Practical approaches to teaching the CS theory module:

nondecision problems and real computer programs

John MacCormick

Dickinson College and UEA School of Computing Sciences

Overview

1. Main talk (practical approaches to the CS theory module)
    - About 35 minutes; real-time questions and interaction are welcomed
2. Other interests (hope to pursue some of these in the next two years)
    - About 5 minutes: computer vision, machine learning, distributed systems, CS education, public understanding of computer science
3. Questions and discussion

Understanding the audience

- As an undergraduate, did you:
  - Take a module that emphasized the distinction between polynomial time and exponential time algorithms (more formally, P vs EXP)?
  - Take a module that explored NP and NP-completeness?
  - Take a module that discussed the equivalence, in terms of time complexity, of all “reasonable” computational models (up to polynomial factors)?
  - Take a module that covered undecidability, including proofs that certain problems (e.g. the halting problem) are undecidable?

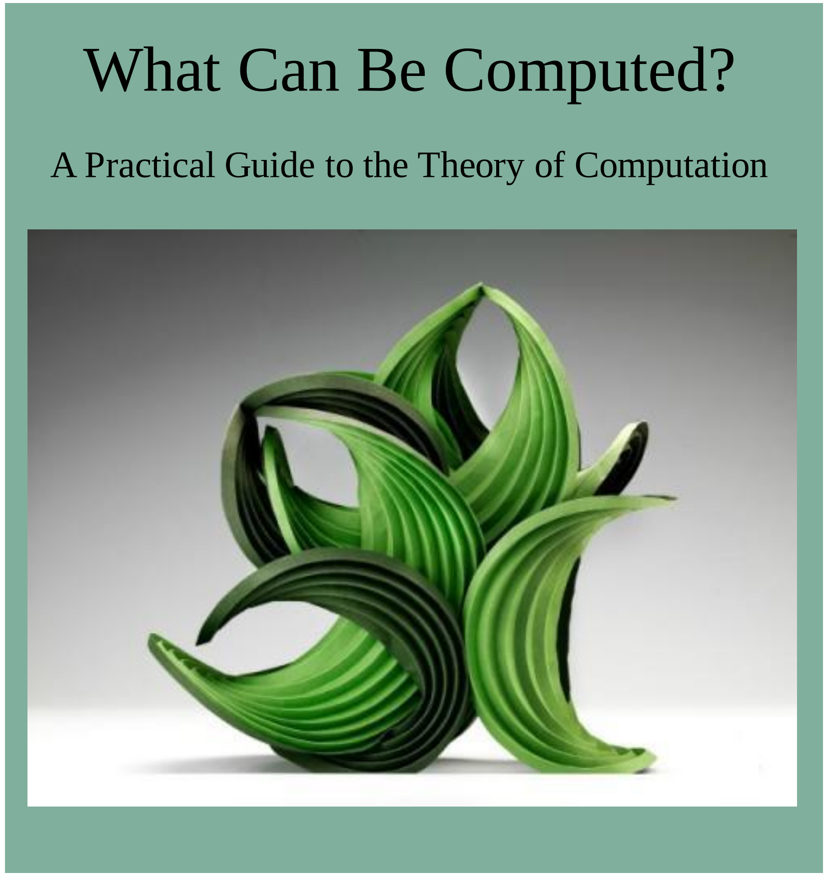
Understanding the audience, part II

- As an *instructor*, have you:
  - *Taught* a module that emphasized the distinction between polynomial time and exponential time algorithms (more formally, P vs EXP)?
  - *Taught* a module that explored NP and NP-completeness?
  - *Taught* a module that discussed the equivalence, in terms of time complexity, of all "reasonable" computational models (up to polynomial factors)?
  - *Taught* a module that covered undecidability, including proofs that certain problems (e.g. the halting problem) are undecidable?

The CS “theory” module? What theory module?

- Most computer science programs in the UK and US offer a “theory” module
  - many require it
- Typical topics drawn from:
  - automata theory (dfas, pdas, regular grammars, cfgs, Turing machines)
  - computability theory (existence of undecidable problems e.g. halting problem, Turing reductions, Rice’s theorem)
  - complexity theory (P, NP, EXP, NP-completeness, Cook-Levin theorem, polynomial time reductions)
- Sometimes the complexity theory is included as part of an advanced algorithms module

High-level point of the talk: the theory module can be taught in a practical and accessible way



vapourware version of front cover
(Erik Demaine origami)

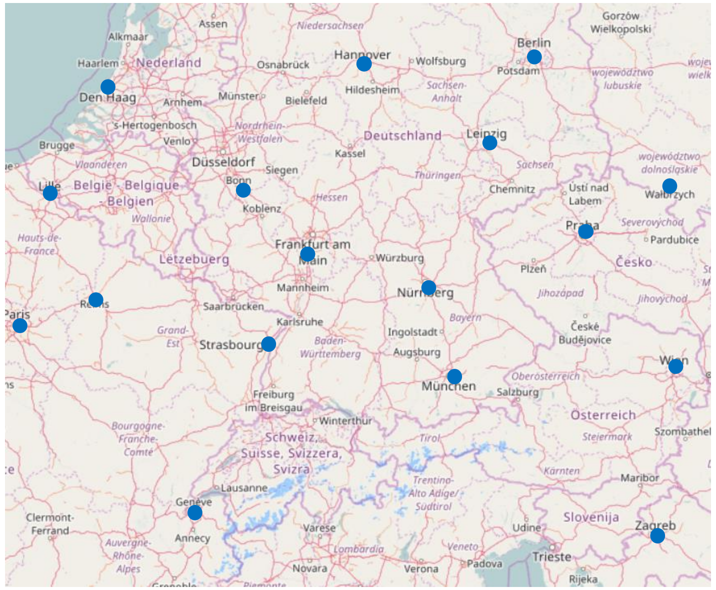
- new undergraduate textbook from Princeton University Press, available February 2018
- Key features:
  - Python programs as the main computational model
  - Focuses on nondecision problems

