The Turing Machine

Algorithms and Complexity Theory

Matei Popovici[1]

[1]POLITEHNICA University of Bucharest
Computer Science and Engineering Department, Bucharest, Romania

December 6, 2012

Preamble

There exist problems which ***cannot be solved efficiently*** *?*

*There exist problems which **cannot be solved efficiently** ?*

- How do we formally express that a problem is **solvable** ?

*There exist problems which **cannot be solved efficiently** ?*

- How do we formally express that a problem is **solvable** ?
- Is it the case that, *finding the correct answer* is harder than *checking if a **given answer is correct*** ?

*There exist problems which **cannot be solved efficiently** ?*

- How do we formally express that a problem is **solvable** ?
- Is it the case that, *finding the correct answer* is harder than *checking if a **given answer is correct*** ?
- How do we express that a problem is **harder** than another problem ?

*There exist problems which **cannot be solved efficiently** ?*

- How do we formally express that a problem is **solvable** ?
- Is it the case that, *finding the correct answer* is harder than *checking if a **given answer is correct*** ?
- How do we express that a problem is **harder** than another problem ?
- Is it possible to construct a **hierarchy** of problems, based on hardness ?

How do we formally express that a problem is solvable ?

Part I: What is a problem ?

*Given a list of **acquaintances** and a list of all **pairs** among them who get along, find the **largest set of acquaintances** you can invite to a dinner party such that **every two invitees get along** with one another.* [1]

*Given a list of **acquaintances** and a list of all **pairs** among them who get along, find the **largest set of acquaintances** you can invite to a dinner party such that **every two invitees get along** with one another.* [1]

Let $G = (V, E)$ be an undirected graph.

*Given a list of **acquaintances** and a list of all **pairs** among them who get along, find the **largest set of acquaintances** you can invite to a dinner party such that **every two invitees get along** with one another.* [1]

Let $G = (V, E)$ be an undirected graph. What is the **maximal** set $C \subseteq V$

*Given a list of **acquaintances** and a list of all **pairs** among them who get along, find the **largest set of acquaintances** you can invite to a dinner party such that **every two invitees get along** with one another.* [1]

Let $G = (V, E)$ be an undirected graph. What is the **maximal** set $C \subseteq V$ such that $\forall x, y \in C$, $x \neq y$, $(x, y) \in E$?

*Given a list of **acquaintances** and a list of all **pairs** among them who get along, find the **largest set of acquaintances** you can invite to a dinner party such that **every two invitees get along** with one another.* [1]

Let $G = (V, E)$ be an undirected graph. What is the **maximal** set $C \subseteq V$ such that $\forall x, y \in C$, $x \neq y$, $(x, y) \in E$? (**MAX-CLIQUE**)

*Given a list of **acquaintances** and a list of all **pairs** among them who get along, find the **largest set of acquaintances** you can invite to a dinner party such that **every two invitees get along** with one another.* [1]

Let $G = (V, E)$ be an undirected graph. What is the **maximal** set $C \subseteq V$ such that $\forall x, y \in C,\ x \neq y,\ (x, y) \in E$? (**MAX-CLIQUE**)

Example: $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)\}$

*Given a list of **acquaintances** and a list of all **pairs** among them who get along, find the **largest set of acquaintances** you can invite to a dinner party such that **every two invitees get along** with one another.* [1]

Problem: Let $G = (V, E)$ be an undirected graph. What is the **maximal** set $C \subseteq V$ such that $\forall x, y \in C$, $x \neq y$, $(x, y) \in E$? (**MAX-CLIQUE**)

Example: $V = \{1, 2, 3, 4, 5\}$ and
$E = \{(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)\}$

*Given a list of **acquaintances** and a list of all **pairs** among them who get along, find the **largest set of acquaintances** you can invite to a dinner party such that **every two invitees get along** with one another.* [1]

Problem: Let $G = (V, E)$ be an undirected graph. What is the **maximal** set $C \subseteq V$ such that $\forall x, y \in C,\ x \neq y,\ (x, y) \in E$? (**MAX-CLIQUE**)

Problem instance: *Example:* $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)\}$

Definition (Problem instance [2])

A **problem instance** is a *mathematical object* of which we ask a *question* and expect an *answer*.

Definition (Problem instance [2])

A **problem instance** is a *mathematical object* of which we ask a *question* and expect an *answer*.

Definition (Problem)

A problem $P : I \rightarrow O$ is a *question* which maps an *answer* from O to each problem instance from I.

Definition (Problem instance [2])

A **problem instance** is a *mathematical object* of which we ask a *question* and expect an *answer*.

Definition (Problem)

A problem $P : I \rightarrow O$ is a *question* which maps an *answer* from O to each problem instance from I.

Definition (Decision problem)

A a problem which requires a **yes**/**no** answer ($P : I \rightarrow \{0, 1\}$).

Definition (Problem instance [2])

A **problem instance** is a *mathematical object* of which we ask a *question* and expect an *answer*.

Definition (Problem)

A problem $P : I \rightarrow O$ is a *question* which maps an *answer* from O to each problem instance from I.

Definition (Decision problem)

A a problem which requires a **yes**/**no** answer ($P : I \rightarrow \{0, 1\}$).

In what follows, we will deal with **decision problems only** !

Decision vs optimisation problems

- what is the decision problem that corresponds to **MAX-CLIQUE** ?

