

Where Is the Algorithm?

Two tic-tac-toe players and what they mean for teaching algorithms

Samridh Limbu
Deakin University
Melbourne, Australia

Abstract

I built two tic-tac-toe players. One was solved by dynamic programming over 5478 board positions in 9 sweeps. The other learned by playing 200,000 games and never had a model of the game at all. Both finish as a table of numbers I did not type, and nobody wrote a rule in either one that says where to open. That is the puzzle people raise about language models and learned agents, sitting inside an exercise from an undergraduate algorithms unit. I ask where the word “algorithm” points in a learned system, which run-time state belongs to a deployed algorithm, and what an algorithms unit should change because of the answers. The method is conceptual analysis. I test candidate answers against a fixed set of cases and five conditions stated up front, and against what the two players and a second solver printed. My answer is that “the algorithm” is relative to a level of description. Name the level, and the boundary I give below does the rest. Relative to the generic executor, the state a player plays from is input. Relative to the deployed player, that state is part of the procedure that runs. So computational knowledge can be part of an algorithm on the second reading. The boundary turns on the role the state plays at run time, not on whether a human wrote it. The split between a procedure and the state it reads is not new with neural networks. It is already inside dynamic programming, and in the way I was taught it the recurrence got all the attention and the table got thrown away. For teaching, I propose three changes to an algorithms unit and a two-hour lab built on the two players, with three learning outcomes, an answer key and a plan to evaluate it. I have not studied any learners.

CCS Concepts

• **Social and professional topics** → **Computer science education**.

Keywords

algorithms education, definition of algorithm, learned computation, levels of description, dynamic programming, reinforcement learning

1 Introduction

The definition most of us were given is that an algorithm is a sequence of instructions for solving a problem. Knuth’s version is a finite set of rules giving a sequence of operations for a specific type of problem, with five features, finiteness, definiteness, input, output and effectiveness [10]. Sitting behind it is Turing’s analysis of what a person following a fixed procedure can compute [19]. The way I was taught the definition also carried an assumption that’s easy to miss. The knowledge of how to solve the problem lives with the person, and the algorithm is just the way that knowledge gets to the machine. The machine is a fast and empty executor.

Modern systems make that assumption look wrong. A language model summarises a document that nobody wrote a summarising rule for. A network reads handwritten digits with no line anywhere saying a closed top loop means an eight. An agent finds strategies outside traditional Go knowledge [16]. A recommender guesses what you want to watch from what people like you watched. In all of these an algorithm is involved. The awkward question is which part of the system the word points at.

I want to make that question concrete before answering it. The usual examples are all too large to inspect. The learning agent that plays a game is the one I can shrink, so I built two players for tic-tac-toe, and the small case stands in for the large one. The first solves the game by dynamic programming. The second learns it by playing, with no model of the game at all. Both play well. Neither one contains an instruction anyone wrote about any particular board, and both end as a table of numbers that no person typed. Two tables, then. One of those tables came out of a textbook algorithm that has been taught for decades. The other came out of reinforcement learning. If the puzzle about learned systems is real, it is already here, in the smaller of the two.

My position has two parts.

The first is that “the algorithm” is a level-relative term. Not that the question is confused, and not that there is no fact of the matter. What the word points at moves with the level of description you are working at, the way “the machine” moves between a CPU, a virtual machine and a process. Relative to the generic executor, learned parameters are input. Relative to the deployed procedure that plays the game or classifies the digit, they are part of what is running. Both are true. Shifting between them inside a paragraph is not allowed, and I think that’s where most of the argument on this topic goes wrong.

The second is that the boundary that matters is not the one people reach for. The line is not human-written against learned-from-data. It is generic executor against persistent control state, and that line is old. It runs through interpreters, parser tables, comparator arguments and the policies dynamic programming reads off its tables, and it was there long before anyone fitted a weight. Saying so costs me the exciting version of the claim. I do not get to say the definition of an algorithm broke when deep learning arrived. I think the duller answer is the right one and I think it is the more useful one for a curriculum.

The overall question is where the word “algorithm” points in a learned system, and what an algorithms unit should do about it. I split it into three research questions.

RQ1. At which levels of description can “the algorithm” name a part of a learned system, and what does it name at each level?

RQ2. Which run-time state belongs to a deployed algorithm, and does it matter whether a person wrote that state?

RQ3. What should an undergraduate algorithms unit change because of the answers, and how could the change be tested?

I answer them by conceptual analysis. Section 3 says how, and a matched pair of small systems I built (Section 4) is the test bed. Sections 5 to 7 answer RQ1, Section 8 answers RQ2 and Section 9 answers RQ3.

The two levels themselves are an old distinction. Turing drew a version of them in 1950 [20], and computing education already talks about programs and algorithms at several levels [7, 11, 15], so that isn't the new part. Marr and Poggio also split one system into levels of description [12]. Their levels are kinds of explanation of one system. My executor and deployed levels sit roughly inside their algorithm level and differ in which run-time state counts as part of the algorithm, which they do not separate out. The contribution has three parts. The first is a three-part test for which run-time state belongs to a deployed algorithm, with a verdict that does not depend on who wrote the state. The second is that test applied to a matched pair of tables nobody typed, one classical and one learned. The third is three changes to an algorithms unit and a two-hour lab with learning outcomes, an answer key and an evaluation plan. I have not studied any learners.

2 Related Work

The question of what an algorithm is has a literature, and some of what I might have claimed as news was answered in it twenty years ago.

2.1 What an Algorithm Is

Blass and Gurevich survey the attempts at an absolute definition. Their verdict is that the notion is not as settled as Kleene thought. Their words are that “the notion of algorithm is richer these days than it was in Turing’s days”, and that there are algorithms “not covered directly by Turing’s analysis”, naming algorithms that interact with their environments, algorithms whose inputs are abstract structures, and non-discrete algorithms [2]. None of those three reasons is machine learning. Gurevich puts it harder in a later paper. His answer to whether the notion can be defined rigorously is yes and no, and the negative half is that “the notion of algorithm cannot be rigorously defined in full generality, at least for the time being”, because “the notion is expanding” [8]. Moschovakis takes a different route to a related place. His own summary of his answer is that “algorithms are recursive definitions while machines model implementations, a special kind of algorithms” [14].

So the claim that the classical definition is incomplete belongs to them. It is old, it is better argued than I could argue it, and it did not need neural networks. My own bit is narrower than that. Those authors were not asking where in a running learned system the word points, and they were not asking what a curriculum should do about it. Their incompleteness is about defining the notion rigorously in full generality, which is not the same as lacking words for learned systems. I take that first one as given and work on the location question and the teaching question.

