# Strategies for basing the CS theory course on non-decision problems…

# … and using real computer programs.

# i.e. a practical approach to teaching the CS theory course

John MacCormick
Dickinson College and
University of East Anglia School of Computing Sciences

# What is the CS “theory course”?

Usually covers some or all of:

- automata theory
  - finite automata, context free languages, Turing machines
- computability theory
  - undecidability of halting problem, Rice’s Theorem
- complexity theory
  - P, NP, NP-completeness, Cook-Levin theorem

# How can we make the theory course more accessible, more practical, and less intimidating?

Previous work has made substantial strides in this direction:

- interactive automata software tools such as JFLAP and DEM
  - Chesñevar et al (2003), Rodger et al (2006, 2009, etc.)
- “NP-completeness for all”
  - Crescenzi (2013), Enstrom (2010), Lobo (2006)
- Different theoretical model
  - Mandrioli (1982), Goldreich (2006, 2010)

Today’s talk

# How can we make the theory course more accessible, more practical, and less intimidating?

- Use *nondecision problems* as the primary paradigm
  - Contrasts with traditional decision problem paradigm

Main emphasis of the paper

- Use *real computer programs* as the primary computational model
  - Contrasts with traditional use of Turing machines

Also important (see the book)

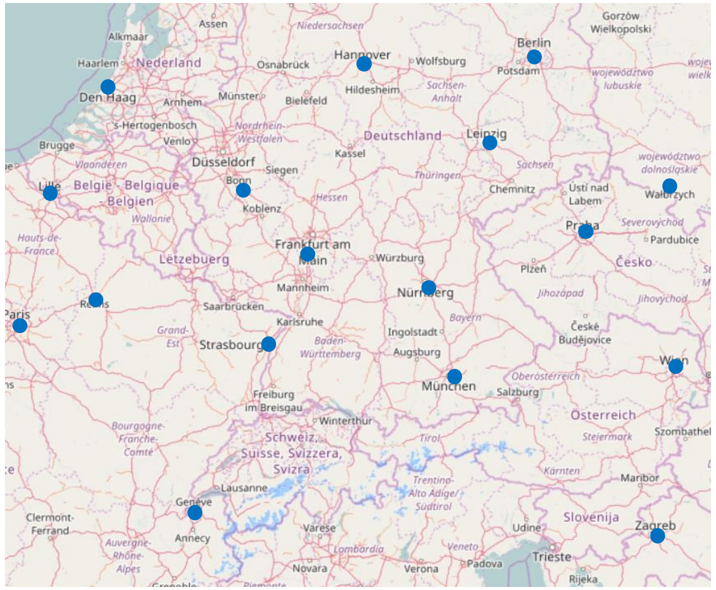
# A long-term vision for the theory course

- Make it more accessible and more practical
- Teach it to a wider range of undergraduates in more institutions
- Place it earlier in the curriculum with fewer prerequisites

- How?
  - Use nondecision problems ← Next: explain what this means
  - Use real computer programs

# 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 courses 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 courses 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 course using *nondecision problems* and *real computer programs*.
- Students may achieve better learning because the content is perceived as relevant and practical.

