"Why Are We Learning This?": The Importance of Theoretical Computer Science

IEEE Canadian Atlantic Computer Society

Taylor J. Smith

Department of Computer Science
St. Francis Xavier University
Antigonish, Nova Scotia, Canada

December 9, 2024

Who am I?

- Assistant professor, StFX
  Dept. of Computer Science
- Director, FLAReLab
- Research specialties:
  - Automata theory
  - Formal languages
- Teaching specialties:
  - Theory of computing
  - Algorithm design & analysis
  - Computational logic
- PhD ’21, Queen’s University
- IEEE membership:
  - GS’16–21
  - M’22–present

FLAReLab
Formal Languages and Automata Research Lab
St. Francis Xavier University

Table of Contents

Table of Contents

Why are we learning TCS?

- There's a reason this question is the title of the talk—it's one of the most common refrains from students!

Why are we learning TCS?

- There's a reason this question is the title of the talk—it's one of the most common refrains from students!

Officially?

- Because we have to.
  - Every Canadian U15 university has some kind of TCS course on the books, usually required to earn a CS degree
  - Ditto for the Maple League universities

Why are we learning TCS?

- There's a reason this question is the title of the talk—it's one of the most common refrains from students!

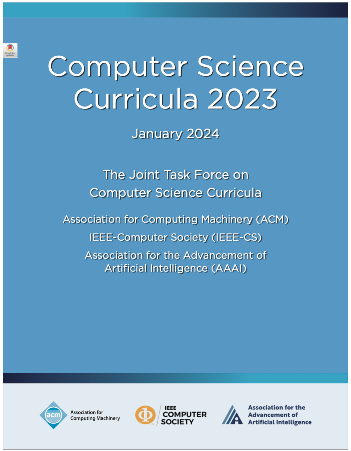
Officially?

- Because we have to.
  - Every Canadian U15 university has some kind of TCS course on the books, usually required to earn a CS degree
  - Ditto for the Maple League universities

Unofficially?

- Because it's the foundation of all of computer science!
  - There isn't a single result in any other CS course that doesn't rely on theory in some way

The official word: CS2023


Computer Science Curricula 2023

January 2024

The Joint Task Force on Computer Science Curricula

Association for Computing Machinery (ACM)

IEEE-Computer Society (IEEE-CS)

Association for the Advancement of Artificial Intelligence (AAAI)

AL-Models: Computational Models and Formal Languages

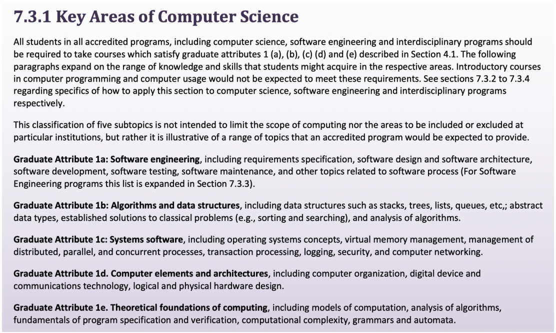
CS Core:
1. Formal automata
   a. Finite State
   b. Pushdown
   c. Linear Bounded
   d. Turing Machine
2. Formal languages, grammars and Chomsky Hierarchy
   (See also: FPL-Translation, FPL-Syntax)
   a. Regular (Type-3)
      i. Regular Expressions
   b. Context-Free (Type-2)
   c. Context-Sensitive (Type-1)
   d. Recursively Enumerable (Type-0)
3. Relations among formal automata, languages, and grammars
4. Decidability, (un)computability, and halting
5. The Church-Turing thesis
6. Algorithmic correctness
   a. Invariants (e.g., in iteration, recursion, tree search)

KA Core:
7. Deterministic and nondeterministic automata
8. Pumping Lemma proofs
   a. Proof of Finite State/Regular-Language limitation
   b. Pushdown Automata/Context-Free-Language limitation
9. Decidability
   a. Arithmetization and diagonalization
10. Reducibility and reductions
11. Time complexity based on Turing Machine
12. Space complexity (e.g., Pspace, Savitch's Theorem)
13. Equivalent models of algorithmic computation
   a. Turing Machines and Variations (e.g., multi-tape, non-deterministic)
   b. Lambda Calculus (See also: FPL-Functional)
   c. Mu-Recursive Functions

https://csed.acm.org

The official (Canadian) word: CSAC

CIPS
Computer Science Accreditation Council
Accreditation Criteria for
Computer Science, Software Engineering and Interdisciplinary Programs
Approved August 2011



7.3.1 Key Areas of Computer Science

All students in all accredited programs, including computer science, software engineering and interdisciplinary programs should be required to take courses which satisfy graduate attributes 1 (a), (b), (c) (d) and (e) described in Section 4.1. The following paragraphs expand on the range of knowledge and skills that students might acquire in the respective areas. Introductory courses in computer programming and computer usage would not be expected to meet these requirements. See sections 7.3.2 to 7.3.4 regarding specifics of how to apply this section to computer science, software engineering and interdisciplinary programs respectively.

This classification of five subtopics is not intended to limit the scope of computing nor the areas to be included or excluded at particular institutions, but rather it is illustrative of a range of topics that an accredited program would be expected to provide.

Graduate Attribute 1a: Software engineering, including requirements specification, software design and software architecture, software development, software testing, software maintenance, and other topics related to software process (For Software Engineering programs this list is expanded in Section 7.3.3).

Graduate Attribute 1b: Algorithms and data structures, including data structures such as stacks, trees, lists, queues, etc,; abstract data types, established solutions to classical problems (e.g., sorting and searching), and analysis of algorithms.

Graduate Attribute 1c: Systems software, including operating systems concepts, virtual memory management, management of distributed, parallel, and concurrent processes, transaction processing, logging, security, and computer networking.

Graduate Attribute 1d. Computer elements and architectures, including computer organization, digital device and communications technology, logical and physical hardware design.

