

Question

Why are we unsuccessful in finding polynomial time algorithms for some problems?

Possible answers:

- Perhaps such problems have, as yet undiscovered, polynomial time algorithm that rest on unknown principles.
- Some of such problems simply cannot be solved in polynomial time. They may be intrinsically difficult.

22c:131 Limits of Computation

Hantao Zhang

<http://www.cs.uiowa.edu/~hzhang/c131>

The University of Iowa
Department of Computer Science

The Class NP

Example

Hamiltonian path problem:

- A Hamiltonian path in a directed graph G is a path that goes through each node of G exactly once.
- Hamiltonian path problem consists of testing whether a directed Graph G contains a Hamiltonian path connecting two specified nodes.
- $HAMPATH = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t \}$

Why Complexity Theory

Complexities of many problems are linked! The discovery of a polynomial time algorithm for one such problem can be used to solve an entire class of problems.

Polynomial Verifiability

The *HAMPATH* problem has a feature called *polynomial verifiability* which is important for understanding its complexity:

If a Hamiltonian path in a graph G is discovered (somehow) we could easily convince someone else of its existence, simply by presenting it!

That is, *verifying* the existence of a Hamiltonian graph may be much easier than *determining* its existence.

A Hamiltonian path

Figure 1 shows a graph that contains a Hamiltonian path between nodes s and t .

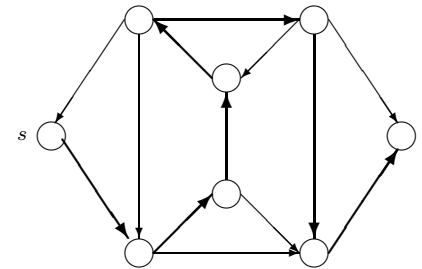


Figure 1: A Hamiltonian path

Note

Some problems may not be polynomially verifiable. For example $\overline{HAMPATH}$, the complement of *HAMPATH*, is not polynomial time verifiable.

Rationale: Even if we could determine that a graph did not have a Hamiltonian path we don't know a way for verifying its non-existence without using the same exponential time algorithm that determined its nonexistence!

Another example

Another example of polynomial verifiable problem is number *compositness*.

A natural number is composite if it is the product of two integers greater than 1

$$COMPOSITES = \{x \mid x \in \mathcal{N} \wedge x = pq, p, q \in \mathcal{N}, p, q > 1\}$$

Although we don't know a polynomial time algorithm for deciding this problems, we can easily verify that a number x is composite: all that is needed is a divisor of x

Observations

- A verifier uses additional information, represented by c in Verifier's definition.
- This info is called a *certificate* or *proof*, of membership.
- For polynomial verifiers the certificate has a polynomial length (in the length of w).
- **Examples:** the certificate for $\langle G, s, t \rangle \in \text{HAMPATH}$ is a Hamiltonian path between s and t ; the certificate for $x \in \text{COMPOSITES}$ is a divisor p of x

Verifier

A verifier for a language A is an algorithm V where:
 $A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$.

Note:

- Time of the verifier is measured only in terms of length of w . A polynomial time verifier runs in polynomial time in the length of w .
- A language A is polynomially verifiable if it has a polynomial time verifier.

A NTM deciding *HAMPATH*

N_1 = "On input $\langle G, s, t \rangle$ where G is direct graph with nodes s, t :

1. Write a list of m numbers, p_1, p_2, \dots, p_m where m is the number of nodes in G . Each number is nondeterministically selected between 1 and m .
2. Check for repetitions in the list. If any are found *reject*.
3. Check whether $s = p_1$ and $t = p_m$. If either fails *reject*
4. For each i , $1 \leq i \leq m$, check whether (p_i, p_{i+1}) is an edge of G . If any are not, *reject*. Otherwise *accept*."

The Class NP

NP is the class of languages that have polynomial time verifiers

Note:

- The term **NP** comes from **nondeterministic polynomial time** and is derived from an alternative characterization using nondeterministic polynomial time Turing machines.
- **NP** class is important because it contains many problems of practical interest
- $\text{HAMPATH}, \text{COMPOSITES} \in \text{NP}$. Note that $\text{COMPOSITES} \in \text{P}$ but proving it is more difficult.

