Using Computers to Design Distributed Algorithms

Jukka Suomela
Aalto University, Finland

Computer science: “what can be automated?”

Next level:

"can we automate our own work?"

Key players in algorithmics

Model of computing

Computational problem — *"what are feasible solutions for any given input?"*

Algorithm — *"how to find a feasible solution for any given input?"*

Key players in algorithmics

Model of computing	e.g. RAM machines
Computational problem	e.g. sorting
Algorithm	e.g. merge sort

Key players in algorithmics

Model of computing	e.g. distributed graph algorithms
Computational problem	e.g. list 3-coloring
Algorithm	e.g. Cole–Vishkin

recall Lecture 1…

How to design algorithms?

Model of computing

Computational problem

Algorithm?

How to design algorithms?

Model of computing

Computational problem

Algorithm?

- Some systematic principles:
  - algorithm design paradigms
  - reductions …
- But largely just “think hard”, years of experience, clever insights, good luck?

How to design algorithms?

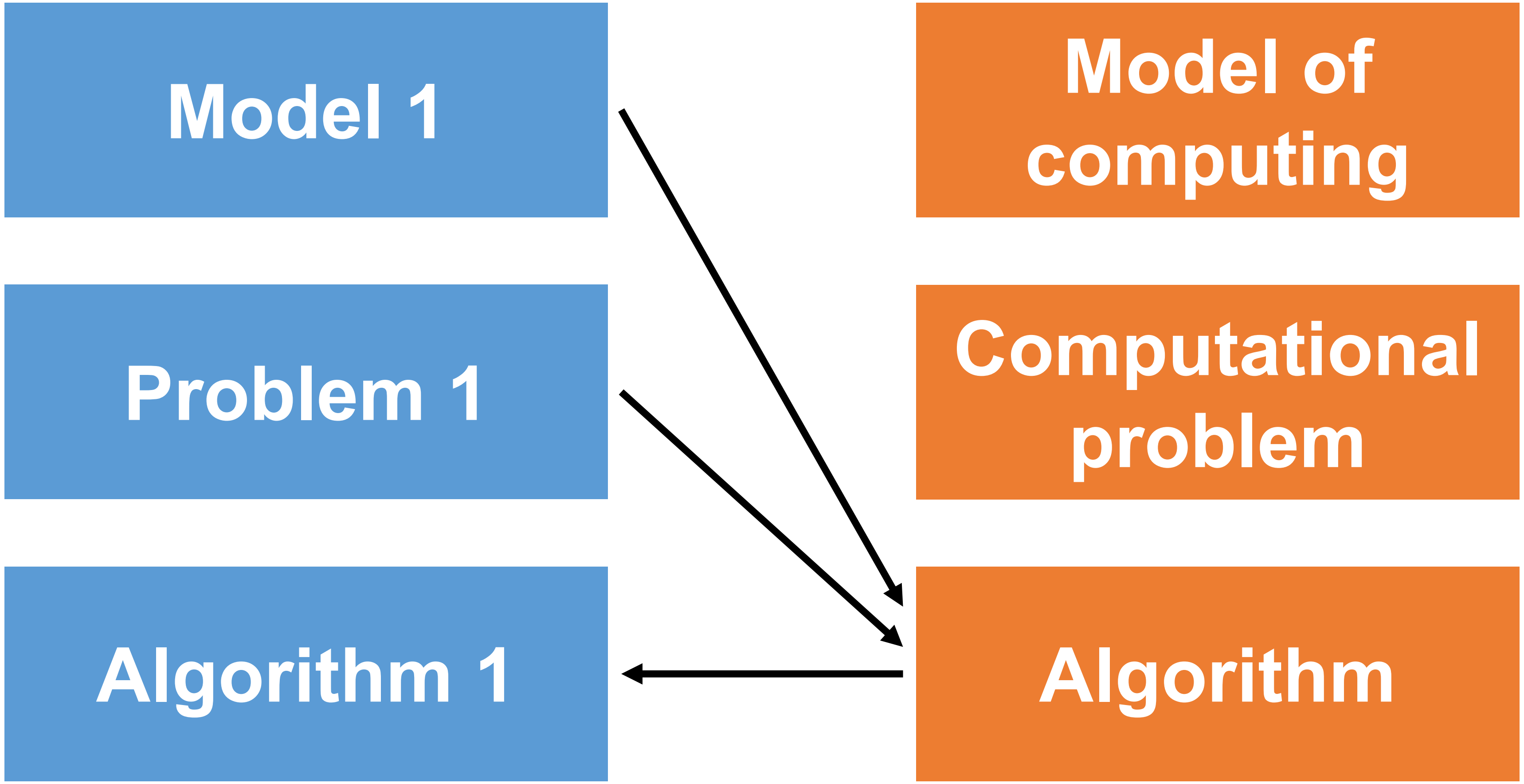
Model of computing

Computational problem

Algorithm?

- Some systematic principles:
  - algorithm design paradigms
  - reductions …
- But largely just “think hard”, years of experience, clever insights, good luck?
- ***Could we automate it?***

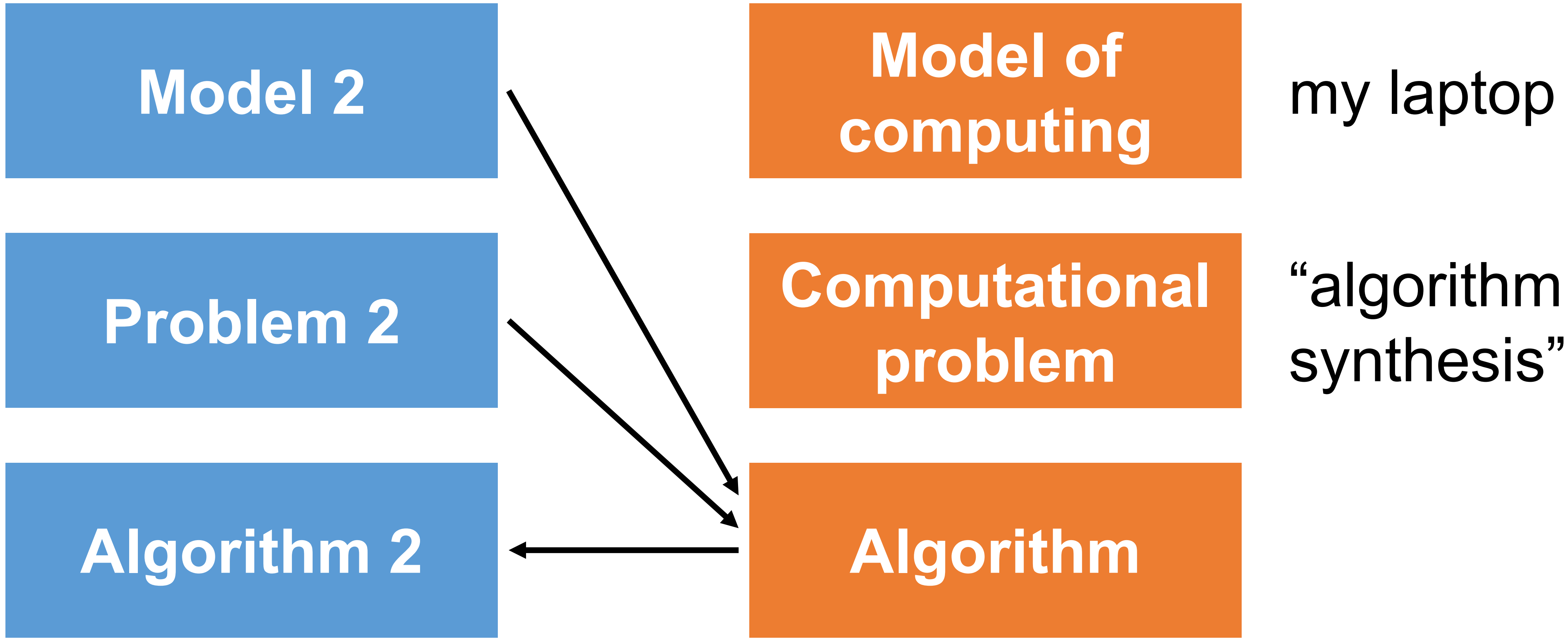
Ultimate meta-algorithm??



