MTH401: Theory Of Computation

Computational Complexity Theory

Havi Bohra (210429), Sharvani (210960), Vekariya Keval (211158)

Department of Mathematics and Statistics
Indian Institute of Technology Kanpur
Course Instructor: Prof. Mohua Banerjee

April 19, 2025

Contents

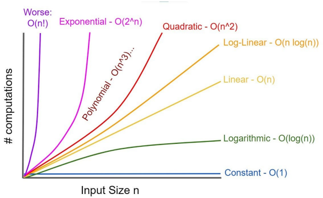
Time Complexity

BIG-O NOTATION

Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$.
We say that $f(n) = O(g(n))$ if there exist positive integers c and n_0 such that for every integer $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

- $g(n)$ is an **upper bound** for $f(n)$
- More precisely, $g(n)$ is an **asymptotic upper bound**, because we are suppressing constant factors



Examples:

- $f_1(n) = 5n^3 + 2n^2 + 22n + 6 \Rightarrow O(n^3)$
- $f_2(n) = 4n \log n + 3n \Rightarrow O(n \log n)$
- $f_3(n) = 2^n + n^5 \Rightarrow O(2^n)$

ANALYZING ALGORITHMS

TM M_1 for $A = \{0^k 1^k \mid k \geq 0\}$

Input: String w

1. Scan across the tape and reject if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s remain after all the 1s are crossed, or 1s remain after all the 0s are crossed, reject. Otherwise, if neither 0s nor 1s remain, accept.

Time Analysis:

- Stage 1: $O(n)$ for scanning + repositioning head
- Stage 2–3: Up to $n/2$ scans, each $O(n) \to O(n^2)$
- Stage 4: Final scan $\to O(n)$

Total Time:

$$O(n) + O(n^2) + O(n) = O(n^2)$$

TYPES OF TURING MACHINES

We will study the **time complexity** of three types of Turing machines:

Deterministic Turing Machines (DTMs)

One computation path

Nondeterministic Turing Machines (NTMs)

Multiple computation branches

Multi-tape Turing Machines

More efficient with multiple tapes and heads

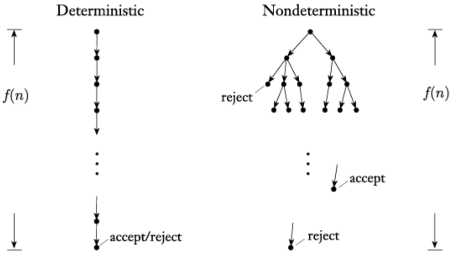
TIME COMPLEXITY OF TURING MACHINES

Let M be a **deterministic TM** that halts on all inputs.

- The time complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps M uses on any input of length n
- **f(n)-Turing Machine:** A Turing machine with time complexity $f(n)$ is called an $f(n)$-Turing machine.

Let N be a **nondeterministic TM** that is a decider.

- The time complexity of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps used *on any branch* of computation for any input of length n



Complexity Relationships between Multitape and Single-tape

Theorem:

Every $t(n)$ time multitape Turing Machine with $t(n) \geq n$,
can be simulated using a $O(t^2(n))$ time single-tape Turing Machine.

Proof:

Notation:

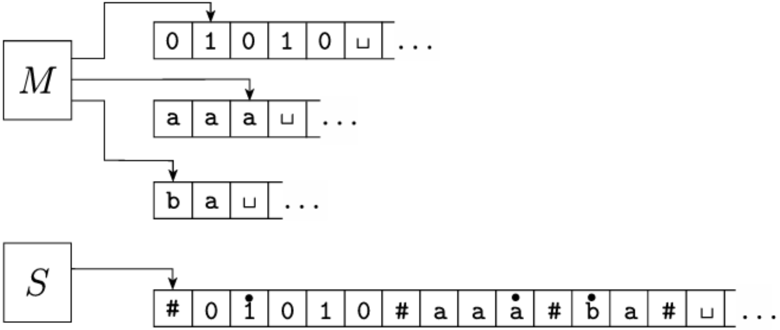
Let $M = (Q, \Sigma, \delta, q_0, h)$ be a k-tape Turing Machine.
Here, Σ denotes the tape alphabet, containing the blank symbol $\#$.
Formally, the transition function is:

$$\delta : Q \times \Sigma^k \to (Q \cup \{h\}) \times (\Sigma \cup \{L, R\})^k$$