Wegner is the cautionary case. He argued that interaction is more powerful than algorithms and that the classical concept is too narrow to cover what computers actually do [23]. The structure of that move is the structure of mine, and the argument over whether an extension like his computes anything the classical model cannot went on for years. Cockshott and Michaelson are one example. Replying to a later version by Wegner and Eberbach, they argue that the input stream of an interactive run can be recorded and fed to an ordinary tape machine, and they conclude that “The Church-Turing thesis remains, for now, secure” [4, Sections 5.2 and 8]. I try to stay on the right side of that by making a claim about description rather than about computational power. Nothing in this paper says a learned system computes a function no Turing machine can.

The level split itself is older than all of this. Writing about learning machines, Turing asks how the rules of operation of a machine can change, when they “should describe completely how the machine will react whatever its history might be”. He grants that those rules are “quite time-invariant”. His answer is that the rules which change in learning “are of a rather less pretentious kind, claiming only an ephemeral validity” [20, p. 458]. That is an early version of my executor level and deployed level. I add a named test for which state belongs at the second level, applied to tables nobody typed, and what follows from it for teaching.

2.2 How Students Understand the Word

Perrenet, Groote and Kaasenbrood defined four levels of abstraction in how computer science students think about the concept of algorithm [15]. At the execution level the algorithm is “a specific run on a concrete specific machine”. At the program level it is a process described in one executable programming language. At the object level it is “not connected with a specific programming language”. At the problem level it is a black box, and the question is which type of algorithm suits a given problem. They found the mean answering level “almost always” varied only from the program level to the object level, and they found little relation between a student’s level and their test results. Their execution level is not my executor level. Theirs is one run on one concrete machine, mine is the generic machinery before any run, with everything it reads counted as input (Section 5). None of their four levels has a place for state that is neither the program nor an input, and that every run reads.

Outside education, Marr and Poggio split one system into “four nearly independent levels of description”, the computation, the algorithm, the mechanism and the hardware [12]. They let an algorithm’s specification draw on information “available from stored models” [12, p. 18], but they don’t separate that stored state from the procedure that uses it.

There is some evidence that the notion is unstable in classrooms already. Bellettini and colleagues ran a three hour workshop for students and teachers, with high school students among them. Pairs got a sheet of eight procedures written in natural language, built on purpose to carry flaws or features that would start an argument, and decided which of them counted as algorithms. Then the pairs merged into groups which compared their decisions and wrote a definition. Over nine workshops the authors coded the definitions from 27 groups against the ACM Encyclopedia entry, which names

the device that runs the method and the notation it is written in. Every definition had an ordered sequence of steps and most asked for the steps to be unambiguous, but only three of the 27 mentioned an interpreter or executor and none mentioned notation. The authors say ambiguity, and getting the interpreter named, was often the turning point. One group worked out that precise means precise enough for the interpreter to know what to do, but that never made it into their written definition. That one comes from one author's notes on the last two workshops, and they do not claim the issues raised there are representative of all group discussions [1]. They place the written definitions at Perrenet's object level, with the talk often moving to the problem level. So the word moves even inside one workshop. That is the gap I am pointing at. The thing participants leave out of their definition is the executor, and the level question is a way of making them put it back.

A working group on how programs are talked about in education draws the line between the two words like this, "an algorithm is the abstract idea of a method to solve a problem, whereas a program is a formal description for such a method" [11, p. 132]. All three works place the algorithm somewhere between a run, a program and a problem. None of them asks which of two runnable things the word names when one of them is a table nobody wrote.

2.3 Notional Machines

Computing education already has a device for showing students the hidden executor. Sorva defines a notional machine as "an idealized abstraction of computer hardware and other aspects of the runtime environment of programs", and says its purpose "is to explain program execution" [17, pp. 59–60]. It is not a student's own mental model, and the typical one is tied to a single language or a few similar ones. Fincher and colleagues call it "a pedagogic device to assist the understanding of some aspect of programs or programming". They point out that running a program is "multi-layered", and that a notional machine deals with only "some layers of description" [7, pp. 22 and 30]. My executor level is close kin to a notional machine but it's not the same kind of thing. A notional machine is a teaching model of how code runs. The executor level is a claim about what the word "algorithm" picks out. For a learned system a notional machine of the forward pass is easy to give, layers, matrix products and an activation function. It can't say why the output is what it is. That last point is my argument and not theirs.

2.4 Machine Learning in the Curriculum

Lonati and colleagues focus on "programs which are implementation of human designed algorithms". They leave out purpose "programs whose implemented logic is not human made but machine produced", and machine learning is one of their examples [11, p. 118]. Following Janlert, they call such programs "dark". A traditional program comes from formalising a method, and a dark one is "produced by processes" like training a neural network. With a dark program you work with "(a sample of) the pairs that belong to the input/output relation" rather than with an algorithm that computes them [11, p. 131]. This paper takes up the case they scoped out. Their choice made sense for their framework and I think the case now needs its own account. Bellettini and colleagues also expect AI tools "to change the general perception of what it means to instruct

a machine" [1, p. 107]. Writing about K–12 teaching, Tedre, Denning and Toivonen argue that several computational thinking concepts, correctness and notional machines among them, "are insufficient for ML and need to be extended" [18]. I agree on correctness, and my third teaching change is one version of that point. The notional machine strand above takes up another item on their list. None of these works says where the algorithm sits in a trained system, and that is the question I take on.

3 Method

This is conceptual analysis, so the method is a way of testing claims and not of collecting data. I state candidate answers and test each one against one fixed set of cases. The candidates are the five places in Section 6 and then the criterion in Section 8. The cases are the six rows of Table 2, plus four harder ones the paper keeps coming back to. A system that keeps its examples, a route planner's map, a system prompt set against what a user types, and a recommender that refits.

Any answer has to meet five conditions, and I state them here, before the analysis. None of them fixes which verdict an answer has to reach.

- (1) It sorts every case without contradicting itself.
- (2) It gives a verdict on two bit-identical tables under the same executor with different histories, one fitted and one typed.
- (3) It says where it stops, and its verdict on the training data, the user and the hardware does not rest on causal influence alone.
- (4) It gives a verdict on a system prompt and on what a user types in the same conversation, and says whether the two differ.
- (5) It says whether two players built on the same value table with different tie rules run the same algorithm.