Graduate Attribute 1e. Theoretical foundations of computing, including models of computation, analysis of algorithms, fundamentals of program specification and verification, computational complexity, grammars and automata.

https://cips.ca/csac/

Traditional TCS courses

- Usually, a traditional TCS course builds upward from a very simple model of computation.
  - finite automata (= regular expressions)
  - pushdown automata (= context-free grammars)
  - linear-bounded automata
  - Turing machines

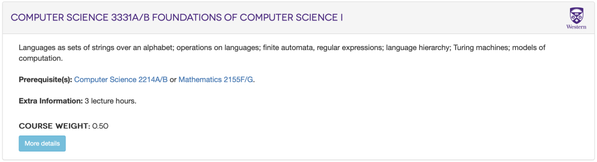
Traditional TCS courses

- Usually, a traditional TCS course builds upward from a very simple model of computation.
  - finite automata (= regular expressions)
  - pushdown automata (= context-free grammars)
  - linear-bounded automata
  - Turing machines
- Once we reach Turing machines, we can start to talk more about what computers *do* rather than what computers *are*.
  - decidable problems
  - undecidable problems

Traditional TCS courses

- Usually, a traditional TCS course builds upward from a very simple model of computation.
  - finite automata (= regular expressions)
  - pushdown automata (= context-free grammars)
  - linear-bounded automata
  - Turing machines
- Once we reach Turing machines, we can start to talk more about what computers *do* rather than what computers *are*.
  - decidable problems
  - undecidable problems
- Finally, once we have a collection of problems, we can start to draw connections between problems.
  - reductions
  - complexity theory
  - hardness and completeness

Traditional TCS courses



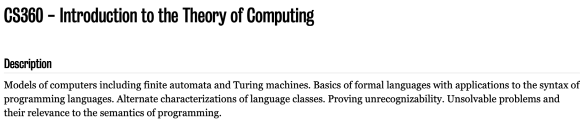
COMPUTER SCIENCE 3331A/B FOUNDATIONS OF COMPUTER SCIENCE I

Languages as sets of strings over an alphabet; operations on languages; finite automata, regular expressions; language hierarchy; Turing machines; models of computation.

Prerequisite(s): Computer Science 2214A/B or Mathematics 2155F/G.

Extra Information: 3 lecture hours.

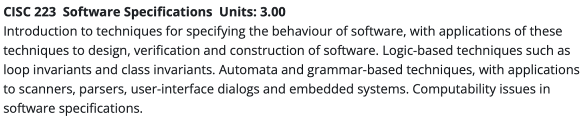
COURSE WEIGHT: 0.50



CS360 – Introduction to the Theory of Computing

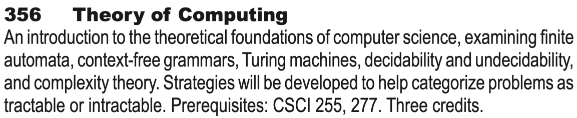
Description

Models of computers including finite automata and Turing machines. Basics of formal languages with applications to the syntax of programming languages. Alternate characterizations of language classes. Proving unrecognizability. Unsolvable problems and their relevance to the semantics of programming.



CISC 223 Software Specifications Units: 3.00

Introduction to techniques for specifying the behaviour of software, with applications of these techniques to design, verification and construction of software. Logic-based techniques such as loop invariants and class invariants. Automata and grammar-based techniques, with applications to scanners, parsers, user-interface dialogs and embedded systems. Computability issues in software specifications.



356 Theory of Computing

An introduction to the theoretical foundations of computer science, examining finite automata, context-free grammars, Turing machines, decidability and undecidability, and complexity theory. Strategies will be developed to help categorize problems as tractable or intractable. Prerequisites: CSCI 255, 277. Three credits.

What do students think about TCS?

Actual evaluation comments:

- "at the end of the semester I still can't understand the real life application of this subject, I never feel I can apply my any coding skills or even mathematical skills here."
- "I think it would help if the supplemental material was presented in a way that's a bit easier to visualize."
- "Maybe [add] some application based questions! I know it is theoretical computer science, but something to contextualize what we're learning."
- "This subject is mostly on theory, it would be better if add any practical examples or projects."

What do students think about TCS?

Actual evaluation comments:

- "I started the semester enrolled in 6 courses with the intention of doing a vibe check and cutting it down to 4. You passed the vibe check by asking if anyone had played Dwarf Fortress. I am not sure how repeatable this strategy is, but it worked."

What do students think about TCS?

Actual evaluation comments:

- "I started the semester enrolled in 6 courses with the intention of doing a vibe check and cutting it down to 4. You passed the vibe check by asking if anyone had played Dwarf Fortress. I am not sure how repeatable this strategy is, but it worked."

Key takeaway:

- Talk about **Dwarf Fortress**

What do academics think about TCS?

- At STOC 1996, a group of researchers presented a report assessing the state of the field and making recommendations for the future.

In order for [the Theory of Computing] to prosper in the coming years, it is essential to strengthen our communication with the rest of computer science and with other disciplines, and to increase our impact on key application areas.

A. V. Aho et al. “Theory of Computing: Goals and Directions”. 1996.

What do academics think about TCS?

- At STOC 1996, a group of researchers presented a report assessing the state of the field and making recommendations for the future.

In order for [the Theory of Computing] to prosper in the coming years, it is essential to strengthen our communication with the rest of computer science and with other disciplines, and to increase our impact on key application areas.

- This report proved to be contentious, and some other researchers quickly drafted a rebuttal.

In order for the Theory of Computing to prosper in the future it is essential that [it] attracts the same calibre of researchers, that Theoretical Computer Scientists concentrate their research efforts in Theory of Computing and that they enjoy the freedom to do so.

O. Goldreich and A. Wigderson. "Theory of Computing: A Scientific Perspective". 1996.

What do academics think about TCS?

- At STOC 1996, a group of researchers presented a report assessing the state of the field and making recommendations for the future.

