PSPACE and the Polynomial Hierarchy

Derek Aoki

June 2024

Difficulty of a Problem for a Computer

Complexity is about resources needed for computers to solve problems. Resources include:

Difficulty of a Problem for a Computer

Complexity is about resources needed for computers to solve problems. Resources include:

- Miminum runtime of an algorithm

Difficulty of a Problem for a Computer

Complexity is about resources needed for computers to solve problems. Resources include:

- Miminum runtime of an algorithm
- Minimum memory usage of an algorithm

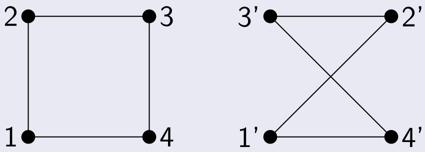
Difficulty of a Problem for a Computer

Complexity is about resources needed for computers to solve problems. Resources include:

- Miminum runtime of an algorithm
- Minimum memory usage of an algorithm

Example

One problem is determining whether or not graphs are "isomorphic":



The immediate algorithm is brute force, but this is not the best (and there has been significant work into improving this).

Goals

Why we care:

Goals

Why we care:

1. Finding proteins to cure cancer

Goals

Why we care:

1. Finding proteins to cure cancer
2. Efficiently attacking cryptography

Goals

Why we care:

1. Finding proteins to cure cancer
2. Efficiently attacking cryptography
3. Playing Go and Chess

Goals

Why we care:

1. Finding proteins to cure cancer
2. Efficiently attacking cryptography
3. Playing Go and Chess

What we want to do:

Goals

Why we care:

1. Finding proteins to cure cancer
2. Efficiently attacking cryptography
3. Playing Go and Chess

What we want to do:

1. Figure out what problems are easier than others

Goals

Why we care:

1. Finding proteins to cure cancer
2. Efficiently attacking cryptography
3. Playing Go and Chess

What we want to do:

1. Figure out what problems are easier than others
2. What are important properties of problems and how can we classify them

Goals

Why we care:

1. Finding proteins to cure cancer
2. Efficiently attacking cryptography
3. Playing Go and Chess

What we want to do:

1. Figure out what problems are easier than others
2. What are important properties of problems and how can we classify them

What we need to approach that:

Goals

Why we care:

1. Finding proteins to cure cancer
2. Efficiently attacking cryptography
3. Playing Go and Chess

What we want to do:

1. Figure out what problems are easier than others
2. What are important properties of problems and how can we classify them

What we need to approach that:

1. What do "computer" and "problem" mean mathematically

Goals

Why we care:

1. Finding proteins to cure cancer
2. Efficiently attacking cryptography
3. Playing Go and Chess

What we want to do:

1. Figure out what problems are easier than others
2. What are important properties of problems and how can we classify them

What we need to approach that:

1. What do "computer" and "problem" mean mathematically
2. What can "easier" mean mathematically

Goals

Why we care:

1. Finding proteins to cure cancer
2. Efficiently attacking cryptography
3. Playing Go and Chess

What we want to do:

1. Figure out what problems are easier than others
2. What are important properties of problems and how can we classify them

What we need to approach that:

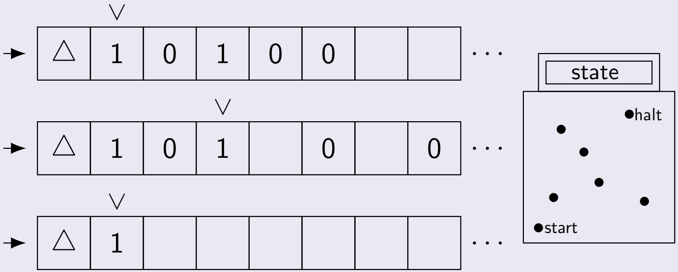
1. What do "computer" and "problem" mean mathematically
2. What can "easier" mean mathematically
3. Kinds of categories and properties we can formalize

Turing Machines

We need to formalize our notion of a "computer":

Definition

A Turing machine (TM) is a set of k tapes, a head on each tape, and a set of "instructions":

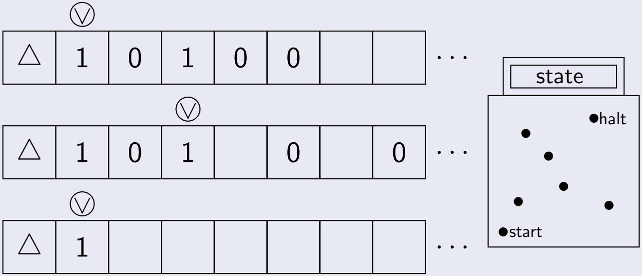


Turing Machines

We need to formalize our notion of a "computer":