The two players, and a second solver that plans against a worst case opponent, are there to sink candidates, so here is what they could have shown. If every seed had opened on the same cell, the claim that the learning procedure is the algorithm that plays would have survived. The five seeds opened on three cells. If the tie rule had not changed the second solver's win rate, the claim that the value table alone fixes the deployed algorithm would have survived. It went from 0.994792 to 0.848958 and no value changed. If the learner had played an optimal move in every state, the point about correctness of the state would have had no example. Its greedy move was optimal in 0.9059 of all decision states and in 108 of the 109 it met in play. I chose the cases and wrote the test myself, and Section 10 says what that costs.

4 Two Players for the Same Game

Tic-tac-toe has 5478 reachable board configurations. Of those, 2423 are non-terminal positions with X to move, which are the states my players have to decide in. O has its own 2097, so it is not a big game. Every number below is from my notebook's saved output.

The solved player. The first player treats the game as a Markov decision process, with the opponent folded into the transition probabilities rather than modelled as a second decision maker. Value iteration converges in 9 sweeps. The last sweep changes nothing

at all. The value of the empty board comes out at 0.994791666667, and a backward induction over the same state graph in exact fractions gives exactly 191/192. Policy iteration, run separately, picks the same move as value iteration in all 2423 decision states. Every rate below is exact, scored over the whole game graph rather than counted from sample games. Against a uniformly random opponent this player wins 0.994792 of games and loses none. Against a perfect opponent every game is a draw.

The learned player. The second player is tabular Q-learning. It gets the legal moves, the board it lands on after each move, and the result of each finished game. It never sees a model of the game. After 200,000 episodes its table covers 2420 of the 2423 decision states. Against the same random opponent it wins 0.989583 of games and loses none, and against a perfect opponent it draws every game.

Setup. The learner uses the tabular, model-free Q-learning of Watkins and Dayan, which keeps one value for each state and move [22]. Mine uses a constant step size of 0.3 and no discount, $\gamma = 1.0$. It explores with an epsilon that starts at 1.0 and is multiplied by 0.99995 after every episode, and it ended at 0.00005. A win scores +1, a loss -1 and a draw 0, and a single move earns nothing. A row of the table starts at zero the first time its state is visited, and ties go to the lowest cell index. It trains for 200,000 episodes against an O that picks uniformly at random among its legal moves, with seed 7 for the learner and seed 11 for the opponent. Every 2000 episodes I saved a checkpoint, 100 in all, each holding the exact win, draw and loss of the greedy policy and the cell it opens on. The whole run took 0.9 seconds on my machine and that time covers scoring the 100 checkpoints and the final player too. Training samples the opponent's moves but scoring does not, every rate is averaged over all of the random O's moves, under the same opponent model. The perfect O plays, at each of its moves, the moves that are best for O under minimax, and the scoring spreads evenly over tied ones. Watkins and Dayan prove convergence for step sizes that shrink. Mine stayed at 0.3, so their theorem does not cover this run, and the evidence that the learner plays well is the exact scoring. The solved player uses the same tie rule. It builds a policy once, with a one-step lookahead through the game model from its value table, and at move time it looks the move up. I report one main run, the one the notebook saved in full. I picked tic-tac-toe because it's small enough to solve exactly, so every choice the learner makes can be checked against a known answer. The pair is matched on purpose. Same game, same opponent model, same tie rule, and the main thing that differs is how the state each player plays from got made. The solved player also makes its choices once, before play, where the learner makes them at every move.

Other seeds. The notebook also trained the learner on five more seeds, in a shorter setup of 100,000 episodes with the same step size and an epsilon that also starts at 1.0 and decays each episode. The opponent seeds changed too. For those runs it saved only summaries. Their win rate against the random opponent is printed as 0.9885 ± 0.0013 , and between them they opened on cells 1, 4 and 6, an edge, the centre and a corner. So one learning procedure gave deployed players that open in three different places. The setup isn't the main run's, so I use it only as it was saved.

Table 1: The two players. The numbers are from the notebook's own output. "Plays from" is what each player reads at move time.

	dynamic programming	Q-learning
what it needs	the whole model	one game at a time
states it works over	5478	2423
states it ends up storing	5478	2420
plays from	a policy (2423)	the Q table (2420)
work	9 sweeps	200,000 episodes
win vs random O	0.994792	0.989583
loss vs random O	0.000000	0.000000
loss vs perfect O	0.000000	0.000000
opening cell	corner	centre

Table 1 puts them side by side. They differ on the opening. The solved player opens on a corner, all four are worth 191/192 against a random opponent. The learner's final table opens on the centre, which is worth 95/96. That choice never settled. Its greedy policy opens on a corner at 21 of the 100 checkpoints I took during training, 16 of them in the second half, and the centre is where the last checkpoint happened to land.

Then the part I did not expect. The learner's greedy action matches a dynamic programming optimal action in 0.9059 of all 2423 decision states. Of the 109 decision states that come up when it plays a random opponent, 108 are optimal, and the one it gets wrong is the empty board, which is the centre opening again. Its values are much further out than its actions are. Over the 8622 legal state action pairs in the states it stored, $|Q^{\text{learned}} - Q^*|$ has a mean of 0.3423 and a maximum of 1.157. Taking the largest of the legal entries survives a lot of error spread across them.

5 The Words I Am Using

Most of the disagreement here is people meaning different things by the same word, so I will fix mine before arguing.

An **algorithm** I take to be an abstract procedure, a rule for getting from an input to an output in finitely many definite steps. A **program** is one representation of such a procedure, in source, in bytecode, in machine code, or in some other encoding. An **executor** is the execution procedure a program runs on, the thing a virtual machine or an interpreter or the forward pass of a network implements. I mean that procedure rather than the metal it happens to sit on. **Persistent control state** is state the executor reads to decide what to do. It outlives any one input. It is read again on the next one. A comparator passed into a sort is persistent control state. So is a parser table, a script fed to an interpreter, the policy a dynamic programming solver reads off its value table, and a weight vector. **Computational knowledge** I take to mean persistent control state that supports reliable performance on a task across some input distribution, when an executor interprets it. That definition is thin on purpose. It says nothing about the state being true, or readable, or free of mistakes, and a table that fails the performance test can still be part of the deployed procedure. Section 10 says what that costs.

The rule I hold to from here is that whenever I write "the algorithm" I say which level I mean. Three matter. The *executor level*,

where the algorithm is the generic machinery and everything it reads is input. The *deployed level*, where the algorithm is the particular procedure running on this task, executor and control state together. The *system level*, where the algorithm would be the whole arrangement including the people and the feedback. I argue below that the first two carry their weight as answers to where the algorithm is and the third one does not, but I would rather name it than leave it out.