my laptop

“algorithm synthesis”

Ultimate meta-algorithm??



Too good
to be true?

Does this make any sense?

- Is “algorithm synthesis” a well-defined computational problem?
- What are the right ***representations***?
  - how to represent computational problems or models of computing as input data?
  - how to represent algorithms as output?

Computability?

- Recall the classical meta-computational question: the ***halting problem***
  - input: “algorithm” (encoded as a Turing machine)
  - output: does it ever halt?
- **Undecidable problem** — there is no “meta-algorithm” that solves it

Computability?

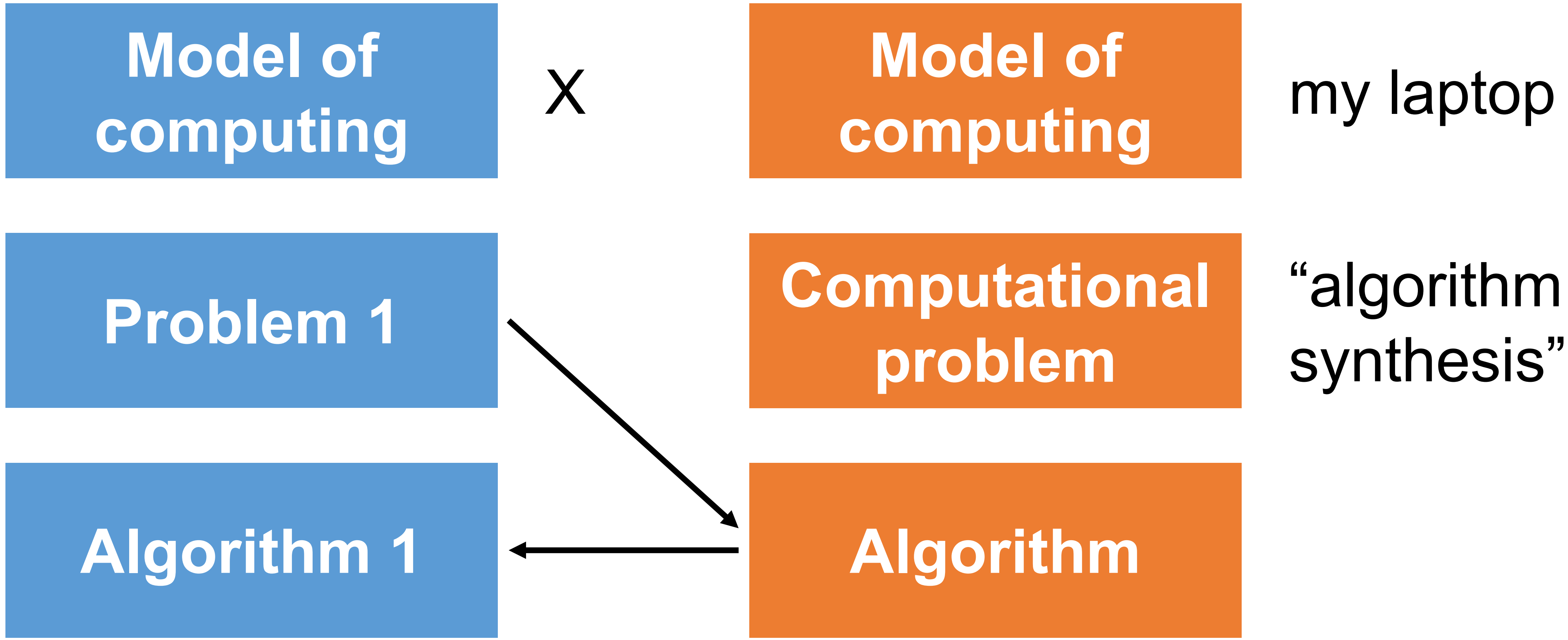
- We are already in trouble if we would like to ***verify*** a given algorithm
- Isn’t it much harder to ***synthesize*** an algorithm than to verify a given algorithm?

Computational complexity?

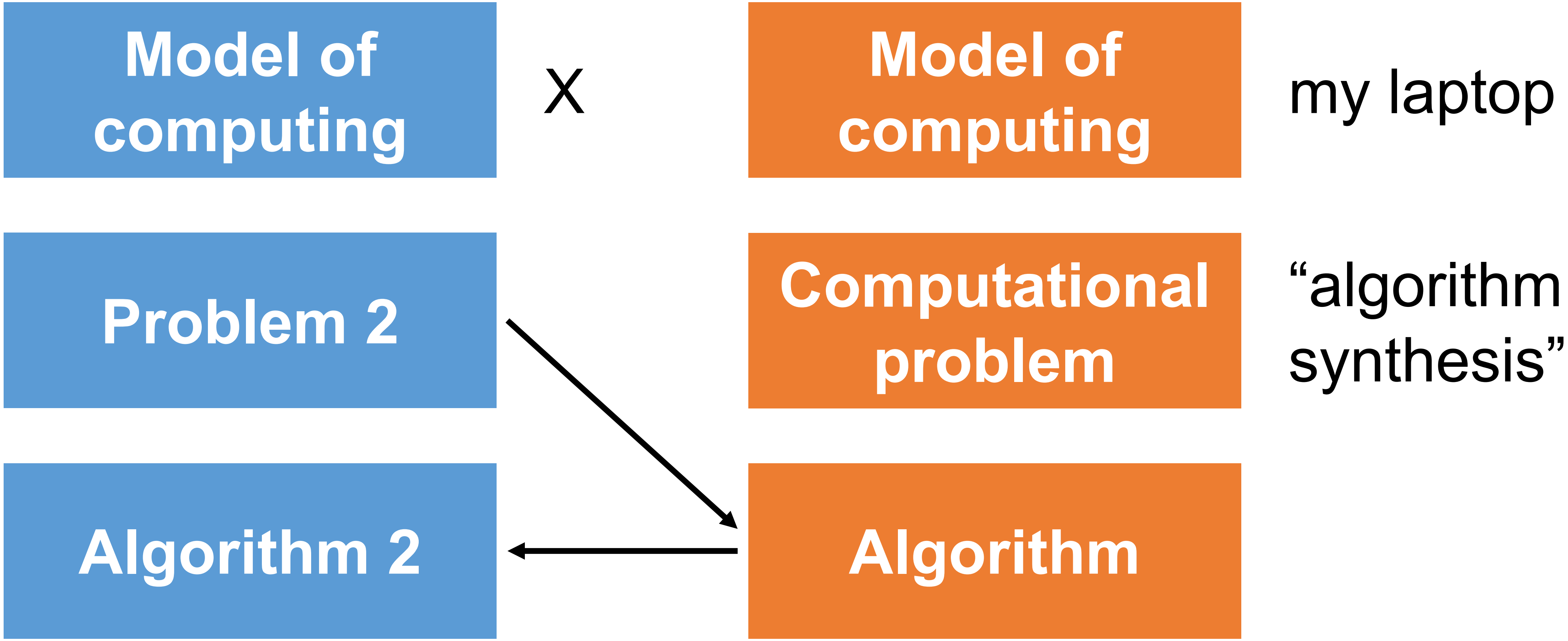
- Even if we could synthesize algorithms in principle, does it work in practice?
- Does anyone have enough ***computational resources*** to do it?