Technical content of today's talk

Next few slides: informal overview of the key distinction between decision and nondecision problems

Which is more “useful”: program *A* or program *B*?

Input: Input to both programs is a roadmap and a list of cities:



Output:

Program *A* outputs

- “**yes**” if there’s a driving route that visits each city and takes less than 100 hours
- “**no**” otherwise

Program *B* outputs

- **a description of a suitable route** if there’s a driving route that visits each city and takes less than 100 hours
- “**no**” otherwise

Which is more relevant for teaching: program *A* or program *B*?

Program *A* outputs

- "`yes`"
- "`no`"

Program *B* outputs

- **a description of a suitable route**
- "`no`"

Which is more relevant for teaching: program *A* or program *B*?

Program *A* outputs { "`yes`" / "`no`" }

- *Decision problem*.

Program *B* outputs { **a description of a suitable route** / "`no`" }

- *Nondecision problem*.

Which is more relevant for teaching: program *A* or program *B*?

Program *A* outputs { "`yes`" / "`no`" }

- *Decision problem*.
- Existing theory-of-computation modules usually focus on decision problems.

Program *B* outputs { **a description of a suitable route** / "`no`" }

- *Nondecision problem*.

Which is more relevant for teaching: program *A* or program *B*?

Program *A* outputs { “`yes`” / “`no`”

- *Decision problem*.
- Existing theory-of-computation modules usually focus on decision problems.

Program *B* outputs { **a description of a suitable route** / “`no`”

- *Nondecision problem*.

- This talk points to a way to teach the theory-of-computation module using nondecision problems.
- Students may achieve better learning because the content is perceived as relevant and practical.

We consider only a *novice audience*

- Novice audience $\equiv$ undergraduate students who are seeing computability and complexity theory for the first time
- Experienced practitioners know that decision programs can often be converted to equivalent non-decision programs with only a logarithmic increase in running time.
- Therefore, experienced practitioners don’t care if we restrict attention to decision problems
- But for the *novice audience*, a program that outputs only a single bit may appear abstruse, irrelevant, and impractical

Conclusion: For the novice audience, start the module with nondecision problems

Advantages of decision problems	Advantages of nondecision problems
Some definitions and proofs are more concise	Solutions are perceived as more meaningful and useful by the novice audience
Almost all existing literature focuses on decision problems	Sometimes, the nondecision variant of a problem has important special properties (e.g. factoring)

Conclusion:

- No clear-cut winner.
- We recommend using nondecision problems for most of the module, then transitioning to decision problems for advanced topics. Specifically:
  - Nondecision problems for decidability, P, EXP, and NP
  - Decision problems for NP-completeness

The talk could end here. The remainder provides additional detail.