In order for [the Theory of Computing] to prosper in the coming years, it is essential to strengthen our communication with the rest of computer science and with other disciplines, and to increase our impact on key applicatio

Why not both?

- This report proved to be contentious, and some other researchers quickly drafted a rebuttal.

In order for the Theory of Computing to prosper in the future it is essential that [it] attracts the same calibre of researchers, that Theoretical Computer Scientists concentrate their research efforts in Theory of Computing and that they enjoy the freedom to do so.

O. Goldreich and A. Wigderson. "Theory of Computing: A Scientific Perspective". 1996.

What is to be done?

The *real* key takeaways:

- TCS is (admittedly) a dry subject to students—but it doesn't have to be this way.
- We should appreciate TCS as a subject in itself.
  - There's nothing wrong with appreciating the fascinating results as they are
  - That's what attracted me to the field!
- At the same time, we should be mindful that TCS is more than just a bunch of theorems and proofs.
  - TCS is a broadly applicable method of reasoning about how to solve/compute problems
  - And it has a rich history!

Table of Contents

We’re *really* learning TCS. . .

- Most traditional TCS courses follow a rigid “definition–theorem–proof” lecture style.
  - Probably the main source of student consternation!
- There’s a myriad of interesting examples, applications, and anecdotes that can help reveal to students why they’re *really* learning TCS.
  - How can we use TCS concepts to do something?
  - How can we use TCS concepts to know our limits?
  - Can we have fun with this?

Table of Contents

. . . To know how to do something

Background:

A *one-dimensional word* is a string of symbols from some alphabet Σ. For example, if $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, one possible 1D word is $\mathtt{abbaba}$.

A *two-dimensional word* is an array of symbols from some alphabet Σ. For example, if $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, one possible 2D word is:

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

- Words (and collections of words, called *languages*) form the foundations of much of computer science.
- All data we can store in a computer's memory/disk is essentially a binary word!

Words

- Some data we care about exhibits certain special properties.
- For example, *periodicity*: the property of some segment of data repeating as a pattern.

Example:

A periodic 1D word over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ is abbaabbaabba. The pattern is abba, repeated three times.
A periodic 2D word over $\Sigma = \{0, 1\}$ is

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

The pattern is $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, repeated 2×3 times.

Words

- Some data we care about exhibits certain special properties.
- For example, *periodicity*: the property of some segment of data repeating as a pattern.
- Periodicity is a fundamental notion in formal languages and combinatorics on words.
- Patterns show up everywhere in the real world!
  - E.g., biology, genomics, astronomy, meteorology, earth sciences, oceanography, etc.

Words

- Some data we care about exhibits certain special properties.
- For example, *periodicity*: the property of some segment of data repeating as a pattern.
- Periodicity is a fundamental notion in formal languages and combinatorics on words.
- Patterns show up everywhere in the real world!
  - E.g., biology, genomics, astronomy, meteorology, earth sciences, oceanography, etc.
- For this reason, many researchers have come up with efficient algorithms to detect periodicity.
  - 1D: $O(n)$ time, word of length n*
  - 2D: $O(mn)$ time, word of size $m \times n$ (Smith et al., 2017)

*R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In Proc. of FOCS 1999, pages 596–604. 1999.

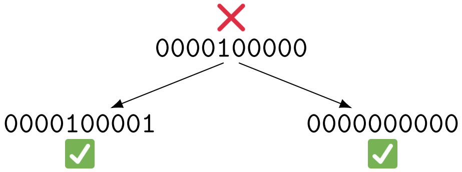
Words and error correction

- Of course, data storage isn't perfect.
- Data can be corrupted with *errors*.
- How can we *detect* and *correct* such errors if they occur?
  - In the context of periodicity, how do we *recover the pattern* from an erroneous word?

Words and error correction

- Of course, data storage isn't perfect.
- Data can be corrupted with *errors*.
- How can we *detect* and *correct* such errors if they occur?
    - In the context of periodicity, how do we *recover the pattern* from an erroneous word?

In one dimension:



- Even with only one error, there could be many candidates.
- However, we just need to check $O(\log(n))$ possibilities.

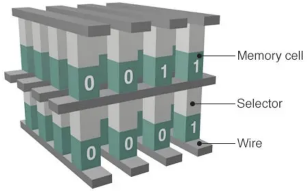
A. Amir et al. Cycle detection and correction. *ACM Transactions on Algorithms* 9(1):13:1–13:20, 2012.

Words and error correction

- Of course, data storage isn't perfect.
- Data can be corrupted with *errors*.
- How can we *detect* and *correct* such errors if they occur?
  - In the context of periodicity, how do we *recover the pattern* from an erroneous word?

In more than one dimension?

- Yes, multidimensional storage media exists!
  - E.g., Intel's *3D XPoint*



Words and error correction in 2D

- Suppose our data has size $m \times n$; the pattern has size $p \times q$.
- As long as the number of errors is bounded above by $\left\lfloor \frac{1}{2+\epsilon} \left\lfloor \frac{n}{p} \right\rfloor \left\lfloor \frac{m}{q} \right\rfloor \right\rfloor$, we need only check $O(\log(mn))$ possibilities!
- How do we correct errors?
  - Perform Karp–Miller–Rosenberg (KMR) renaming to assign "labels" to unique subwords
  - Find a candidate/approximate pattern having size $p \times q$
  - Verify that this candidate/approximate pattern is primitive (not periodic)
  - Compute the Hamming distance between pattern and data; if distance is close enough, correct the error

A. Amir et al. Multidimensional period recovery. *Algorithmica* 84:1490–1510, 2022.

Words and error correction in dD

- The same error correction procedure can be extended to $d \geq 3$ dimensions.
- As we might expect, we need to check $O(\log^d(N))$ possibilities.
    - Here, $N = \prod_{i=1}^{d} n_i$
- With a small tweak, however, it is possible to make the number of possibilities independent of the dimensions; i.e., just $O(\log(N))$ possibilities!

