Introduction to Computational Complexity

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Introduction

- Computational problems, solvable in principle, but with solutions requiring so much time or space that they cannot be used in practice, are called **intractable**.
- Several problems we have seen are thought to be intractable but none have been proven to be intractable. E.g., the SAT problem and all other NP-complete problems.
- We visit examples of problems that can be shown to be intractable.
- We also develop several theorems that relate the power of Turing machines to the amount of time or space available for computation.
- We close with a discussion of the possibility of proving that problems in NP are intractable, thereby solving the P versus NP question.
- In summary, the topics we discuss are:
  - The relativization technique that is used to show that certain methods are unsuitable for resolving the P versus NP.
  - Circuit complexity theory, which is believed to be a promising approach.

Subsection 1

Hierarchy Theorems

Space Constructible Functions

- Intuition suggests that giving a Turing machine more time or more space should increase the class of problems that it can solve.
- The hierarchy theorems prove that this intuition is correct, subject to certain conditions.
- We start with the hierarchy theorem for space complexity.

Definition (Space Constructible Function)

A function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is at least $\mathrm{O}(\log n)$, is called **space constructible** if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in space $\mathrm{O}(f(n))$.

- Note that f is space constructible if some $\mathrm{O}(f(n))$ space TM M exists that always halts with the binary representation of $f(n)$ on its tape when started on input 1^n.
- Fractional functions, e.g., $n \log_2 n$ and $\sqrt{n}$, are rounded down to the next lower integer for the purposes of time and space constructibility.

Remarks on Space Constructibility

- All commonly occurring functions that are at least $O(\log n)$ are space constructible, including the functions $\log_2 n$, $n \log_2 n$ and n^2.
- Example: n^2 is space constructible because a machine may take its input 1^n:
  - Obtain n in binary by counting the number of 1s;
  - Multiply $n \cdot n$ using any standard multiplication method;
  - Output n^2.

  The total space used is $O(n)$ which is $O\left(n^2\right)$.
- When showing functions $f(n)$ that are $o(n)$ to be space constructible, we use a separate read only input tape, as we did with sublinear space complexity.
  Example: Such a machine can compute the function which maps 1^n to the binary representation of $\log_2 n$ as follows:
  - It first counts the number of 1s in its input in binary, using its work tape as it moves its head along the input tape.
  - With n in binary on its work tape, it can compute $\log_2 n$ by counting the number of bits in the binary representation of n.

The Role of Space Constructibility

- If $f(n)$ and $g(n)$ are two space bounds, where $f(n)$ is asymptotically larger than $g(n)$, we would expect a machine to be able to compute more languages in $f(n)$ space than in $g(n)$ space.
- Consider, however, the case in which $f(n)$ exceeds $g(n)$ by only a very small and hard to compute amount.

  Then, the machine may not be able to use the extra space profitably, since even computing the amount of extra space may require more space than is available.

  In such a case, a machine may not be able to compute more languages in $f(n)$ space than it can in $g(n)$ space.
- Imposing space constructibility on $f(n)$ avoids this situation and allows proving that a machine can compute more than it would in any asymptotically smaller bound.

Space Hierarchy Theorem

The Space Hierarchy Theorem

For any space constructible function $f : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $O(f(n))$ space but not in $o(f(n))$ space.

- We must demonstrate a language A that has two properties:
  - A is decidable in $O(f(n))$ space.
  - A is not decidable in $o(f(n))$ space.

  We describe A by giving an algorithm D that decides it:
  - D runs in $O(f(n))$ space, ensuring the first property.
  - Furthermore, D guarantees that A is different from any language that is decidable in $o(f(n))$ space, ensuring the second property.

  In order to ensure that A is not be decidable in $o(f(n))$ space, we use the diagonalization method: If M is a TM that decides a language in $o(f(n))$ space, D guarantees that A differs from M's language in at least one place. The place corresponds to a description of M itself.

Essentials of the Operation of D

- D takes its input to be the description of a TM M.

  D runs M on input $\langle M \rangle$ within the space bound $f(n)$.
  - If M halts within that much space, D accepts iff M rejects.
  - If M does not halt, D just rejects.

  Thus:
  - If M runs within space $f(n)$, D has enough space to ensure that its language is different from M's.
  - If not, D does not have enough space to figure out what M does, but fortunately D has no requirement to act differently from machines that do not run in $o(f(n))$ space. So D's action on this input is inconsequential.

Critical Remarks on the Operation of D

- If M runs in $o(f(n))$ space, D must guarantee that its language is different from M's language.
  But even when M runs in $o(f(n))$ space, it may use more than $f(n)$ space for small n, before assuming the asymptotic behavior. Possibly, D might not have enough space to run M to completion on input $\langle M \rangle$. Hence, D will miss its one opportunity to avoid M's language.
  - We modify D to give it additional opportunities to avoid M's language. Instead of running M only on input $\langle M \rangle$, it runs M also on inputs of the form $\langle M \rangle 10^*$. If M really is running in $o(f(n))$ space, D will have enough space to run it to completion on input $\langle M \rangle 10^k$ for some large value of k because the asymptotic behavior must eventually kick in.
- When D runs M on some string, M may get into an infinite loop while using only a finite amount of space. But D is supposed to be a decider, so we must ensure that D does not loop while simulating M.
  - Any machine that runs in space $o(f(n))$ uses only $2^{o(f(n))}$ time. We modify D so that it counts the number of steps used in simulating M. If this count ever exceeds $2^{f(n)}$, then D rejects.

The Algorithm D

- The following $\mathrm{O}(f(n))$ space algorithm D decides a language A that is not decidable in $\mathrm{o}(f(n))$ space:
  D: On input w:
  1. Let n be the length of w.
  2. Compute $f(n)$ using space constructibility, and mark off this much tape. If later stages ever attempt to use more, reject.
  3. If w is not of the form $\langle M \rangle 10^*$ for some TM M, reject.
  4. Simulate M on w while counting the number of steps used in the simulation. If the count ever exceeds $2^{f(n)}$, reject.
  5. If M accepts, reject. If M rejects, accept.
- In Stage 4 the simulated TM M has an arbitrary tape alphabet and D has a fixed tape alphabet. So each cell of M's tape is represented with several cells on D's tape. Therefore, the simulation introduces a constant factor overhead in the space used, i.e., if M runs in $g(n)$ space, then D uses $dg(n)$ space to simulate M for some constant d that depends on M.

The Language that Algorithm D Decides

- Machine D is a decider because each of its stages can run for a limited time.
- Let A be the language that D decides.
  - A is decided by D in space $O(f(n))$.
  - A is not decidable in $o(f(n))$ space: Assume to the contrary that some Turing machine M decides A in space $g(n)$, where $g(n)$ is $o(f(n))$. D can simulate M, using space $dg(n)$, for some constant d. Because $g(n)$ is $o(f(n))$, some constant n_0 exists, such that $dg(n) < f(n)$, for all $n \geq n_0$. Therefore, D's simulation of M will run to completion so long as the input has length n_0 or more.
    When D is run on input $\langle M \rangle 10^{n_0}$, which is longer than n_0, the simulation in Stage 4 will complete. Therefore, D will do the opposite of M on the same input. Hence, M does not decide A, contradicting our assumption. Therefore, A is not decidable in $o(f(n))$ space.

Separating Polynomial Space Complexity Classes

Corollary

For any two functions $f_1, f_2 : \mathbb{N} \to \mathbb{N}$, where $f_1(n)$ is $o(f_2(n))$ and f_2 is space constructible, $\text{SPACE}(f_1(n)) \subsetneq \text{SPACE}(f_2(n))$.