Overcoming some challenges: specialization and semi-automation

Fix the model of computing



Fix the model of computing



Good news

- For some **models of distributed computing**, algorithm synthesis is possible!
  - both ***in theory*** and ***in practice***!
  - there are computer-designed distributed algorithms that outperform the best human-designed algorithms!

More good news

- **Human beings** are not yet obsolete!
  - many success stories of ***computer–human collaboration***
  - “computer-aided” algorithm design instead of “fully automatic” algorithm design

Case study 1:
robust counters

Case study: robust counters

- Multiple devices connected to each other
- Common clock pulse coming to all devices
- Devices have to **count pulses**
  - ***in agreement***: if one device thinks this is pulse number *x*, then all devices agree
  - ***in a fault-tolerant manner*** (more about this soon)

Case study: robust counters

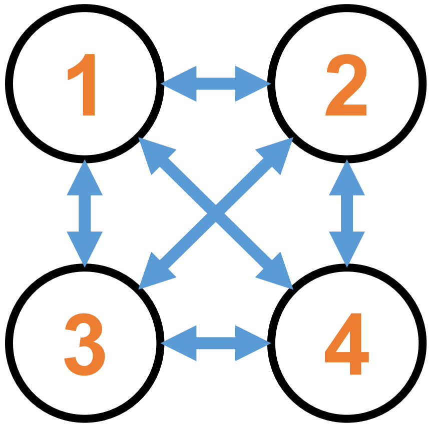
- Running example:
  - ***4 devices***
  - all devices can directly communicate with each other
  - task: count pulses ***modulo 2***

device 1:	0	1	0	1	0	1	…
device 2:	0	1	0	1	0	1	…
device 3:	0	1	0	1	0	1	…
device 4:	0	1	0	1	0	1	…

Case study: robust counters

- Nodes labeled with 1, 2, 3, 4
- At each clock pulse, each node can also receive a ***message*** from every other node

device 1:	0	1	0	1	0	1	…
device 2:	0	1	0	1	0	1	…
device 3:	0	1	0	1	0	1	…
device 4:	0	1	0	1	0	1	…



Case study: robust counters

- Very easy to solve **if there are no failures** and all nodes start in the same state

- ***How would you do it?***

device 1:	0	1	0	1	0	1	…
device 2:	0	1	0	1	0	1	…
device 3:	0	1	0	1	0	1	…
device 4:	0	1	0	1	0	1	…

Case study: robust counters

- What if we wanted to tolerate **Byzantine failures**?
- Still easy to solve — ***how?***

device 1:	0	1	0	1	0	1	…
device 2:	???	???	???	???	???	???	…
device 3:	0	1	0	1	0	1	…
device 4:	0	1	0	1	0	1	…

recall Lecture 4…

Case study: robust counters

- What if we wanted to design a **self-stabilizing algorithm**?
- Still easy to solve — ***how?***

device 1:	**garbage**	1	0	1	0	1	…
device 2:	**garbage**	1	0	1	0	1	…
device 3:	**garbage**	1	0	1	0	1	…
device 4:	**garbage**	1	0	1	0	1	…

recall Lecture 9…

Case study: robust counters

- Can we get both **self-stabilization** and **Byzantine fault tolerance** simultaneously?
- Very difficult to solve — ***try it!***

device 1:	garbage	1	0	1	0	1	…
device 2:	garbage	???	???	???	???	???	…
device 3:	garbage	1	0	1	0	1	…
device 4:	garbage	1	0	1	0	1	…

Case study: robust counters

- Goal: reach correct behavior
  - **self-stabilization:** starting from any configuration
  - **Byzantine fault tolerance:** even if one node is misbehaving
- We want to ***ask computers to find a good algorithm*** for this problem!

How to represent algorithms?

- Human-readable **pseudocode**?
  - can computers understand it at all?
- Machine-readable **programing language**, e.g. Python, Java, C++, x86 assembly?
  - very easy to write a ***short program that nobody can analyze***, not human beings, not computers

How to represent algorithms?

- Let’s try to keep things very simple
- **Computer** = ***finite state machine***
- **Communication** = each node simply tells everyone else its ***current state***
- **Algorithm** = ***lookup table***

How to represent algorithms?

- Example: 4 nodes, 3 states per node
- ***Algorithm*** = lookup table that tells what is the new state for each combination of states
  - $3^4 = 81$ rows
  - easy to represent with computers

old state	new state
0, 0, 0, 0	1, 1, 1, 1
0, 0, 0, 1	1, 1, 1, 1
…	…
0, 1, 1, 1	2, 0, 0, 0
0, 1, 1, 2	0, 0, 0, 1
…	…
2, 2, 2, 2	1, 1, 1, 1