That is, on reading the current tape alphabets $a_1, a_2, \ldots, a_k$ on the k tapes, the machine transitions to some other state and, on each tape, either writes a symbol or moves the head left or right

Multi-tape to Single-tape: Design

We construct a single-tape machine S that simulates the execution of M



The alphabet for the single-tape machine is constructed as follows:

- Contents of the k tapes are separated by a delimiter ⊔.
- For each section representing one tape, a dotted symbol indicates the current head position

Multitape to Single-tape: Execution

On input w,

1. S puts its tape into a format representing all the k tapes of M. The formatted tape is:

$$\sqcup \dot{w}_1 w_2 w_3 \ldots w_n \sqcup \dot{\#} \sqcup \dot{\#} \sqcup \dot{\#} \ldots \sqcup$$

2. To simulate a single move, S scans its tape from the first $\#$ to the last $\#$ to determine the symbols under the heads of each tape. Then, it makes a second pass to update the tapes according to the transition function of M
3. If at any point S moves one of the virtual heads from onto a $\sqcup$, this means that M has equivalently moved the head of the respective tape into a previously unread blank portion of the tape. Thus, whenever this happens, we write a $\#$ there and move the contents of S's tape from this $\#$ to the right-end, one cell to the right

Multitape to Single-tape: Time Complexity

- Let the part of a tape that we have already seen be the **"active part"** of the tape
- We will need the following key observation: since we can only move one cell in one step on any tape, the active part of any tape cannot have length greater than $t(n)$. Thus, in order to do one pass of S's tape, it will take at most $O(kt(n)) = O(t(n))$ time
- The first step, where S puts the tape into the starting configuration, takes $O(n)$ time. Afterwards, it takes $O(t(n))$ time for any step of M
- Thus, the execution takes $O(t^2(n))$ time. The total time complexity is thus,

$$O(n + t^2(n)) = O(t^2(n)).$$

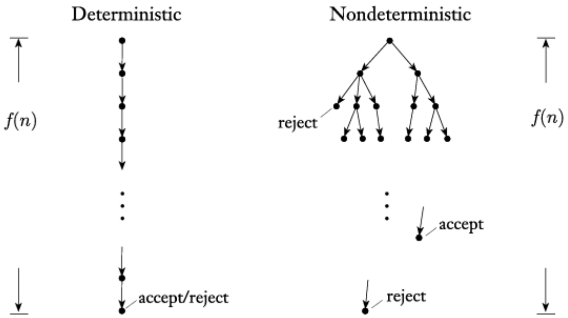
Remark: The $t(n) \geq n$ is a reasonable assumption, because it takes $O(n)$ time to just read the input completely.

Complexity Relationships between NTM and TM

Theorem

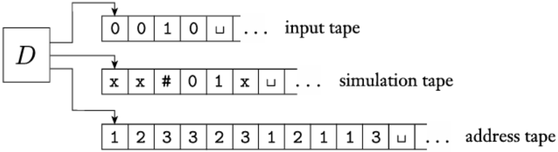
Let $t(n) \geq n$. Then every $t(n)$-NTM can be simulated using a $2^{O(t(n))}$-TM.

Let M be a $t(n)$-NTM. Let b be the number of legal transitions allowed by M.
Then we can visualize the set of all possible computations of M as proceeding along a b-ary tree of depth at most $t(n)$.



Simulating a Non-Deterministic TM with a Deterministic TM

The simulating deterministic TM D has **three tapes**.



By previous theorem, multitape can be converted to a single tape.

- Tape 1 always contains the input string and is never altered.
- Tape 2 maintains a copy of N's tape on some branch of its nondeterministic computation.
- Tape 3 keeps track of D's location in N's nondeterministic computation tree.