- This corollary separates various space complexity classes.
  - The function n^c is space constructible for any natural number c. Hence, for any natural numbers $c_1 < c_2$, $\text{SPACE}(n^{c_1}) \subsetneq \text{SPACE}(n^{c_2})$.
  - A bit more work yields that n^c is space constructible for any rational number $c > 0$ and helps extend the preceding containment to hold for any rational numbers $0 < c_1 < c_2$.
  - Observing that two rational numbers c_1 and c_2 always exist between any two real numbers $\epsilon_1 < \epsilon_2$, such that $\epsilon_1 < c_1 < c_2 < \epsilon_2$, we obtain:

Corollary

For any two real numbers $0 < \epsilon_1 < \epsilon_2$, $\text{SPACE}(n^{\epsilon_1}) \subsetneq \text{SPACE}(n^{\epsilon_2})$.

Non-Deterministic Logarithmic and Polynomial Space

- The space hierarchy theorem also separates two space complexity classes encountered before:

Corollary

$\mathrm{NL} \subsetneq \mathrm{PSPACE}$.

- Savitch's theorem shows that $\mathrm{NL} \subseteq \mathrm{SPACE}(\log^2 n)$. The space hierarchy theorem shows that $\mathrm{SPACE}(\log^2 n) \subsetneq \mathrm{SPACE}(n)$. Hence, the corollary follows.
- This separation shows that $\mathrm{TQBF} \notin \mathrm{NL}$ because TQBF is PSPACE-complete with respect to log space reducibility.

Existence of Intractable Languages

- We establish the existence of problems that are decidable in principle but not in practice, i.e., problems that are decidable but intractable.
- By hierarchy, $\text{SPACE}(n^k) \subsetneq \text{SPACE}(n^{\log n}) \subsetneq \text{SPACE}(2^n)$. Therefore, we can separate PSPACE from $\text{EXPSPACE} = \bigcup_k \text{SPACE}(2^{n^k})$:

Corollary

$\text{PSPACE} \subsetneq \text{EXPSPACE}$.

- This corollary establishes the existence of decidable problems that are intractable, in the sense that their decision procedures must use more than polynomial space.
- The languages themselves are somewhat artificial - interesting only for the purpose of separating complexity classes. We use them later to prove the intractability of other, more natural, languages.

Time Constructible Functions

Definition (Time Constructible Function)

A function $t : \mathbb{N} \to \mathbb{N}$, where $t(n)$ is at least $O(n \log n)$, is called **time constructible** if the function that maps the string 1^n to the binary representation of $t(n)$ is computable in time $O(t(n))$.

- Thus, t is time constructible if some $O(t(n))$ time TM M exists that always halts with the binary $t(n)$ on its tape when started on 1^n.
- Example: All commonly occurring functions that are at least $n \log n$ are time constructible, including $n \log n$, $n\sqrt{n}$, n^2 and 2^n.
  To see that $n\sqrt{n}$ is time constructible:
  - First, design a TM to count the number of 1s in binary.
    - The TM moves a binary counter along the tape, incrementing it by 1 for every input position. This part uses $O(n \log n)$ steps because $O(\log n)$ steps are used for each of the n input positions.
  - Then, we compute $n\sqrt{n}$ in binary from the binary representation of n.
    - Any reasonable method needs $O(n \log n)$ time because the length of the numbers involved is $O(\log n)$.

The Time Hierarchy Theorem

Time Hierarchy Theorem

For any time constructible function $t : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $O(t(n))$ time but not decidable in time $o\left(\frac{t(n)}{\log t(n)}\right)$.

- We construct a TM D that decides a language A in time $O(t(n))$, where A cannot be decided in $o\left(\frac{t(n)}{\log t(n)}\right)$ time. D takes an input w of the form $\langle M \rangle 10^*$ and simulates M on input w, making sure not to use more than $t(n)$ time. If M halts within that much time, D gives the opposite output.
- In counting the number of steps used while simulating M, D incurs a time cost. The simulation must be efficient so that D runs in $O(t(n))$ time while avoiding all languages decidable in $o\left(\frac{t(n)}{\log t(n)}\right)$ time. The simulation introduces a logarithmic factor overhead, which is the reason for the $\frac{1}{\log t(n)}$ factor in the statement.

Description of the Algorithm D

- The following $O(t(n))$ time algorithm D decides a language A that is not decidable in $o\left(\frac{t(n)}{\log t(n)}\right)$ time:

  D: On input w:

  1. Let n be the length of w.
  2. Compute $t(n)$ using time constructibility, and store the value $\frac{t(n)}{\log t(n)}$ in a binary counter. Decrement this counter before each step used to carry out stages 3, 4, and 5. If the counter ever hits 0, reject.
  3. If w is not of the form $\langle M \rangle 10^*$ for some TM M, reject.
  4. Simulate M on w.
  5. If M accepts, then reject. If M rejects, then accept.
- We examine each stage to determine the running time.
  - Stages 1, 2 and 3 can be performed within $O(t(n))$ time.
  - In Stage 4, every time D simulates one step of M, it takes M's current state together with the tape symbol under M's tape head and looks up M's next action in its transition function so that it can update M's tape appropriately. All three of these objects (state, tape symbol, and transition function) are stored on D's tape somewhere.

Stage 4: Organizing D's Tape into "Tracks"

- To keep D efficient, the information needed in Stage 4 to simulate M, has to be stored close together.
  - We can think of D's single tape as organized into tracks.
    - One way to get two tracks is by storing one track in the odd positions and the other in the even positions.
    - Another, by enlarging D's tape alphabet to include each pair of symbols, one from the top track and the second from the bottom track.
  - We can get the effect of additional tracks similarly. Multiple tracks introduce only a constant factor overhead in time, provided that only a fixed number of tracks are used.

  D has three tracks.

Stage 4: Simulating Machine M

- One of the tracks contains the information on M's tape.
- A second contains its current state and a copy of M's transition function. During the simulation, D keeps the information on the second track near the current position of M's head on the first track. If M's head position moves, D shifts this information to keep it near the head. Because the size of the information on the second track depends only on M and not on the length of the input to M, the shifting needs constant time. Furthermore, because the required information is kept close together, the cost of looking up M's next action in its transition function and updating its tape is only a constant.

  If M runs in $g(n)$ time, D can simulate it in $\mathrm{O}(g(n))$ time.
- At every step in Stages 3 and 4, D must decrement the step counter originally set in Stage 2. D keeps the counter in binary on a third track and moves it to keep it near the present head position. This counter has a magnitude of about $\frac{t(n)}{\log t(n)}$, so its length is $\log \frac{t(n)}{\log t(n)}$, which is $\mathrm{O}(\log t(n))$. The cost of updating and moving it at each step adds a $\log t(n)$ factor to the simulation time, thus bringing the total running time to $\mathrm{O}(t(n))$.

A not Decidable in $o(t(n)/\log t(n))$ Time

- A is not decidable in $o(t(n)/\log t(n))$:

  Assume to the contrary that TM M decides A in time $g(n)$, where $g(n)$ is $o(t(n)/\log t(n))$. D can simulate M, using time $dg(n)$, for some constant d. If the total simulation time (not counting the time to update the step counter) is at most $\frac{t(n)}{\log t(n)}$, the simulation will run to completion. Because $g(n)$ is $o\left(\frac{t(n)}{\log t(n)}\right)$, some constant n_0 exists where $dg(n) < \frac{t(n)}{\log t(n)}$, for all $n \geq n_0$. Therefore D's simulation of M will run to completion as long as the input has length n_0 or more.

