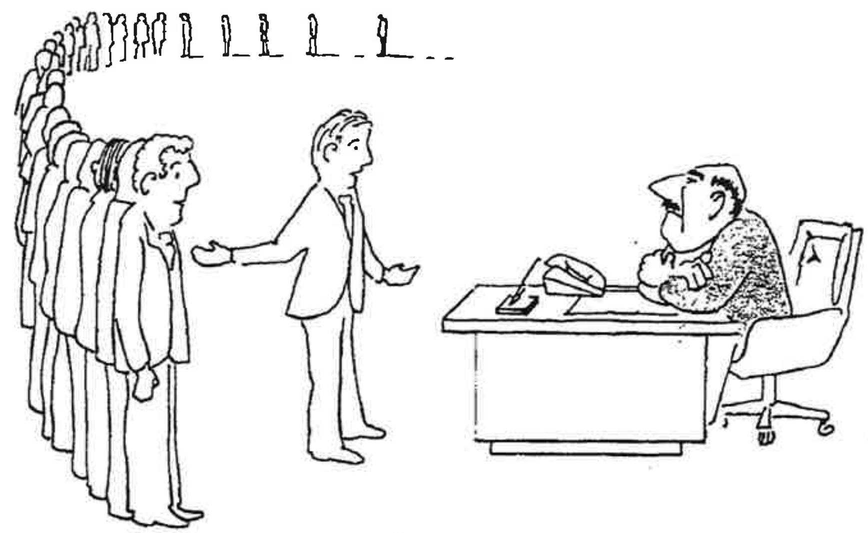
"I can't find an efficient algorithm, I guess I'm just too dumb."

To avoid serious damage to your position within the company, it would much better if you could prove that the bandersnatch problem is *in-:ntly* intractable, that no algorithm could possibly solve it quickly. You ı could stride confidently into the boss's office and proclaim:

"I can't find an efficient algorithm, because no such algorithm is possible!"

Unfortunately, proving inherent intractability can be just as hard as ing efficient algorithms. Even the best theoreticians have been stymied heir attempts to obtain such proofs for commonly encountered hard ›lems. However, having read this book, you have discovered something

almost as good. The theory of NP-completeness provides many straightforward techniques for proving that a given problem is "just as hard" as a large number of other problems that are widely recognized as being difficult and that have been confounding the experts for years. Armed with these techniques, you might be able to prove that the bandersnatch problem is NP-complete and, hence, that it is equivalent to all these other hard problems. Then you could march into your boss's office and announce:



"I can't find an efficient algorithm, but neither can all these famous people."

At the very least, this would inform your boss that it would do no good to fire you and hire another expert on algorithms.

Of course, our own bosses would frown upon our writing this book if its sole purpose was to protect the jobs of algorithm designers. Indeed, discovering that a problem is NP-complete is usually just the beginning of work on that problem. The needs of the bandersnatch department won't disappear overnight simply because their problem is known to be NP-complete. However, the knowledge that it is NP-complete does provide valuable information about what lines of approach have the potential of being most productive. Certainly the search for an efficient, exact algorithm should be accorded low priority. It is now more appropriate to concentrate on other, less ambitious, approaches. For example, you might look for efficient algorithms that solve various special cases of the general problem. You might look for algorithms that, though not guaranteed to run quickly, seem likely to do so most of the time. Or you might even relax the problem somewhat, looking for a fast algorithm that merely finds designs that

We make two assumptions about TM M in defining the notion of a tableau. First, as we mentioned in the proof idea, M accepts only when its head is on the leftmost tape cell and that cell contains the ␣ symbol. Second, once M has halted, it stays in the same configuration for all future time steps. So by looking at the leftmost cell in the final row of the tableau, $cell[t(n), 1]$, we can determine whether M has accepted. The following figure shows part of a tableau for M on the input 0010.

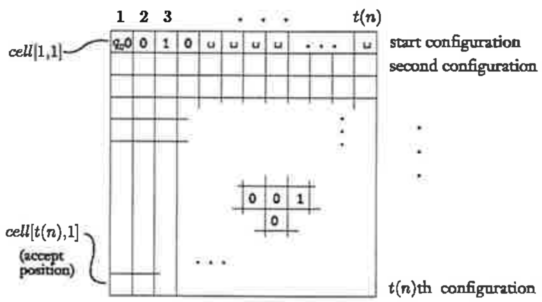
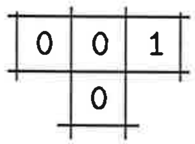


FIGURE 9.31
A tableau for M on input 0010

The content of each cell is determined by certain cells in the preceding row. If we know the values at $cell[i-1, j-1]$, $cell[i-1, j]$, and $cell[i-1, j+1]$, we can obtain the value at $cell[i, j]$ with M's transition function. For example, the following figure magnifies a portion of the tableau in Figure 9.31. The three top symbols, 0, 0, and 1, are tape symbols without states, so the middle symbol must remain a 0 in the next row, as shown.



Now we can begin to construct the circuit C_n. It has several gates for each cell in the tableau. These gates compute the value at a cell from the values of the three cells that affect it.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

To make the construction easier to describe, we add lights that show the output of some of the gates in the circuit. The lights are for illustrative purposes only and don't affect the operation of the circuit.

Let k be the number of elements in $\Gamma \cup (Q \times \Gamma)$. We create k lights for each cell in the tableau—one light for each member of Γ, and one light for each member of $(Q \times \Gamma)$—or a total of $kt^2(n)$ lights. We call these lights $light[i, j, s]$, where $1 \le i, j \le t(n)$ and $s \in \Gamma \cup (Q \times \Gamma)$. The condition of the lights in a cell indicates the contents of that cell. If $light[i, j, s]$ is on, $cell[i, j]$ contains the symbol s. Of course, if the circuit is constructed properly, only one light would be on per cell.

Let's pick one of the lights—say, $light[i, j, s]$ in $cell[i, j]$. This light should be on if that cell contains the symbol s. We consider the three cells that can affect $cell[i, j]$ and determine which of their settings cause $cell[i, j]$ to contain s. This determination can be made by examining the transition function δ.

Suppose that if the cells $cell[i-1, j-1]$, $cell[i-1, j]$, and $cell[i-1, j+1]$ contain a, b, and c, respectively, $cell[i, j]$ contains s, according to δ. We wire the circuit so that if $light[i-1, j-1, a]$, $light[i-1, j, b]$, and $light[i-1, j+1, c]$ are on, then so is $light[i, j, s]$. We do so by connecting the three lights at the $i-1$ level to an AND gate whose output is connected to $light[i, j, s]$.

In general, several different settings $(a_1, b_1, c_1), (a_2, b_2, c_2), \ldots, (a_l, b_l, c_l)$ of $cell[i-1, j-1]$, $cell[i-1, j]$, and $cell[i-1, j+1]$ may cause $cell[i, j]$ to contain s. In this case, we wire the circuit so that for each setting a_i, b_i, c_i, the respective lights are connected with an AND gate, and all the AND gates are connected with an OR gate. This circuitry is illustrated in the following figure.

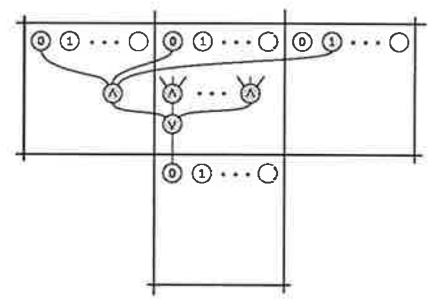


FIGURE 9.32
Circuitry for one light

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.