Definition

A Turing machine (TM) is a set of k tapes, a head on each tape, and a set of "instructions":

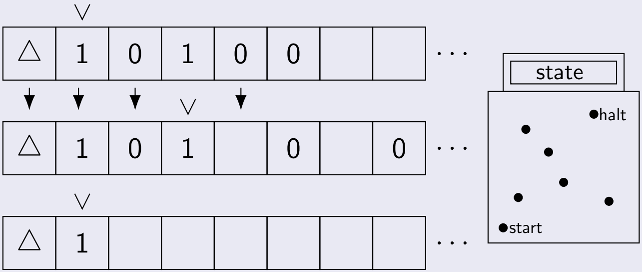


Turing Machines

We need to formalize our notion of a "computer":

Definition

A Turing machine (TM) is a set of k tapes, a head on each tape, and a set of "instructions":

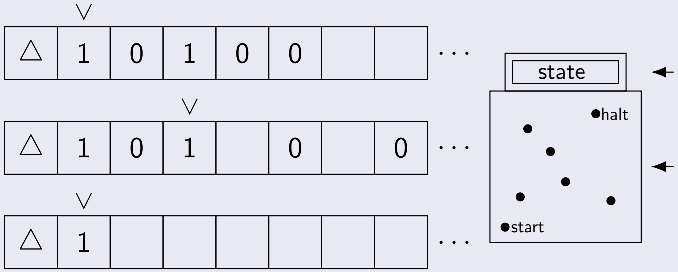


Turing Machines

We need to formalize our notion of a "computer":

Definition

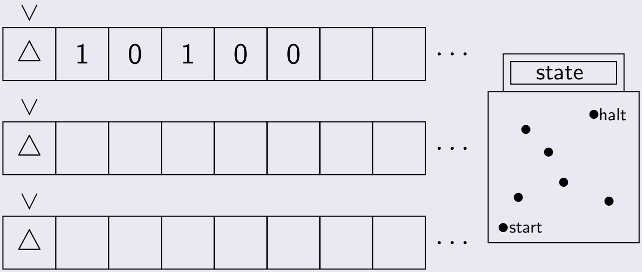
A Turing machine (TM) is a set of k tapes, a head on each tape, and a set of "instructions":



Turing Machines

Definition

A Turing machine (TM) is a set of k tapes, a head on each tape, and a set of "instructions":

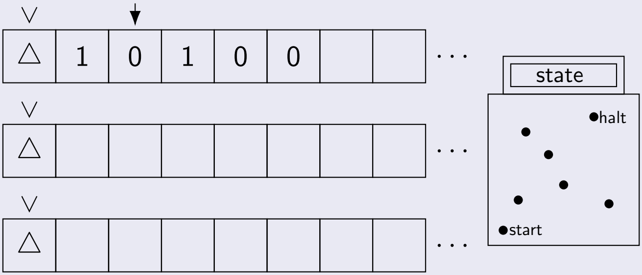


Steps: (1) check symbols under heads and state (2) write symbols under heads (3) move heads (4) change state

Turing Machines

Definition

A Turing machine (TM) is a set of k tapes, a head on each tape, and a set of "instructions":

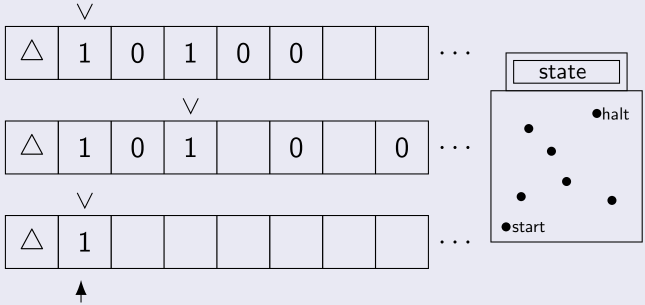


Steps: (1) check symbols under heads and state (2) write symbols under heads (3) move heads (4) change state

Turing Machines

Definition

A Turing machine (TM) is a set of k tapes, a head on each tape, and a set of "instructions":



Steps: (1) check symbols under heads and state (2) write symbols under heads (3) move heads (4) change state

Why TMs?

This is a fairly involved definition, and seemingly arbitrarys, so one might ask why we specifically choose TMs as our formalism of computers.

1. They are "universal"
2. They let us formalize properties such as "time" and "space" naturally

Problems

We also need to introduce the idea of a "problem" for our computer to solve:

Problems

We also need to introduce the idea of a "problem" for our computer to solve:

Definition

Define $\{0,1\}^* = \bigcup_n \{0,1\}^n$ as the set of all finite binary strings.