Proof

$A \in \mathbf{NP} \Rightarrow A$ is decidable by a polynomial time NTM:

Assume that V is polynomial time verifier deciding A in time n^k . The NTM N_V equivalent to V works as follows:

$N_V =$ "On input w of length n :

1. Nondeterministically select a string c of length at most n^k .
2. Run V on input $\langle w, c \rangle$
3. If V accepts, *accept*, if V rejects *rejects*."

Theorem 7.20

A language is in \mathbf{NP} iff it is decided by some nondeterministic polynomial time Turing machine.

Proof idea:

- We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and viceversa.
- The verifier simulates the NTM by using the accepting branches as certificates

Class $\mathbf{NTIME}(t(n))$

The nondeterministic time complexity class $\mathbf{NTIME}(t(n))$ is defined by:

$\mathbf{NTIME}(t(n)) = \{L \mid L \text{ is a language decide by an } \mathcal{O}(t(n)) \text{ NTM} \}$

Corollary: $\mathbf{NP} = \bigcup_k \mathbf{NTIME}(n^k)$.

Proof: obvious from previous considerations

Proof, continuation

Assume that A is decided by a polynomial time NTM N and construct a polynomial verifier V_N that decides A .

$V_N =$ "On input $\langle w, c \rangle$ where w and c are strings:

1. Simulates N on input w , using each symbol of c as a description of the nondeterministic choice to make at each step (see NTM computation simulation).
2. If this branch of N 's computation accepts, *accept*, otherwise *reject*

Undirected graphs and cliques

- A clique in an undirected graph is a subgraph wherein every two nodes are connected by an edge.
- A k -clique is a clique containing k nodes. Figure 2 illustrates a graph with a 5-clique.

Clique problem:

CLIQUE = $\{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}$

Observations

- The class **NP** is insensitive to the choice of reasonable nondeterministic computational model because all such models are polynomial equivalent
- When describing and analyzing nondeterministic polynomial time algorithm we follow the notational conventions set up for deterministic polynomial time algorithms
- Each stage of a nondeterministic polynomial time algorithm must have an obvious implementation in nondeterministic polynomial time on a reasonable nondeterministic computational model.
- Algorithm analysis shows that every branch uses at most polynomially many stages

Theorem 7.24

CLIQUE \in NP

Proof idea: the clique is the certificate.

The following is a verifier V for CLIQUE:

$V =$ "On input $\langle \langle G, k \rangle, c \rangle$:

1. Test whether c is a set of k nodes in G .
2. Test whether G contains all edges connecting nodes in c .
3. If both pass, *accept*, otherwise *reject*."

Example graph with a clique

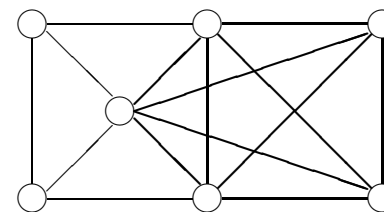


Figure 2: A graph with a 5-clique

SUBSET-SUM Problem

A collection of k integers x_1, x_2, \dots, x_k and a target number t are given. We want to determine whether this collection contains a subcollection that adds up to t

$SUBSET-SUM = \{ \langle S, t \rangle \mid S = \{x_1, x_2, \dots, x_k\} \text{ and for some } C = \{y_1, y_2, \dots, y_l\} \subseteq S \text{ we have } \sum_{i=1}^{i=l} y_i = t \}$

Example: $\langle \{4, 11, 16, 21, 27\}, 25 \rangle \in SUBSET-SUM$ because $4 + 21 = 25$.

Note: $\{x_1, x_2, \dots, x_k\}$ and $\{y_1, y_2, \dots, y_l\}$ may be multisets.

Alternative proof

If one prefers to think in terms of polynomial time NTM one can construct the following NTM N that decides CLIQUE:

$N =$ "On input $\langle G, k \rangle$ where G is undirected graph:

1. Nondeterministically select a subset C of k nodes of G .
2. Test whether all nodes in c are connected and whether G contains all edges connecting nodes in C .
3. If yes, *accept*, otherwise, *rejects*."

Alternative proof

We can also prove this theorem given the NTM N that decides $SUBSET-SUM$:

$N =$ "On input $\langle S, t \rangle$:

1. Nondeterministically select a subset c of numbers in S
2. Test whether c is a collection of numbers that sum to t
3. If the test passes, *accept*, otherwise, *reject*."