  We run D on input $\langle M \rangle 10^{n_0}$: This input is longer than n_0 so the simulation in Stage 4 will complete. Therefore D will do the opposite of M on the same input. Hence, M does not decide A, which contradicts our assumption.

  Therefore A is not decidable in $o\left(\frac{t(n)}{\log t(n)}\right)$ time.

Consequences of Time Hierarchy

Corollary

For any two functions $t_1, t_2 : \mathbb{N} \to \mathbb{N}$, where $t_1(n)$ is o $\left(\frac{t_2(n)}{\log t_2(n)}\right)$ and t_2 is time constructible,

$$\text{TIME}(t_1(n)) \subsetneq \text{TIME}(t_2(n)).$$

Corollary

For any two real numbers $1 \leq \epsilon_1 < \epsilon_2$,

$$\text{TIME}(n^{\epsilon_1}) \subsetneq \text{TIME}(n^{\epsilon_2}).$$

Corollary

$\text{P} \subsetneq \text{EXPTIME}$.

Subsection 2

Exponential Space Completeness

Intractability and Regular Expressions

- We can use the preceding results to show that a specific language is actually intractable. We follow several steps:
  - First, the hierarchy theorems tell us that a Turing machine can decide more languages in EXPSPACE than it can in PSPACE.
  - Then, we show that a particular language concerning generalized regular expressions is complete for EXPSPACE and hence cannot be decided in polynomial time or even in polynomial space.
- Recall that regular expressions are built up from the atomic expressions $\emptyset$, ε and members of the alphabet, by using the regular operations union, concatenation, and star, denoted $\cup$, $\circ$ and *, respectively.
- It can be shown that we can test the equivalence of two regular expressions in polynomial space.
- We show that, by allowing regular expressions with more operations than the usual regular operations, the complexity of analyzing the expressions may grow dramatically.

Generalized Regular Expressions

- Let $\uparrow$ be the **exponentiation operation**. If R is a regular expression and k is a nonnegative integer, writing $R \uparrow k$ is equivalent to the concatenation of R with itself k times. The notation R^k is shorthand for $R \uparrow k$:
$$R^k = R \uparrow k = R \circ R \circ \cdots \circ R.$$
- **Generalized regular expressions** allow the exponentiation operation in addition to the usual regular operations.
- Generalized regular expressions still generate the same class of regular languages as do the standard regular expressions.
- $\mathrm{EQ}_{\mathrm{REX}\uparrow} = \{\langle Q, R\rangle : Q \text{ and } R \text{ are equivalent regular expressions with exponentiation}\}$.
- To show that $\mathrm{EQ}_{\mathrm{REX}\uparrow}$ is intractable, we demonstrate that it is complete for the class EXPSPACE: Any EXPSPACE-complete problem cannot be in PSPACE, much less in P, since, otherwise, EXPSPACE would equal PSPACE, contradicting the preceding corollary.

EXPSPACE-Completeness

Definition (EXPSPACE-Complete Language)

A language B is EXPSPACE-**complete** if

1. $B \in$ EXPSPACE;
2. Every A in EXPSPACE is polynomial time reducible to B.

Theorem

$\mathrm{EQ}_{\mathrm{REX}\uparrow}$ is EXPSPACE-complete.

- We assume that all exponents are written as binary integers. The length of an expression is the total number of symbols it contains.
- We sketch an EXPSPACE algorithm for $\mathrm{EQ}_{\mathrm{REX}\uparrow}$.
  - We first use repetition to eliminate exponentiation.
  - Then convert the resulting expressions to NFAs.
  - Finally, we use an NFA equivalence testing procedure.
- To show that a language A in EXPSPACE is polynomial time reducible to $\mathrm{EQ}_{\mathrm{REX}\uparrow}$, we use "reductions via computation histories".

Reduction Via Computation Histories

- Given a TM M for A, we design a polynomial time reduction mapping an input w to a pair of expressions, R_1 and R_2, that are equivalent exactly when M accepts w.
  The expressions R_1 and R_2 simulate the computation of M on w.
  - Expression R_1 simply generates all strings over the alphabet consisting of symbols that may appear in computation histories.
  - Expression R_2 generates all strings that are not rejecting computation histories.

  So, if the TM accepts its input, no rejecting computation histories exist, and expressions R_1 and R_2 generate the same language.
- A rejecting computation history is the sequence of configurations that the machine enters in a rejecting computation on the input.
- The difficulty is that the size of the expressions constructed must be polynomial in n (so that the reduction can run in polynomial time), whereas the simulated computation may have exponential length. Exponentiation helps in expressing the long computation compactly.

Nondeterministic Test for Inequivalence of NFAs

- First we present a nondeterministic algorithm for testing whether two NFAs are inequivalent:

  N: On input $\langle N_1, N_2 \rangle$, where N_1 and N_2 are NFAs:

  1. Place a marker on each of the start states of N_1 and N_2.
  2. Repeat $2^{q_1+q_2}$ times, where q_1,q_2 are the numbers of states in N_1,N_2:
     3. Nondeterministically select an input symbol and change the positions of the markers on the states of N_1,N_2 to simulate reading that symbol.
  4. If at any point, a marker was placed on an accept state of one of the finite automata and not on any accept state of the other finite automaton, accept. Otherwise, reject.

Correctness and Complexity of the Inequivalence Test

- If N_1 and N_2 are equivalent, N clearly rejects because it only accepts when it determines that one machine accepts a string that the other does not accept.
- If the automata are not equivalent, some string is accepted by one machine and not by the other. Some such string must be of length at most $2^{q_1+q_2}$: Otherwise, consider using the shortest such string as the sequence of nondeterministic choices. Only $2^{q_1+q_2}$ different ways exist to place markers on the states of N_1 and N_2, so in a longer string the positions of the markers would repeat. By removing the portion of the string between the repetitions, a shorter such string would be obtained. Hence algorithm N would guess this string among its nondeterministic choices and would accept.
- Algorithm N runs in nondeterministic linear space. By applying Savitch's theorem, we obtain a deterministic $O\left(n^2\right)$ space algorithm for this problem.

EXPSPACE-Algorithm for $\mathrm{EQ}_{\mathrm{REX}\uparrow}$

- We design algorithm E that decides $\mathrm{EQ}_{\mathrm{REX}\uparrow}$.
  E: On input $\langle R_1, R_2 \rangle$, where R_1 and R_2 are regular expressions with exponentiation:
  1. Convert R_1 and R_2 to equivalent regular expressions B_1 and B_2 that use repetition instead of exponentiation.
  2. Convert B_1 and B_2 to equivalent NFAs N_1 and N_2, using the well-known conversion procedure from the theory of languages.
  3. Use the deterministic version of N to determine if N_1,N_2 are equivalent.
- Algorithm E obviously is correct.
- Using repetition to replace exponentiation may increase the length of an expression by a factor of 2^{ℓ}, where ℓ is the sum of the lengths of the exponents. Thus, B_1 and B_2 have a length of at most $n2^n$, n input length. The conversion procedure increases the size linearly and hence NFAs N_1 and N_2 have at most $\mathrm{O}(n2^n)$ states. Thus, the deterministic version of N uses space $\mathrm{O}\left((n2^n)^2\right) = \mathrm{O}\left(n^2 2^{2n}\right)$. Hence $\mathrm{EQ}_{\mathrm{REX}\uparrow}$ is decidable in exponential space.