A. Amir et al. Multidimensional period recovery. *Algorithmica* 84:1490–1510, 2022.

Words and error correction in dD

- The same error correction procedure can be extended to $d \geq 3$ dimensions.
- As we might expect, we need to check $O(\log^d(N))$ possibilities.
  - Here, $N = \prod_{i=1}^{d} n_i$
- With a small tweak, however, it is possible to make the number of possibilities independent of the dimensions; i.e., just $O(\log(N))$ possibilities!
- In practice, we must also account for $d \cdot 2^d$ overhead.
- But still, simple techniques (periodicity checking) allow us to do crucial real-world tasks!

A. Amir et al. Multidimensional period recovery. *Algorithmica* 84:1490–1510, 2022.

Table of Contents

. . . To understand what we can't do

Background:

The *Boolean satisfiability problem* (or SAT) is defined as follows:

SATISFIABILITY

Given a Boolean logic formula in conjunctive normal form, does there exist an assignment of true/false values to each variable that makes the formula true?

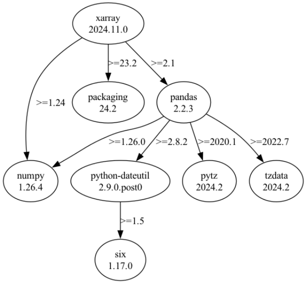
Conjunctive normal form means the formula consists of conjuncted clauses of disjuncted variables. For example:

$$\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_4) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_5}).$$

- There are many heuristic SAT-solving algorithms for practical problems (and competitions).
- However, no efficient algorithm is known to solve instances of SAT in general!

Packages and dependencies

- Many programmers rely on packages made by other programmers to perform common tasks.
  - E.g., if you're doing some kind of numerical computation in Python, you won't start from scratch…you'll `import numpy`
- These packages themselves don't always start from scratch; some packages depend on other packages.



An example of a dependency tree produced by `pipdeptree`

Resolving dependencies

- How might we automatically determine/resolve dependencies?

Resolving dependencies

- How might we automatically determine/resolve dependencies?
- Let's use Boolean logic: formulate the problem in terms of the SATISFIABILITY problem.

Resolving dependencies

- How might we automatically determine/resolve dependencies?
- Let's use Boolean logic: formulate the problem in terms of the SATISFIABILITY problem.

Example:

Suppose we want to install v1.0 of the `foo` package.

(`foo` 1.0)

Resolving dependencies

- How might we automatically determine/resolve dependencies?
- Let's use Boolean logic: formulate the problem in terms of the SATISFIABILITY problem.

Example:

`foo` v1.0 depends on the `bar` package: either v1.5 or v2.0.

$$(\texttt{foo}\ 1.0) \land (\neg\texttt{foo}\ 1.0 \lor \texttt{bar}\ 1.5 \lor \texttt{bar}\ 2.0)$$

Resolving dependencies

- How might we automatically determine/resolve dependencies?
- Let's use Boolean logic: formulate the problem in terms of the SATISFIABILITY problem.

Example:

`bar` v1.5 depends on v3.4 of the `baz` package.

$$(\texttt{foo}\ 1.0) \land (\neg\texttt{foo}\ 1.0 \lor \texttt{bar}\ 1.5 \lor \texttt{bar}\ 2.0) \land (\neg\texttt{bar}\ 1.5 \lor \texttt{baz}\ 3.4)$$

Resolving dependencies

- How might we automatically determine/resolve dependencies?
- Let's use Boolean logic: formulate the problem in terms of the SATISFIABILITY problem.

Example:

bar v2.0 depends on v3.6 of the baz package.

$$(\texttt{foo}\ 1.0) \wedge (\neg\texttt{foo}\ 1.0 \vee \texttt{bar}\ 1.5 \vee \texttt{bar}\ 2.0) \wedge (\neg\texttt{bar}\ 1.5 \vee \texttt{baz}\ 3.4) \wedge (\neg\texttt{bar}\ 2.0 \vee \texttt{baz}\ 3.6)$$

Resolving dependencies

- How might we automatically determine/resolve dependencies?
- Let's use Boolean logic: formulate the problem in terms of the SATISFIABILITY problem.

Example:

We can't install both `bar` v1.5 and v2.0 at the same time.

$$(\texttt{foo}\ 1.0) \land (\neg\texttt{foo}\ 1.0 \lor \texttt{bar}\ 1.5 \lor \texttt{bar}\ 2.0) \land (\neg\texttt{bar}\ 1.5 \lor \texttt{baz}\ 3.4)$$
$$\land\ (\neg\texttt{bar}\ 2.0 \lor \texttt{baz}\ 3.6) \land (\neg\texttt{bar}\ 1.5 \lor \neg\texttt{bar}\ 2.0)$$

Resolving dependencies

- How might we automatically determine/resolve dependencies?
- Let's use Boolean logic: formulate the problem in terms of the SATISFIABILITY problem.

Example:

We can't install both `baz` v3.4 and v3.6 at the same time.

$$(\texttt{foo}\ 1.0) \wedge (\neg\texttt{foo}\ 1.0 \vee \texttt{bar}\ 1.5 \vee \texttt{bar}\ 2.0) \wedge (\neg\texttt{bar}\ 1.5 \vee \texttt{baz}\ 3.4)$$
$$\wedge\ (\neg\texttt{bar}\ 2.0 \vee \texttt{baz}\ 3.6) \wedge (\neg\texttt{bar}\ 1.5 \vee \neg\texttt{bar}\ 2.0) \wedge (\neg\texttt{baz}\ 3.4 \vee \neg\texttt{baz}\ 3.6)$$

Resolving dependencies

- How might we automatically determine/resolve dependencies?
- Let's use Boolean logic: formulate the problem in terms of the SATISFIABILITY problem.