Theorem 7.25

$SUBSET-SUM \in NP$.

Proof idea: the subset is the certificate

The following is a verifier V for $SUBSET-SUM$:

$V =$ "On input $\langle \langle S, t \rangle, c \rangle$:

1. Test whether C is collection of numbers that sum to t
2. Test whether S contains all the numbers in C .
3. If both pass, *accept*, otherwise, *reject*."

The P versus NP question

P: the class of languages for which membership can be *decided* quickly

NP: the class of languages for which membership can be *verified* quickly

Question: is **P=NP**?

This is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics.

Observations

- It seems that verifying that something is *not* present is more difficult than that it is present.
- Hence, \overline{CLIQUE} and $\overline{SUBSET - SUM}$ are not obviously members of **NP**.
- A separate complexity class denoted **coNP** contains languages that are complements of languages in **NP** class.
- We don't know whether **NP** is different from **coNP**.

Advance on the P versus NP

- Certain problems in **NP** have their individual complexity related to that of the entire class (Stephen Cook and Leonid Levin, 1970s)
- If a polynomial time algorithm exists for any of these problems, all problems in **NP** class would be polynomial time solvable.
- These problems are called **NP-complete** and the phenomenon of **NP-completeness** is important for both theoretical and practical reasons.

Observations

- If **P** is equal to **NP** then any polynomial verifiable problem would be polynomially decidable.
- Most researchers believe that **P≠NP** because people have invested enormous effort to find polynomial time algorithms for some problems in **NP** without success.
- A proof that **P≠NP** would mean that no fast algorithm exists to replace brute-force search for some problems. This may be beyond scientific reach.
- Best method known for solving problems in **NP** deterministically is based on deterministic simulation of NTM, which is exponential, i.e., $NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k})$. But we don't know whether **NP** is contained in a smaller deterministic time complexity class.

Practical Importance

- The phenomenon of **NP-completeness** may prevent wasting time searching for a non-existent polynomial time algorithm to solve a particular problem.
- Even though we may not have the necessary mathematics to prove that the problem is unsolvable in polynomial time, showing that it is **NP-complete** will suffice, because we believe that $P \neq NP$.

Theoretical Importance

- Research trying to show that $P \neq NP$ may focus on an **NP-complete** problem.
- If any problem in **NP** requires more than polynomial time, an **NP-complete** does.
- Research trying to show that $P=NP$ only need to find a polynomial time algorithm for **one NP-complete** problem.

Cook-Levin Theorem

$SAT \in P$ iff $P = NP$.

Proof idea: Use polynomial time reducibility method.

Example NP-complete problem

- A Boolean formula is an expression involving Boolean variables and operations. For example, $\Phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$ is a Boolean formula.
- A Boolean formula is satisfiable if some assignments of 1 and 0 (true and false) to its variables makes the formula evaluates to 1. For example, the assignment $x = 0, y = 1, z = 0$ makes Φ evaluate to 1.
- The satisfiability problem (SAT) is to test whether a Boolean formula is satisfied, i.e., $SAT = \{\langle \Phi \rangle \mid \Phi \text{ is a satisfiable boolean formula}\}$.

Polynomial time computability

A function $f : \Sigma^* \rightarrow \Sigma^*$ is a polynomial time computable function if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input w

Comments on reducibility

- When problem A reduces to problem B a solution to B can be used to solve A .
- Polynomial time reducibility is a reducibility method that takes the efficiency of computation into account.
- When problem A is efficiently reducible to problem B , an efficient solution to B can be used to solve A efficiently.

Observations

- A polynomial time reduction of A to B provides an efficient way of converting membership testing in A to membership testing in B
- To test whether $w \in A$ we use the reduction f to map w into $f(w)$ and test $f(w) \in B$.
- If language A is polynomial reducible to language B which has a polynomial time solution then A has a polynomial time solution

Polynomial time reducibility

A language A is **polynomial time mapping reducible**, or **polynomial time reducible** to a language B , written $A \leq_P B$, if a polynomial computable function $f : \Sigma^* \rightarrow \Sigma^*$ exists where for every $w \in \Sigma^*$, $w \in A \iff f(w) \in B$

Note: f is called the **polynomial time reduction** of A to B .