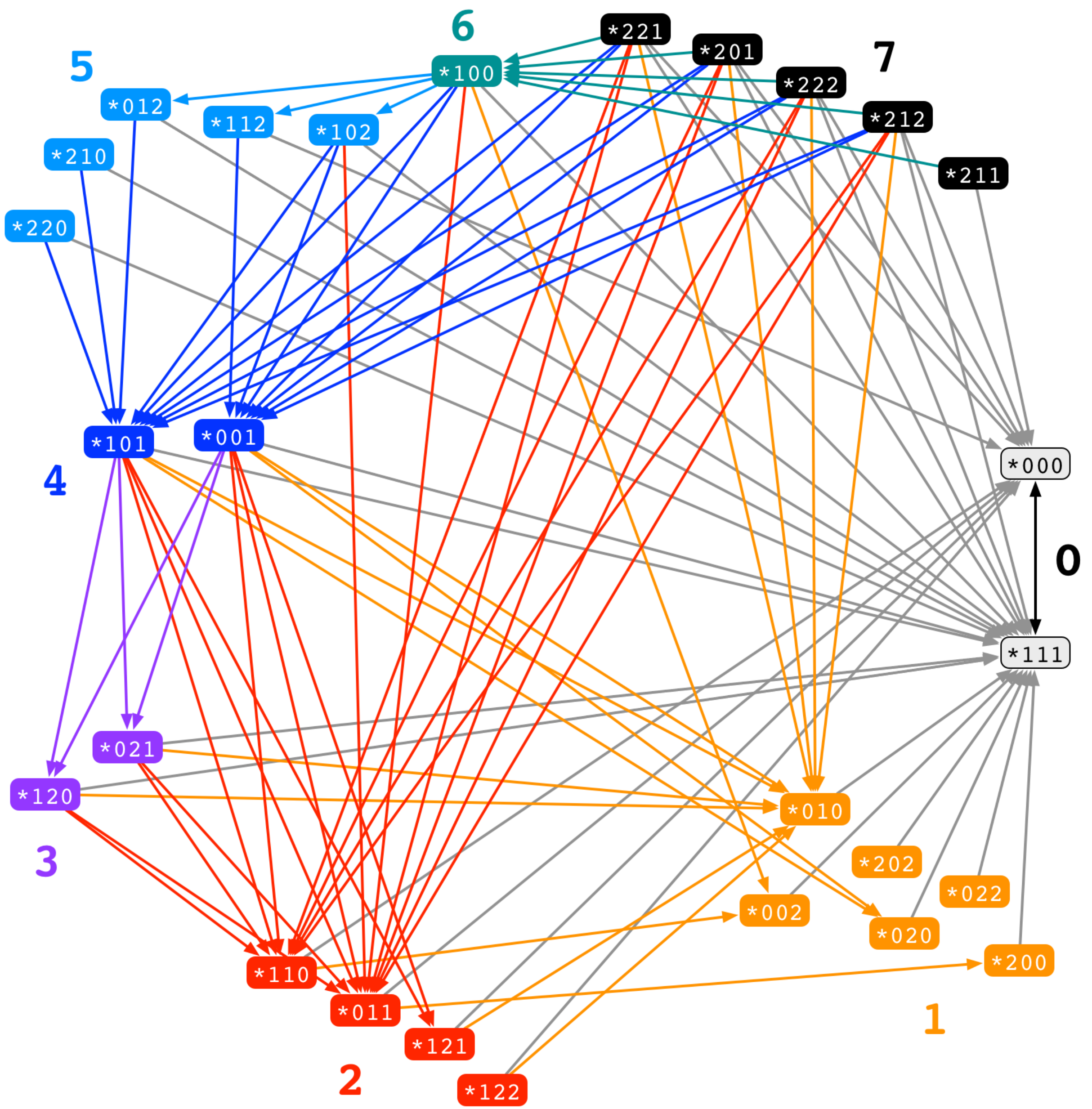
How to represent executions?

- *Algorithm* = lookup table
- Possible state transitions:
  - example: node 4 misbehaves
  - possible: 0,0,1,* → 1,1,1,*
  - possible: 0,0,1,* → 0,2,0,*
  - possible: 0,0,1,* → 1,2,0,* (!!)

old state	new state
0, 0, 0, 0	1, 1, 1, 1
0, 0, 0, 1	1, 1, 1, 1
…	…
0, 0, 1, 0	1, 1, 1, 1
0, 0, 1, 1	0, 2, 0, 1
0, 0, 1, 2	1, 1, 1, 1
…	…
2, 2, 2, 2	1, 1, 1, 1

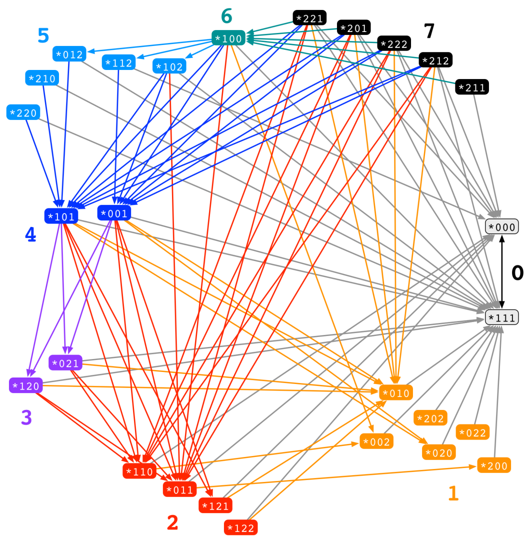
Given an algorithm, we can construct a ***directed graph*** that represents all possible state transitions

Directed path = possible **execution**



Graph representations

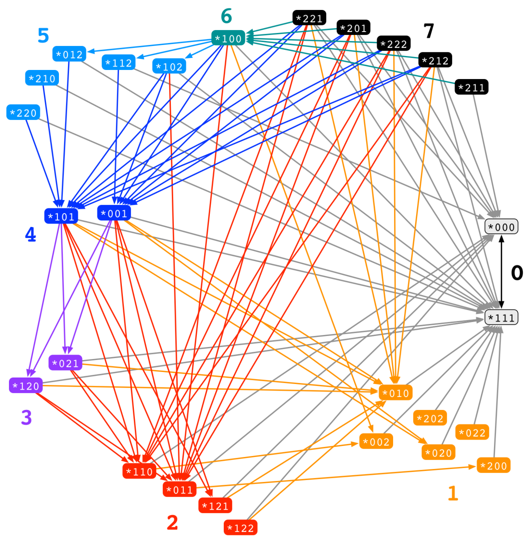
- Seemingly hard, open-ended questions:
  - is this ***algorithm correct***?
  - does it ***recover quickly*** from all failures?
- Simple, well-defined questions:
  - do ***all paths in this graph*** lead to nodes “*000” and “*111”?
  - are ***all such paths short***?