Example:

The baz developers just took down v3.6 due to a bug.

$$(\texttt{foo } 1.0) \land (\neg\texttt{foo } 1.0 \lor \texttt{bar } 1.5 \lor \texttt{bar } 2.0) \land (\neg\texttt{bar } 1.5 \lor \texttt{baz } 3.4)$$
$$\land\ (\neg\texttt{bar } 2.0 \lor \texttt{baz } 3.6) \land (\neg\texttt{bar } 1.5 \lor \neg\texttt{bar } 2.0) \land (\neg\texttt{baz } 3.4 \lor \neg\texttt{baz } 3.6)$$
$$\land\ (\neg\texttt{baz } 3.6)$$

Resolving dependencies

- How might we automatically determine/resolve dependencies?
- Let's use Boolean logic: formulate the problem in terms of the SATISFIABILITY problem.

Example:

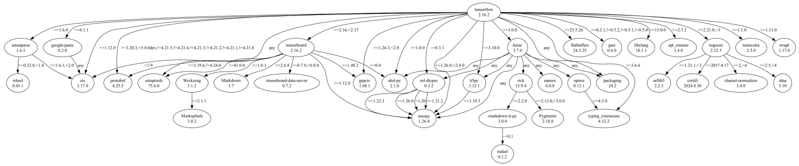
So, how do we solve this dependency problem?

$$(\texttt{foo }1.0) \wedge (\neg\texttt{foo }1.0 \vee \texttt{bar }1.5 \vee \texttt{bar }2.0) \wedge (\neg\texttt{bar }1.5 \vee \texttt{baz }3.4)$$
$$\wedge\, (\neg\texttt{bar }2.0 \vee \texttt{baz }3.6) \wedge (\neg\texttt{bar }1.5 \vee \neg\texttt{bar }2.0) \wedge (\neg\texttt{baz }3.4 \vee \neg\texttt{baz }3.6)$$
$$\wedge\, (\neg\texttt{baz }3.6)$$

- `baz` 3.6 = False (clause 7; we can't get this package)
- `bar` 2.0 = False (clause 4, since we can't get `baz` 3.6)
- `bar` 1.5 = True (clause 2, since we need some `bar` package)
- `baz` 3.4 = True (clause 3, since `bar` 1.5 depends on this)
- `foo` 1.0 = True (clause 1; we're installing this package!)

Resolving dependencies

- Unfortunately, the example we just saw was an easy one.
- Some package dependency trees are. . . less easy to follow.



- Since we have no efficient algorithm for SATISFIABILITY, there is no efficient way in general to resolve dependencies!
- In practice, package managers like `pip` use *backtracking algorithms* and *heuristics*.
  - Still, there are some cases where untangling dependencies will take a *long* time!

`https://pip.pypa.io/en/stable/topics/more-dependency-resolution/`

. . . To understand what we can't do

Background:

The *halting problem* is defined as follows:

HALT

Given a description of a program and an input to that program, does the program halt when run on that input?

The halting problem is *undecidable*; that is, there does not exist an algorithm that always returns a yes/no answer for every pair of program and input.

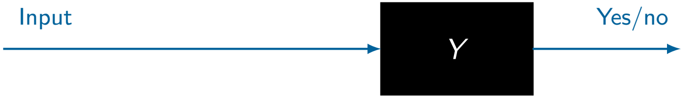
- The halting problem is one of the most famous problems in all of computer science.
- Note that while the problem is *undecidable*, this doesn't mean it's *impossible*.
    - If a program halts on an input, we can definitely detect that; if it gets caught in an infinite loop, we'll never get an answer

Undecidable problems

- If we want to show that other decision problems (i.e., "yes/no" problems) are undecidable, then we can use something known as a *reduction*.
- How does a reduction work?
  - Let's suppose we know problem X is undecidable
  - We want to learn more about a new problem Y

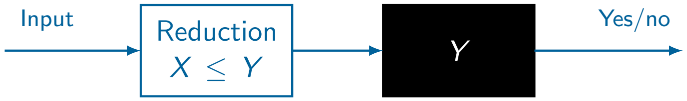
Undecidable problems

- If we want to show that other decision problems (i.e., "yes/no" problems) are undecidable, then we can use something known as a *reduction*.
- How does a reduction work?
  - Let's suppose we know problem X is undecidable
  - We want to learn more about a new problem Y
  - If we assume that we can write an algorithm to decide Y...



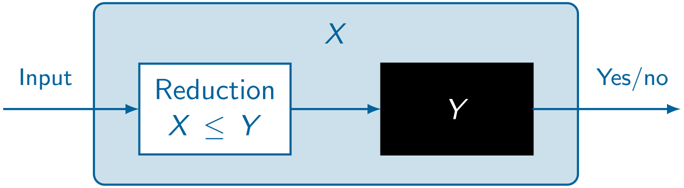
Undecidable problems

- If we want to show that other decision problems (i.e., "yes/no" problems) are undecidable, then we can use something known as a *reduction*.
- How does a reduction work?
  - Let's suppose we know problem X is undecidable
  - We want to learn more about a new problem Y
  - If we assume that we can write an algorithm to decide Y...
  - ...and show that we can turn X into Y...



Undecidable problems

- If we want to show that other decision problems (i.e., "yes/no" problems) are undecidable, then we can use something known as a *reduction*.
- How does a reduction work?
  - Let's suppose we know problem X is undecidable
  - We want to learn more about a new problem Y
  - If we assume that we can write an algorithm to decide $Y \dots$
  - $\dots$ and show that we can turn X into $Y \dots$
  - $\dots$ then we just created an algorithm to "decide" X!
  - But this can't be, so Y must be undecidable like X



Undecidable problems

- There are all kinds of undecidable problems beyond HALT.

Undecidable problems

- There are all kinds of undecidable problems beyond $\textsc{Halt}$.
- Some of these problems are computational:
  - Given a Turing machine and an input word, does it accept that word?
  - Given a second-order λ-calculus expression, can we detect its type?