Execution Details of the Simulation

Addressing in the Tree:

- Each node in the computation tree has at most b children.
- Every node is addressed using a string over the alphabet $\Gamma_b = \{1, 2, \ldots, b\}$.
- For example, address 231 means: 2nd child of root $\to$ 3rd child $\to$ 1st child.
- The empty string ε represents the root.

Steps of the simulation by D:

1. Initially, tape 1 contains the input w, tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2, and initialize tape 3 with ε.
3. Simulate N using tape 2 for one branch. Use symbols on tape 3 to guide choices.
4. If:
   - No symbol remains or choice is invalid
   - A rejecting configuration is reached
     $\Rightarrow$ Go to step 4.
   - If accepting configuration is reached, **accept**.
5. (Step 4) Replace tape 3's string with next string in lexicographic order and repeat from step 2.

Proof: Simulating NTM with Deterministic TM

- The simulation travels N's nondeterministic computation tree using **breadth-first search**.
- That is, it visits all nodes at depth d before visiting those at depth $d+1$.
- The algorithm starts at the root and travels down to a node for every visit.

Tree Structure:

- Each branch of N has depth $\leq t(n)$.
- Each node has at most b children ($b =$ max number of legal choices in N).
- Total number of leaves: $\leq b^{t(n)}$.
- Total number of nodes: $\leq 2b^{t(n)} = O(b^{t(n)})$.

Simulation:

- Time to travel from the root to a node: $O(t(n))$.
- Total nodes to simulate: $O(b^{t(n)})$.
- $\Rightarrow$ Total simulation time: $O(t(n) \cdot b^{t(n)}) = 2^{O(t(n))}$.

Proof Continued: Single-Tape Simulation

Final Step:

- The TM D uses three tapes to perform the simulation.
- By previous theorem, converting a multi-tape TM to a single-tape TM at most squares the running time.

$$\text{Time on single-tape TM} = \left(2^{O(t(n))}\right)^2 = 2^{O(2t(n))} = 2^{O(t(n))}$$

Hence, the theorem is proved!

Classes P & NP

The Class P

Definition:

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine.
In other words,

$$P = \bigcup_k \mathrm{TIME}(n^k)$$

The class P plays a central role in our theory and is important because:

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine.
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

The Class NP

Definition:

NP is the class of languages that are decidable in polynomial time on a non-deterministic single-tape Turing machine.
In other words,

$$NP = \bigcup_k \mathrm{NTIME}(n^k)$$

As we shall see, **NP** contains several algorithmically significant problems and admits a distinct algorithmic characterization.

Examples of Problems in P

PATH:

- Given a graph $G = (V, E)$ and nodes S, T, does a path from S to T exist?

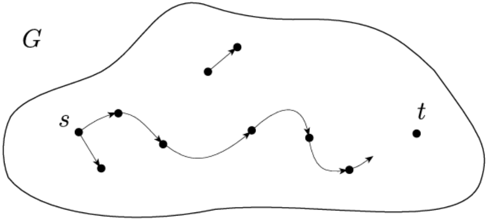


Figure: PATH PROBLEM

Proof of PATH $\in$ P:

- M = "On input $\langle$ G, s, t $\rangle$:
  1. Mark node s
  2. Repeat until no new nodes marked:
     - For each edge (a, b), if a is marked and b isn't, mark b
  3. Accept if t is marked; reject otherwise
- Stage 3 runs at most m times (m = nodes), giving total stages $\leq 1 + 1 + m$, hence polynomial time decidability.

HAMPATH

Hamiltonian Path:

- A Hamiltonian path visits each node exactly once.
- HAMPATH = $\langle$ G, s, t $\rangle$ — G has a Hamiltonian path from s to t

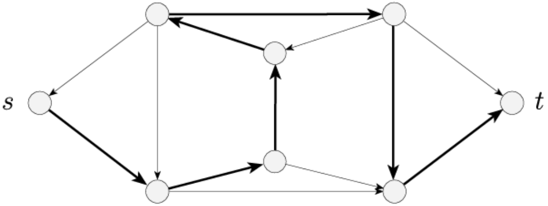


Figure: HAMPATH PROBLEM

HAMPATH $\in$ NP

- A non-deterministic TM N decides HAMPATH as follows:
  1. Guess a permutation $p_1, \ldots, p_n$ of vertices
  2. Check for repetition; reject if found
  3. Ensure $p_1 = s$, $p_n = t$
  4. Verify every consecutive pair $(p_i, p_{i+1}) \in E$
- All steps run in polynomial time