EXPSPACE-Hardness of $\mathrm{EQ}_{\mathrm{REX}\uparrow}$ I

- Next, we show that $\mathrm{EQ}_{\mathrm{REX}\uparrow}$ is EXPSPACE-hard: Let A be a language decided by TM M running in space $2^{(n^k)}$, for a constant k. The reduction maps an input w to a pair of regular expressions, R_1, R_2.
  - Expression R_1 is Δ^*, where, if Γ and Q are M's tape alphabet and states, $\Delta = \Gamma \cup Q \cup \{\#\}$ is the alphabet consisting of all symbols that may appear in a computation history.
  - We construct expression R_2 to generate all strings that are not rejecting computation histories of M on w.
- M accepts w iff M on w has no rejecting computation histories.

Therefore, the two expressions are equivalent iff M accepts w.

A rejecting computation history for M on w is a sequence of configurations separated by #s. We assume all configurations have length $2^{(n^k)}$ and are padded on the right by blank symbols if they otherwise would be too short. The first configuration is the start configuration of M on w. The last configuration is a rejecting configuration. Each configuration must follow from the preceding one according to the rules specified in the transition function.

EXPSPACE-Hardness of $\mathrm{EQ}_{\mathrm{REX}\uparrow}$ II

- A string may fail to be a rejecting computation in several ways:
  - It may fail to start or end properly;
  - It may be incorrect somewhere in the middle.

  Expression R_2 equals $R_{\mathrm{bad-start}} \cup R_{\mathrm{bad-window}} \cup R_{\mathrm{bad-reject}}$, where each subexpression corresponds to one of the three ways a string may fail.
- $R_{\mathrm{bad-start}}$ generates all strings not starting with start configuration:

  Configuration C_1 looks like $q_0 w_1 w_2 \ldots w_n \sqcup \cdots \sqcup \#$. We write $R_{\mathrm{bad-start}}$ as the union of several subexpressions to handle each part of C_1:

  $$R_{\mathrm{bad-start}} = S_0 \cup S_1 \cup \cdots \cup S_n \cup S_b \cup S_{\#}.$$

  - S_0 generates all strings that do not start with q_0. Let S_0 be $\Delta_{-q_0}\Delta^*$, where Δ_{-q_0} is shorthand for the union of all symbols in Δ but q_0.
  - S_1 generates all strings that do not contain w_1 in the second position. Let S_1 be $\Delta\Delta_{-w_1}\Delta^*$. For $1 \leq i \leq n$, expression S_i is $\Delta^i\Delta_{-w_i}\Delta^*$.
  - S_b generates all strings that fail to contain a blank symbol in some position $n+2$ through $2^{(n^k)}$. Let $S_b = \Delta^{n+1}(\Delta \cup \varepsilon)^{2^{(n^k)}-n-2}\Delta_{-\sqcup}\Delta^*$.
  - $S_{\#}$ generates all strings that do not have a $\#$ in position $2^{(n^k)}+1$. Let $S_{\#}$ be $\Delta^{2^{(n^k)}}\Delta_{-\#}\Delta^*$.

EXPSPACE-Hardness of $\mathrm{EQ}_{\mathrm{REX}\uparrow}$ III

- We still need to construct $R_{\mathsf{bad-reject}}$ and $R_{\mathsf{bad-window}}$:
  - We turn to $R_{\mathsf{bad-reject}}$: It generates all strings that fail to contain a rejecting configuration. Any rejecting configuration contains the state q_{reject}. So we let $R_{\mathsf{bad-reject}} = \Delta^*_{-q_{\mathsf{reject}}}$.
  - Finally, we construct $R_{\mathsf{bad-window}}$ that generates all strings whereby one configuration does not properly lead to the next configuration. One configuration legally yields another whenever every three consecutive symbols in the first configuration correctly yield the corresponding three symbols in the second configuration according to the transition function. Hence, if one configuration fails to yield another, the error will be apparent from an examination of the appropriate six symbols. We use this idea to construct:

$$R_{\mathsf{bad-window}} = \bigcup_{\mathsf{bad}(abc,def)} \Delta^* abc \Delta^{(2^{(n^k)}-2)} def \Delta^*,$$

    $\mathsf{bad}(abc, def)$ means abc does not yield def according to the transition.
- Several exponents of magnitude roughly $2^{(n^k)}$ appear, and their total length in binary is $O\left(n^k\right)$. Thus, the length of R is polynomial in n.

Subsection 3

Relativization

Idea of Relativization

- The proof that $\mathrm{EQ}_{\mathrm{REX}^{\uparrow}}$ is EXPSPACE-complete uses diagonalization.
- The question arises on whether diagonalization could be used to show that a ND poly time TM can decide a language that is not in P.
- The method of relativization gives strong evidence against the possibility of solving the P versus NP question using diagonalization.
  - The model of computation is strengthened by giving the Turing machine certain extra information.
  - Depending on which information is actually provided, the TM may be able to solve some problems more easily than before.

Example: Suppose that we grant the TM the ability to solve the satisfiability problem in a single step, for any size Boolean formula. This imaginary single step solver is called an oracle. The enriched machine could use the oracle to solve any NP problem in polynomial time, regardless of whether P equals NP, because every NP problem is polynomial time reducible to the satisfiability problem. The term relativization refers to computing relative to the satisfiability problem.

Oracle Turing Machines

Definition (Oracle Turing Machine)

An **oracle** for a language A is a device that is capable of reporting whether any string w is a member of A. An **oracle Turing machine** M^A is a modified Turing machine that has the additional capability of querying an oracle. Whenever M^A writes a string on a special oracle tape, it is informed whether that string is a member of A in a single computation step.
Let P^A be the class of languages decidable with a polynomial time oracle Turing machine that uses oracle A. Define NP^A analogously.

- Example:
  - Polynomial time computation relative to the satisfiability problem contains all of NP, i.e., $\mathrm{NP} \subseteq \mathrm{P}^{\mathrm{SAT}}$.
  - Since $\mathrm{P}^{\mathrm{SAT}}$ is a deterministic complexity class, it is closed under complementation. Therefore, we also have $\mathrm{coNP} \subseteq \mathrm{P}^{\mathrm{SAT}}$.

An Additional Example of an Oracle Machine

- The class $\mathrm{NP}^{\mathrm{SAT}}$ contains languages believed not to be in NP.
- Example: Call two Boolean formulas ϕ and ψ over the variables $x_1, \ldots, x_\ell$ **equivalent** if the formulas have the same value on any assignment to the variables. Call a formula **minimal** if no smaller formula is equivalent to it. Define the language

$$\textsc{NonMinFormula} = \{\langle\phi\rangle : \phi \text{ is not a minimal Boolean formula}\}.$$

NonMinFormula does not seem to be in NP (though whether it actually belongs to NP is not known). However, NonMinFormula is in $\mathrm{NP}^{\mathrm{SAT}}$: A nondeterministic polynomial time oracle Turing machine with a SAT oracle can test whether ϕ is a member:

- The inequivalence problem for two Boolean formulas is solvable in NP, since a nondeterministic machine can guess the distinguishing assignment. Hence, the equivalence problem is in coNP.
- The nondeterministic oracle machine for NonMinFormula nondeterministically guesses a smaller equivalent formula, tests whether it is equivalent, using the SAT oracle, and accepts if it is.

Subsection 4

Limits of the Diagonalization Method

Oracles and the Diagonalization Method

- We show the existence of oracles A and B for which:
  - P^A and NP^A are provably different;
  - P^B and NP^B are provably equal.
- So, it is unlikely to resolve the P vs NP using diagonalization:
  - The diagonalization method is a simulation of one Turing machine by another. The simulation is done so that the simulator can determine the behavior of the other machine and then behave differently.
  - Suppose that both of these Turing machines were given identical oracles. Then, whenever the simulated machine queries the oracle, so can the simulator, and therefore the simulation can proceed as before. Consequently, any theorem proved about Turing machines by using only the diagonalization method would still hold if both machines were given the same oracle. In particular, if we could prove that P and NP were different by diagonalizing, we could conclude that they are different relative to any oracle as well. But, clearly, this conclusion is false.
- Similarly, no proof that relies on a simple simulation could show $P = NP$, because that would show that $P^A = NP^A$, for any oracle A.