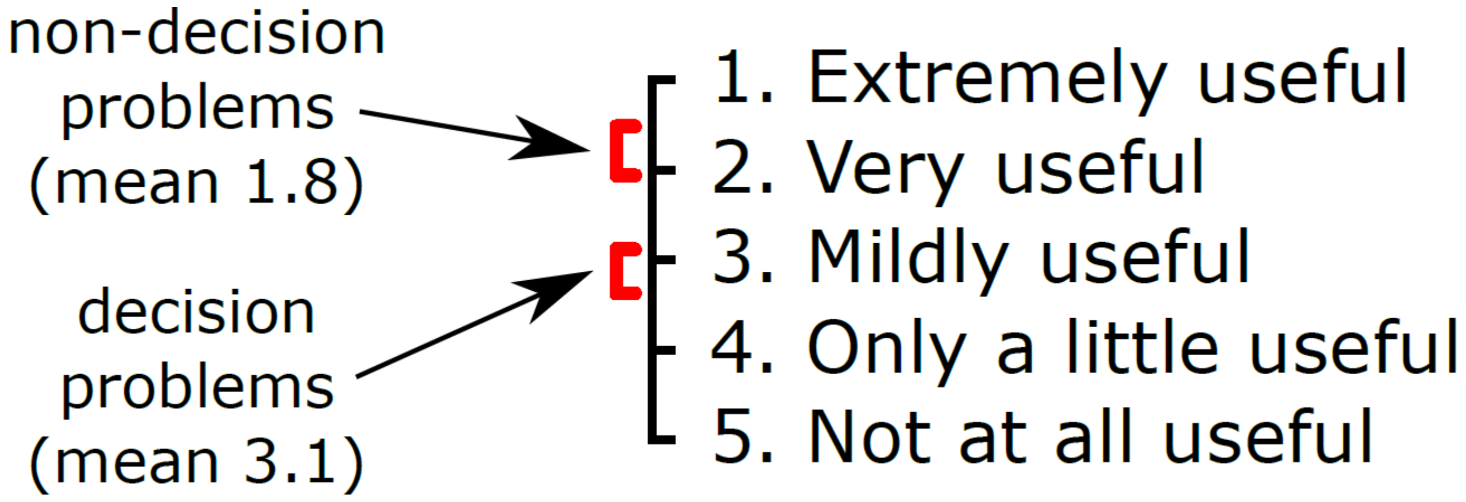
Remainder of the talk

1. Empirical evidence of student perceptions favoring nondecision problems
2. Technical details of how to teach the content using nondecision problems
   a) Definitions, including *formal languages* vs *computational problems*
   b) Computability
   c) Complexity

A survey of computer science students gathered empirical evidence

- 41 computer science students given descriptions of four computer programs
  - The programs solve decision and non-decision variants of two different problems (TSP and knapsack)
- rate “usefulness” from 1 (extremely useful) to 5 (not at all useful)

Programs that solve nondecision problems are perceived as much more "useful" by the novice audience



- The difference has overwhelming statistical significance
  - Wilcoxon signed-rank test has $p < 10^{-11}$
- The effect size is also substantial
  - Additional tests show the effect size exceeds difference between "very useful" and "mildly useful"

Educational theory implies that perceived usefulness will lead to improved outcomes

- Education researchers have demonstrated that effectiveness of learning is enhanced when concepts are perceived as *useful* or *applicable*
  - See e.g. L. D. Fink, *Creating significant learning experiences: An integrated approach to designing college courses* (2013)
- Therefore, we conclude the use of nondecision problems in the CS theory course should improve learning outcomes

We have not attempted to measure improved learning outcomes directly. I welcome suggestions on how to do that!

Remainder of the talk

1. Empirical evidence of student perceptions favoring nondecision problems
2. Technical details of how to teach the content using nondecision problems
   a) Definitions, including *formal languages* vs *computational problems*
   b) Computability
   c) Complexity

Details of the traditional approach

- ***alphabet*** $\equiv$ finite set of symbols, denoted Σ
- ***string*** $\equiv$ finite sequence of symbols
- set of all possible strings on Σ is denoted Σ^*
- ***language*** or ***formal language*** $\equiv$ subset of Σ^*
- Given Turing machine M with input string s, we say
  - M ***accepts*** s if it terminates in an accepting state
  - M ***rejects*** s if it terminates in any other state
  - but remember the machine may not terminate, so it could neither accept nor reject
- Machine M ***decides*** language L if
  - M accepts all $s \in L$ and rejects all $s \notin L$

so it had better terminate on all inputs!

What is the connection between "deciding a language" and "solving a problem"?

- For decision problems, these concepts are equivalent
- Example: Hamilton cycle
  - asks the yes/no question "does this graph have a Hamilton cycle?"
  - e.g. the string $s =$ "`a,b b,c c,a`" is a *positive instance*,
    but $s' =$ "`a,b b,c`" is a *negative instance*
- Let language L be the set of strings that are positive instances
- Then a Turing machine that decides L implicitly answers the question "does this graph have a Hamilton cycle?"

$D_L =$ "is string s in language L?"

$L_D =$ set of strings that are positive instances of D

Details of the traditional approach

- ***alphabet*** $\equiv$ finite set of symbols, denoted Σ
- ***string*** $\equiv$ finite sequence of symbols
- set of all possible strings on Σ is denoted Σ^*
- ***language*** or ***formal language*** $\equiv$ subset of Σ^*
- Given Turing machine M with input string s, say
  - M ***accepts*** s if terminates in special accepting state
  - M ***rejects*** s if terminates in any other state
  - but remember the machine may not terminate, so it could neither accept nor reject
- Machine M ***decides*** language L if
  - M accepts all $s \in L$ and rejects all $s \notin L$

Key recommendation: instead of *formal language*, use *computational problem*

- A ***computational problem*** (which may or may not be a decision problem) is a function F, mapping ASCII strings to sets of ASCII strings.

- If $F(x) = \{s_1, s_2, \ldots\}$, we call $\{s_1, s_2, \ldots\}$ the ***solution set*** for x, and each s_i is a ***solution*** for x.

- If $F(x) = \{\text{“}\texttt{no}\text{”}\}$, then x is a ***negative instance*** of F; otherwise x is a ***positive instance***.