Graph representations

- **Algorithm verification** was replaced with a simple ***graph problem***

- Candidate algorithm
  → lookup table
  → graph of all executions
  → reachability problem
  → is this algorithm good

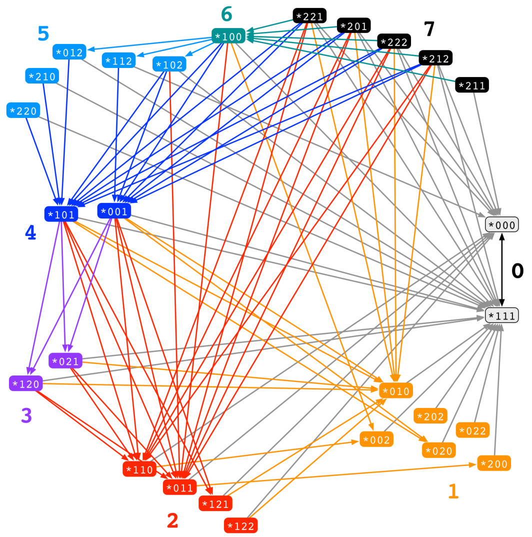


Graph representations

- We now know how to ***test*** with computers if an algorithm candidate is good

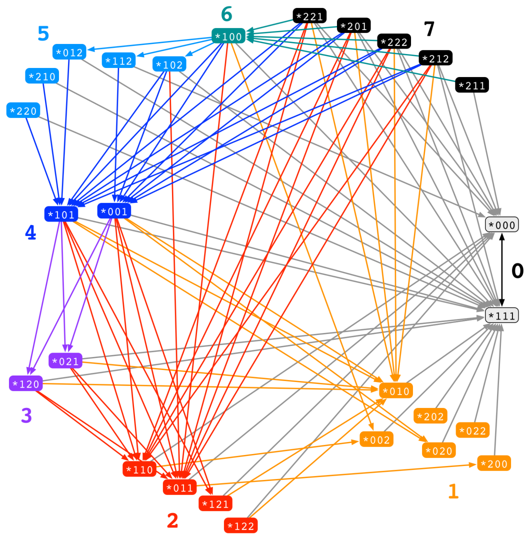
- How to use computers to ***find*** a good algorithm?

- In principle easy: we could **check all candidates**



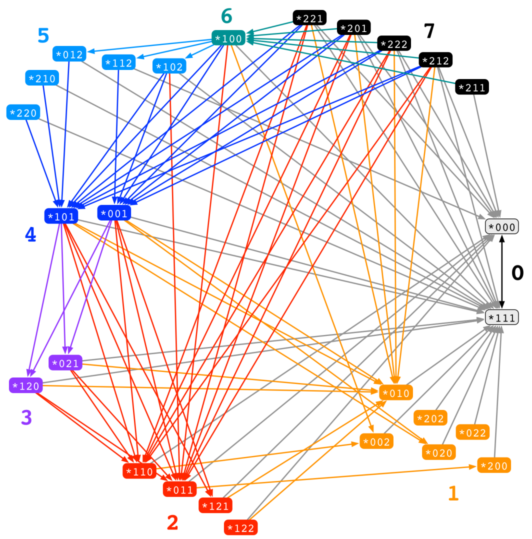
Graph representations

- Algorithm = lookup table with 81 entries
- Each entry has 81 possible values
- Just test $81^{81} \approx$ **10^{154}** candidates?



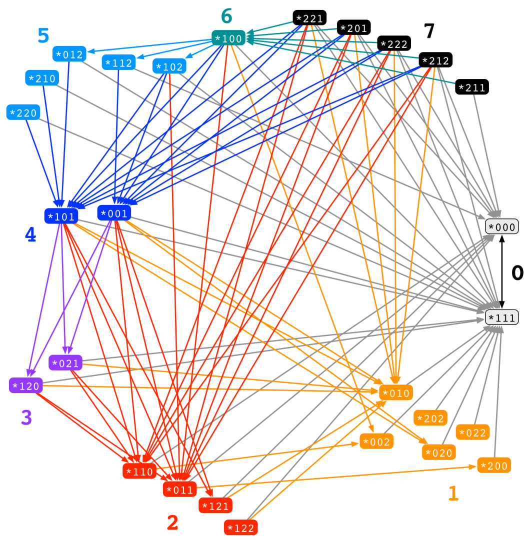
Logical representations

- Again just a matter of **representations**
  - lookup table ≈ Boolean variables x_1, x_2, …
  - this lookup table is good ≈ formula $f(x_1, x_2, \ldots)$ is true
- Apply modern ***SAT solvers*** to find values x_1, x_2, … such that $f(x_1, x_2, \ldots)$ is true



Graph representations

- **Algorithm verification** was replaced with a simple ***graph problem***
- **Algorithm synthesis** was replaced with a ***Boolean satisfiability problem***
  - NP-hard, but often (?) solvable in practice



High-throughput algorithmics

We can ask computers:

"Is there an algorithm for ***n* nodes** that uses only ***s* states** per node and always stabilizes in at most ***t* steps**?"

	$t = 3$	$t = 4$	$t = 5$	$t = 6$	$t = 7$	$t = 8$	...
$n = 4$	—	—	$s \geq 4$	$s \geq 4$	$s \geq 3$	$s \geq 3$	...
$n = 5$	—	$s \geq 3$	$s \geq 3$	$s \geq 3$	$s \geq 3$	$s \geq 3$	...
$n = 6$	$s \geq 3$	$s \geq 3$	$s \geq 3$	$s \geq 2$	$s \geq 2$	$s \geq 2$	...
$n = 7$	$s \geq 3$	$s \geq 3$	$s \geq 3$	$s \geq 2$	$s \geq 2$	$s \geq 2$	...
$n = 8$	$s \geq 3$	$s \geq 2$	$s \geq 2$	$s \geq 2$	$s \geq 2$	$s \geq 2$	...
$n = 9$	$s \geq 3$	$s \geq 2$	$s \geq 2$	$s \geq 2$	$s \geq 2$	$s \geq 2$	...