A language is some set $L \subseteq \{0,1\}^*$.

Problems

We also need to introduce the idea of a "problem" for our computer to solve:

Definition

Define $\{0,1\}^* = \bigcup_n \{0,1\}^n$ as the set of all finite binary strings.

A language is some set $L \subseteq \{0,1\}^*$.

Languages correspond to "decision problems".

Problems

We also need to introduce the idea of a "problem" for our computer to solve:

Definition

Define $\{0,1\}^* = \bigcup_n \{0,1\}^n$ as the set of all finite binary strings.

A language is some set $L \subseteq \{0,1\}^*$.

Languages correspond to "decision problems".

Example

Given encodings of two graphs, determine whether or not they are isomorphic.

"Easy" Problems

In complexity, there is a general assumption that:

1. A problem is "easy" if it can be decided in "polynomial time" (there exists a TM that decides it in $p(n)$ steps on inputs of length n)
2. A problem is "hard" if it cannot (the stereotypical example is problems that require "exponential" time, or $2^{p(k)}$ steps on inputs of length n)

The collection of languages that are "easy" in this sense are the class **P** and this notion of "hard" defines the class **EXPTIME**.

Reductions

We care not only about general ease but also relative ease. For example, some problems are "easier" than others in a certain sense:

Definition

A "polynomial-time reduction" from a language A to a language B is a function $f : \{0,1\}^* \to \{0,1\}^*$ such that there is a TM that can compute $f(x)$ in $p(n)$ steps on inputs of length n and such that $f(x)$ is in B if and only if x is in A. We say that A is "polynomial-time reducible" to B if there exists a reduction from A to B, denoted $A \leq_p B$.

We can see that if a language A is reducible to another language B, then if we can easily solve B we can easily solve A. We can take a string, perform the reduction, and then see whether or not it is in B, which in total determines membership in A.

Complete Problems

This idea of "easier" problems naturally leads into the idea of problems that are harder than an entire set of problems:

Definition

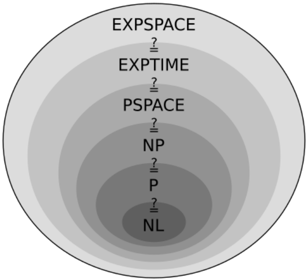
Let $\mathcal{C}$ be a collection of languages and L a language.

- We say that L is hard for $\mathcal{C}$ via polynomial-time reductions if $A \leq_p L$ for every A in $\mathcal{C}$ (L is $\mathcal{C}$-hard).
- We say L is complete for $\mathcal{C}$ via polynomial-time reductions if it is $\mathcal{C}$-hard and is in $\mathcal{C}$.

Complete problems are the "hardest", but they aren't unique.

The Poset of Languages

We can see that reductions create a "partial ordering" on the set of all languages. We are interested in certain subsets of this partially ordered set and how they compare to each other. On the right, you can see a variety of classes that interest us and their inclusions.



[1]Credit to Wikipedia for this image

PSPACE

We care about "memory" used by TMs when solving problems:

PSPACE

We care about "memory" used by TMs when solving problems:

Definition

"Space" or memory used by a TM is the number of cells it uses. **PSPACE** is the set of languages decided by a TM using $p(n)$ cells on an input of length n.

PSPACE

We care about "memory" used by TMs when solving problems:

Definition

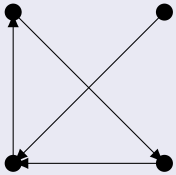
"Space" or memory used by a TM is the number of cells it uses. **PSPACE** is the set of languages decided by a TM using $p(n)$ cells on an input of length n.

We will now introduce the following specific problem:

Geography

Example

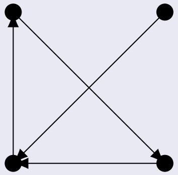
The problem Geography (GG) is determining whether or not player 1 has a winning strategy for a given digraph:



Geography

Example

The problem Geography (GG) is determining whether or not player 1 has a winning strategy for a given digraph:

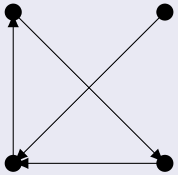


We have the following theorem:

Geography

Example

The problem Geography (GG) is determining whether or not player 1 has a winning strategy for a given digraph:



We have the following theorem:

Theorem

The language GG of encodings of digraphs in which player 1 has a winning strategy is **PSPACE**-complete.

NP

We will now introduce a famous class that interests many:

Definition

A language L is in **NP** if and only if there exists a polynomial-time TM M and a polynomial p such that, for every binary string x,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ such that } M \text{ accepts on input } \langle x, u \rangle$$

where $|x|$ is the length of x and $\langle x, u \rangle$ is an encoding of x and u.