“Deciding a language” vs “solving a problem”

- Computer program P ***solves*** the computational problem F if $P(x) \in F(x)$ for all x. That is, the program always terminates and outputs a correct solution.
- Contrast with: Turing machine M ***decides*** language L if M accepts all $s \in L$ and rejects all $s \notin L$

Helpful examples of computational problems: HamCycle and Factor

	Traditional (decision)	**Practical (nondecision)**
HamCycle	Does this graph have a Hamilton cycle? e.g. “`a,b b,c c,a`”↦ “`yes`”	Please give me a Hamilton cycle of this graph. e.g. “`a,b b,c c,a`”↦ “`a,b,c`”
Factor	Does this integer have a nontrivial factor? e.g. “`51295697`” ↦ “`yes`”	Please give me a nontrivial factor of this integer. e.g. “`51295697`” ↦ “`8779`”

Remainder of the talk

1. Empirical evidence of student perceptions favoring nondecision problems
2. Technical details of how to teach the content using nondecision problems
   a) Definitions, including *formal languages* vs *computational problems*
   b) Computability
   c) Complexity

Computability replaces *decidability*

- The notion of *computable function* is well known, but here we generalize to the notion of *computable problem*:
  - F is *computable* if there exists a Python program P that computes F
  - i.e. require $P(x) \in F(x)$ for all x — but for given x, P needs to compute only one solution, not all of them (e.g. find one Hamilton cycle, not all Hamilton cycles)
- Uncomputable problems include old favorites such as the halting problem, but also include interesting nondecision problems, e.g.
  - can view Hilbert's 10^{th} problem as a nondecision problem: find integer solutions to Diophantine equations
  - given a program, how many steps will it execute before it halts?

Using real computer programs also helps understanding

Example: A classical diagonalization + proof by contradiction can be done explicitly in Python

```
from yesOnString import yesOnString
def weirdYesOnString(progString):
    if yesOnString(progString, progString)=='yes':
        return 'no'
    else:
        return 'yes'
```

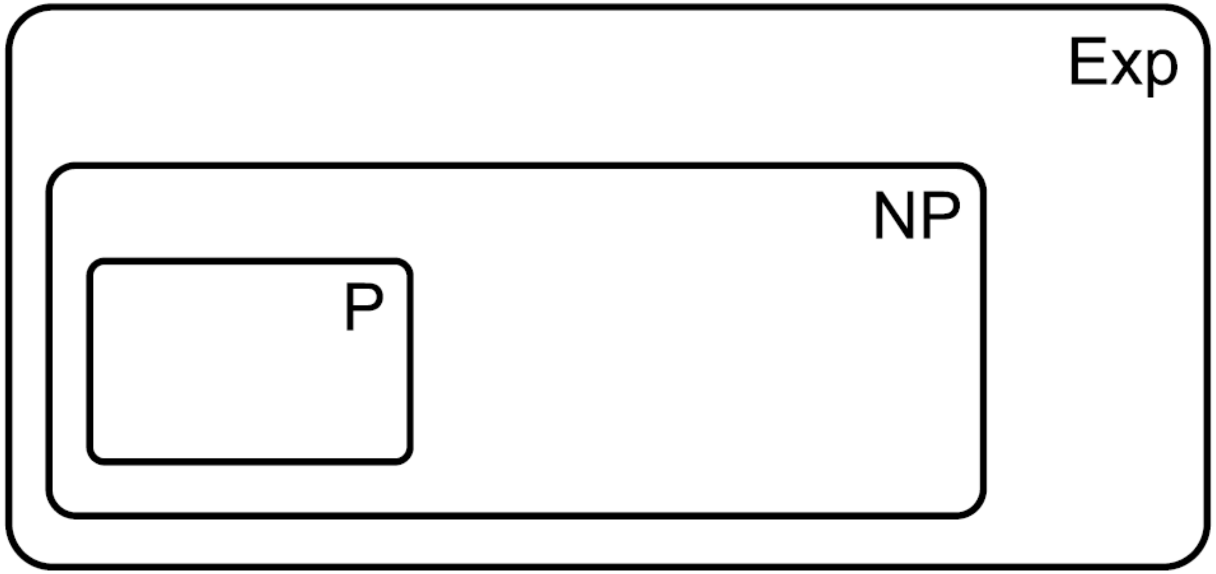
Remainder of the talk

1. Empirical evidence of student perceptions favoring nondecision problems
2. Technical details of how to teach the content using nondecision problems
   a) Definitions, including *formal languages* vs *computational problems*
   b) Computability
   c) Complexity