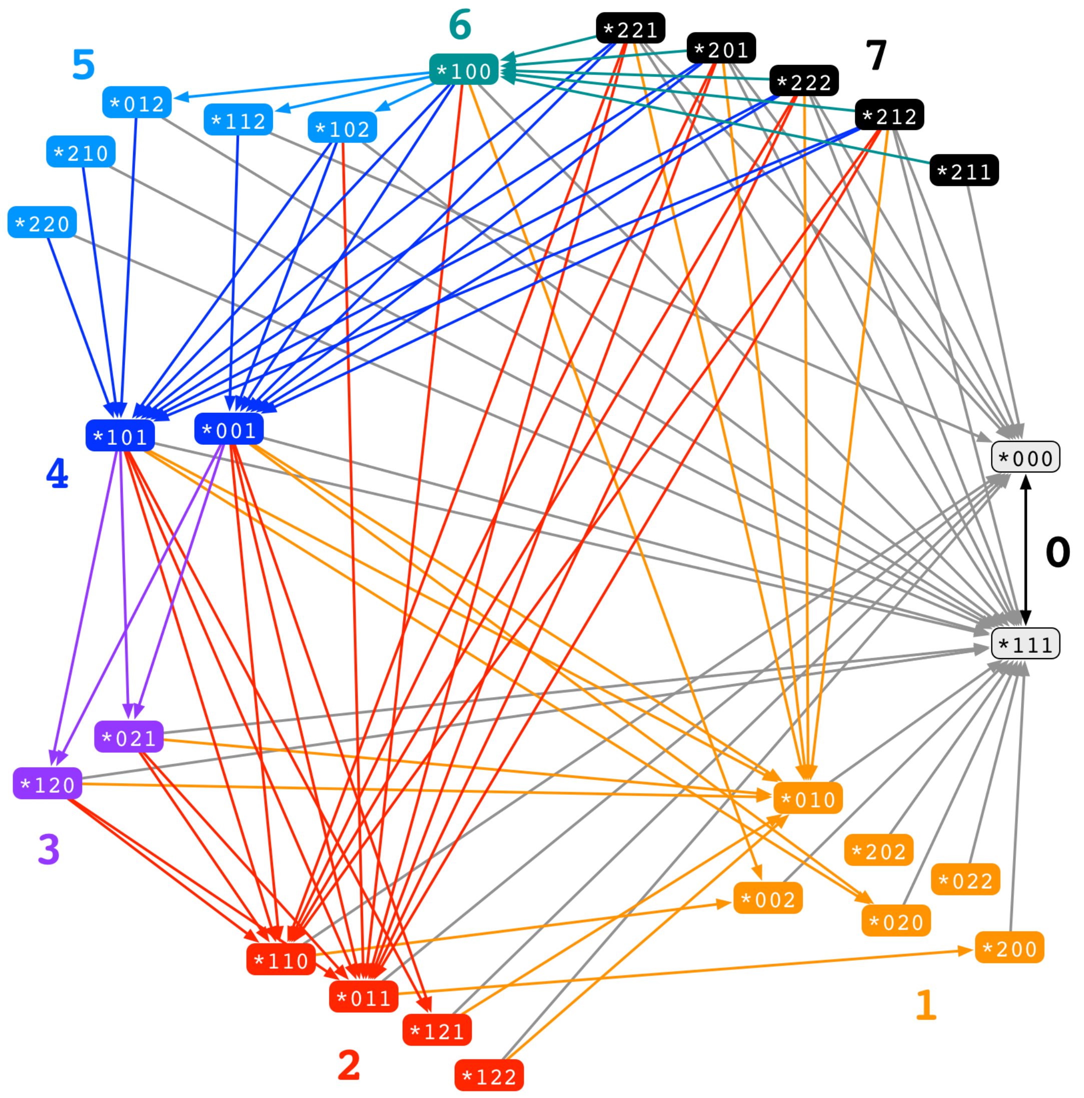
Example:

4 nodes

1 faulty node

3 states per node

always stabilizes
in at most 7 steps



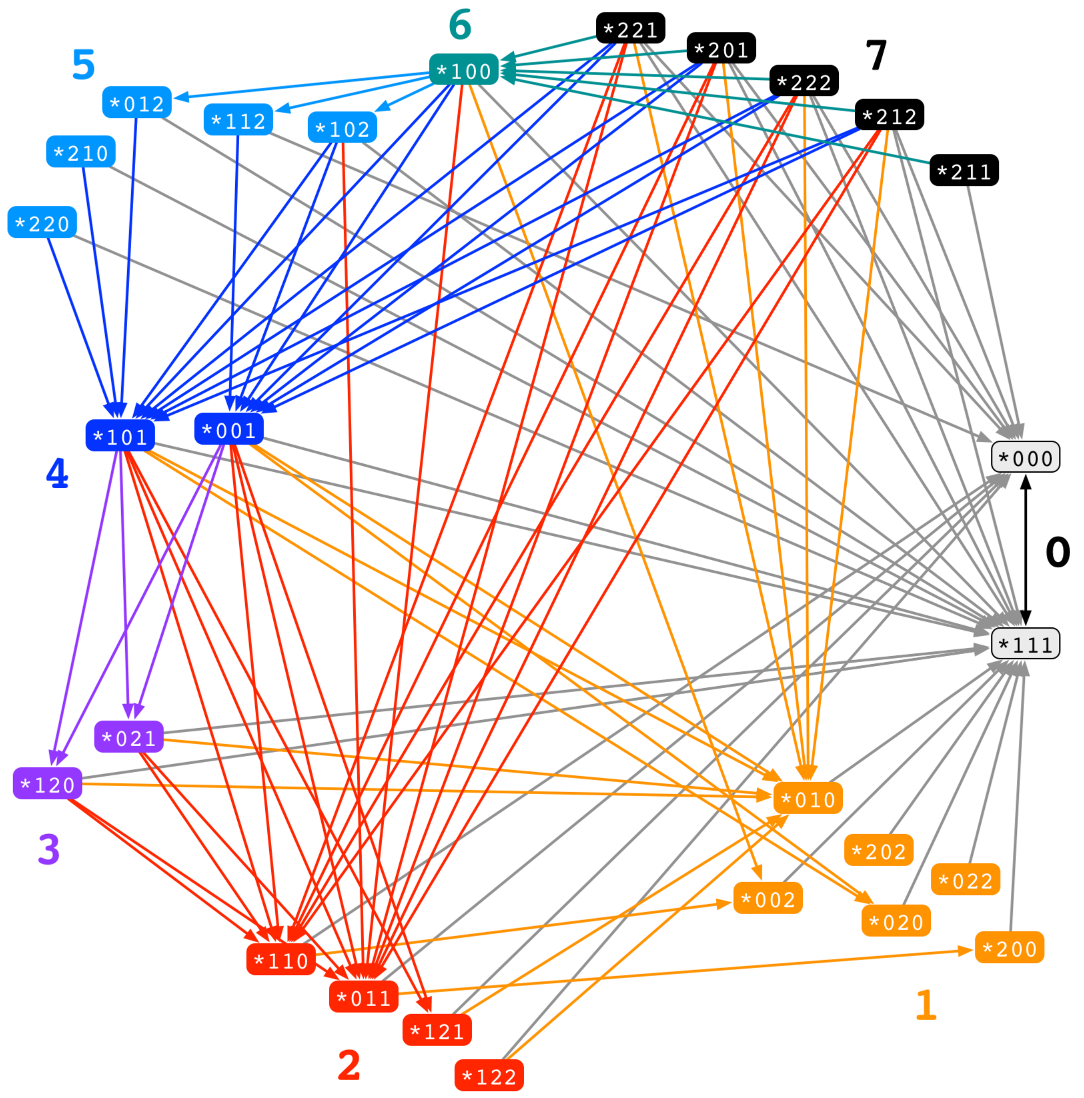
Efficient computer-designed solution for the ***base case***