Verifiers

Verifier:

- A verifier for a language A is an algorithm V such that $A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some } c\}$
- A language is polynomially verifiable if such a V runs in polynomial time in $|w|$

Remark:

The string c is called a certificate for w. If A is a polynomial time verifier, then there must be a certificate with length polynomial in |w|.

The Class NP

Definition:

NP is the class of languages that have polynomial-time verifiers.

Theorem:

A language is in NP if and only if it is decided by some NTM.

Proof ($\Rightarrow$):

Assume $A \in$ NP, and let V be the polynomial-time verifier for A, running in time n^k for some constant k. Construct an NTM N to decide A as follows:

- On input w of length n:
  1. Nondeterministically guess a string c of length at most n^k.
  2. Run V on input $\langle w, c \rangle$.
  3. Accept if V accepts; reject otherwise.

The Class NP (contd.)

Theorem:

A language is in NP if and only if it is decided by some NTM.

Proof ($\Leftarrow$):

Assume A is decided by a NTM N that runs in polynomial time.
Construct a polynomial-time verifier V for A as follows:

- On input $\langle w, c \rangle$, where w is a string and c is a certificate (sequence of nondeterministic choices):
  1. Simulate N on input w, using each symbol of c to direct the nondeterministic choices.
  2. Accept if this computation path leads to acceptance; otherwise, reject.

Since N runs in polynomial time and the simulation is deterministic using c, the verifier V runs in polynomial time.

NP Completeness

Vertex Cover

- $S \subseteq V$ is a vertex cover if every edge in G touches some vertex in S
- VERTEX COVER $= \{ \langle \text{G, k} \rangle : \text{G has a vertex cover of size} \leq k \}$

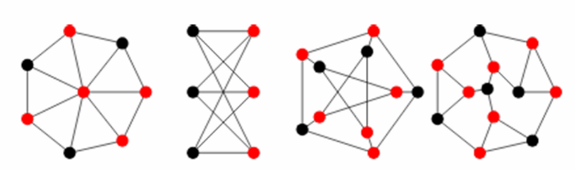


Figure: Vertex Cover examples

It can be checked that Vertex-Cover $\in$ NP as it is verifiable in polynomial time.

Independent Set

- $S \subseteq V$ is an independent set if no two vertices in S are connected
- INDEPENDENT SET $= \{ \langle \text{G, k} \rangle : \text{G has an independent set of size} \geq k \}$

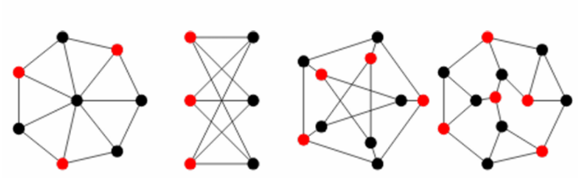


Figure: Independent Set examples

It can be checked that Independent-Set $\in$ NP as it is verifiable in polynomial time.

Theorem:

$X \subseteq V$ is a vertex cover of $G \iff V \setminus X$ is an independent set

Proof Sketch:

- ($\Rightarrow$) If X is a vertex cover, no edge can exist between nodes in $V \setminus X$
- ($\Leftarrow$) If $V \setminus X$ is an independent set, then X must touch all edges

Computational Consequence

We can reduce VERTEX COVER to INDEPENDENT SET:
$\langle G, k\rangle \in \text{VERTEX COVER} \iff \langle G, n-k\rangle \in \text{INDEPENDENT SET}$

Polynomial Time Reducibility

Definition:

A function $f : \Sigma^* \rightarrow \Sigma^*$ is polynomial-time computable if a TM computes $f(w)$ in polynomial time

Reduction:

$A \leq_P B$ if there exists a polynomial-time computable f such that:

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

NP Hard and NP Complete

NP Hard:

A language L is NP Hard if for all $J \in \text{NP}$, $J \leq_P L$

NP Complete:

A language L is NP Complete if it is both NP Hard and is in NP

Importance of NP Complete Problems

Proving P $=$ NP

Solving an NP-complete problem in polynomial time implies P $=$ NP

Proving P $\neq$ NP

Showing no polynomial-time algorithm exists for any NP-complete problem proves P $\neq$ NP

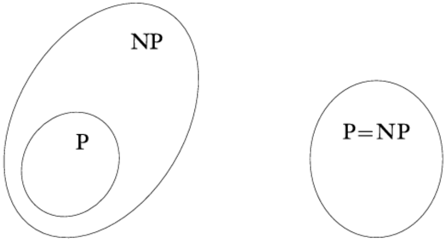


Figure: Only one of these possibilities is correct

The Cook-Levin Theorem

Does any NP-complete problem exists ?

Why such a question arises?
Because:

- Every problem, known as well as unknown, from class NP has to be reducible to this problem.
- Such a problem would indeed be the hardest of all problems in NP.

SATISFIABILITY Problem (SAT)

- **Boolean Variables**: variables that can take on the values TRUE(1) and FALSE(0)
- **Boolean operations**: AND($\wedge$), OR($\vee$), and NOT($\neg$)
- **Boolean Formula**: expression involving Boolean variables and operations.

  For example: $\varphi = (\neg x \wedge y) \vee (x \wedge \neg z)$

Satisfiable: A Boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1

Satisfiability Problem (SAT):

Given a Boolean formula F, is it satisfiable?

$$\text{SAT} = \{<\varphi>| \ \varphi \text{ is a satisfiable Boolean formula}\}$$

The Cook-Levin Theorem

Theorem:

SAT is NP-complete

Proof Sketch:

- Showing that SAT is in NP is easy, here it is:
- A nondeterministic polynomial Turing machine can guess an assignment to a given formula φ and accept if the assignment satisfies φ.
- Difficult part of the proof is showing that any language in NP is polynomial time reducible to SAT

Proving NP-Completeness

Theorem:

Let $\mathcal{L}_1$ be NP-Complete and $\mathcal{L}_2 \in NP$. If $\mathcal{L}_1 \leq_P \mathcal{L}_2$, then $\mathcal{L}_2$ is NP-Complete.

Proof:

- Let L be any language in NP.
- Since L_1 is NP-Complete, $L \leq_P L_1$ via some reduction τ_1.
- Given $L_1 \leq_P L_2$ via some reduction τ_2,
- By transitivity: $L \leq_P L_2$ via $\tau = \tau_2 \circ \tau_1$.
- Hence, L_2 is NP-Complete.

We will show **SAT** is NP-Complete, by proving **Bounded-Tiling** problem is NP-Complete and **Bounded-Tiling** $\leq_P$ **SAT**

Bounded Tiling Problem

Tiling System: A system $D = (D, H, V)$, where:

- D: finite set of tiles
- $H \subseteq D^2$: valid horizontal pairs of tiles
- $V \subseteq D^2$: valid vertical pairs of tiles

Bounded Tiling Problem

Given a tiling system $D = (D, H, V)$, an integer s, a function for assignment of first row of the tiling $f_0 : \{0, \dots, s-1\} \to D$

Is there a function $f : \{0, \dots, s-1\}^2 \to D$ such that:

- $f(m, 0) = f_0(m)$ for all $m < s$
- $(f(m, n), f(m+1, n)) \in H$ for all $m < s-1, n < s$
- $(f(m, n), f(m, n+1)) \in V$ for all $m < s, n < s-1$

Bounded Tiling is NP-Complete

Theorem:

The Bounded Tiling Problem is NP-Complete.