Decision vs optimisation problems

- what is the decision problem that corresponds to **MAX-CLIQUE** ?
  - Given a graph G and an integer k, does G have a clique of size at least k ?

Decision vs optimisation problems

- what is the decision problem that corresponds to **MAX-CLIQUE** ?
  - Given a graph G and an integer k, does G have a clique of size at least k ? (**k-CLIQUE**)

Decision vs optimisation problems

- what is the decision problem that corresponds to **MAX-CLIQUE** ?
  - Given a graph G and an integer k, does G have a clique of size at least k ? (**k-CLIQUE**)

Claim

An optimisation problem[a] is as hard to solve as it's equivalent decision problem[b].

[a] such as MAX-CLIQUE
[b] in our example, k-CLIQUE

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

Example:

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

Example:

- Assume we can solve **MAX-CLIQUE**

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

Example:

- Assume we can solve **MAX-CLIQUE**
  - $c = MAX - CLIQUE(G)$

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

Example:

- Assume we can solve **MAX-CLIQUE**
  - $c = MAX - CLIQUE(G)$
  - Solve k-CLIQUE by verifying $k \leq c$

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

Example:

- Assume we can solve **MAX-CLIQUE**
  - $c = MAX - CLIQUE(G)$
  - Solve k-CLIQUE by verifying $k \leq c$
- Assume we can solve **k-CLIQUE**

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

Example:

- Assume we can solve **MAX-CLIQUE**
  - $c = MAX - CLIQUE(G)$
  - Solve k-CLIQUE by verifying $k \leq c$
- Assume we can solve **k-CLIQUE**
  - Fix $k = |V|$.

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

Example:

- Assume we can solve **MAX-CLIQUE**
  - $c = MAX - CLIQUE(G)$
  - Solve k-CLIQUE by verifying $k \leq c$
- Assume we can solve **k-CLIQUE**
  - Fix $k = |V|$.
  - Solve $k - CLIQUE$. If answer is **no**, $k = k - 1$

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

Example:

- Assume we can solve **MAX-CLIQUE**
  - $c = MAX - CLIQUE(G)$
  - Solve k-CLIQUE by verifying $k \leq c$
- Assume we can solve **k-CLIQUE**
  - Fix $k = |V|$.
  - Solve $k - CLIQUE$. If answer is **no**, $k = k - 1$
  - Repeat until we get a **yes** or $k = 0$. The answer to MAX-CLIQUE is k.

An optimisation problem is *as hard* to solve as it's equivalent decision problem.

Example:

- Assume we can solve **MAX-CLIQUE**
  - $c = MAX - CLIQUE(G)$
  - Solve k-CLIQUE by verifying $k \leq c$
- Assume we can solve **k-CLIQUE**
  - Fix $k = |V|$.
  - Solve $k - CLIQUE$. If answer is **no**, $k = k - 1$
  - Repeat until we get a **yes** or $k = 0$. The answer to MAX-CLIQUE is k.

If k-CLIQUE can be solved in $O(n^p)$ time then MAX-CLIQUE can be solved in $O(n^{p+1})$ time

When analysing the running time of an algorithm, we agreed to **ignore** constants. This lead to asymptotic notations.

When analysing the running time of an algorithm, we agreed to **ignore** constants. This lead to asymptotic notations.

In Complexity Theory, we will further agree to **ignore** the degrees of polynomials.

Problems vs algorithms

Definition (Algorithm)

Given a decision problem $P : I \to \{0, 1\}$, an algorithm for P is a finite specification of a computation process, which takes an encoding of a problem instance from I, and produces 0 or 1.

Computation process

Problems vs algorithms

Definition (Algorithm)

Given a decision problem $P : I \to \{0, 1\}$, an algorithm for P is a finite specification of a computation process, which takes an encoding of a problem instance from I, and produces 0 or 1.

Computation process

- goto

Problems vs algorithms

Definition (Algorithm)

Given a decision problem $P : I \to \{0, 1\}$, an algorithm for P is a finite specification of a computation process, which takes an encoding of a problem instance from I, and produces 0 or 1.

Computation process

- goto, loops

Problems vs algorithms

Definition (Algorithm)

Given a decision problem $P : I \to \{0, 1\}$, an algorithm for P is a finite specification of a computation process, which takes an encoding of a problem instance from I, and produces 0 or 1.

Computation process

- goto, loops, recursion

Problems vs algorithms

Definition (Algorithm)

Given a decision problem $P : I \to \{0, 1\}$, an algorithm for P is a finite specification of a computation process, which takes an encoding of a problem instance from I, and produces 0 or 1.

Computation process

- goto, loops, recursion, ...

Problems vs algorithms

Definition (Algorithm)

Given a decision problem $P : I \to \{0, 1\}$, an algorithm for P is a finite specification of a computation process, which takes an encoding of a problem instance from I, and produces 0 or 1.

Computation process

- goto, loops, recursion, ...
- *How do we find a simple mathematical model that captures all these ways to compute? [1]*

Problems vs algorithms

Definition (Algorithm)

Given a decision problem $P : I \to \{0, 1\}$, an algorithm for P is a finite specification of a computation process, which takes an encoding of a problem instance from I, and produces 0 or 1.