Conjunctive Normal Form

- **Literal:** a Boolean variable or a negated Boolean variable.
Examples: x and \bar{x} are literals
- **Clause:** several literals connected with \vee .
Example: $(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4)$ is a clause
- **Conjunctive normal form:** a Boolean formula which is a conjunction of several clauses (i.e., connected with \wedge).
Example: $(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_5 \vee x_6) \wedge (x_3 \vee \bar{x}_6)$ is a cnf-formula
- **3CNF-formula:** a CNF-formula where each clause has three literals.
Example: $(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_5) \wedge (x_3 \vee x_6 \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$

Theorem 7.31

If $A \leq_P B$ and $B \in \mathbf{P}$ then $A \in \mathbf{P}$

Proof: let M be a polynomial time algorithm deciding B and f be a polynomial time reduction from A to B . Then the algorithm N is a polynomial time decider of A :

$N =$ "On Input w :

1. Compute $f(w)$
2. Run M on input $f(w)$ and output whatever M outputs."

Note Since composition of two polynomial is polynomial, N is a polynomial time algorithm.

Proof idea:

- The polynomial time reduction f that we demonstrate from $3SAT$ to $CLIQUE$ converts formulas to graphs.
- In the constructed graphs, cliques of a specialized size correspond to satisfying assignments of the formula.
- Structures within the graph are designed to mimic the behavior of the variables and clauses.

3SAT Problem

$3SAT$ is a special case of the SAT problem where the Boolean formula is a *3cnf – formula*. That is,

$$3SAT = \{ \langle \Phi \rangle \mid \Phi \text{ is a satisfiable 3cnf-formula} \}$$

Theorem 7.32 $3SAT$ problem is polynomial time reducible to $CLIQUE$

Nodes of G

- Nodes in G are organized in k groups of three nodes each called the triplets t_1, t_2, \dots, t_k .
- Each triplet correspond to one of the clauses in Φ and each node in the triplet corresponds to a literal in the associated clause.
- Label each node of G with its corresponding literal in Φ

Proof

Let Φ be a formula with k clauses:

$\Phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \dots \wedge (a_k \vee b_k \vee c_k)$ The reduction f generates the string $\langle G, k \rangle$ where G is a undirected graph.

Example 3SAT reduction to graphs

3SAT formula $\Phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$ in transformed in the graph in Figure 3.

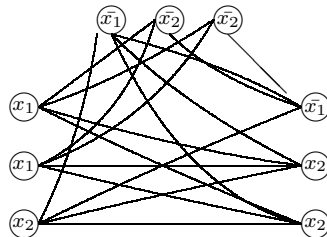


Figure 3:

Edges of G

Edges of G connect all but two types of pairs of nodes in G :

- No edge is present between the nodes of the same triplet
- No edge is present between two nodes with contradictory labels, such as x_2 and \bar{x}_2 .

Proof, continuation

A k -clique in G implies Φ is satisfiable. Suppose G has a k -clique:

- No two of the clique nodes can occur in the same triplet because nodes in the same triplet are not connected. Therefore each the k -triplets contains contains one of the k -clique nodes.
- Assign truth value to the variables of Φ so that each literal labeling a clique node is made true. This is possible because two nodes with contradictory labels are not connected.
- This assignment satisfies the formula Φ . Since each node corresponds to a clause that has a true value in it the clause is true; since classes are connected by \wedge the formula is true.

Why this construction works?

Φ is satisfiable iff G has a k -clique.

Proof: $3SAT \Rightarrow CLIQUE$. Suppose that Φ has a satisfying assignment.

- At least one literal is true in every clause (required by \vee)
- In each triplet of G we select one node corresponding to a true literal in the satisfying assignment. If more literals are true in some clause we select the true literal arbitrarily.
- The nodes just selected form a k -clique; number of nodes is k (there are k clauses in Φ) and each pair of selected nodes are joined, by construction.
- Selected node are not from the same triplet, by construction; they could not have contradictory labels because otherwise the associated labels would be both true in the satisfying assignment. Hence G contains a k -clique.

Definition of NP-Completeness

A language B is **NP**-complete if it satisfies two conditions:

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

Conclusions

- If **CLIQUE** is solvable in polynomial time, so is $3SAT$.
- Polynomial time reducibility allows us to link these two very different problems.
- Similar link may be made among other problems.