\+

human-designed ***recursive step***

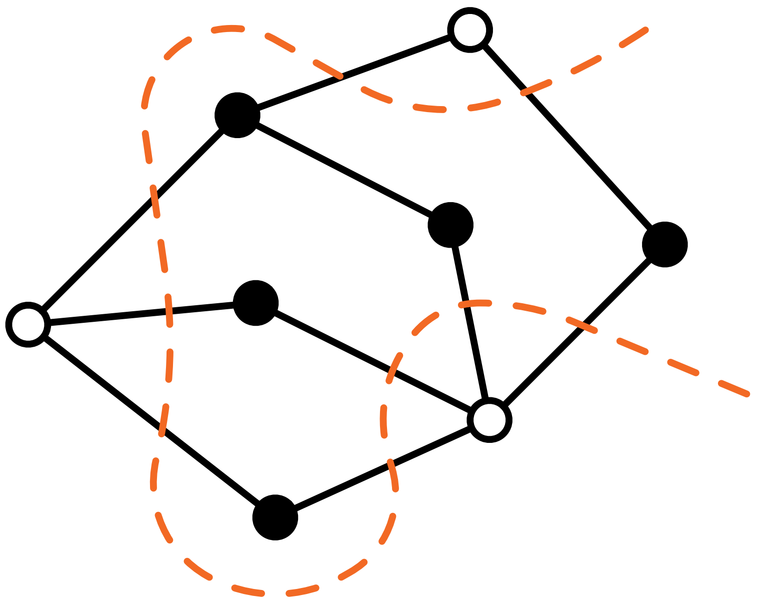
=

efficient solution for the ***general case***



Case study 2:
large cuts

Large cuts



- **Goal:** find a **large cut**
- **Setting:**
  - ***1-round randomized algorithms***
  - 1 bit of randomness per node
  - *d*-regular graphs, no short cycles

Large cuts

- Again we can represent algorithms as ***lookup tables***:
  - **input:** random bits of myself and my neighbors
  - **output:** black or white
- For each lookup table we can calculate ***probability that a given edge is a cut edge***

Large cuts

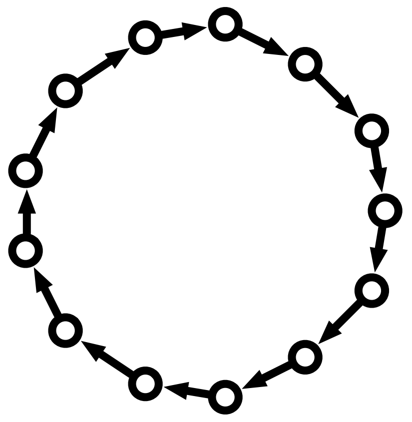
- ***Computer:***
  - find optimal algorithm for d = 2, 3, 4, …
- ***Human:***
  - look at the structure of optimal algorithms
  - generalize the idea

Large cuts

- **Algorithm:**
  - Pick a random cut
  - Change sides if **at least $\left\lceil \frac{d+\sqrt{d}}{2} \right\rceil$ neighbours** on the same side
- ***How well does this work for d = 2?***

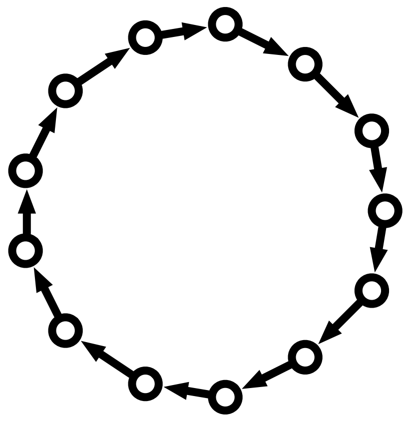
Case study 3:
local problems on cycles

LCLs on cycles



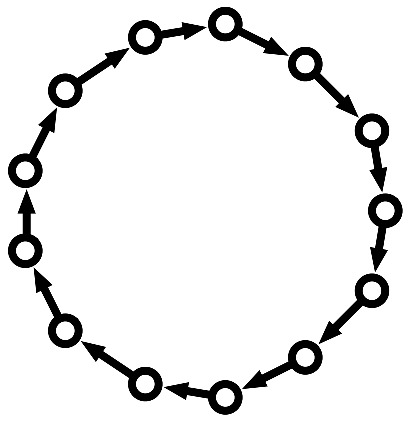
- Computer network = directed n-cycle
  - nodes labelled with **$O(\log n)$-bit identifiers**
  - each round: each node exchanges (arbitrarily large) **messages** with its neighbors and updates its state
  - each node has to output its **own part of the solution**
  - ***time = number of rounds*** until all nodes stop

LCLs on cycles



- LCL problems:
  - solution is globally good if it **looks good in all local neighborhoods**
  - examples: vertex coloring, edge coloring, maximal independent set, maximal matching…
  - cf. class NP: solution ***easy to verify***, not necessarily easy to find

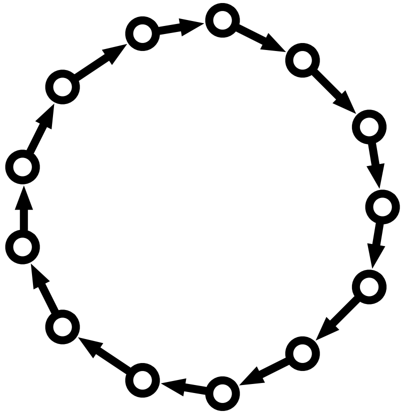
LCLs on cycles



- 2-colouring: inherently global
  - $\Theta(n)$ rounds
  - solution does not always exist
- 3-colouring: local
  - $\Theta(\log^* n)$ rounds
  - solution always exists

recall Lecture 1…

LCLs on cycles

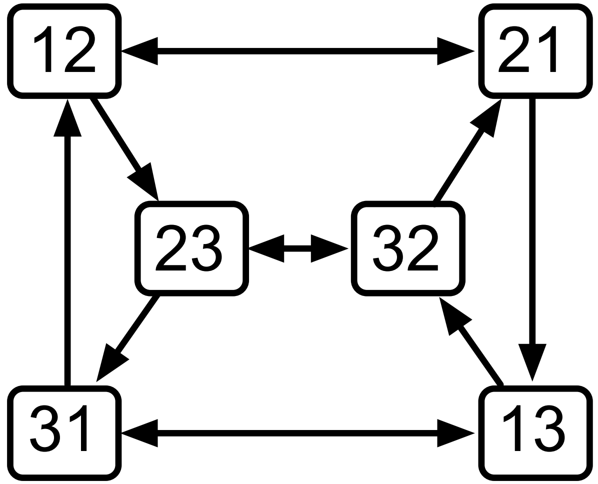


- Given an algorithm, it may be very difficult to **verify**
  - easy to encode e.g. halting problem
  - running time can be any function of n
- However, given an LCL problem, it is very easy to **synthesize** optimal algorithms!