Proof Sketch:

1. **Bounded Tiling** $\in$ NP:
   A non-deterministic TM can guess a tiling and verify constraints in polynomial time.
2. **For any language** $L \in \mathrm{NP}$, $L \leq_P$ Bounded Tiling:
   That is, we define a polynomial-time computable reduction τ such that:

$$x \in L \iff \tau(x) = (D = (D, H, V), s, f_0) \text{ has a valid } s \times s \text{ tiling}$$

The Reduction τ

Since $L \in$ NP, there exists a non-deterministic Turing Machine

$$M = (K, \Sigma, \delta, s)$$

such that:

- All computations on input x halt within $p(|x|)$ steps (for some polynomial p)
- $x \in L \iff M$ has an accepting computation on input x

Define the reduction $\tau(x) = (\mathcal{D} = (D, H, V), s, f_0)$ as follows:

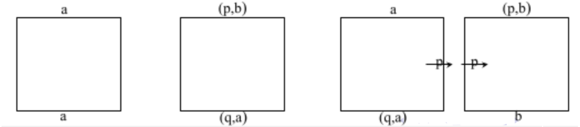
1. Set $s = p(|x|) + 2$
2. Construct the tiling system D using the Turing Machine
   - The tiling system is arranged so that if a tiling is possible, then the markings on the horizontal edges between the n^{th} and$(n+1)^{th}$ rows of tiles, read off the configuration of M after n steps of its computation.
   - Successive configurations appear one above the next.

Goal: Accepting computation $\Rightarrow$ valid tiling exists

The Reduction τ

3. Define tiles in D to encode transitions of the TM M:
   - For each $a \in \Sigma$: a tile that passes symbol a upward (unchanged)
   - For $a, b \in \Sigma$, $p, q \in K$ such that $\delta(q, a) = (p, b)$:
     A tile that changes $a \to b$, updates state $q \to p$, and marks head
   - For moves:
     - $\delta(q, a) = (p, \to)$: tile encodes right move and state change
     - $\delta(q, a) = (p, \leftarrow)$: tile encodes left move and state change

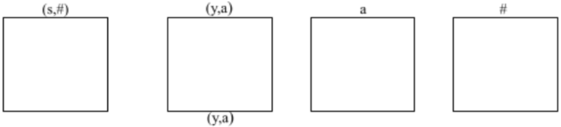
   These tiles ensure vertical compatibility reflects valid TM execution.



The Reduction τ

- Tiles for the initial configuration:
  - ★ Tile with upper marking $\triangleright$ (start of tape)
  - ★ Tile with upper marking $(s, \#)$: TM starts in state s scanning blank symbol
  - ★ Tiles with upper marking $\#$ to pad rest of the row
- Halting behavior: (y: accepting state, n: non-accepting state)
  - ★ For each $a \in \Sigma$, define a tile with both upper and lower marking (y, a): accepts
  - ★ No tile with lower marking (n, a): rejects

These enforce that a tiling is only possible for accepting computations.



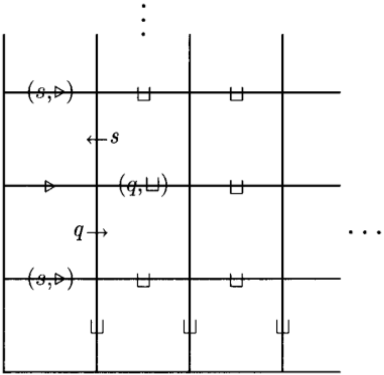
The Reduction τ

4. The function $f_0 : \{0, \ldots, s-1\} \to D$ encodes the initial configuration of M on input x:
   - $f_0(0)$: tile with upper marking $\triangleright$ (start-of-tape symbol)
   - $f_0(1)$: tile with upper marking $(s, \#)$, indicating state s scanning blank
   - $f_0(i+1)$: tile with upper marking x_i, for $i = 0, 1, \ldots, |x|$
   - $f_0(i)$: tile with upper marking $\#$, for $i > |x| + 1$

   This row forms the base from which TM simulation proceeds upward in the tiling.
5. We define the sets H and V such that two tiles can be adjacent iff the markings on their touching edges match:
   - $(d, d') \in H$ the **right edge** of tile d matches the **left edge** of tile d'
   - $(d, d') \in V$ the **top edge** of tile d matches the **bottom edge** of tile d'

   These constraints ensure:
   - Configurations are locally consistent left-to-right (row-wise)
   - Transitions between configurations follow TM behavior (row-to-row)

Example Tiling System



This machine simply oscillates its head from left to right and back again, never moving beyond the first tape square.