Undecidable problems

- There are all kinds of undecidable problems beyond HALT.
- Some of these problems are computational:
  - Given a Turing machine and an input word, does it accept that word?
  - Given a second-order λ-calculus expression, can we detect its type?
- Some of these problems are mathematical:
  - Given a set of $n \times n$ integral matrices, can we obtain the zero matrix by taking some product of matrices from this set?
  - Given two finite group presentations, are these groups isomorphic?

Undecidable problems

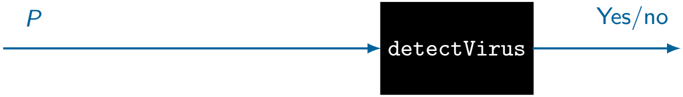
- There are all kinds of undecidable problems beyond HALT.
- Some of these problems are computational:
  - Given a Turing machine and an input word, does it accept that word?
  - Given a second-order λ-calculus expression, can we detect its type?
- Some of these problems are mathematical:
  - Given a set of $n \times n$ integral matrices, can we obtain the zero matrix by taking some product of matrices from this set?
  - Given two finite group presentations, are these groups isomorphic?
- Some of these problems are just plain interesting:
  - Does one player have a winning strategy in *Magic: The Gathering*?
  - **Can we detect a virus in a computer file?**

Virus detection

- Virus detecting software exists, but it suffers occasionally from false positives/false negatives.
- Why don't we have 100% flawless virus detection software?

Virus detection

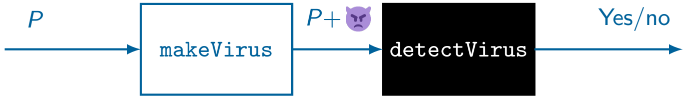
- Virus detecting software exists, but it suffers occasionally from false positives/false negatives.
- Why don't we have 100% flawless virus detection software?
- Let's suppose we do (and see what goes wrong)!
- Assume we have a function `detectVirus`(P) that takes as input a program's source code P and returns "yes" if it's a virus, and "no" otherwise.



D. Evans. On the impossibility of virus detection. 2017.

Virus detection

- Let's write a new function `makeVirus`(P) that takes a program's source code P and appends "`sudo rm -rf /*`" to the end of the program, then pass this to `detectVirus`.
- If `detectVirus` returns "yes", then P (whatever it was) must have finished running before executing `rm -rf`.
- If `detectVirus` returns "no", then we never reached `rm -rf`, and so P must not have finished running.

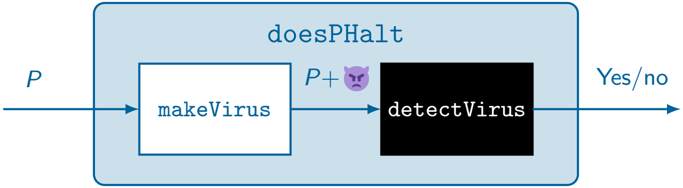


D. Evans. On the impossibility of virus detection. 2017.

Virus detection

- But wait. . . what did we just do?
- We implicitly decided whether or not P halts!
  - We only reach the bad code inserted by `makeVirus` if P halts
  - Thus, the answer from `detectVirus` will be the same as the answer for `doesPHalt`
- Since HALT is undecidable, so is virus detection!

(*Note.* It's possible that P may already contain a virus and also never halt. This is easy for us to fix: just modify P so it's incapable of infecting anything on the computer itself; e.g., redirect P's output to a VM, etc.)



D. Evans. On the impossibility of virus detection. 2017.

Table of Contents

. . . Because it's fun!

Background:

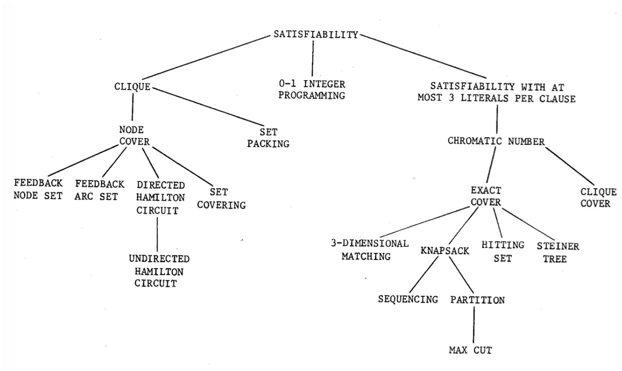
A problem is *NP-complete* if:

- using a deterministic computer, we can efficiently *verify* whether a solution is correct (i.e., *polynomial time* in the input size); and
- every other problem that can be verified in polynomial time can be transformed into this problem efficiently (i.e., via a *polynomial-time reduction*).

- The first criterion indicates that the problem is in NP.
  - It's "easy" to *check* a solution, but "difficult" to *find* a solution
- The second criterion indicates that the problem is *NP-hard*.
  - Every other problem in NP can be represented as an instance of this problem
  - Alternatively, this problem is at least as difficult to *solve* as any other problem in NP

NP-complete problems

- There are many, many examples of NP-complete problems!



- However, some of these problems aren't exactly run-of-the-mill. . .

R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.

Snow removal

February 4, 2024:

My car, pre-shovelling

Snow removal

February 4, 2024:

My car, mid-shovelling

Snow removal

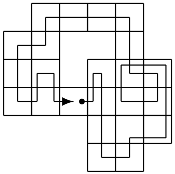
- Could this job have been easier if I had a snowblower?
- Preferably, an automatic snowblower that routes itself around my driveway?

Snow removal

- Could this job have been easier if I had a snowblower?
- Preferably, an automatic snowblower that routes itself around my driveway?
- Let's formulate this as a problem:

Snowblowing
Does there exist a path to allow a snowblower to clear a given polygonal region without piling snow deeper than some depth d within this region?