# We consider only a *novice audience*

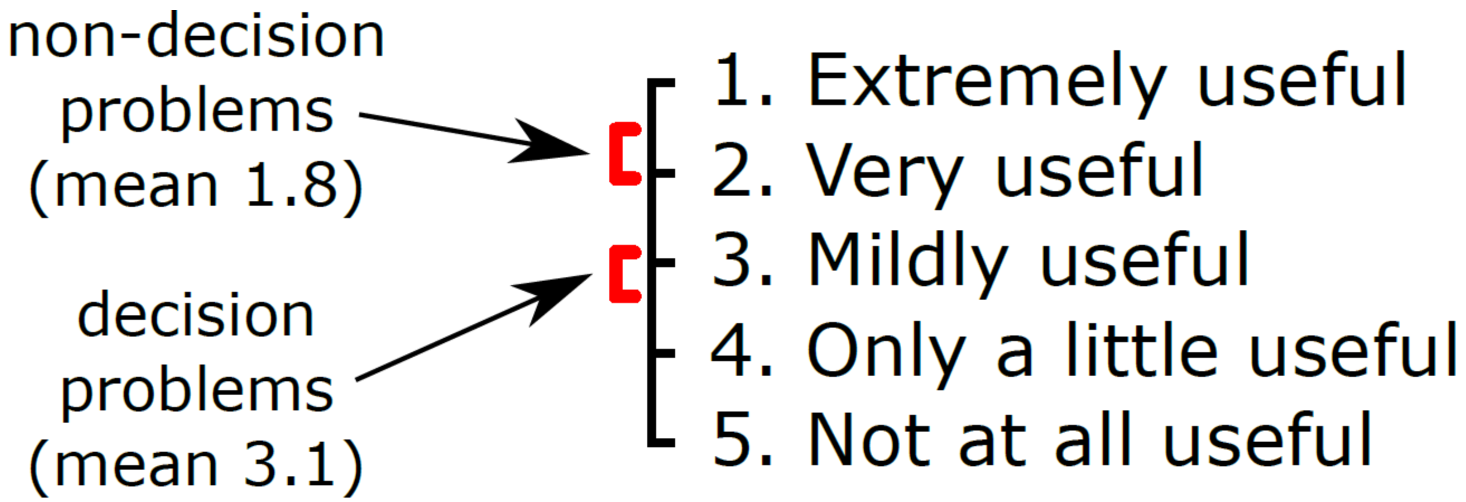
- Novice audience $\equiv$ undergraduate students who are seeing computability and complexity theory for the first time

- Experienced practitioners know that decision programs can often be converted to equivalent non-decision programs with only a logarithmic increase in running time.

- But for the *novice audience*, a program that outputs only a single bit may appear abstruse, irrelevant, and impractical

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

Survey of undergraduates compared decision and nondecision variants of TSP and knapsack problems




- The difference has overwhelming statistical significance and large effect size
- Perceived usefulness translates to better learning outcomes (Fink 2013)

# OK, we need to use nondecision problems. But how do we do that?

- Answer: adjust certain definitions
- A brief example is shown next
- Please see the paper for details

# Example of a technical detail: computational problems

- 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 $x$.

This allows us to talk about "solving a problem" instead of "deciding a language":

- Computer program $P$ ***solves*** the computational problem $F$ if $P(x) \in F(x)$ for all $x$.
- Contrast with: Turing machine $M$ ***decides*** language $L$ if $M$ accepts all $s \in L$ and rejects all $s \notin L$

# Examples of “solving a problem” instead of “deciding a language”

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

# Other examples of technical details

- To incorporate nondecision problems, we need generalizations of traditional complexity classes, e.g.:
  - P becomes Poly
  - NP becomes NPoly

- The generalized definition of a *verifier* offers new pedagogical opportunities:
  - The traditional role of the *certificate* is separated into two clearer, independent notions: the *solution* and the *hint*

# How can we make the theory course more accessible, more practical, and less intimidating?

- Use *nondecision problems* as the primary paradigm
  - Contrasts with traditional decision problem paradigm

Main emphasis of the paper

- Use *real computer programs* as the primary computational model
  - Contrasts with traditional use of Turing machines

Also important (see the book)

Why? As Turing himself wrote (1936):

> This proof, although perfectly sound, has the disadvantage that it may leave the reader with a feeling that "there must be something wrong".