The Oracle Theorem

Theorem

1. An oracle A exists, such that $\mathrm{P}^A \neq \mathrm{NP}^A$.
2. An oracle B exists, such that $\mathrm{P}^B = \mathrm{NP}^B$.

- Exhibiting oracle B is easy: Let B be any PSPACE-complete problem such as TQBF.
- We exhibit oracle A by construction: We design A so that a certain language L_A in NP^A provably requires brute-force search, whence L_A cannot be in P^A. Hence, we can conclude that $\mathrm{P}^A \neq \mathrm{NP}^A$.

  The construction considers every polynomial time oracle machine in turn and ensures that each fails to decide the language L_A.

Proof of the Oracle Theorem

- Let B be TQBF. We have $\mathrm{NP}^{\mathrm{TQBF}} \subseteq \mathrm{NPSPACE} \subseteq \mathrm{PSPACE} \subseteq \mathrm{P}^{\mathrm{TQBF}}$.
  - Containment 1 holds because we can convert the nondeterministic polynomial time oracle TM to a nondeterministic polynomial space machine that computes the answers to queries regarding TQBF instead of using the oracle.
  - Containment 2 follows from Savitch's theorem.
  - Containment 3 holds because TQBF is PSPACE-complete.

  Hence, we have $\mathrm{P}^{\mathrm{TQBF}} = \mathrm{NP}^{\mathrm{TQBF}}$.
- We show, next, how to construct oracle A. For any oracle A, let L_A be the collection of all strings for which a string of equal length appears in A: $L_A = \{w : \exists x \in A(|x| = |w|)\}$.
  - For any A, the language L_A is in NP^A.
  - To show L_A is not in P^A, we design A as follows: Let $M_1, M_2, \ldots$ be a list of all polynomial time oracle TMs. We assume for simplicity that M_i runs in time n^i. The construction proceeds in stages, where stage i constructs a part of A, which ensures that M_i^A does not decide L_A.

Construction of A

- Stage i constructs a part of A, which ensures that M_i^A does not decide L_A. We construct A by declaring that certain strings are in A and others are not in A. Each stage determines the status of only a finite number of strings.
  Stage i: Assume a finite number of strings have already been declared to be in or out of A. We choose n greater than the length of any such string and such that $2^n > n^i$. We extend our information about A so that M_i^A accepts 1^n whenever that string is not in L_A:
  - We run M_i on input 1^n and respond to its oracle queries as follows:
    - If M_i queries a string y whose status has already been determined, we respond consistently.
    - If y's status is undetermined, we respond `NO` to the query and declare y to be out of A.

    We continue the simulation of M_i until it halts.

  Consider the situation from M_i's perspective:
  - If it finds a string of length n in A, it should accept since 1^n is in L_A.
  - If M_i determines that all strings of length n are not in A, it should reject because it knows that 1^n is not in L_A.

Construction of A (Cont'd)

- We are at Stage i:
  - M_i does not have enough time to ask about all strings of length n, and we have answered `NO` to each of the queries it has made. Hence when M_i halts and must decide whether to accept or reject, it does not have enough information to be sure that its decision is correct.

  Our objective is to ensure that its decision is not correct. We do so by observing its decision and then extending A with the reverse.
  - If M_i accepts 1^n, we declare all the remaining strings of length n to be out of A and so determine that 1^n is not in L_A.
  - If M_i rejects 1^n, we find a string of length n that M_i has not queried and declare that string to be in A to guarantee that 1^n is in L_A. Such a string must exist because M_i runs for n^i steps, which is fewer than 2^n, the total number of strings of length n.

  Either way, we have ensured that M_i^A does not decide L_A.
- Stage i is completed and we proceed with stage $i+1$.
- After finishing all stages, we arbitrarily declare any string of status still undetermined to be out of A.

Subsection 5

Circuit Complexity

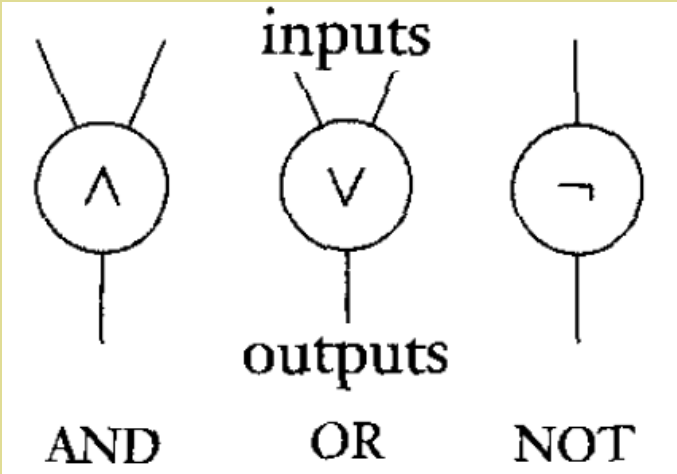
Boolean Circuits

- Computers are built from electronic devices wired together in a design called a **digital circuit**.
- **Boolean circuits** are theoretical counterpart to digital circuits, which can be used to simulate theoretical models, such as Turing machines.
- A connection between TMs and Boolean circuits serves two purposes:
  - Circuits provide a convenient computational model for attacking the P versus NP and related questions.
  - Circuits provide an alternative proof that SAT is NP-complete.

Definition (Boolean Circuit)

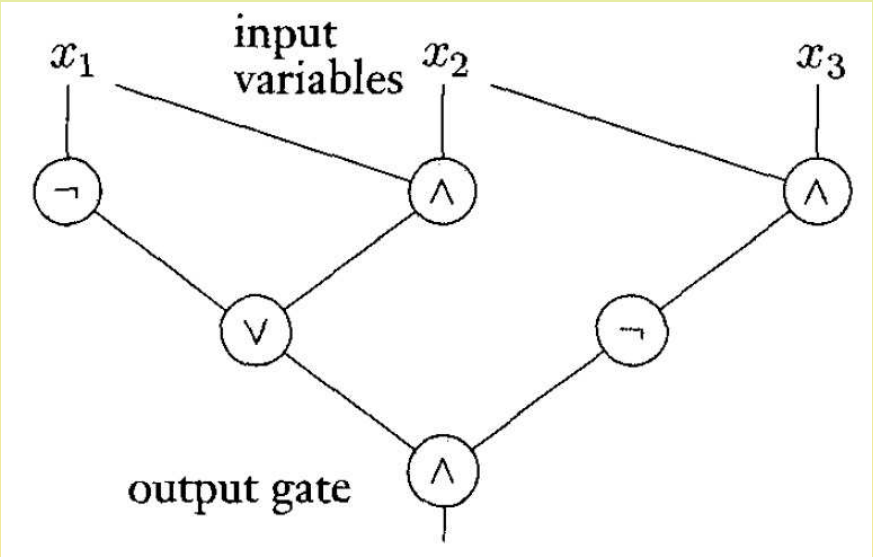
A **Boolean circuit** is a collection of gates and inputs connected by wires. Cycles are not permitted. Gates take three forms:

- AND gates;
- OR gates;
- NOT gates.