Theorem 7.36

If B is NP -complete and $B \leq_P C$ for $C \in NP$ then C is NP -complete.

Proof: Since $C \in NP$ we only need to show that every $A \in NP$ is polynomial time reducible to C .

- Since B is NP -complete A is polynomial time reducible to B
- Since B is polynomial time reducible to C , A is polynomial time reducible to C by first reducing it to B and then reducing its image to C .
- Hence, every language in NP is polynomial time reducible to C

Theorem 7.35

If B is NP -complete and $B \in P$ then $P = NP$.

Proof: this follows directly from the definition of NP -completeness.

Proof

1. $SAT \in NP$: a nondeterministic polynomial time machine can guess an assignment to the variables of a given formula Φ and accept if assignment satisfies Φ .
2. Let $A \in NP$: show that A is polynomial time reducible to SAT . For a NTM N that decides A in n^k time for some constant k , construct a formula Φ that simulates N .
3. Construction of Φ : based on organizing the computation performed by N into an $n^k \times n^k$ tableau as seen in Figure 4

The Cook-Levin Theorem

Theorem 7.37 SAT is NP -complete

Proof idea:

- Show that $SAT \in NP$, which is easy
- Show that any language $A \in NP$ is polynomial time reducible to SAT
- The reduction of A takes a string w and produces a Boolean formula Φ that simulates the NTM N that decides A operating on w .
- If N accepts, Φ has a satisfying assignment that corresponds to that computation; if N doesn't accept, no assignment satisfies Φ . Hence, $w \in A$ iff Φ is satisfiable.

Tableau(N,w)

- Every accepting tableau for N and w correspond to an accepting computation branch of N on w .
- Problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N and w exists.

Configurations tableau of N

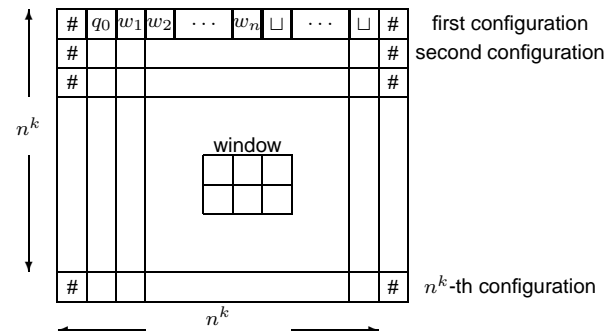


Figure 4: A tableau is an $n^k \times n^k$ table of configurations

- Observations:** (1) Each configuration starts and ends with a # symbol.
 (2) A tableau is **accepting** if any of its rows is an accepting configuration.

Assignment-tableau correspondence

The assignment turns on exactly one variable for each cell, using the following constructs:

1. at least one variable that is associated with a cell is on, by: $\bigvee_{s \in C} x_{i,j,s}$
2. no more than one symbol in each cell, by: $\bigwedge_{s,t \in C, s \neq t} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}})$

Thus, any satisfying assignment specifies one symbol in each cell by:

$$\Phi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} [(\bigvee_{s \in C} x_{i,j,s}) \wedge (\bigwedge_{s \neq t \in C} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}))]$$

Polynomial time $f : A \rightarrow SAT$

- On input w , f produces Φ_w
- **Variables of Φ_w :** Let $N = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, and $C = Q \cup \Gamma \cup \{\#\}$. For each $1 \leq i, j \leq n^k \wedge s \in C$ we have a variable $x_{i,j,s}$ in Φ_w .
- **Cells:** of the $(n^k)^2$ entries of a tableau is called a *cell*. $\forall s \in C$ $x_{i,j,s} = 1$ if $cell[i, j] = s$.
- Formula Φ_w is $\Phi_w = \Phi_{cell} \wedge \Phi_{start} \wedge \Phi_{move} \wedge \Phi_{accept}$.