Computation process

- goto, loops, recursion, ...
- *How do we find a simple mathematical model that captures all these ways to compute? [1]*
- *How do we formally specify **algorithms** ?*

Part II: *The Turing Machine*

A **deterministic Turing Machine** (DTM) is a tuple $M = (K, F, \Sigma, \delta, s)$ where:

A **deterministic Turing Machine** (DTM) is a tuple $M = (K, F, \Sigma, \delta, s)$ where:

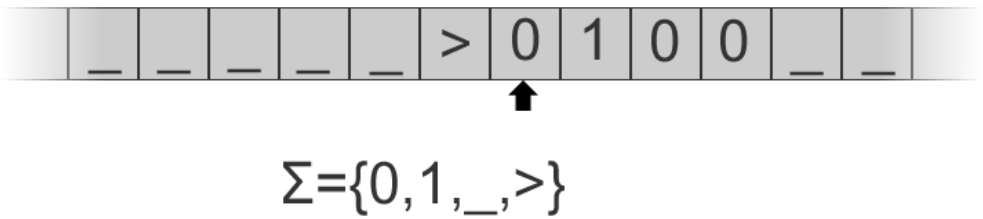
We assume *data* is *read/written* on a **tape**.

A **deterministic Turing Machine** (DTM) is a tuple $M = (K, F, \Sigma, \delta, s)$ where:

Σ={0,1,_,>}

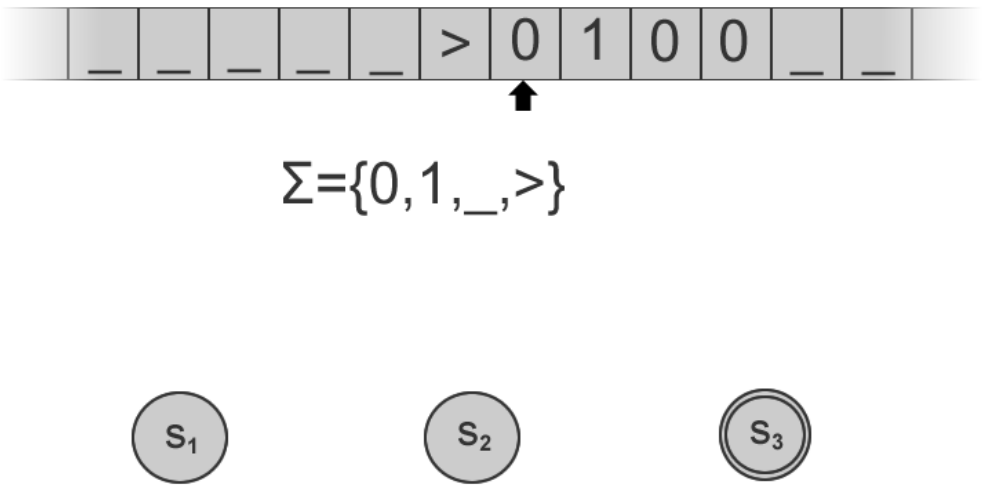
The tape contains **cells**, and each cell contains **symbols**. Σ is the set of symbols. Σ always contains _ (the blank symbol) and $>$ (the first symbol).

A **deterministic Turing Machine** (DTM) is a tuple $M = (K, F, \Sigma, \delta, s)$ where:



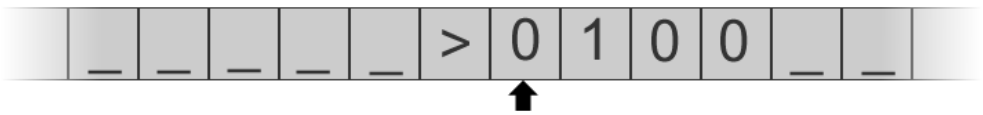
A **tape head** indicates the current position on the tape.

A **deterministic Turing Machine** (DTM) is a tuple $M = (K, F, \Sigma, \delta, s)$ where:

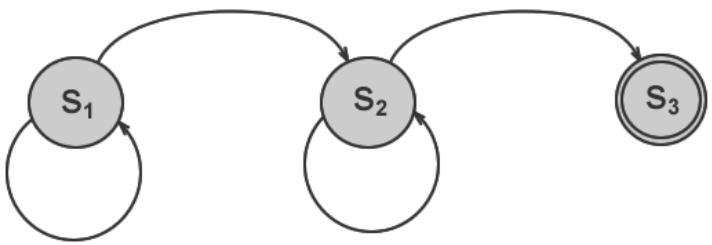


A set of **states** K and a set of **accepting states** $F \subseteq K$. Here $K = \{s_1, s_2, s_3\}$ and $F = \{s_3\}$.

A **deterministic Turing Machine** (DTM) is a tuple $M = (K, F, \Sigma, \delta, s)$ where:

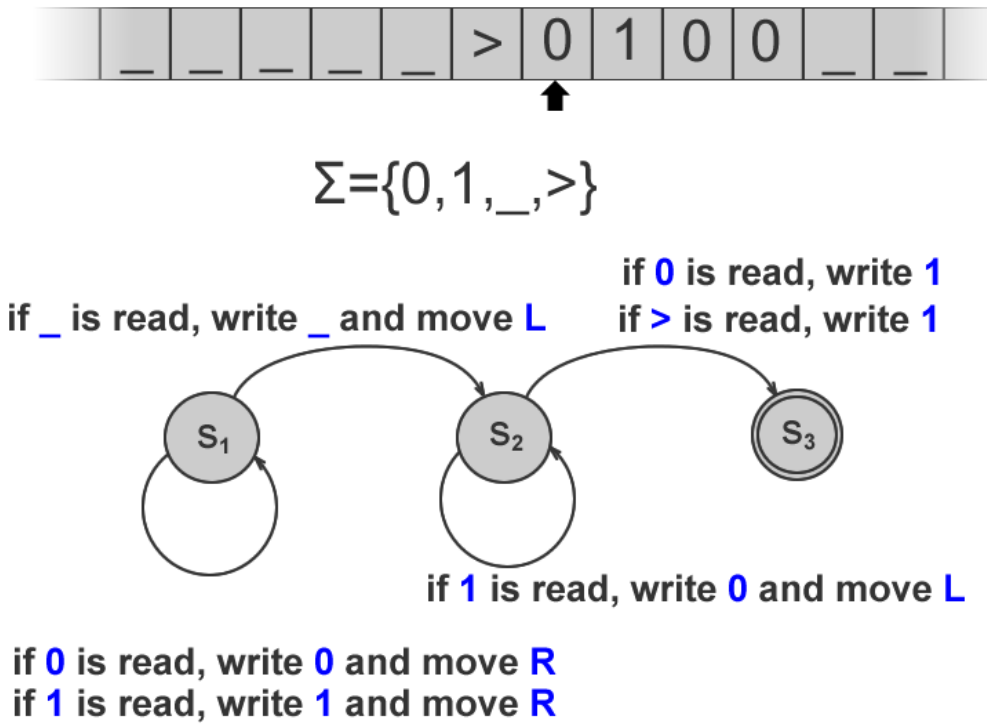


Σ={0,1,_,>}



A **transition function** $\delta : K \times \Sigma \rightarrow K \times \Sigma \times \{L, H, R\}$

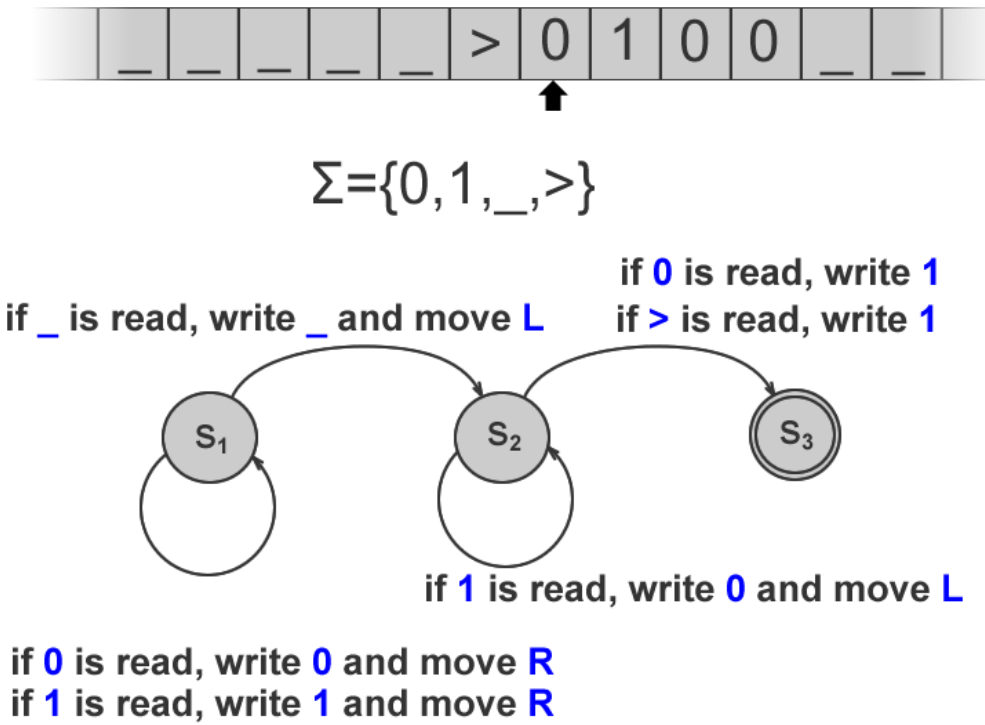
A **deterministic Turing Machine** (DTM) is a tuple $M = (K, F, \Sigma, \delta, s)$ where:



$\delta : K \times \Sigma \to K \times \Sigma \times \{L, H, R\}$

Write down δ for our example

A **deterministic Turing Machine** (DTM) is a tuple $M = (K, F, \Sigma, \delta, s)$ where:



An **initial state** s. In our case $s = s_1$.

Turing Machines as algorithms for decision problems

- Let $P : I \rightarrow \{0, 1\}$ be a **decision problem**

Turing Machines as algorithms for decision problems

- Let $P : I \rightarrow \{0, 1\}$ be a **decision problem**
- Let *M* be a Turing Machine, and *w* be a string encoding a **problem instance** from *I*

Turing Machines as algorithms for decision problems

- Let $P : I \rightarrow \{0, 1\}$ be a **decision problem**
- Let M be a Turing Machine, and w be a string encoding a **problem instance** from I
- Once M enters one of the accepting states during the processing of w, we say it **halts**