6 The Five Places It Could Be

Five candidates get put on the table. I will take each and say at which level it is right, and where it fails.

The learning procedure. Stochastic gradient descent, value iteration, Q-learning. Classical algorithms in the fullest sense, all three of them. They are finite, each one stops on a rule you set, a tolerance or an episode count, you can state their complexity, and in the value iteration case I can tell you it took 9 sweeps. None of that is in doubt. It does not account for the deployed behaviour though. Q-learning is one program, and the run I use here gave a player that opens on the centre. The procedure does not tell you that. The five shorter runs in Section 4 used the same update rule every time and opened on an edge, the centre and a corner between them. So the learning procedure is an algorithm, and it is not the algorithm that plays the game.

The inference procedure. The code that reads the table and picks the largest entry, or in a network the sequence of matrix multiplications. Also a classical algorithm, also not the answer on its own. My move selector is a few lines in each player and it ends the same way in both, take the largest entry among the legal cells and break a tie on the lowest index. The learner runs it at move time, over its Q row and the list of legal moves. The solved player runs it once for every position before any game is played, scoring each move with a one-step lookahead through the game model, and keeps the answers as a policy it looks up at move time. Handed a different table, either one plays differently. For the learner, at the executor level the selector is the whole algorithm and the Q table is its input, and at the deployed level it is half of one. For the solved player the selector finished before the first game, so at move time the split is between a lookup and its policy.

I should be careful here, because the strength of a learned system is not always concentrated in the parameters. AlphaGo Zero is a case that cuts against a lazy version of my own argument. Its playing strength comes from a learned network read inside Monte Carlo tree search at move time, and the search does a lot of the work. The authors ran the raw network on its own, picking the most probable move with no lookahead, and it rated 3055 Elo against 5185 for the full system [16]. So the competence there is split between a learned artefact and a procedure. That is a better argument for the pair than for either half of it.

The parameters. The contested one, and I take it up in Sections 7 and 8. Short version. At the executor level they are input, at the deployed level they are part of the procedure. They are never the algorithm alone. A table with nothing to read it is an inert array.

The interaction between model and user. It is tempting to wave this one away as not an algorithm. That would be a dodge, the interaction can be written down. A dialogue with a language model is a loop over turns, with a state that is the conversation so far, a policy that maps state to a next message, and an update that appends. A recommender that refits on what you clicked is a control loop with a delay in it. Those are specifications, not vague things, and Wegner built a whole position on taking them seriously [23].

My claim about it is narrower than his, and it depends on whether the loop changes the state. A chat with a model whose weights and prompt stay fixed is one run of a deployed algorithm. The conversation so far is that run's input, as Section 8 says. A recommender that refits on your clicks is different. Every refit changes the control state, so a new deployed algorithm starts at each one. The loop there is the learning procedure running online with the user supplying its data, and it decides which control state obtains at a given time. Write the deployed player as the pair of an executor and the state it reads, and an online system is a time-indexed sequence of those pairs, with the loop as the process that maintains the index. That keeps the recommender inside the account instead of excluding it by accident, and it explains why the two questions people ask about such a system need different answers. If I want to know why my learner opens on the centre, the loop is irrelevant, the table settles it. If I want to know why a recommender's suggestions narrowed over six months, no single table will tell me, and the loop is the only place the answer lives.

All of it together. The composite answer sounds the most reasonable and I think it is the weakest, because nobody who gives it says where it stops. If the algorithm includes the model and the user, does it include the people who wrote the training data. The engineers who set the reward. The electricity. Each of those changes the output if you change it, so a criterion built on causal influence has no way to exclude any of them, and an algorithm that includes the power grid is not a concept anyone can use. I need a stopping rule and I give one in Section 8. It is not causal influence.

7 The Strongest Objection

The best argument against me is not that AI is secretly hand written. It is this.

A universal machine has a fixed transition procedure and it reads a program off a tape. Change the tape and it computes a different function. The standard reading is that the tape is not part of the universal machine's algorithm. Inference code plus weights has exactly that shape, generic execution machinery and an array that decides what gets computed. So the weights are a program-like input to the inference algorithm and not a part of it, and classical theory already has every word it needs. Interpreter, program, tape, input, table, state. Nothing is missing and nothing needs expanding.

I think this objection is right at the executor level, and I think it kills a tempting move. The move is: swap the weights and the system computes a different function, therefore the weights are part of the algorithm. That does not follow, and it proves too much. Swap the comparator in a sort and it computes a different function, swap the tape and the universal machine computes a different function, swap the table in a table driven parser and it parses a different grammar. Determining an output is not the same as being

a constituent of the procedure. If it were, the input would be part of the algorithm too, and then the word means nothing at all.

The compilation analogy, which is the other move people reach for, runs the wrong way as well. Weights are like machine code, the argument goes, machine code is an algorithm, so weights are an algorithm. But machine code is not part of the CPU. If weights are machine code then the analogy puts them outside the inference engine, which is the objection's conclusion and not mine. The analogy is still worth something, it shows that being found by optimisation rather than typed does not stop a thing being program-like, which is the point Karpathy was gesturing at [9]. It just does not put the weights inside the executor.

The other level survives. The universal machine argument settles the executor level and it does not touch the deployed one. The tape is not part of the universal machine's procedure. It belongs to the program being executed, and that program is a real object which we name, analyse and argue about all the time. When I ask which sorting algorithm a process is running I do not answer "the CPU". I answer by naming the thing the code encodes. That is the deployed level, and at that level the algorithm is the executor and its state together, so the state is part of what the thing is.

Partial evaluation gives the deployed level a classical form. A partial evaluator takes a program and part of its input, and produces a version of the program specialised to that input. Specialise an interpreter to one fixed program and the result "behaves exactly the same" as that program. This is the first Futamura projection, which Williams and Perugini credit to Futamura [24]. Cook and Lämmel work a case close to mine, an interpreter for table-driven state machines specialised to one machine, where in the result they aim for "The accept states and the transition table are no longer present as data structures". Their first attempt stops early, because the table is looked up with a key known only at run time, and they rewrite the interpreter so partial evaluation can carry on [5]. So a table can disappear into the procedure that reads it, and whether it does depends on how that procedure is written. My solved player does a version of this by hand. It runs its move selector over the value table once for every position before any game, and at move time there's only the policy left to read. Neither source does this to a network and its weights. I only use the idea to show the deployed level is old, it's the thing a specialiser would hand you.