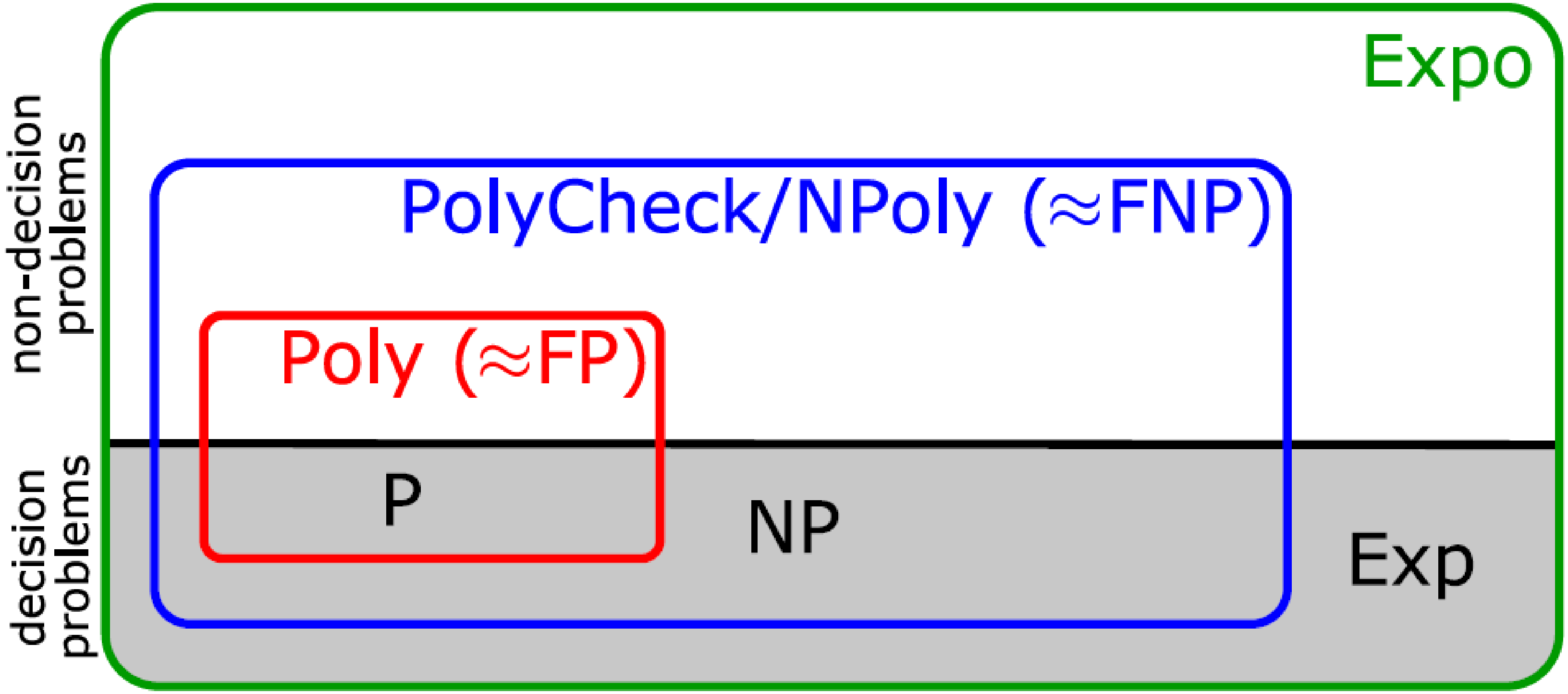
Need new notation to generalize standard complexity classes

P, NP, Exp generalize to
Poly, NPoly, Expo

Decision
problems
only



General
problems



Poly, NPoly, Expo yield pedagogical benefits

Examples:

- Students can write multithreaded programs that find factors, Hamilton cycles, etc. in nondeterministic polynomial time
  - Leads to concrete experience of the power of nondeterminism
- The impact of complexity theory on cryptography is obvious to the novice audience
  - No polynomial time algorithm for *finding* factors is known
  - But the AKS algorithm determines the *existence* of a factor in polynomial time! So in the decision framework, it’s hard to make the link to cryptography.

The generalization of *verifier* presents some interesting challenges and opportunities

- The definition involves multiple conditions and quantifiers
- The definition separates the proposed solution s and any required "hint" h
  - Contrast this with the traditional approach, where s and h are incorporated into a single string c called the *witness* or *certificate*
  - It can be difficult for a novice audience to interpret the certificate c

Let F be a computational problem. A *verifier* for F is a program $V(w, s, h)$ with the following properties:

- V receives three string parameters: an instance w, a proposed solution s, and a hint h.
- V halts on all inputs, returning either "yes" or "no".
- **Every positive instance can be verified:** If w is a positive instance of F, then $V(w, s, h) =$ "yes" for **some** correct positive solution s and **some** hint h.
- **Negative instances can never be verified:** If w is a negative instance of F, then $V(w, s, h) =$ "no" for **all** values of s and h.
- **Incorrect proposed solutions can never be verified:** If s is not a correct solution (i.e. $s \notin F(w)$), then $V(w, s, h) =$ "no" for **all** h.

We recommend the traditional approach to polynomial time reductions, with one small tweak

- As with the strong majority of other treatments, stick with *Karp reductions* (also known as *mapping reductions* or *many-one reductions*)
- One small generalization: can reduce from decision problems to *nondecision* problems, without altering the definition
  - Leads to a nice definition of NP-hardness later
- In principle, can teach a more general approach, reducing *nondecision* problems to *nondecision* problems
  - Experiments led to some success, but on balance this is not recommended for the novice audience

For NP-completeness, stay firmly within the traditional realm

- It is possible to teach “NPoly-completeness,” but not recommended
- Even while restricting to decision problems, the benefits of using nondecision problems earlier in the course are felt:
  - The practical impacts of routing, scheduling and knapsack problems are obvious
  - Holistic discussions of “P versus NP” have a more practical flavour

Ten CS theory textbooks

		Mention FP, FNP; Define and/or focus on nondecision problems	Karp reductions
Sipser (2013)	Introduction to the Theory of Computation	✗	✓
Linz (2011)	An Introduction to Formal Languages and Automata	✗	✓
Hopcroft, Motwani and Ullman (2006)	Introduction to Automata Theory, Languages, and Computation	✗	✓
Rich (2008)	Automata, Computability and Complexity: Theory and Applications	✓	✓
Davis, Sigal and Weyuker (1994)	Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science	✗	✓
Lewis and Papadimitriou (1997)	Elements of the Theory of Computation	✗	✓
Papadimitriou (1994)	Computational Complexity	✓✓	✗
Goldreich (2010)	P, NP, and NP-Completeness: The Basics of Computational Complexity	✓✓✓	✗
Arora and Barak (2009)	Computational Complexity: A Modern Approach	✓	✓
Moore and Mertens (2011)	The Nature of Computation	✓	✓