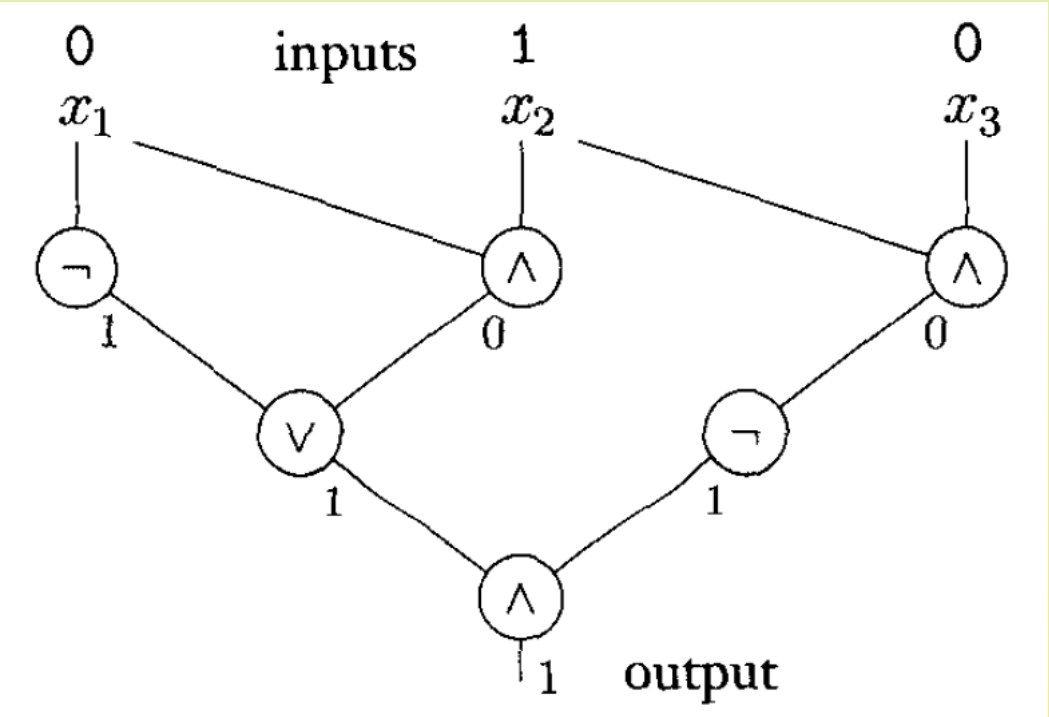
Gates, Inputs and Output

- The wires in a Boolean circuit carry the Boolean values 0 and 1.
- The gates are simple processors that compute AND, OR, and NOT:
  - The AND function outputs 1 if both of its inputs are 1 and 0 otherwise.
  - The OR function outputs 0 if both of its inputs are 0 and 1 otherwise.
  - The NOT function outputs the opposite of its input.
- The inputs are labeled $x_1, \ldots, x_n$.
- One of the gates is designated the **output gate**.



Computing an Output Vale

- A Boolean circuit computes an output value from a setting of the inputs by propagating values along the wires and computing the function associated with the respective gates until the output gate is assigned a value:



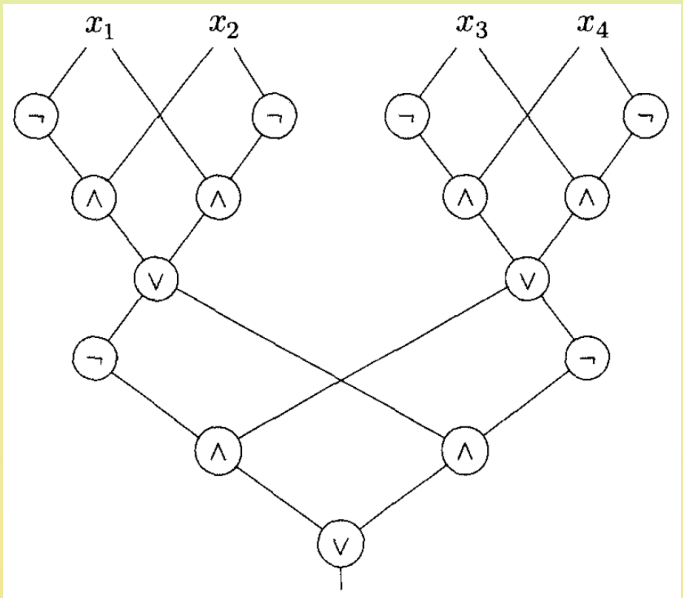
Functions Computed by Boolean Circuits

- To a Boolean circuit C with n input variables, we associate a function $f_C : \{0,1\}^n \to \{0,1\}$, where, if C outputs b when its inputs $x_1, \ldots, x_n$ are set to $a_1, \ldots, a_n$, we write

$$f_C(a_1, \ldots, a_n) = b.$$

  We say that C **computes** the function f_C.
- We sometimes consider Boolean circuits with multiple output gates. A circuit with k output gates computes a function with range $\{0,1\}^k$.
- Example:

  The n-input **parity function** $\text{parity}_n : \{0,1\}^n \to \{0,1\}$ outputs 1 if an odd number of 1s appear in the input variables.
  The circuit shown computes parity_4:



Circuit Families

- We want to use circuits to test membership in languages, once they have been suitably encoded into $\{0, 1\}$.
- One problem is that any particular circuit can handle only inputs of some fixed length, whereas a language may contain strings of different lengths.
- So, instead of using a single circuit to test language membership, we use an entire family of circuits, one for each input length.

Definition (Circuit Family)

A **circuit family** C is an infinite list of circuits $(C_0, C_1, C_2, \ldots)$, where C_n has n input variables. We say that C **decides** a language A over $\{0, 1\}$ if, for every string w,

$$w \in A \quad \text{iff} \quad C_n(w) = 1,$$

where n is the length of w.

Size and Depth Complexity of Circuit Families

- The **size** of a circuit is the number of gates that it contains.
- Two circuits are **equivalent** if they have the same input variables and output the same value on every input assignment.
- A circuit is **size minimal** if no smaller circuit is equivalent to it.
- The problem of minimizing circuits has obvious engineering applications but is very difficult to solve in general. Even testing a particular circuit for minimality does not appear to be in P or in NP.
- A circuit family for a language is **minimal** if every C_i on the list is a minimal circuit.
- The **size complexity** of a circuit family $(C_0, C_1, C_2, \ldots)$ is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the size of C_n.
- The **depth** of a circuit is the length (number of wires) of the longest path from an input variable to the output gate.
- We define **depth minimal** circuits and circuit families, and the **depth complexity** of circuit families, as we did with circuit size.

Circuit Size and Circuit Depth Complexity of Languages

Definition (Circuit Size and Circuit Depth Complexity)

The **circuit size complexity** of a language is the size complexity of a size minimal circuit family for that language.
The **circuit depth complexity** of a language is the depth complexity of a depth minimal circuit family for that language.

- Example: We can generalize the preceding example to give circuits that compute the parity function on n variables with $O(n)$ gates. One way to do so is to:
  - Build a binary tree of gates that compute the XOR function, where the XOR function is the same as the 2-parity function;
  - Then implement each XOR gate with 2 NOTs, 2 ANDs, and 1 OR, as we did in that earlier example.

  Let A be the language of strings that contain an odd number of 1s. Then A has circuit size complexity $O(n)$.

Relating Time with Circuit Complexity

- A language with small time complexity has small circuit complexity:

Theorem

Let $t : \mathbb{N} \to \mathbb{N}$ be a function, where $t(n) \geq n$. If $A \in \mathrm{TIME}(t(n))$, then A has circuit size complexity $\mathrm{O}\left(t^2(n)\right)$.