There is a place where the analogy inside the objection breaks and it is worth stating, though it does not go the way the objection wants. A tape usually has an independent specification. Someone wrote the program, someone can read it, and a compiler that produced it can be related to a source. A trained weight vector has none of that. There is a data set, an objective and a training run behind it, but no source to recover, and any rule you want out of it has to be dug out afterwards. That is a difference in what we can know about the state, not a difference in the role the state plays at run time. It is the reason most of the questions we want to ask about a trained model have to be asked at the deployed level, and it is not a reason to redraw the executor level.

The objection therefore forces a level distinction. It does not pick the level.

8 Can Computational Knowledge Be Part of an Algorithm?

Yes, at the deployed level, with a boundary that does not depend on where the state came from. This yes is cheap once the definitions in Section 5 are in place, and I would rather say so. The part that needs arguing is that the deployed level deserves the word algorithm. Section 7 is where I argue it does.

The criterion. The test ranges over state the executor reads at run time. Anything that is not that is out by category, and the rest gets sorted like this. State counts as part of a deployed algorithm when it is persistent, when the executor reads it to decide what to do, and when the deployment holds it fixed across the inputs the procedure ranges over instead of taking it as one of them. By a deployment I mean what gets fixed and released to run over a stated class of inputs. The test admits weights, a Q table, the policy my solved player reads, a comparator, a parser table and a script fed to an interpreter. For that player the value table is out, because nothing reads it at move time. A player that did its lookahead at move time would read the value table and the game model, and then both would be in. A memo table that fills while a program runs is persistent and gets read, but the deployment does not hold it fixed, so it is not in. For a pure function it only changes how long an answer takes, and it filling up isn't a refit like the recommender's. A system prompt held fixed across a deployment gets in too, and the words a user types in one conversation do not. That last split is not visible inside a single forward pass and it is not meant to be. The deployment decides which state counts as control. The same bytes can be control in one deployment and input in another. A deployment fixes one procedure and runs it over a class of conversations, so the prompt is control at the deployed level. A conversation is one run of that procedure, so what gets typed into it is that run's input.

What it rules out. The input to a single run. Activations that live and die inside one forward pass. The user. The training data, which shaped the state without being interpreted at run time. The hardware. The last three are not run-time state at all, so they never reach the sorting step, and that stops the composite answer expanding out to the power grid. A system that keeps its examples and reads them at run time is a different case. There the examples are persistent, read to decide the output and held fixed by the deployment, so the same rule puts them in. A route planner's map gets in on the same terms. I accept that, it's the same kind of thing as a comparator. It is a rule about the role a thing plays at run time. It says nothing about how far up the causal chain you are willing to walk.

The price. My criterion does not care about provenance and I do not think it should. Two systems running bit-identical tables under the same executor have the same status, whether that table was fitted, evolved, copied off a disk or typed by hand. Nothing about the computation is different. So the category I am pointing at is not one that machine learning invented, it is large and it is old, and neural networks are a recent and extreme member of it. I would rather say that than draw a boundary that turns on a fact about history which the running system cannot see. The same test says when two deployed algorithms are one. Same executor, same tie rule and same state make the same deployed algorithm. A different tie

rule over the same value table gives a different policy, so a different deployed algorithm, as the tie break result below shows.

Statistics had a version of this argument. Breiman describes an algorithmic modelling culture whose approach is to find a function $f(x)$, in his words “an algorithm that operates on x to predict the responses y ”, and sets it against a data modelling culture that fits a stochastic model of the process instead [3]. He is calling the fitted predictor an algorithm, in those words. Whatever you think of the rest of that argument, the move is the one I am making, and it was made in 2001 about decision trees, forests and the neural nets of the day rather than about anything with a billion parameters.

Where the behaviour nobody wrote comes from. I am half agreeing with the other side, so I owe the other side’s explanation too. Nothing in my learner was ever told to open on the centre. Nothing in the solved player was told to open on a corner either. In both cases a person wrote a general update rule, a way of scoring an outcome and a way of picking among scores, and then a lot of arithmetic ran. The behaviour is the arithmetic’s output. A finite rule applied many times over data produces consequences nobody enumerated in advance, and my two players show the same effect twice with two very different rules. This is an old answer. Turing took on the same worry in 1950 as Lady Lovelace’s objection. As he quotes her, the Analytical Engine “can do *whatever we know how to order it to perform*” [20, p. 450]. To the variant that says a machine can never take us by surprise, he replied that “Machines take me by surprise with great frequency”. Thinking otherwise, he said, assumes that all the consequences of a fact spring into the mind together with the fact [20, pp. 450–451]. My two players are that answer run twice, on tables nobody typed. The surprise belongs to me. The system was only doing arithmetic. I could not have predicted the centre opening by reading the Q-learning update, and I could not have predicted the corner opening by reading the Bellman equation. The corner was sitting there the whole time waiting to be computed. The centre was too, but only once the seed and the stopping point were fixed. So the answer to how a system does what it was never programmed to do does not need a new kind of causation. It needs the reminder that specifying a process is not the same as enumerating its results, which is a thing algorithms have done since long before they were fitted to anything.

What I am not claiming. Not that all the competence lives in the parameters. It does not. Architecture, the objective, the training distribution, how input gets encoded and what the decode rule is all carry problem-specific information. I solved the same game a second way to check that, against a worst case opponent rather than a random one. The policy read off that table scores 0.994792 against a random opponent when ties go to the lowest index, the same rate as my solved player, and 0.848958 when they go to the worst of the tied best moves, worst as scored against the random opponent. Not one value in the table changed. All that moved was the rule for picking among moves the table rates equally. Not that learned state is knowledge in any strong sense. My learner picks a non-optimal move in roughly a tenth of the 2423 decision states and it plays well anyway. And not that a trained model computes a deterministic function, which is only true once you have fixed the sampling and a stopping rule.

9 What I Think This Means for Teaching

If the split between a procedure and the state it reads is old, then the fix for algorithms teaching is not a second subject bolted onto the first. Denning’s complaint about computing education is that a list of core technologies and their great ideas “does little to convey the great principles of computing”, which he says have been “interred beneath layers of technology in our understanding and our teaching” [6]. He also turns down the obvious picture of a pantheon with one pillar per principle, and calls it an unsatisfactory portrayal. An extra pillar for learned computation is the kind of picture he rejects. I want a change inside the unit that is already there. The workshop results in Section 2 suggest the executor gets left out of definitions, and the changes below try to put it back.

Three changes, in the order I would make them.