Related work

- Focus on nondecision problems
  - Goldreich, *On Teaching the Basics of Complexity Theory* (2006) + books (2008, 2010); Mandrioli (1982)
- Interactive automata software tools e.g. JFLAP, DEM
  - Chesñevar et al. (2003); Rodger et al. (2006, …);
- “NP-completeness for all”
  - Crescenzi et al. (2013); Enström and Kann (2010); Lobo and Baliga (2006)

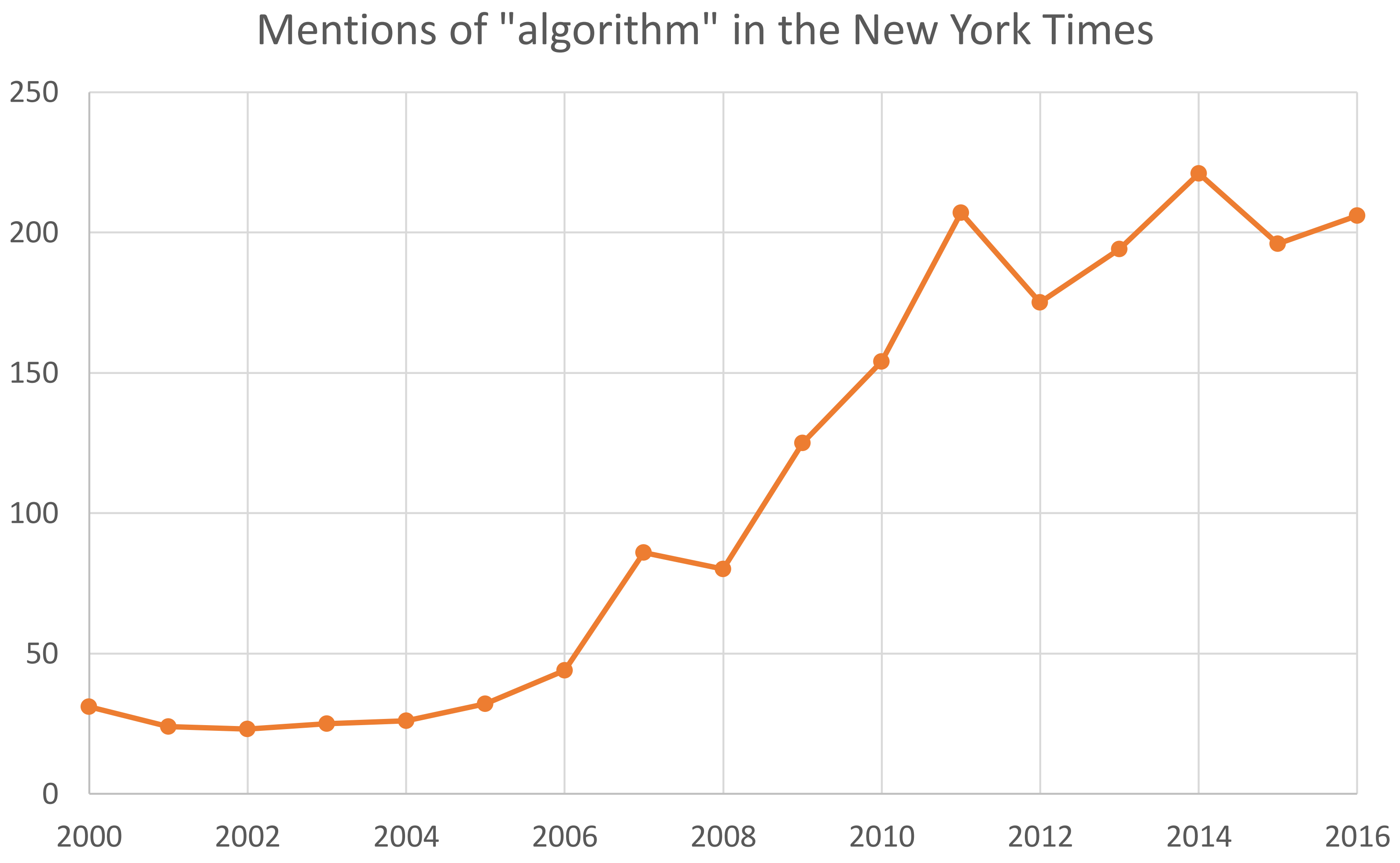
Summary: The CS theory course can be made practical and accessible

- Key ideas: focus on nondecision problems, use real computer programs
- Two main components of today's talk:
  - Survey of CS students shows they perceive nondecision problems as more useful; educational theory implies this leads to better learning outcomes
  - Presented definitions and techniques useful for achieving this with a novice audience
- Approach has been refined over four years' experimentation in classroom; details appear in forthcoming textbook