Turing Machines as algorithms for decision problems

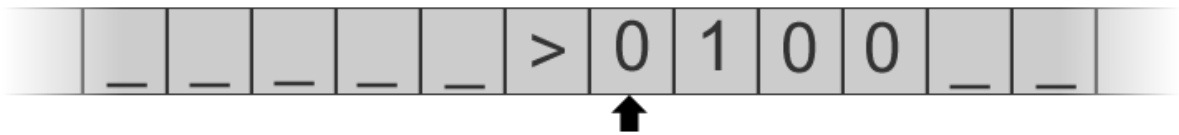
- Let $P : I \rightarrow \{0, 1\}$ be a **decision problem**
- Let M be a Turing Machine, and w be a string encoding a **problem instance** from I
- Once M enters one of the accepting states during the processing of w, we say it **halts**
- Once M halts, the **output** is the contents of the **tape**

Turing Machines as algorithms for decision problems

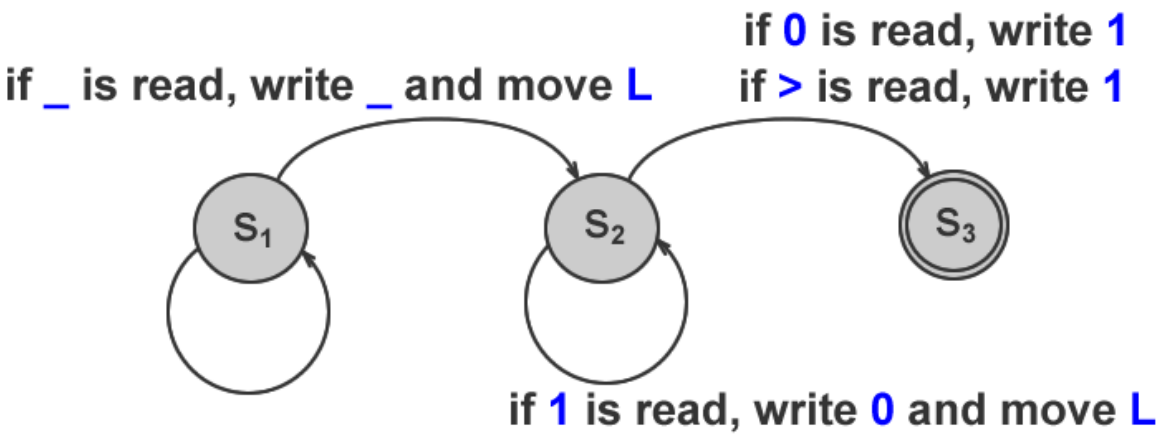
- Let $P : I \rightarrow \{0, 1\}$ be a **decision problem**
- Let M be a Turing Machine, and w be a string encoding a **problem instance** from I
- Once M enters one of the accepting states during the processing of w, we say it **halts**
- Once M halts, the **output** is the contents of the **tape**
- We also write $M(w)$ to refer to the output of the Turing Machine M, on string w.

Example

What does our machine do ?



$\Sigma = \{0, 1, \_, >\}$



if 0 is read, write 0 and move R
if 1 is read, write 1 and move R

Another example: `http://www.youtube.com/watch?v=WJ-ODmFjmrU&feature=relmfu`

Properties of Turing Machines

Properties of Turing Machines

Let Σ be an alphabet. We denote by Σ^* the **set of finite words** built with characters from Σ.

Properties of Turing Machines

Let Σ be an alphabet. We denote by Σ^* the **set of finite words** built with characters from Σ.

Definition (Running time [1])

Let $P : I \to \{0, 1\}$, $T : \mathbb{N} \to \mathbb{N}$, and M be a Turing Machine. We say M solves P if for every encoding $w \in \Sigma^*$ of a problem instance from I, if the tape is initialized with w, then it **halts** with $P(w)$ written as output. If the computation takes **at most** $T(|w|)$ steps **for every** w, then we say the running time of M is $T(n)$.

Properties of Turing Machines

Let Σ be an alphabet. We denote by Σ^* the **set of finite words** built with characters from Σ.

Definition (Running time [1])

Let $P : I \to \{0, 1\}$, $T : \mathbb{N} \to \mathbb{N}$, and M be a Turing Machine. We say M solves P if for every encoding $w \in \Sigma^*$ of a problem instance from I, if the tape is initialized with w, then it **halts** with $P(w)$ written as output. If the computation takes **at most** $T(|w|)$ steps **for every** w, then we say the running time of M is $T(n)$.

Definition

If M is a Turing Machine with running time $T(n)$, which solves the problem P, we say that P is **solvable** in time $T(n)$

Questions (you should ask):

[1] Recall our assumption regarding *distinction* between running times !

Questions (you should ask):

- *The choice of an encoding doesn't seem to appear in the definition. Is this correct ?*

[1] Recall our assumption regarding *distinction* between running times !

Questions (you should ask):

- *The choice of an encoding doesn't seem to appear in the definition. Is this correct ?*
- *Turing Machines can perform any kind of computation ?*

[1] Recall our assumption regarding *distinction* between running times !

Questions (you should ask):

- *The choice of an encoding doesn't seem to appear in the definition. Is this correct ?*
- *Turing Machines can perform any kind of computation ?*
- *Is there no other formalism capable of doing computations faster*[1] *?*

[1] Recall our assumption regarding *distinction* between running times !