Teach the table as the artefact. The way I was taught it, dynamic programming is a recurrence and the assessment is the recurrence. The table is scaffolding, it gets printed once for a small example and then thrown away. But the play comes from the table. My solved player reads its policy off the table once, and that policy plays the game. A student who has filled a memo table and then been asked what that table *is* after the run, whether it’s data or program, whether it would still work if the opponent changed, has met the whole question on ground where they can check the answer themselves. The comparison is not expensive to set up.

Teach the level question directly. Ask where the algorithm is for a short list of cases and require the level to be named in the answer. Sorting with a comparator passed in, a universal machine with a tape, a table driven parser, a dynamic programming player, a Q-learning player and a trained digit classifier will do for a start. The classifier is the one worth spending time on. The same convolution code with a different weight file could sort small photos of clothes instead. So at the executor level that code is the whole algorithm and the weights are its argument. But anyone who uses it would call it a digit recogniser. Students should be able to say that at the executor level the table is input and at the deployed level it is part of the procedure without finding that answer evasive. They should also spot the slide between the two when they read it in a paper, or hear it in an argument.

Teach correctness of the state as well as of the steps. This is the one I would fight for. A loop invariant proves the steps. For a learner it does not tell you whether the table those steps produced is right in the places nobody looked. My learner is a clean example. Its policy is optimal in 0.9059 of the 2423 positions overall, and in 108 of the 109 positions that come up when it plays a random opponent. The update rule can tell you some of this and not the rest. It says where errors are likely, in states the learner rarely meets and in near-ties, since a step size that stays at 0.3 keeps the values moving. It can’t tell you which moves are wrong. For that I had to score the learner against a solved reference. The one wrong state it meets in play is the empty board, visited in every game, where the corner beats the centre by 1/192 against a random opponent. That gap is small next to the 0.3423 mean error in the learner’s values. A curriculum that only ever asks whether a procedure is correct has no vocabulary for that gap, and the gap is where deployed systems fail. Classical correctness tools still apply where they apply and I am not proposing

Table 2: The lab’s answer key. At the executor level the algorithm is the second column on its own. At the deployed level it is the second and third columns together. The fourth column is input at both levels.

case	executor	state the deployment holds fixed	input to one run
sorting with a comparator passed in	the sort routine	the comparator	the list to sort
universal machine	its transition procedure	the program on the tape	the data on the tape
table driven parser	the parser driver	the parse table	the token string
dynamic programming player	a policy lookup	the policy, read once off the value table	the board
Q-learning player	the greedy selector and its tie rule	the Q table	the board
trained digit classifier	the forward pass	the weight file	the image

to drop any of them. Alongside them students need generalisation treated as a correctness notion, PAC style guarantees and what they assume [21], and the practice of holding out a reference and measuring where the two disagree. This is my version of the point Tedre, Denning and Toivonen make about correctness [18].

None of this needs a new unit. It needs the existing one to give the table the same attention it gives the recurrence.

9.1 A Two-Hour Lab

The three changes fit in one lab. I wrote it for students who have met dynamic programming but not reinforcement learning. The teacher hands out my notebook’s code for both players and the worst case solver, with its tie rules and seeds, so the good answers below are what that code prints. Students work in pairs and fill in a sheet with one box per step. By the end of the lab a student should be able to do three things.

LO1. Name the level (executor, deployed or system) whenever they say where the algorithm is in a system, and say what the word picks out at that level.

LO2. Apply the test in Section 8 to a piece of run-time state and justify whether it belongs to the deployed algorithm.

LO3. Check a learned table against a solved reference, and explain what reasoning about the update rule alone can and cannot tell you about where the two disagree.

LO1 goes after the gap in the workshop study, where 3 of 27 definitions named an executor and getting the interpreter named was often the turning point [1]. LO3 goes after the correctness point of Tedre, Denning and Toivonen [18]. Perrenet and colleagues found little relation between a student’s level and their test results [15]. Their levels measure how abstractly a student talks about algorithms. LO1 aims at a narrower slip, moving between levels without saying so, and their result doesn’t say whether that slip costs students anything. A review of abstraction in computing education reports the advice that students “should be made aware of any change of abstraction level” [13, p. 632], and LO1 is one way to do that. The lab also gives students more time with dynamic programming, which they have already met.

Materials. Cells are numbered 0 to 8 row by row, so the corners are 0, 2, 6 and 8 and the centre is 4. The random O picks uniformly among its legal cells. The perfect O is the one from Section 4. The solved player’s value iteration folds the random O into the transitions. The worst case solver runs the same value iteration with O taking its best reply at every move. Its “worst” tie rule picks, among the moves its own value table rates best, the one the random-model

value table scores lowest. Scoring is exact. Every rate is averaged over all the moves the opponent could make, which for the perfect O means its tied best ones.

The steps below give the time, what students run, what they write down and what a good answer looks like. The good answers are the ones my own run gives, and the numbers are from Sections 4 and 8.

- (1) *Before (0–10 min).* Each student writes a definition of “algorithm” in their own words and hands it in.
- (2) *Solve (10–30 min).* The teacher opens with a five to eight minute briefing on the game as a Markov decision process and on the Q update. Then students run value iteration and read the policy off the value table. Write down the number of sweeps, the value of the empty board and the opening cell. A good answer has 9 sweeps, 0.994791666667, which is 191/192, and a corner, and notes that all four corners are worth the same.
- (3) *Learn (30–55 min).* Train the learner on several seeds, one at a time, at 100,000 episodes per seed, the setup my five saved seeds used. The notebook printed “200000 episodes in 0.9s” on my machine, and that includes scoring the 100 checkpoints, so a pair should be able to train a handful of players in a few minutes. Write down each player’s opening cell and its exact win rate against the random opponent, then pool the openings on the board. A good answer notices that the openings can differ from seed to seed while the update rule never changed, and concludes that the learning procedure is not the algorithm that plays. My five saved seeds opened on three different cells. If a class gets the same opening every time, the checkpoints of the main run still show it moving. In my 200,000 episode run the greedy policy opened on a corner at 21 of the 100.
- (4) *Swap the tie rule (55–70 min).* Read two policies off the worst case solver’s table, one with ties going to the lowest index and one with ties going to the worst tied move, worst as scored against the random opponent, and score both. Write down both win rates and what changed between them. A good answer has 0.994792 and 0.848958, says that no value in the table moved, and concludes that the table alone does not fix the deployed algorithm.
- (5) *Score (70–90 min).* Compare the greedy move of the main run’s learner with the solved player’s optimal moves in every decision state, and then only in the states met against the random opponent. Write down both rates and one state where the learner is wrong. A good answer finds the second

rate much higher than the first (mine were 0.9059 and 108 of 109). It says the Q-learning update can point at rarely met states and near-ties, but it cannot say which moves are wrong.