- To prove P $\neq$ NP one might attempt to show that some language in NP has more than polynomial circuit size complexity.
- We let M be a TM that decides A in time $t(n)$ (ignoring the constant in $\mathrm{O}\left(t(n)\right)$. For each n, we construct a circuit C_n, that simulates M on inputs of length n. The gates of C_n are organized in rows, one for each of the $t(n)$ steps in M's computation on an input of length n. Each row represents the configuration of M at that step. Each row calculates its configuration from the previous row's configuration. We modify M so that the input is encoded into $\{0, 1\}$. Moreover, when M is about to accept, it moves its head onto the leftmost cell and writes the $\sqcup$ symbol on that cell prior to entering the accept state.

Proof of the Theorem

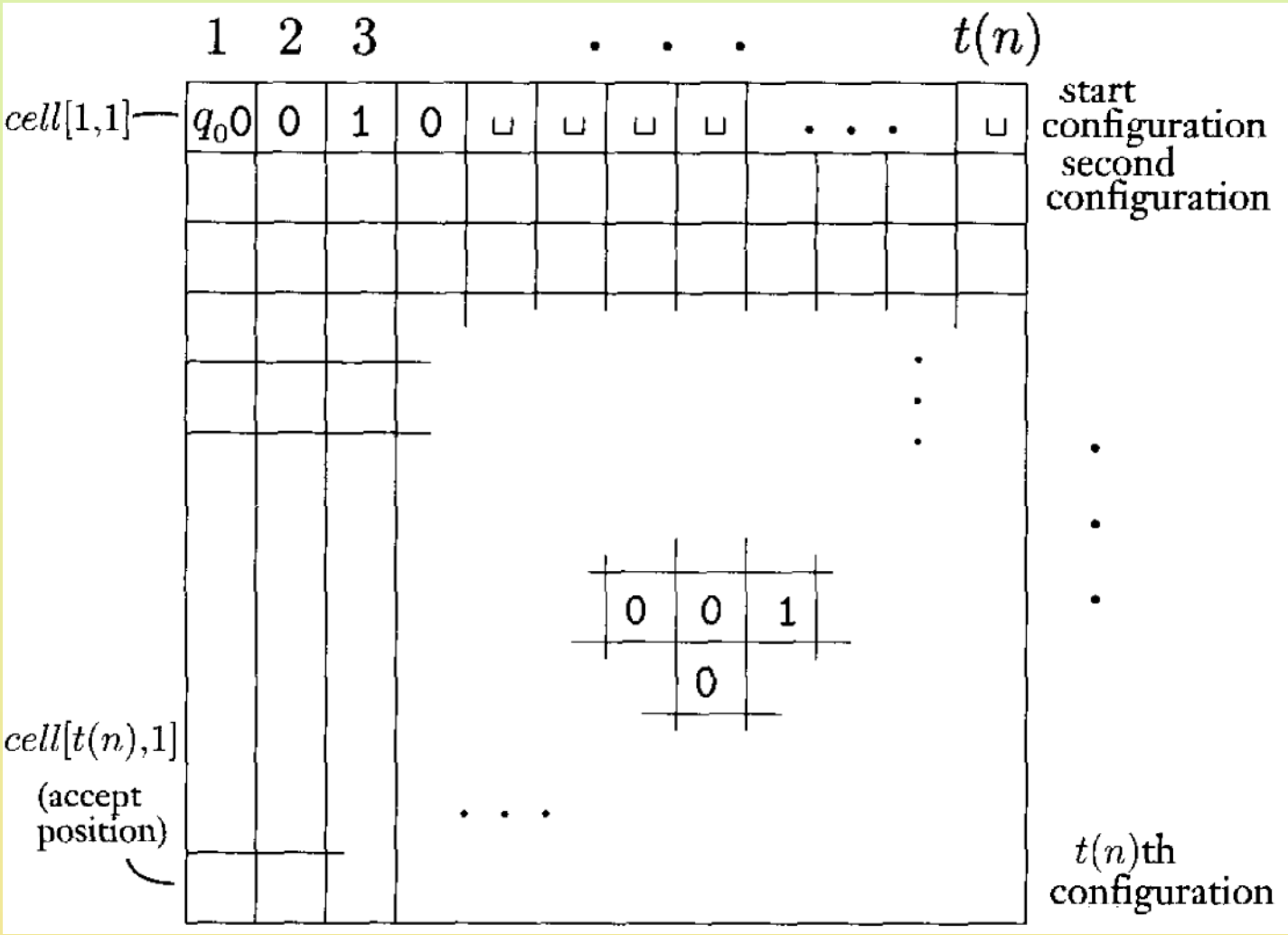
- Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ decide A in time $t(n)$. Let w be an input of length n to M. Define a **tableau** for M on w to be a $t(n) \times t(n)$ table whose rows are configurations of M.
  - The top row of the tableau contains the start configuration of M on w;
  - The i-th row the configuration at the i-th step of the computation.

  We represent both the state and the tape symbol under the tape head by a single character:
  - E.g., if M is in state q and its tape contains 1011 with the head at 0, the old format is $1q011$ and the new format $1\boxed{q0}11$.

  Each entry of the tableau can contain a tape symbol (member of Γ) or a combination of a state and a tape symbol (member of $Q \times \Gamma$). The entry at the i-th row and j-th column of the tableau is cell$[i, j]$. We make two assumptions about TM M:
  1. M accepts only when its head is on the leftmost tape cell and that cell contains the $\sqcup$ symbol.
  2. Once M has halted it stays in the same configuration for all future time steps. So, cell$[t(n), 1]$ determines whether M has accepted.

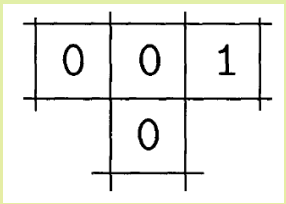
Example of a Tableau

- The following is a possible tableau for M on 0010:



Interaction Between Contents of Cells in a Tableau

- The contents of each cell are determined by certain cells in the preceding row:
  - Knowing 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$[i,j]$ via M's transition function



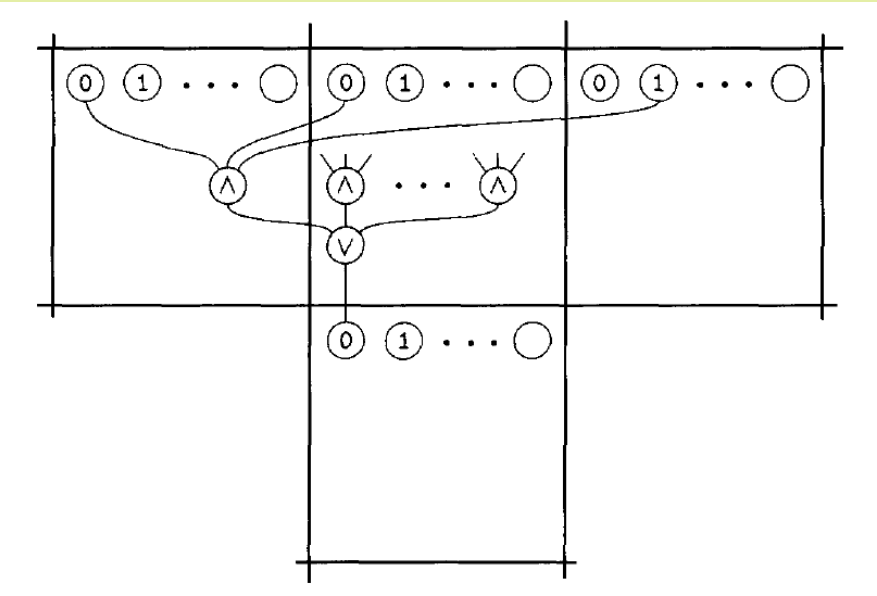
E.g., if the three top symbols, 0, 0, and 1, are tape symbols without states, the middle symbol must remain a 0 in the next row.