Encoding does not matter

Proposition

Every problem $P : I \to \{0, 1\}$ *that is* ***solvable*** *in time* $T(n)$ *by a Turing Machine with alphabet* Σ *(which is used for encoding problem instances in* I*), is also solvable in time* $O(\log(|\Sigma|)) \cdot T(n)$ *using the alphabet* $\{0, 1, >, \_\}$.

Encoding does not matter

Proposition

Every problem $P : I \to \{0, 1\}$ *that is* ***solvable*** *in time* $T(n)$ *by a Turing Machine with alphabet* Σ *(which is used for encoding problem instances in* I*), is also solvable in time* $O(\log(|\Sigma|)) \cdot T(n)$ *using the alphabet* $\{0, 1, >, \_\}$.

Proof.

Blackboard !

Encoding does not matter

Proposition

Every problem $P : I \to \{0,1\}$ *that is* ***solvable*** *in time* $T(n)$ *by a Turing Machine with alphabet* Σ *(which is used for encoding problem instances in* I*), is also solvable in time* $O(\log(|\Sigma|)) \cdot T(n)$ *using the alphabet* $\{0,1,>,\_\}$.

Proof.

Blackboard !

□

The encoding does not matter !

Universal Turing Machines

- Turing Machine $\equiv$ algorithm

Universal Turing Machines

- Turing Machine $\equiv$ algorithm
- Turing Machines vs *actual computers*

Universal Turing Machines

- Turing Machine $\equiv$ algorithm
- Turing Machines vs *actual computers*
- **The (Universal) Turing Machine !** $U(M, w) = M(w)$

Universal Turing Machines

- Turing Machine $\equiv$ algorithm
- Turing Machines vs *actual computers*
- **The (Universal) Turing Machine !** $U(M, w) = M(w)$
- U is an algorithm which simulates another algorithm

Universal Turing Machines

- Turing Machine $\equiv$ algorithm
- Turing Machines vs *actual computers*
- **The (Universal) Turing Machine !** $U(M, w) = M(w)$
- U is an algorithm which simulates another algorithm
- software $\leftrightarrow$ hardware $\leftrightarrow$ data

Universal Turing Machines

Definition (Efficient Turing Machine [1])

For every Turing Machine M having running time $T(n)$, and every word $w \in \{0,1\}$ there exists a **Universal Turing Machine** U, such that $U(enc(M), w) = M(w)$.
Moreover, the **running time of** U is $c \cdot T(n)$, where c only depends on M, and not on the size of w.

Universal Turing Machines

Definition (Efficient Turing Machine [1])

For every Turing Machine M having running time $T(n)$, and every word $w \in \{0,1\}$ there exists a **Universal Turing Machine** U, such that $U(enc(M), w) = M(w)$.
Moreover, the **running time of** U is $c \cdot T(n)$, where c only depends on M, and not on the size of w.

Proof.

Blackboard ! □

Turing Machines as Strings

Let $M = (K, F, \Sigma, \delta, s)$ be a Turing Machine.

Turing Machines as Strings

Let $M = (K, F, \Sigma, \delta, s)$ be a Turing Machine.

- We encode any each state in K as an integer in $\{1, 2, \ldots, |K|\}$

Turing Machines as Strings

Let $M = (K, F, \Sigma, \delta, s)$ be a Turing Machine.

- We encode any each state in K as an integer in $\{1, 2, \ldots, |K|\}$
- We encode each final state F as an integer in $\{|K| + 1, \ldots, |K| + |F|\}$

Turing Machines as Strings

Let $M = (K, F, \Sigma, \delta, s)$ be a Turing Machine.

- We encode any each state in K as an integer in $\{1, 2, \ldots, |K|\}$
- We encode each final state F as an integer in $\{|K| + 1, \ldots, |K| + |F|\}$
- We encode s as $|K| + |F| + 1$ and L, H, R as $|K| + |F| + i$, with $i \in \{2, 3, 4\}$

Turing Machines as Strings

Let $M = (K, F, \Sigma, \delta, s)$ be a Turing Machine.

- We encode any each state in K as an integer in $\{1, 2, \ldots, |K|\}$
- We encode each final state F as an integer in $\{|K| + 1, \ldots, |K| + |F|\}$
- We encode s as $|K| + |F| + 1$ and L, H, R as $|K| + |F| + i$, with $i \in \{2, 3, 4\}$

Each integer x will be represented as a string $enc(x)$ using $\lceil \log(|K| + |F| + 4) \rceil$ **bits**

Turing Machines as Strings

Encoding δ:

- Each transition $\delta(s, c) = (s', c', pos)$ is encoded as:

Turing Machines as Strings

Encoding δ:

- Each transition $\delta(s, c) = (s', c', pos)$ is encoded as:
- ((enc(s),c), (enc(s'),c',enc(pos))

Turing Machines as Strings

Encoding δ:

- Each transition $\delta(s, c) = (s', c', pos)$ is encoded as:
- ((enc(s),c), (enc(s'),c',enc(pos))
- using the special characters "(", ")" and ","

Turing Machines as Strings

Encoding δ:

- Each transition $\delta(s, c) = (s', c', pos)$ is encoded as:
- ((enc(s),c), (enc(s'),c',enc(pos))
- using the special characters "(", ")" and ","

enc(δ) is a **sequence** of transitions, encoded as above

Turing Machines and computability

Claim (Church)

For every program $P : \Sigma^* \to \{0, 1\}$, written in any programming language, and with running time $T(n)$, there exists an ***equivalent*** *Turing Machine with running time $O(T(n)^p)$, where p is a* ***fixed*** *natural number.*