- (6) *Six cases (90–105 min).* In groups, fill in Table 2 with only the first column given, then say where the algorithm is at the executor level and at the deployed level for each row. The teacher also gives each group a short paragraph that slides between levels, and the group finds the slide.
- (7) *After (105–120 min).* Each student writes the definition again, without seeing the first one.

The paragraph for step 6 can be short. This one I wrote to slide on purpose. “The network reads its weights on every input and they stay fixed once the model ships, so the weights are part of the algorithm. But the weights are only numbers the inference code reads, so every model trained this way runs the same algorithm.” The first sentence works at the deployed level and the second at the executor level, and nothing in the text marks the change. A good answer finds the word “algorithm” changing level between the two sentences and says both can be true once the level is named.

Steps 3, 4 and 6 serve LO1 and LO2, and step 5 serves LO3. The system level is left out of the table on purpose. It’s for the discussion at the end, where the interaction loop in Section 6 makes a good closing case.

9.2 How the Lab Could Be Evaluated

I have not run this lab and I have not studied any learners. The rest of this section is a plan. Running it would need ethics approval before any student work is collected, and consent from each student.

Primary measure. After the lab each student works alone on four new items. Two are cases not in Table 2, a route planner’s map and a chat deployment where the student sorts the system prompt and the user’s messages. The third is a new paragraph that slides between levels. These three score 0, 1 or 2. A 0 names no level, a 1 names a level but the answer slides, and a 2 names the level, says what the word picks out there and holds it. For LO2 the marker also checks whether all three parts of the test are applied and whether the verdict matches the key. The fourth item is for LO3. It gives a handful of states from a learner’s table next to the solved values, and it is marked on finding a wrong move and on what the student says the update rule can and cannot tell them. The same four items come back a week later to separate learning from recall.

Secondary measure. Code the before and after definitions with the scheme Bellettini and colleagues used, against the same encyclopedia features, and count how many name an interpreter or executor. This measures naming an executor without being asked, it is not a measure of LO1. Their 3 of 27 [1] is only a loose reference point. It came from groups of teachers, of university students, of high school students, or of teachers and university students together, writing after deciding which procedures counted as algorithms, so the population and the task both differ from mine. As a second code, place each definition on Perrenet’s four levels [15]. Perrenet’s execution level is one run on one machine, and a coder should not read it as my executor level. Two people should code independently and report how well they agree. Perrenet and

colleagues report a Cohen’s kappa of .64, after dropping four items [15], so I expect the level coding to be the hard part.

Analysis. Within a class, the before and after definitions are paired proportions, so I would compare them with McNemar’s test. Between the two classes, I would compare scores on the new items and on the delayed items.

A single class with no comparison group can show change but not its cause, a fairer design gives a second class the same definitions and the same new items around a dynamic programming lab that never asks the level question. Writing a definition first is itself a prompt to think about the word, so the before task will prime students in both classes, and the comparison has to carry that.

The plan also says in advance what would count against the lab. If the lab class scores no better than the comparison class on the new items, or loses the gain a week later, the lab didn’t do its main job. If students name a level on the six cases and then miss the slide in the step 6 paragraph, they have learned the level words as labels and are not using them. Either result would send me back to the lab design before any claim about the level question itself.

10 Limits, and What Would Change My Mind

My evidence is one game with 5478 states, two solvers and one learner, all tabular. A Q table is a much friendlier object than a weight vector. Far too friendly, maybe. It’s indexed by states I can name and I can diff it against an exact answer, which is the only reason I can quote a 0.9059 at all. You cannot do that for a language model, and someone could say the analogy carries less than I want it to. My reply is that the analogy is doing structural work and not quantitative work. I am claiming the two artefacts have the same shape, an executor plus persistent control state, and I am not claiming a network is as legible as a Q table. There is a second gap I should name. My solved player’s value table can be checked against the recurrence that produced it. Nothing that produced my Q table gives me a check like that, so I scored it against a solved reference instead. Provenance changes how you audit a table, even when it changes nothing about how the table runs. There is a third gap. For a network the executor is less generic than an interpreter. The forward pass fixes an architecture, and Section 8 already counts architecture as carrying some of what the system knows. So my line is cleanest for tables and blurrier for networks. I would still be more confident with a case built on something non-tabular though. The next one to build is the digit classifier from Section 9, one executor with two weight files, put through the same test.

I also wrote the definitions, the test and the cases myself, so the cases may fit the test partly because I chose them. The route planner and the system that keeps its examples are in Section 8 because they strain it, and a reader who finds a case that breaks it has found a real flaw in the account, not a quirk of the example.

The numbers come from one main run with one pair of seeds. The five other seeds ran in a shorter setup and only their summaries were saved, so I can’t say how much the 200,000 episode player varies from seed to seed. The convergence theorem does not cover the run either, so its quality rests on the exact scoring alone.

My definition of computational knowledge is thin on purpose and it pays for that. It counts a table full of mistakes as knowledge, since it only asks for reliable performance on some distribution.

Anyone who wants knowledge to mean something nearer to true or justified will not accept it, and I do not have an answer for them beyond saying the stronger notions did not seem to locate anything. The third clause of my test is partly a choice too. People already draw a line between what ships with a system and what arrives while it runs, and the clause leans on it.

This is also a paper about teaching that contains no data about learners. I have not run the lab in Section 9.1 with a class and I do not know whether the level distinction is teachable in an undergraduate algorithms unit or just sounds like hair splitting to someone who has not yet been surprised by a table. That is the obvious next thing to do and it is the thing I cannot claim.

What would change my mind. If someone gave a criterion for algorithmic membership that turns on how the state was produced and that survives the bit-identical case, I would give up the provenance-free part of Section 8. If someone showed the deployed level can be eliminated, that every question we ask about what a trained model is doing can be answered at the executor level with the tape treated as input, then my position collapses into the objection in Section 7 and the classical account wins outright. I do not think that can be done, because naming which program a process is running is not optional, but I have not proved it cannot.

11 Conclusion

Where is the algorithm in a modern AI system? The learning procedure is one, the inference procedure is another, and the deployed thing that recognises the digit or plays the game is a third, made of the inference procedure together with the state it reads. An interaction loop is not a fourth algorithm. A chat with a fixed model is one run of the third, and a loop that refits is the first one run online, settling which deployed one you have at a given moment. The answer that says all of it together is not an answer until it says where it stops. Which one the word points at depends on the level you are asking at, and the failure mode in this debate is not picking a wrong level. It is moving between levels without saying so.