LCLs on cycles

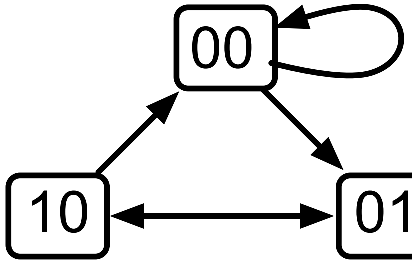
- LCL problem ≈ set of feasible local neighborhoods in the solution
- Can be encoded as a graph:
  - node = neighborhood
  - edge = “compatible” neighborhoods
  - **walk ≈ sliding window**

3-coloring

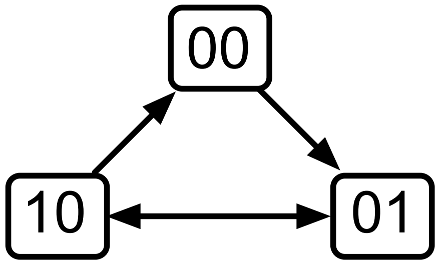


LCLs on cycles

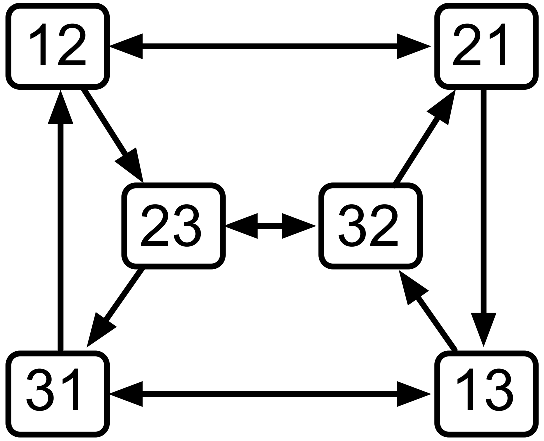
independent set



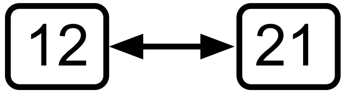
maximal independent set



3-coloring



2-coloring

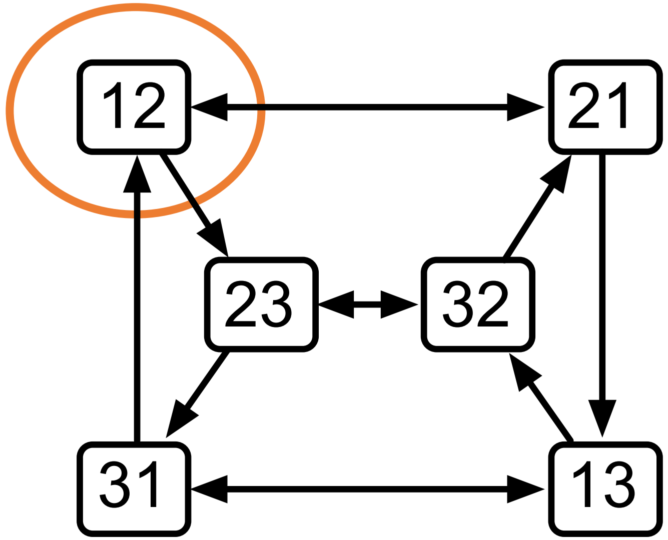


LCLs on cycles

Neighborhood v is "***flexible***" if for all sufficiently large k there is **a walk $v \rightarrow v$ of length k**

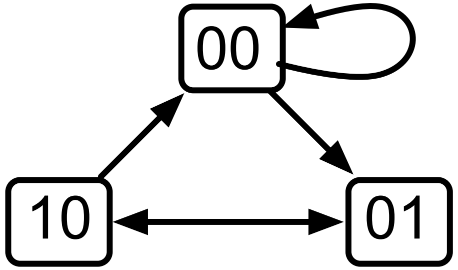
- equivalent: there are walks of coprime lengths
- "**12**" is flexible here, $k \geq 2$

3-coloring



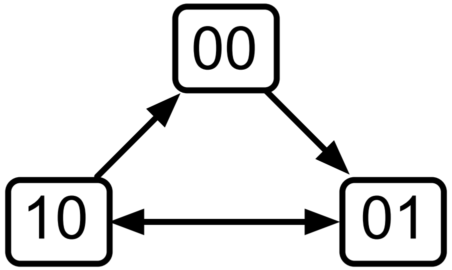
LCLs on cycles

independent set

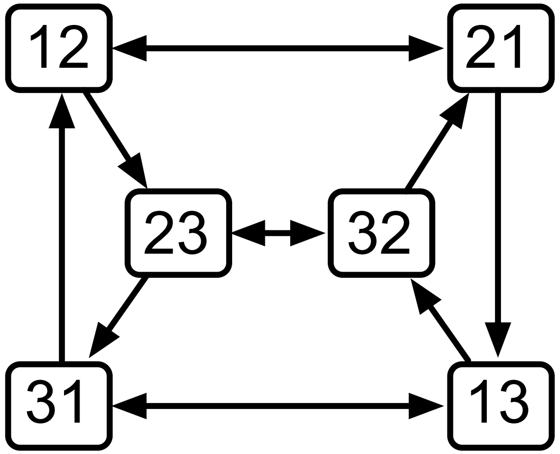


self-loops:
$O(1)$

maximal independent set

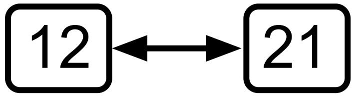


3-coloring



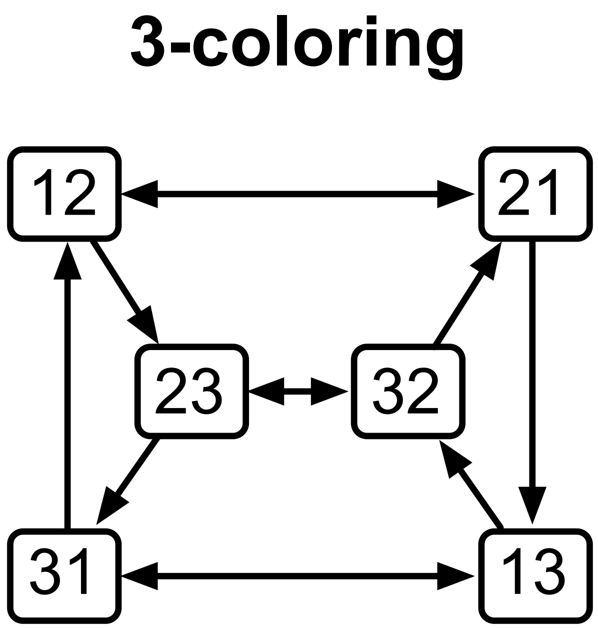
flexible states:
$\Theta(\log^* n)$

2-coloring



otherwise:
$\Theta(n)$

LCLs on cycles



- Given **any LCL problem on cycles**, we can mechanically:
  - represent it as a graph
  - analyze the structure of the graph
  - construct an optimal algorithm for the problem!
- Algorithm synthesis easy with the ***right representation*** of the problem!

Conclusions

Recap of techniques

- Case study 1: **robust counters**
  - computer solves the base case, ***use as a black box***
- Case study 2: **large cuts**
  - computers solves small cases, ***generalize the idea***
- Case study 3: **LCL problems on cycles**
  - algorithm synthesis can be ***fully automated!***

Take-home messages

- ***You are allowed to use computers*** to do theoretical computer science!
- Sometimes algorithm design can be turned into mechanical work that is well-suited for computers

Take-home messages

- We need the right **representations** for:
  - computational problems (inputs)
  - algorithms (outputs)
- Computers are very good at solving ***combinatorial puzzles***
  - graph problems, satisfiability of logical formulas…

Something to think about...

- Do you see possible applications of computational algorithm design ***outside distributed computing***?
- Would it be possible to use computers to ***automatically prove lower bounds***?