Turing Machines and computability

Claim (Church)

*For every program $P : \Sigma^* \to \{0, 1\}$, written in any programming language, and with running time $T(n)$, there exists an **equivalent** Turing Machine with running time $O(T(n)^p)$, where p is a **fixed** natural number.*

- Evidence: **Lambda Calculus**, **RAM**, **μ-recursive functions**

Turing Machines and computability

Claim (Church)

*For every program $P : \Sigma^* \to \{0, 1\}$, written in any programming language, and with running time $T(n)$, there exists an **equivalent** Turing Machine with running time $O(T(n)^p)$, where p is a **fixed** natural number.*

- Evidence: **Lambda Calculus**, **RAM**, **μ-recursive functions**
- For this reason, we convene to describe Turing Machines in **pseudocode**, for simplicity and clarity.

Turing Machines and computability

Claim (Church)

*For every program $P : \Sigma^* \to \{0, 1\}$, written in any programming language, and with running time $T(n)$, there exists an **equivalent** Turing Machine with running time $O(T(n)^p)$, where p is a **fixed** natural number.*

- Evidence: **Lambda Calculus**, **RAM**, **μ-recursive functions**
- For this reason, we convene to describe Turing Machines in **pseudocode**, for simplicity and clarity.

Example - blackboard

Part III: *Is finding the correct answer **harder** than checking if **a given answer** is correct ?*

Nondeterministic Turing Machines

Example:

Nondeterministic Turing Machines

Example:

- Let M be a TM which solves 3-CLIQUE: $M(w_{G(V,E)}) = w_C$, where $w_{G(V,E)}$ encodes a graph and w_C encodes a set of nodes)

Nondeterministic Turing Machines

Example:

- Let M be a TM which solves 3-CLIQUE: $M(w_{G(V,E)}) = w_C$, where $w_{G(V,E)}$ encodes a graph and w_C encodes a set of nodes)
- Let M' be a TM such that $M(w_{G(V,E)}, w_C) = 1$ iff w_C is a 3-CLIQUE for $w_{G(V,E)}$.

Nondeterministic Turing Machines

Example:

- Let M be a TM which solves 3-CLIQUE: $M(w_{G(V,E)}) = w_C$, where $w_{G(V,E)}$ encodes a graph and w_C encodes a set of nodes)
- Let M' be a TM such that $M(w_{G(V,E)}, w_C) = 1$ iff w_C is a 3-CLIQUE for $w_{G(V,E)}$.

One could construct M, based on M'. (blackboard).

Nondeterministic Turing Machines

Example:

- Let M be a TM which solves 3-CLIQUE: $M(w_{G(V,E)}) = w_C$, where $w_{G(V,E)}$ encodes a graph and w_C encodes a set of nodes)
- Let M' be a TM such that $M(w_{G(V,E)}, w_C) = 1$ iff w_C is a 3-CLIQUE for $w_{G(V,E)}$.

One could construct M, based on M'. (blackboard). If the running time of M' is polynomial, then the running time of M is exponential !

Nondeterministic Turing Machines

Example:

- Let M be a TM which solves 3-CLIQUE: $M(w_{G(V,E)}) = w_C$, where $w_{G(V,E)}$ encodes a graph and w_C encodes a set of nodes)
- Let M' be a TM such that $M(w_{G(V,E)}, w_C) = 1$ iff w_C is a 3-CLIQUE for $w_{G(V,E)}$.

One could construct M, based on M'. (blackboard). If the running time of M' is polynomial, then the running time of M is exponential !
Could we do better ?

Nondeterministic Turing Machines

Example:

- Let M be a TM which solves 3-CLIQUE: $M(w_{G(V,E)}) = w_C$, where $w_{G(V,E)}$ encodes a graph and w_C encodes a set of nodes)
- Let M' be a TM such that $M(w_{G(V,E)}, w_C) = 1$ iff w_C is a 3-CLIQUE for $w_{G(V,E)}$.

One could construct M, based on M'. (blackboard). If the running time of M' is polynomial, then the running time of M is exponential !
Could we do better ?
We need a tool to formally express **exhaustive search**

Nondeterministic Turing Machines

Definition

Nondeterministic Turing Machine A non-deterministic Turing Machine (NTM) is a TM $M = (K, F, \Sigma, \delta, s)$, where δ is a **relation**: $\delta \subset K \times \Sigma \times K \times \Sigma \times \{L, H, R\}$

Nondeterministic Turing Machines

Definition

Nondeterministic Turing Machine A non-deterministic Turing Machine (NTM) is a TM $M = (K, F, \Sigma, \delta, s)$, where δ is a **relation**: $\delta \subset K \times \Sigma \times K \times \Sigma \times \{L, H, R\}$

Definition (Success)

We say $M(w) = 1$, if there **exists** a sequence of transitions such that M halts and the output is 1.

Nondeterministic Turing Machines

Definition

Nondeterministic Turing Machine A non-deterministic Turing Machine (NTM) is a TM $M = (K, F, \Sigma, \delta, s)$, where δ is a **relation**: $\delta \subset K \times \Sigma \times K \times \Sigma \times \{L, H, R\}$

Definition (Success)

We say $M(w) = 1$, if there **exists** a sequence of transitions such that M halts and the output is 1.