Bounded-Tiling problem is NP-Complete

Theorem:

$x \in L \iff \exists\ s \times s$ tiling f extending f_0

Proof:

- $(\Leftarrow)$ Since $p(|x|) = s - 2$, the upper markings of the$(s-2)^{th}$ row must contain a halting state (y or n)
  - But since the $(s-1)^{th}$ row exists and there is no tile with lower markings (n, a) for any $a \in \Sigma$ there must be y in the upper markings of the $(s-2)^{th}$ row
  - So the computation must have halted in an accepting state $x \in L$
- $(\Rightarrow)$ If $x \in L$, then TM M has an accepting computation.
  - We simulate this computation using a tiling consistent with the TM's transitions.
  - Hence, a valid $s \times s$ tiling exists extending f_0

SATISFIABILITY is NP-Complete

We now show that **Bounded Tiling** $\leq_P$ **SATISFIABILITY**.

$$\tau(\mathcal{D} = (D, H, V), s, f_0) \longmapsto \text{Boolean formula } \varphi$$

$$\exists\ s \times s \text{ tiling } f \text{ extending } f_0 \iff \varphi \in \text{SATISFIABILITY}$$

Construction:

- Introduce Boolean variables $x_{m,n,d}$ for all $0 \leq m, n < s$, $d \in D$
- Interpretation: $x_{m,n,d} = \top \iff f(m, n) = d$

We now encode constraints as clauses to ensure f is a valid tiling.

SATISFIABILITY is NP-Complete

We now describe the clauses that ensure f is a legal $s \times s$ tiling.

1. **At least one tile per cell:**

$$(x_{m,n,d_1} \vee x_{m,n,d_2} \vee \cdots \vee x_{m,n,d_k}) \quad \forall m, n < s$$

2. **At most one tile per cell:**

$$(\neg x_{m,n,d} \vee \neg x_{m,n,d'}) \quad \forall m, n < s, \; d \neq d'$$

3. **Initial row constraint:**

$$x_{i,0,f_0(i)} \quad \forall i < s$$

4. **Horizontal compatibility:**

$$(\neg x_{m,n,d} \vee \neg x_{m+1,n,d'}) \quad \forall m < s-1, \; n < s, \; (d, d') \notin H$$

5. **Vertical compatibility:**

$$(\neg x_{m,n,d} \vee \neg x_{m,n+1,d'}) \quad \forall m < s, \; n < s-1, \; (d, d') \notin V$$

These clauses form the Boolean formula $\varphi = \tau(D, s, f_0)$.

SATISFIABILITY is NP-Complete

Equivalence:

$$\exists\ s \times s \text{ tiling } f \text{ extending } f_0 \iff \tau(D, s, f_0) \in \text{SATISFIABILITY}$$

- **(⇐)** Suppose $\tau(D, s, f_0)$ is satisfiable with assignment T.
  Then for each position (m, n), exactly one variable $x_{m,n,d}$ is true (from clauses (1-2)).
  Define:
  $$f(m, n) = d \iff x_{m,n,d} = \top$$
  This gives a complete tiling. The remaining clauses (3–5) guarantee:
  - Initial row matches f_0
  - Horizontal and vertical compatibility are satisfied
- **(⇒)** Given a valid tiling f, assign $x_{m,n,d} = \top \iff f(m, n) = d$.
  This assignment satisfies all clauses in $\tau(D, s, f_0)$.

□

References

[1] R. M. Karp. *Reducibility Among Combinatorial Problems*.
In: *Complexity of Computer Computations*, 1972, pp. 85–103.

[2] H. R. Lewis, C. H. Papadimitriou. *Elements of the Theory of Computation*.
Pearson Education India, 2015. ISBN: 9789332549890.

[3] M. Sipser. *Introduction to the Theory of Computation*.
PWS Publishing, 1997. ISBN: 0-534-95097-3.

Thank You!

Questions?