# Using real computer programs permits interactive experimentation by students

Example 1: 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'
```

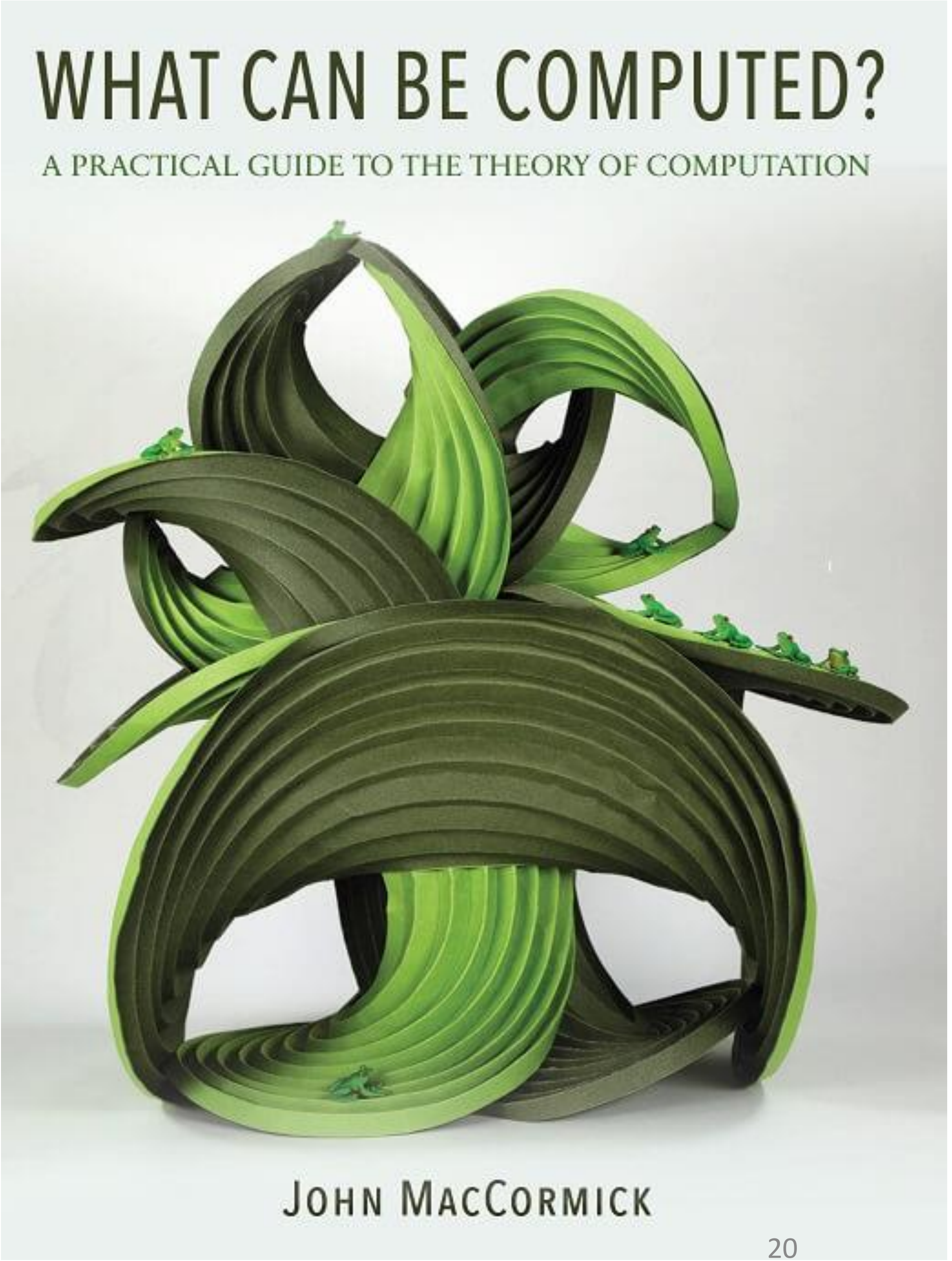
# Using real computer programs permits interactive experimentation by students

Example 2: a “universal program” is much simpler than a universal Turing machine.

```
def universal(progString, inString):
    # Execute the definition of the function in progString. This defines
    # the function, but doesn't invoke it.
    exec(progString)
    # Now that the function is defined, we can extract a reference to it.
    progFunction = utils.extractMainFunction(progString)
    # Invoke the desired function with the desired input string.
    return progFunction(inString)
```

# That sounds interesting. But how can I actually teach a theory course using nondecision problems and real computer programs?

- Answer: There is a new text book from Princeton University Press that takes this approach
  - *What Can Be Computed?: A Practical Guide to the Theory of Computation*
  - Available Spring 2018
  - Visit the Princeton University Press booth or email me for more details



# Conclusion: a long-term vision for the theory course

- Make it more accessible and more practical
- Teach it to a wider range of undergraduates in more institutions
- Place it earlier in the curriculum with fewer prerequisites
- How?
  - Use nondecision problems
  - Use real computer programs
- Thanks for listening!