The circuit C_n 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. Let k be the number of elements in $\Gamma \cup (\Gamma \times Q)$. We create k lights for each cell in the tableau, for a total of $kt^2(n)$ lights. We call them light$[i,j,s]$, where $1 \leq i,j \leq t(n)$, $s \in \Gamma \cup (\Gamma \times Q)$. If light$[i,j,s]$ is on, cell$[i,j]$ contains the symbol s. Consider the three cells that can affect cell$[i,j]$ and determine through δ which of their settings cause cell$[i,j]$ to contain s.

Determining the Contents of Cells in a Tableau

- Suppose that, if 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$[i, j, s]$.

  Since several different settings (a_1, b_1, c_1), (a_2, b_2, c_2), $\ldots$, (a_ℓ, b_ℓ, c_ℓ) may cause cell$[i, j]$ to contain s, 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.



Setting the Input and Output Variables

- The circuitry is repeated for each light, except at the boundaries:
  - Each cell$[i, 1]$ at the left boundary of the tableau has only two preceding cells that affect its contents.
  - The cells at the right boundary are similar.

  In these cases, we modify the circuitry accordingly.
  The cells in the first row have no predecessors. Their lights are wired to the input variables:
  - Thus, light$[1, 1, \boxed{q_0 1}]$ is connected to input w_1;
  - light$[1, 1, \boxed{q_0 0}]$ is connected through a NOT gate to input w_1.
  - light$[1, 2, 1], \ldots,$ light$[1, n, 1]$ are connected to inputs $w_2, \ldots, w_n$.
  - light$[1, 2, 0], \ldots,$ light$[1, n, 0]$ are connected via NOTs to $w_2, \ldots, w_n$.
  - light$[1, n + 1, \sqcup], \ldots,$ light$[1, t(n), \sqcup]$ are on.
  - Finally, all other lights in the first row are off.
- M accepts w if it is in an accept state q_{accept} on a cell containing $\sqcup$ at the left-hand end of the tape at step $t(n)$. The output gate, thus, is the one attached to light$[t(n), 1, \boxed{q_{\text{accept}} \sqcup}]$.

The Circuit Satisfiability Problem

- We obtain an alternative proof of the Cook-Levin theorem.
- We say that a Boolean circuit is **satisfiable** if some setting of the inputs causes the circuit to output 1:

$$\textsc{CircuitSat} = \{\langle C \rangle : C \text{ is a satisfiable Boolean circuit}\}.$$

Theorem

$\textsc{CircuitSat}$ is NP-complete.

- We must show that
  - $\textsc{CircuitSat}$ is in NP;
  - Any language A in NP is reducible to $\textsc{CircuitSat}$.

  The first is obvious. For the second, we must give a polynomial time reduction f that maps strings to circuits, where $f(w) = \langle C \rangle$ implies that $w \in A$ iff the Boolean circuit C is satisfiable.

The Reduction to CircuitSat

- Because A is in NP, it has a polynomial time verifier V whose input has the form $\langle x, c \rangle$, where c may be the certificate showing that x is in A. To construct f, we obtain the circuit simulating V using the method of the theorem. We fill in the inputs to the circuit that correspond to x with the symbols of w. The only remaining inputs to the circuit correspond to the certificate c. We call this circuit C and output it.
- If C is satisfiable, a certificate exists, so w is in A. Conversely, if w is in A, a certificate exists, so C is satisfiable.
- The reduction runs in polynomial time, since the construction of the circuit can be done in time that is polynomial in n:
  - The running time of the verifier is n^k for some k. Thus, the size of the circuit constructed is $O\left(n^{2k}\right)$.
  - The structure of the circuit is quite simple, so the running time of the reduction is $O\left(n^{2k}\right)$.

Alternative Proof of the Cook-Levin Theorem

- We present an alternative proof of the Cook-Levin theorem:

Theorem (The Cook-Levin Theorem)

3SAT is NP-complete.

- 3SAT is obviously in NP.
- We show that all languages in NP reduce to 3SAT in polynomial time. We do so by reducing CIRCUITSAT to 3SAT in polynomial time. The reduction converts a circuit C to a formula ϕ, such that C is satisfiable iff ϕ is satisfiable. The formula contains one variable for each variable and each gate in the circuit and simulates the circuit. A satisfying assignment for ϕ contains a satisfying assignment to C. It also contains the values at each of C's gates in C's computation on its satisfying assignment.
  - ϕ's satisfying assignment "guesses" C's entire computation on its satisfying assignment.
  - ϕ's clauses check the correctness of that computation.
  - Finally, ϕ contains a clause stipulating that C's output is 1.

Construction of the Formula

- Let C be a circuit containing inputs $x_1, \ldots, x_\ell$ and gates $g_1, \ldots, g_m$. The reduction builds a formula ϕ with variables $x_1, \ldots, x_\ell, g_1, \ldots, g_m$. Each of ϕ's variables corresponds to a wire in C.
  - The x_i variables correspond to the input wires.
  - The g_i variables correspond to the wires at the gate outputs.

  We relabel ϕ's variables as $w_1, \ldots, w_{\ell+m}$.
- We describe ϕ's clauses. We write them using implications, but they can be converted in usual form.
  - Each NOT gate in C with input wire w_i and output wire w_j is equivalent to the expression $(\overline{w_i} \rightarrow w_j) \wedge (w_i \rightarrow \overline{w_j})$, or, equivalently, $(w_i \vee w_j) \wedge (\overline{w_i} \vee \overline{w_j})$.
  - Each AND gate in C with inputs w_i and w_j and output w_k is equivalent to $((\overline{w_i} \wedge \overline{w_j}) \rightarrow \overline{w_k}) \wedge ((\overline{w_i} \wedge w_j) \rightarrow \overline{w_k}) \wedge ((w_i \wedge \overline{w_j}) \rightarrow \overline{w_k}) \wedge ((w_i \wedge w_j) \rightarrow w_k)$, or, equivalently, $(w_i \vee w_j \vee \overline{w_k}) \wedge (w_i \vee \overline{w_j} \vee \overline{w_k}) \wedge (\overline{w_i} \vee w_j \vee \overline{w_k}) \wedge (\overline{w_i} \vee \overline{w_j} \vee w_k)$.

Correctness and Complexity

- We continue with the construction of ϕ:
  - Similarly, each OR gate in C with inputs w_i and w_j and output w_k is equivalent to $((\overline{w_i} \wedge \overline{w_j}) \rightarrow \overline{w_k}) \wedge ((\overline{w_i} \wedge w_j) \rightarrow w_k) \wedge ((w_i \wedge \overline{w_j}) \rightarrow w_k) \wedge ((w_i \wedge w_j) \rightarrow w_k)$, or, equivalently, $(w_i \vee w_j \vee \overline{w_k}) \wedge (w_i \vee \overline{w_j} \vee w_k) \wedge (\overline{w_i} \vee w_j \vee w_k) \wedge (\overline{w_i} \vee \overline{w_j} \vee w_k)$.
  - Finally, we add to ϕ clause (w_m), where w_m is C's output gate.

  If some of the clauses contain fewer than three literals, we can easily expand them to the desired size by repeating literals.
- Correctness:
  - If a satisfying assignment for C exists, we obtain a satisfying assignment for ϕ by assigning the g_i variables according to C's computation on this assignment.
  - If a satisfying assignment for ϕ exists, it gives an assignment for C, because it describes C's entire computation with output value 1.
- Time complexity: The reduction can be done in polynomial time because it is simple to compute and the output size is polynomial (linear) in the size of the input.