Computational knowledge can be part of an algorithm, at the deployed level, when it is persistent state that the executor reads and the deployment holds fixed across the inputs the procedure ranges over. So the classical definition does not need a new clause. It needs applying at the deployed level too, and at that level the rules include the state they read. That admits learned weights. It also admits the policy a dynamic programming solver builds, a parser table and a comparator, and having to admit those is a feature. It means the thing people find unsettling about learned systems was already in the curriculum, sitting in an exercise where a student can check it in an afternoon. The lab in Section 9.1 is one way to let them check it, and finding out whether it works needs a class.

Table 3 puts the answers against the research questions.

Two tables, one game, both written by a machine. The classical one has been in the syllabus the whole time, and I was taught around it.

References

- [1] C. Bellettini, V. Lonati, M. Monga, and A. Morpurgo. 2024. To Be Or Not To Be ... An Algorithm: The Notion According to Students and Teachers. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2024)*. ACM, 102–108. doi:10.1145/3626252.3630950

Table 3: The research questions, answered.

RQ	answer	section	evidence and test cases
1	The executor at one level, the executor with its state at the next. The system level gives no answer until it says where it stops.	5–7	One update rule, five seeds, three openings. The universal machine objection holds at the executor level only.
2	State that is persistent, read to decide what to do and held fixed by the deployment. Who wrote it does not matter.	8	The tie rule moves 0.994792 to 0.848958 with no value changed. The bit-identical case.
3	Teach the table as the artefact, the level question and correctness of the state, and test it with the lab.	9	108 of 109 met states against 0.9059 overall. No learner data yet.

- [2] A. Blass and Y. Gurevich. 2003. Algorithms: A Quest for Absolute Definitions. *Bulletin of the European Association for Theoretical Computer Science* 81 (2003), 195–225.
- [3] L. Breiman. 2001. Statistical Modeling: The Two Cultures. *Statist. Sci.* 16, 3 (2001), 199–215. The 199–231 range often quoted for this paper takes in the comments and the rejoinder.
- [4] P. Cockshott and G. Michaelson. 2007. Are There New Models of Computation? Reply to Wegner and Eberbach. *Comput. J.* 50, 2 (2007), 232–247. doi:10.1093/comjnl/bxl062
- [5] W. R. Cook and R. Lämmel. 2011. Tutorial on Online Partial Evaluation. In *Proceedings of the IFIP Working Conference on Domain-Specific Languages (DSL 2011) (EPTCS, Vol. 66)*, O. Danvy and C. Shan (Eds.), 168–180. doi:10.4204/EPTCS.66.8
- [6] P. J. Denning. 2003. Great Principles of Computing. *Commun. ACM* 46, 11 (2003), 15–20.
- [7] S. Fincher, J. Jeuring, C. S. Miller, P. Donaldson, B. du Boulay, M. Hauswirth, A. Hellas, F. Hermans, C. Lewis, A. Mühling, J. L. Pearce, and A. Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITICSE-WGR '20)*. ACM, 21–50. doi:10.1145/3437800.3439202
- [8] Y. Gurevich. 2012. What Is an Algorithm?. In *SOFSEM 2012: Theory and Practice of Computer Science (Lecture Notes in Computer Science, Vol. 7147)*. Springer, 31–42. Read in Microsoft Research Technical Report MSR-TR-2011-116, which contains the SOFSEM text.
- [9] A. Karpathy. 2017. Software 2.0. Medium. <https://karpathy.medium.com/software-2-0-a64152b37c35> 11 November 2017.
- [10] D. E. Knuth. 1997. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (3rd ed.). Addison-Wesley. Section 1.1.
- [11] V. Lonati, A. Brodnik, T. Bell, A. P. Csizmadia, L. De Mol, H. Hickman, T. Keane, C. Mirolo, and M. Monga. 2022. What We Talk About When We Talk About Programs. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education (ITICSE-WGR '22)*. ACM, 117–164. doi:10.1145/3571785.3574125
- [12] D. Marr and T. Poggio. 1976. *From Understanding Computation to Understanding Neural Circuitry*. A.I. Memo 357. Artificial Intelligence Laboratory, Massachusetts Institute of Technology.
- [13] C. Mirolo, C. Izu, V. Lonati, and E. Scapin. 2021. Abstraction in Computer Science Education: An Overview. *Informatics in Education* 20, 4 (2021), 615–639. doi:10.15388/infedu.2021.27
- [14] Y. N. Moschovakis. 2001. What Is an Algorithm? In *Mathematics Unlimited, 2001 and Beyond*, B. Engquist and W. Schmid (Eds.). Springer, 919–936.
- [15] J. Perrenet, J. F. Groote, and E. Kaasenbrood. 2005. Exploring Students' Understanding of the Concept of Algorithm: Levels of Abstraction. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '05)*. ACM, 64–68. doi:10.1145/1067445.1067467
- [16] D. Silver et al. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2017), 354–359.
- [17] J. Sorva. 2012. *Visual Program Simulation in Introductory Programming Education*. Ph.D. Dissertation. Aalto University.
- [18] M. Tedre, P. J. Denning, and T. Toivonen. 2021. CT 2.0. In *Proceedings of the 21st Koli Calling International Conference on Computing Education Research (Koli Calling '21)*. ACM, Article 3, 8 pages. doi:10.1145/3488042.3488053
- [19] A. M. Turing. 1936–7. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Math. Soc., ser. 2* 42, 1 (1936–7), 230–265.
- [20] A. M. Turing. 1950. Computing Machinery and Intelligence. *Mind, New Series* 59, 236 (1950), 433–460. doi:10.1093/mind/LIX.236.433

Where Is the Algorithm?

- [21] L. G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27, 11 (1984), 1134–1142.
- [22] C. J. C. H. Watkins and P. Dayan. 1992. Q-learning. *Machine Learning* 8, 3–4 (1992), 279–292. doi:10.1007/BF00992698
- [23] P. Wegner. 1997. Why Interaction Is More Powerful Than Algorithms. *Commun. ACM* 40, 5 (1997), 80–91.
- [24] B. M. Williams and S. Perugini. 2016. Revisiting the Futamura Projections: A Diagrammatic Approach. *Theoretical and Applied Informatics* 28, 4 (2016), 15–32. Read in the arXiv preprint 1611.09906v3.