Example:

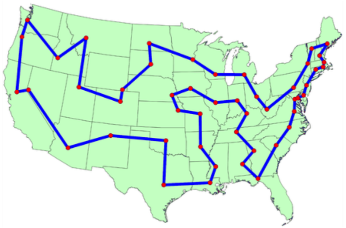


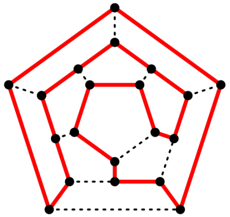
Snow removal

- This looks quite similar to two very well-studied (and NP-complete) problems!

TRAVELLING-SALESPERSON
Does there exist a route of length at most ℓ that visits a list of cities and also returns you to your home city?

HAMILTONIAN-CYCLE
Does there exist a path in a graph that visits each vertex exactly once and returns to the starting vertex?





Snow removal

- It's possible to reduce the Hamiltonian-Cycle problem to the Snowblowing problem.
- This means that the Snowblowing problem is NP-complete!
  - I.e., we can easily *check* whether a snowblower path clears the region, but it's much harder to *find* a path from scratch
- Does this mean all is lost?

E. M. Arkin et al. The snowblower problem. *Computational Geometry* 44(8):370–384, 2011.

Snow removal

- It's possible to reduce the HAMILTONIAN-CYCLE problem to the SNOWBLOWING problem.
- This means that the SNOWBLOWING problem is NP-complete!
    - I.e., we can easily *check* whether a snowblower path clears the region, but it's much harder to *find* a path from scratch
- Does this mean all is lost?
- Actually, no: for the simplest variant of the SNOWBLOWING problem, there exists an 8-*approximation algorithm*.
    - This means that we can find a path that is at most *eight* times as long as the optimal path

E. M. Arkin et al. The snowblower problem. *Computational Geometry* 44(8):370–384, 2011.

. . . Because it's fun!

Background:

A problem is *PSPACE-complete* if:

- we can *solve* the problem using a "reasonable" amount of memory (i.e., *polynomial space* in the input size); and
- every other problem that can be solved using polynomial space can be transformed into this problem efficiently (i.e., PSPACE-hard via a *polynomial-time reduction*).

- Note that we're changing our perspective from *time* to *space*.
- We know that NP $\subseteq$ PSPACE, since there exist problems whose solutions can be found using more than polynomial time, but only polynomial space.

PSPACE-complete problems

- There are many problems in CS/mathematics that are PSPACE-complete.
- Polynomial space also frequently arises when we play games!

GEOGRAPHY

HEX

OTHELLO

CHESS

GO

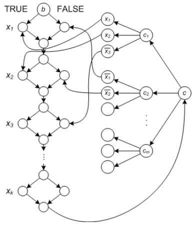
(Actually PSPACE-hard)

Proving PSPACE-completeness

- To obtain the reduction that is vital to proving a problem is PSPACE-complete, one often uses *gadgets*.
  - Gadgets can also be used in NP-completeness proofs
- A gadget is like a “piece” of one problem; multiple “pieces” can be assembled into an instance of another problem we’re reducing to.

Example:

We can reduce to GEOGRAPHY from another PSPACE-complete problem called FORMULA-GAME using the collection of gadgets shown at right.



Proving PSPACE-completeness

- If gadgets are like “pieces” of problems, where might these gadgets appear in something we’re familiar with?

Video games

- If gadgets are like “pieces” of problems, where might these gadgets appear in something we’re familiar with?
- How about video games? (Specifically, game levels/objects)

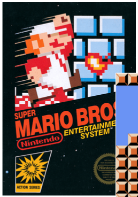
Video games

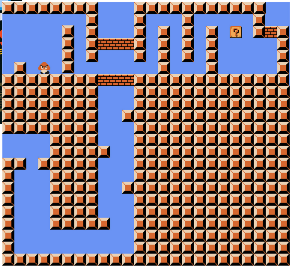
- If gadgets are like "pieces" of problems, where might these gadgets appear in something we're familiar with?
- How about video games? (Specifically, game levels/objects)

Super Mario Bros.:

Deciding whether we can get from the start of a level to the goal is NP-complete

(Reduce from $3\mathrm{SAT}$)





G. Aloupis et al. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science* 586:135–160, 2015.

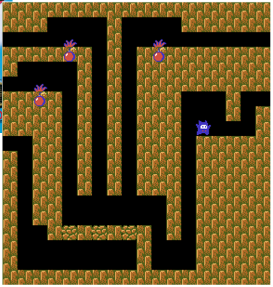
Video games

- If gadgets are like "pieces" of problems, where might these gadgets appear in something we're familiar with?
- How about video games? (Specifically, game levels/objects)

Super Mario Bros. 2:

Deciding whether we can get from the start of a level to the goal is NP-complete

(Reduce from $\mathrm{3SAT}$)



G. Aloupis et al. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science* 586:135–160, 2015.

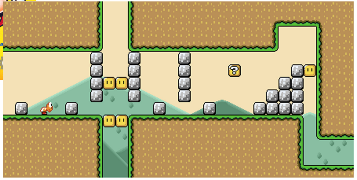
Video games

- If gadgets are like "pieces" of problems, where might these gadgets appear in something we're familiar with?
- How about video games? (Specifically, game levels/objects)

Super Mario Bros. 3:

Deciding whether we can get from the start of a level to the goal is NP-complete

(Reduce from 3SAT)



G. Aloupis et al. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science* 586:135–160, 2015.

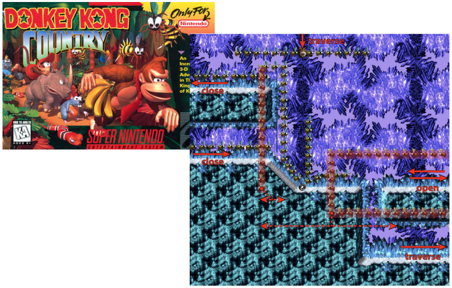
Video games

- If gadgets are like "pieces" of problems, where might these gadgets appear in something we're familiar with?
- How about video games? (Specifically, game levels/objects)

Donkey Kong Country:

Deciding whether we can get from the start of a level to the goal is PSPACE-complete

(Reduce from TQBF)



G. Aloupis et al. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science* 586:135–160, 2015.

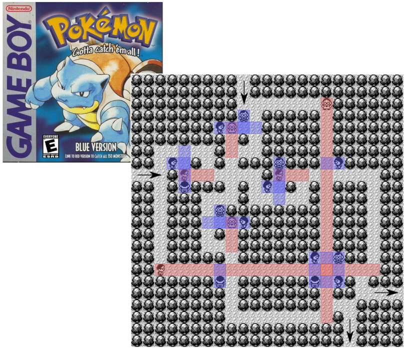
Video games

- If gadgets are like "pieces" of problems, where might these gadgets appear in something we're familiar with?
- How about video games? (Specifically, game levels/objects)

Pokémon:

Deciding whether we can reach a given target location from a given start location is PSPACE-complete

(Reduce from TQBF)



G. Aloupis et al. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science* 586:135–160, 2015.

Video games

- If gadgets are like "pieces" of problems, where might these gadgets appear in something we're familiar with?
- How about video games? (Specifically, game levels/objects)

And many others:

- Donkey Kong Country 2 and 3
- Legend of Zelda
  - A Link to the Past
  - Link's Awakening
  - Ocarina of Time
  - Majora's Mask
- Metroid

G. Aloupis et al. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science* 586:135–160, 2015.

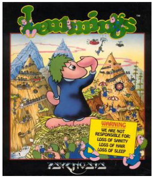
Video games

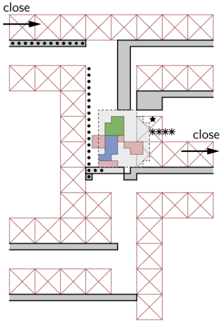
- If gadgets are like "pieces" of problems, where might these gadgets appear in something we're familiar with?
- How about video games? (Specifically, game levels/objects)

Lemmings:

Deciding whether we can save one Lemming using Builders and Bashers is PSPACE-complete

(Even harder if we try to maximize the number of Lemmings saved!)



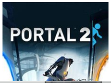
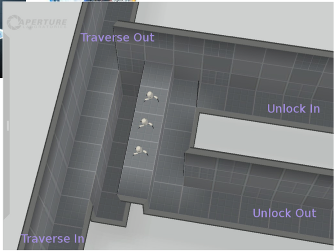


G. Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science* 586:120–134, 2015.

Video games

- If gadgets are like "pieces" of problems, where might these gadgets appear in something we're familiar with?
- How about video games? (Specifically, game levels/objects)

Portal 1 and 2:
Deciding whether we can get from the start of a level to the goal is NP-hard
(if we just have turrets)

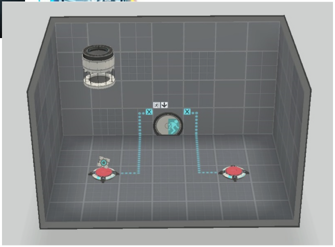
E. D. Demaine et al. The computational complexity of Portal and other 3D video games. In *Proc. of FUN 2018*, pages 19:1–19:22, 2018.

Video games

- If gadgets are like "pieces" of problems, where might these gadgets appear in something we're familiar with?
- How about video games? (Specifically, game levels/objects)

Portal 1 and 2:

Deciding whether we can get from the start of a level to the goal is PSPACE-complete (if we have cubes, weighted buttons, and doors)



E. D. Demaine et al. The computational complexity of Portal and other 3D video games. In *Proc. of FUN 2018*, pages 19:1–19:22, 2018.

Table of Contents

How do we make learning TCS better?

- Let's return to our two key takeaways:
  - We should appreciate TCS as a subject in itself
  - We should be mindful that TCS is more than just a bunch of theorems and proofs

How do we make learning TCS better?

- Let's return to our two key takeaways:
  - We should appreciate TCS as a subject in itself
  - We should be mindful that TCS is more than just a bunch of theorems and proofs
- The majority of TCS courses seem to lean toward the former direction.
  - I.e., the rigid "definition–theorem–proof" lecture style
- The majority of students seem to want to move in the latter direction.
  - "I still can't understand the real life application of this subject"
  - "something to contextualize what we're learning"
  - "add any practical examples or projects"

How do we make learning TCS better?

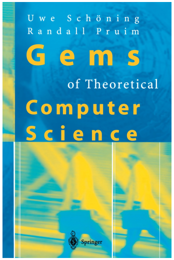
- Let's return to our two key takeaways:
  - We should appreciate TCS as a subject in itself
  - We should be mindful that TCS is more than just a bunch of theorems and proofs
- The majority of TCS courses seem to lean toward the former direction.
  - I.e., the ure style
- The majorit ve in the latter direction.
  - "I still can't understand the real life application of this subject"
  - "something to contextualize what we're learning"
  - "add any practical examples or projects"

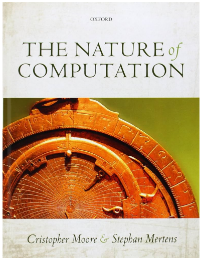
Why not both?

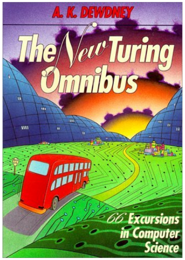
How do we make learning TCS better?

- Presenting the theorems, the proofs, etc., is inescapable in any TCS course. (Sorry, students.)
  - Abstraction and mathematical reasoning are vital skills for any computer scientist
  - It's important to get students comfortable with this mode of thinking early
- But we should couch these results in real-world applications and notions with which students are familiar.
  - Don't "meet the students where they are", but meet them halfway
  - If you can present reductions and completeness in terms of Pokémon, *do it!*

Some inspirational literature







Thank you!

StFXCS

FLAReLab

Formal Languages and Automata Research Lab

St. Francis Xavier University

`taylorjsmith.xyz` `taylorjsmith.xyz/flarelab/`