Legal window

A 2×3 window of cells is legal if that window does not violate the actions specified by N 's transition function. To explain, consider the transitions:

$$\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}.$$

Examples of legal windows for this machine are in Figure 5:

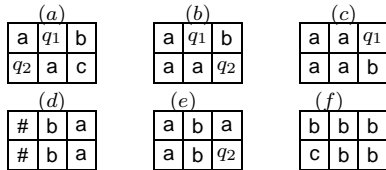


Figure 5: Examples of legal windows

An accepting tableau

is specified by Φ_{start} , Φ_{move} , Φ_{accept}

- Φ_{start} ensures that the first row of the tableau is the starting configuration of N on w by the equality:

$$\Phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \dots \wedge x_{1,n+3,w_n} \wedge \dots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}$$
- Φ_{accept} guarantees that an accepting configuration occurs in the tableau by placing q_a in one of the cells by: $\Phi_{accept} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_a}$
- Φ_{move} guarantees that each row of the tableau correspond to a configuration that legally follows the preceding row's configuration according the N 's transition rules.

Illegal windows

Figure 6 shows illegal windows of N . tiny

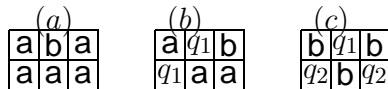


Figure 6: Examples of illegal windows

- (a) is illegal because the central symbol on the top can't be changed because it has no adjacent state
- (b) is illegal because the transition specifies that b get changed to c not to a
- (c) is illegal because two states appear in the bottom row.

Comments:

- Windows (a) and (b) are legal because the transition allows N to move this way.
- Window (c) could be either illegal or illegal because q_1 appears to the right side of the top row and we don't know what symbol is the head over.
- Window (d) is legal because top and bottom are identical, what could happen if the head weren't adjacent to the location of the window.
- Window (e) is legal because state q_1 reading a b might have been immediately to the right of the top row and would have moved to the left
- Window (f) is legal because state q_1 might have been immediately to the left of the top row changing b to c and moving left.

Construction of Φ_{move}

Φ_{move} stipulates that all windows in the table are legal.

- Each windows contain six cells which may be set in a fixed number of ways to yield a legal window. Φ_{move} says that the setting of those six cells is done this way by:

$$\Phi_{move} = \bigwedge_{1 \leq i, j \leq n^k} (\text{window}[i, j] \text{ is legal})$$

- Replace $\text{window}[i, j] \text{ is legal}$ with the following formula where $a_1, a_2, a_3, a_4, a_5, a_6$ are the contents of the six cells:

$$\bigvee_{a_1 \dots a_6} \text{legal}(x_{i, j-1, a_1} \wedge x_{i, j, a_2} \wedge x_{i, j+1, a_3} \wedge x_{i+1, j-1, a_4} \wedge x_{i+1, j, a_5} \wedge x_{i+1, j+1, a_6})$$

Claim

If the top row of the tableau is the start configuration and every every window is legal then each row is a configuration that legally follows the configuration represented by the preceding row.

Proof: show the claim for any two adjacent configurations.

3SAT is NP-Complete

Proof: provide a polynomial time reduction from SAT to $3SAT$.

- Transform first Φ_{SAT} into cnf
- Represent each component of the cnf $(a_1 \vee a_2 \vee \dots \vee a_n)$ by n-2 clauses: $(a_1 \vee a_2 \vee z_1) \wedge (\bar{z}_1 \vee a_3 \vee z_2) \wedge \dots \wedge (z_{n-3} \vee a_{n-1} \vee a_n)$

Complexity of the reduction

- Tableau is $n^k \times n^k$ and thus contains n^{2k} cells; each cell has $|C| = l$ variables associated with it where l depends only on N . Hence total number of variables is $\mathcal{O}(n^{2k})$.
- Estimating the size of four components of Φ : Φ_{cell} is a fixed fragment of Φ so its size is fixed and is $\mathcal{O}(n^{2k})$; Φ_{start} has the size $\mathcal{O}(n^k)$; Φ_{move} and Φ_{accept} have sizes $\mathcal{O}(n^{2k})$. Hence total size of Φ is $\text{mathcal{O}}(n^{2k})$, i.e., size of Φ is polynomial in n .
- Each component of Φ can be produced in polynomial time. Therefore we conclude that we can construct a reduction that produces Φ from N in polynomial time.

This concludes the proof of Cook-Levin Theorem, showing that SAT is **NP-complete**.

Other NP-complete languages

To show that A is NP-complete provide a polynomial time reduction from A to $3SAT$ or to other NP-complete languages.