In effect, this says that we can easily "verify" that a string is in an **NP** language with a "solution".

The Polynomial Hierarchy

We can now generalize this notion of **NP**:

Definition

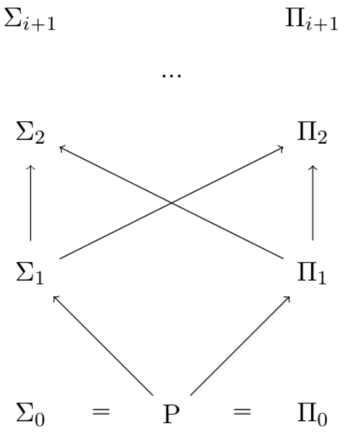
1. Define Σ_i^p as the set of languages L such that there exists a TM M_L and polynomial p such that for all x,

$$\begin{aligned} x \in L \Leftrightarrow & \exists u_1 \in \{0,1\}^{p(|x|)} \forall u_2 \in \{0,1\}^{p(|x|)} \ldots \\ & Q_i u_i \in \{0,1\}^{p(|x|)} \text{ such that} \\ & M_L \text{ accepts on input } \langle x, u_1, \ldots u_i \rangle \end{aligned}$$

2. Define Π_i^p as the set of complements of the languages in Σ_i^p, ie $\Pi_i^p = \{\{0,1\}^* \setminus L : L \in \Sigma_i^p\}$
3. Define **PH** as $\bigcup_i \Sigma_i^p$.

Hierarchy of Inclusions

The diagram of inclusions on the right is not difficult to derive.



[2]Credit to [Bar22] for this image

PH in PSPACE

The polynomial hierarchy and **PSPACE** are quite closely related. One of the most direct relationships is the following:

Theorem

PH $\subseteq$ **PSPACE**.

We have the following proof sketch:

1. Take a language L in Σ_i^p
2. Make "guesses" for each u_n
3. Simulate M_L on the guess
4. Track previous guesses and outputs of M_L

This tells us where the polynomial hierarchy fits into our hierarchy of classes from earlier: it spans from **P** up to (possibly) **PSPACE**, and learning more about it tells us about the problems and classes in that part of the poset of languages.

Final Notes

Some final thoughts:

Final Notes

Some final thoughts:

1. **PSPACE**-complete problems are not necessarily **PH**-complete

Final Notes

Some final thoughts:

1. **PSPACE**-complete problems are not necessarily **PH**-complete
2. We have Σ_i^p-complete problems for every i

Final Notes

Some final thoughts:

1. **PSPACE**-complete problems are not necessarily **PH**-complete
2. We have Σ_i^p-complete problems for every i
3. It is an open problem whether or not $\Sigma_i^p = \Pi_i^p$ for every $i \geq 1$

Final Notes

Some final thoughts:

1. **PSPACE**-complete problems are not necessarily **PH**-complete
2. We have Σ_i^p-complete problems for every i
3. It is an open problem whether or not $\Sigma_i^p = \Pi_i^p$ for every $i \geq 1$
4. It is an open problem whether or not $\Sigma_i^p = \Sigma_{i+1}^p$ for every i (including whether or not $\mathbf{P} = \mathbf{NP}$)

Final Notes

Some final thoughts:

1. **PSPACE**-complete problems are not necessarily **PH**-complete
2. We have Σ_i^p-complete problems for every i
3. It is an open problem whether or not $\Sigma_i^p = \Pi_i^p$ for every $i \geq 1$
4. It is an open problem whether or not $\Sigma_i^p = \Sigma_{i+1}^p$ for every i (including whether or not $\mathbf{P} = \mathbf{NP}$)
5. It is open whether or not $\mathbf{PH} = \mathbf{PSPACE}$

Final Notes

Some final thoughts:

1. **PSPACE**-complete problems are not necessarily **PH**-complete
2. We have Σ_i^p-complete problems for every i
3. It is an open problem whether or not $\Sigma_i^p = \Pi_i^p$ for every $i \geq 1$
4. It is an open problem whether or not $\Sigma_i^p = \Sigma_{i+1}^p$ for every i (including whether or not $\mathbf{P} = \mathbf{NP}$)
5. It is open whether or not $\mathbf{PH} = \mathbf{PSPACE}$
6. This has been about computer science, but there are interesting connections to logic and the theory of finite models that I explore in-depth in my paper

Thank you.

References

Sanjoy Baruah.
Feasibility analysis for hpc-dag tasks.
Real-Time Systems, 58:1–19, 06 2022.