Definition (Fail)

We say $M(w) = 0$, if **for all** sequences of transitions, M halts and the output is 0.

Nondeterministic Turing Machines

Definition

Nondeterministic Turing Machine A non-deterministic Turing Machine (NTM) is a TM $M = (K, F, \Sigma, \delta, s)$, where δ is a **relation**: $\delta \subset K \times \Sigma \times K \times \Sigma \times \{L, H, R\}$

Definition (Success)

We say $M(w) = 1$, if there **exists** a sequence of transitions such that M halts and the output is 1.

Definition (Fail)

We say $M(w) = 0$, if **for all** sequences of transitions, M halts and the output is 0.

Example (blackboard)

Nondeterministic Turing Machines

Definition (Running time of a NTM)

We say a NTM M has running time $T(n)$ iff **any** sequence of transitions takes **at most** $T(|w|)$ steps **for every input** w.

Nondeterministic Turing Machines

Definition (Running time of a NTM)

We say a NTM M has running time $T(n)$ iff **any** sequence of transitions takes **at most** $T(|w|)$ steps **for every input** w.

Comments:

Nondeterministic Turing Machines

Definition (Running time of a NTM)

We say a NTM M has running time $T(n)$ iff **any** sequence of transitions takes **at most** $T(|w|)$ steps **for every input** w.

Comments:

- A computation of a TM is a **path** $\leftrightarrow$ A computation of a NTM is a **tree**

Nondeterministic Turing Machines

Definition (Running time of a NTM)

We say a NTM M has running time $T(n)$ iff **any** sequence of transitions takes **at most** $T(|w|)$ steps **for every input** w.

Comments:

- A computation of a TM is a **path** $\leftrightarrow$ A computation of a NTM is a **tree**
- The running time of a TM is the **path's length** $\leftrightarrow$ The running time of a NTM is the **tree's height**

Nondeterministic Turing Machines

Definition (Running time of a NTM)

We say a NTM M has running time $T(n)$ iff **any** sequence of transitions takes **at most** $T(|w|)$ steps **for every input** w.

Comments:

- A computation of a TM is a **path** $\leftrightarrow$ A computation of a NTM is a **tree**
- The running time of a TM is the **path's length** $\leftrightarrow$ The running time of a NTM is the **tree's height**
- NTMs can explore **exponential space** in **polynomial time**

Nondeterministic Turing Machines

Definition (Running time of a NTM)

We say a NTM M has running time $T(n)$ iff **any** sequence of transitions takes **at most** $T(|w|)$ steps **for every input** w.

Comments:

- A computation of a TM is a **path** $\leftrightarrow$ A computation of a NTM is a **tree**
- The running time of a TM is the **path's length** $\leftrightarrow$ The running time of a NTM is the **tree's height**
- NTMs can explore **exponential space** in **polynomial time**

NTM's as **pseudocode**: (`choice`, `success`, `fail`) - blackboard

NTMs vs DTMs

Proposition

*For every NTM M', there exists an **equivalent** DTM M. If M' runs in polynomial time, M runs in exponential time.*

Complexity classes

Definition (DTIME)

We denote by $DTIME(f(n))$ the **set of problems** which are solvable by a (D)TM with running time $O(f(n))$.

Complexity classes

Definition (DTIME)

We denote by $DTIME(f(n))$ the **set of problems** which are solvable by a (D)TM with running time $O(f(n))$.

Definition (NTIME)

We denote by $NTIME(f(n))$ the **set of problems** which are solvable by a NTM with running time $O(f(n))$.

Complexity classes

Definition (DTIME)

We denote by $DTIME(f(n))$ the **set of problems** which are solvable by a (D)TM with running time $O(f(n))$.

Definition (NTIME)

We denote by $NTIME(f(n))$ the **set of problems** which are solvable by a NTM with running time $O(f(n))$.

Definition (PTIME)

We denote by $PTIME$ (or simply P): $PTIME = \cup_{d \in \mathbb{N}} DTIME(n^d)$.

Complexity classes

Definition (DTIME)

We denote by $DTIME(f(n))$ the **set of problems** which are solvable by a (D)TM with running time $O(f(n))$.

Definition (NTIME)

We denote by $NTIME(f(n))$ the **set of problems** which are solvable by a NTM with running time $O(f(n))$.

Definition (PTIME)

We denote by $PTIME$ (or simply P): $PTIME = \cup_{d \in \mathbb{N}} DTIME(n^d)$.

Definition (NPTIME)

We denote by $NPTIME$ (or simply NP): $NPTIME = \cup_{d \in \mathbb{N}} NTIME(n^d)$.

Complexity classes

*Is finding the correct answer **harder** than checking if **a given answer** is correct ?*

Complexity classes

*Is finding the correct answer **harder** than checking if **a given answer** is correct ?*

Can we **simulate** any NTM with **polynomial** running time by a DTM with **polynomial** running time ?

Complexity classes

*Is finding the correct answer **harder** than checking if **a given answer** is correct ?*

Can we **simulate** any NTM with **polynomial** running time by a DTM with **polynomial** running time ?

P = NP (?!)

Bibliography I


Sanjeev Arora and Boaz Barak.
Computational Complexity: A Modern Approach.
Cambridge University Press, New York, NY, USA, 1st edition, 2009.


C.H. Papadimitriou.
Computational complexity.
Addison-Wesley, 1994.