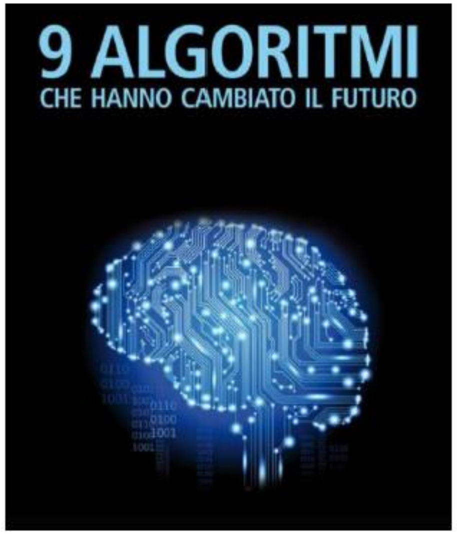
Overview

1. Main talk (practical approaches to the CS theory module)
    - About 35 minutes; real-time questions and interaction are welcomed
2. Other interests (hope to pursue some of these in the next two years)
    - About 5 minutes: computer vision, machine learning, distributed systems, CS education, public understanding of computer science
3. Questions and discussion

Do we live in an age of algorithms?

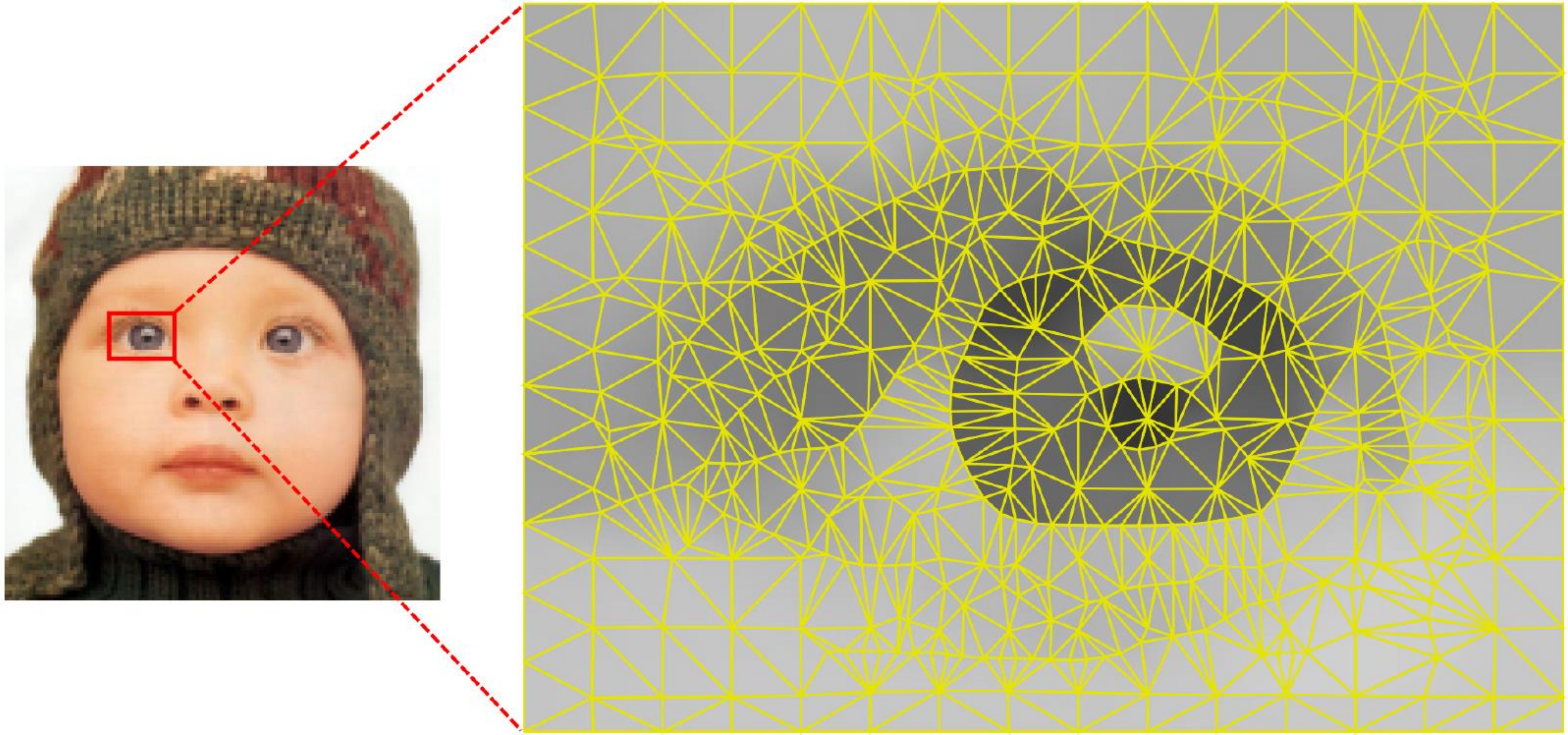


Public understanding of computer science

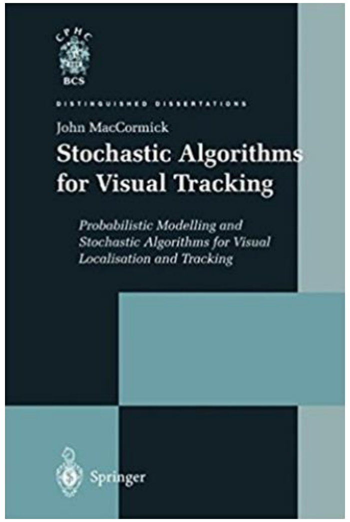


Princeton University Press (2012)

Computer vision and machine learning

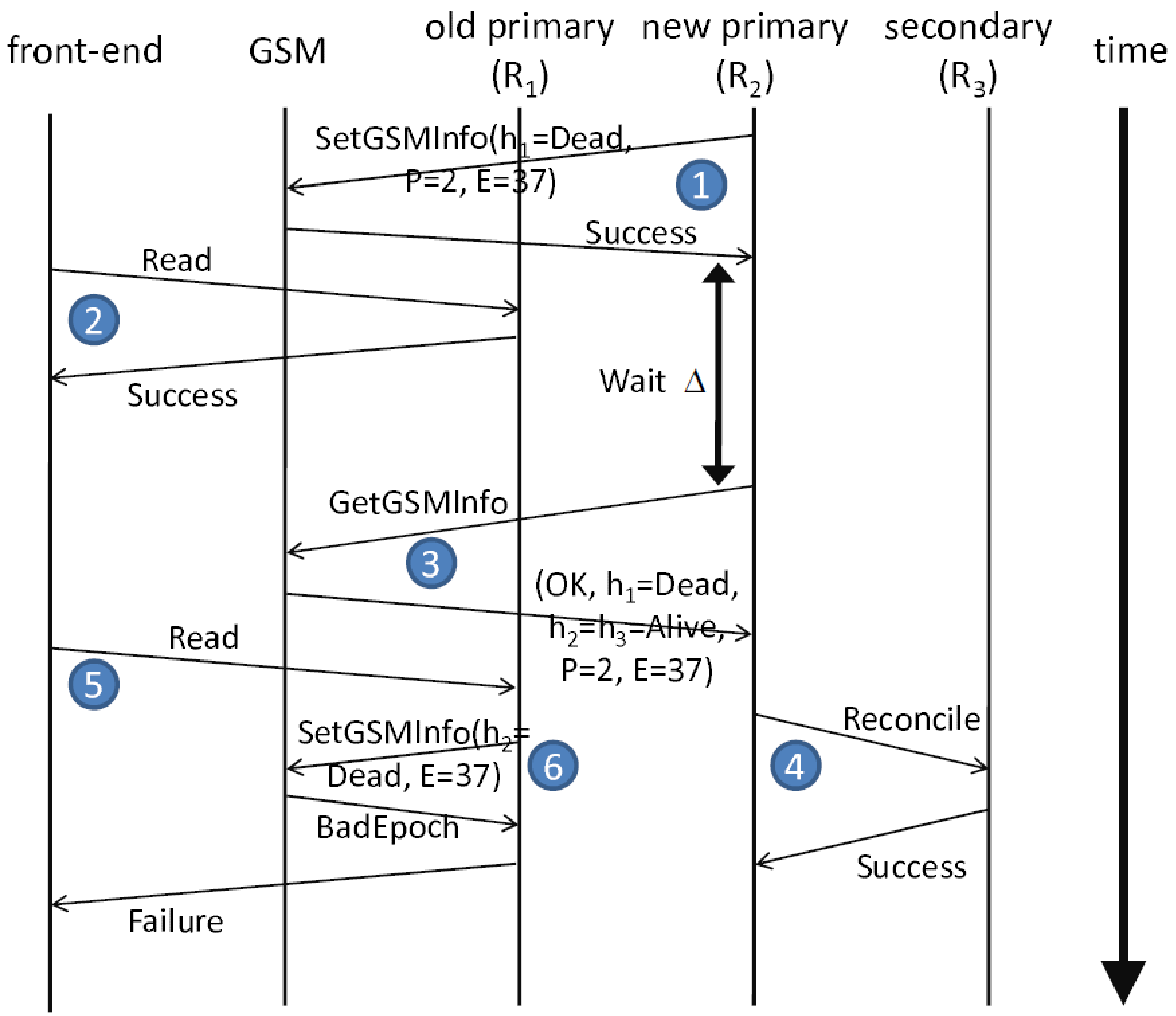


EMMCVPR (2013)



Springer (2002)

Distributed systems



ACM Trans. Storage (2008)

CS education

A Multi-Institutional Perspective on H/FOSS Projects in the Computing Curriculum

GRANT BRAUGHT and JOHN MACCORMICK, Dickinson College
JAMES BOWRING and QUINN BURKE, College of Charleston
BARBARA CUTLER, DAVID GOLDSCHMIDT, MUKKAI KRISHNAMOORTHY, and WESLEY TURNER, Rensselaer Polytechnic Institute
STEVEN HUSS-LEDERMAN, Beloit College
BONNIE MACKELLAR, St. John's University
ALLEN TUCKER, Bowdoin College

ACM Trans. Computing Education (to appear, pending final review)

Overview

1. Main talk (practical approaches to the CS theory module)
    - About 35 minutes; real-time questions and interaction are welcomed
2. Other interests (hope to pursue some of these in the next two years)
    - About 5 minutes: computer vision, machine learning, distributed systems, CS education, public understanding of computer science
3. Questions and discussion

+ thank you for welcoming me into the School of Computing Sciences, and thanks for listening today!