On Teaching Discrete Mathematics
to Freshman Computer Science Students

Faron Moller and Liam O'Reilly

Swansea University, Swansea, United Kingdom
{F.G.Moller,L.P.OReilly}@swansea.ac.uk

Abstract. Discrete Mathematics is an inevitable part of any undergraduate computer science degree programme. However, today's computer science student typically finds this to be at best a necessary evil with which they struggle to engage. Twenty years ago, we started to address this issue seriously in our university, and we have instituted a number of innovations throughout the years which have had a positive effect on engagement and, thus, attainment. In this paper, we describe and motivate the innovations which we introduced, and provide a detailed analysis of how and why engagement and attainment levels varied over two decades as a direct result of these innovations.

1 Introduction

A typical first-year undergraduate student likes writing computer programs as this provides instant gratification: the computer does what you tell it to do. This is often why they choose to do computer science at university. As they proceed through their undergraduate education, they learn how to be more and more creative and to get the computer to do more and more exciting things.

Stopping to think about whether the things that they make the computer do are in fact the *right* things to do – both in a technical sense as well as an ethical sense – is often unattractive to these students. Technical considerations tend to mean unwelcome mathematics disrupting what they want to do; and ethical considerations tend to mean unwelcome philosophy doing the same.

If ethical considerations are ignored when producing correct software, the implications are societal and generally predictable; whilst potentially serious, this is not the concern of this paper. If technical (i.e., logical) errors are inadvertently introduced, the consequences can be devastating, even fatal in the case of safety-critical applications. Only by developing a rigorous, mathematically-based approach to programming – often the anathema of a budding computer hacker – can trust-worthy software engineers be produced.

There are a great number of excellent textbooks for teaching computer science students the discrete mathematics which they will find necessary in their pursuit of the subject. Without prejudice, we can cite [6, 9, 17] as exemplars which have gone through multiple editions and commonly appear in the reading lists of relevant courses. However, whilst often written with computer science

applications in mind, the standard presentation in such texts is inevitably mathematical in nature, with a methodical approach to formal syntax and semantics taking centre stage. As the modern computer science student often lacks the mathematical maturity of their predecessors (as argued below), this can be a hindrance to engagement and, thus, academic attainment.

That the modern computer science student is in general less mathematically minded than a generation ago is well recognised, and its causes now understood. Moller and Crick [13] give a detailed account of the history of computing education in UK schools: from a strong position in the 1980's with the introduction of the BBC Micro into every school along with a curriculum for teaching the fundamentals of programming including hardware, software, Boolean logic and number representation; through the 1990's and beyond where the emergence of pre-installed office productivity software led to the computing curricula being permeated – and overwritten – by basic IT skills; "Death-By-Powerpoint" became a common epithet for the subject. Beyond the arguments and references provided in [13], we can note a trend towards omitting mathematics as a prerequisite subject for studying computer science: of the 164 undergraduate computer science programmes offered by 105 universities in the UK, over 60% of these do not require high-school mathematics as a prerequisite [7].

There is a recognised digital skills shortage providing a high demand for computer science graduates [8], and an eagerness on the part of universities to fill places. However, with ever more students declaring in entrance statements that they are choosing to study computer science due to a love of digital devices rather than a love of the subject – and thus ever less prepared for the intellectual, logical and mathematical problem-solving challenges this entails – it can be a challenge in making some of the mathematical content of the curriculum palatable. This is especially true in the current climate where student satisfaction is a key indicator which universities are required by law in the UK to publish in their recruitment and marketing.

This paper describes an innovative approach that we have developed for teaching discrete mathematics to first-year university computer science students. By adopting and adapting our approach over the past twenty years from a traditional starting point, we have substantially increased the success rate – and substantially decreased the failure rate – of our students. Fig. 1 shows how the percentage of students attaining a first-class mark (one over 70%) rose from 2% in 2000-2001 to over 60% in 2017-2018 and 2018-2019, whilst those failing the course (with a mark under 40%) dropped over the same time frame from 56% to under 2%. The figure also shows the class sizes which have more than tripled over the most recent five years which explains a noticeable dip in attainment which, we show, required further tweaking of our delivery model to address. The fact that this success is based on our approach is borne out by reflecting on annual student feedback for the various courses which students take across their programme of study; our delivery model is contrasted favourably against traditional approaches used in other courses taken by the same students, and recorded attendance (and hence engagement) is highest in this course.
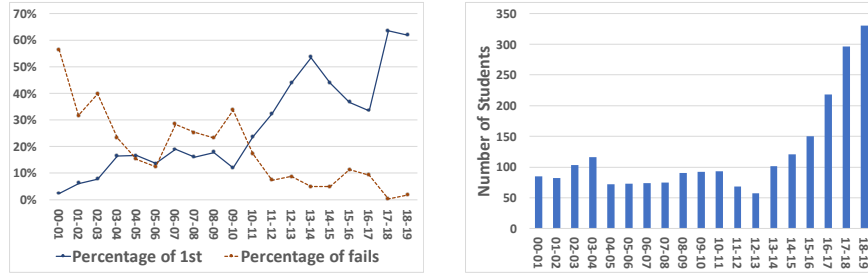
Fig. 1. Trends of students achieving 1st-class and failing results; and class sizes.

2   Background

The nature of computer science education is changing, reflecting the increasing ubiquity and importance of its subject matter. In the last decades, computational methods and tools have revolutionised the sciences, engineering and technology. Computational concepts and techniques are starting to influence the way we think, reason and tackle problems; and computing systems have become an integral part of our professional, economic and social lives. The more we depend on these systems – particularly for safety-critical or economically-critical applications – the more we must ensure that they are safe, reliable and well designed, and the less forgiving we can be of failures, delays or inconveniences caused by the notorious "computer glitch."

Unlike for traditional engineering disciplines, the mathematical foundations underlying computer science are often not afforded the attention they deserve. The civil engineering student learns exactly how to define and analyse a mathematical model of the components of a bridge design so that it can be relied on not to fall down, and the aeronautical engineer learns exactly how to define and analyse a mathematical model of an aeroplane wing for the same purpose. However, software engineers are typically not as robustly drilled in the use of mathematical modelling tools. In the words of the eminent computer scientist Alan Kay [10], "most undergraduate degrees in computer science these days are basically Java vocational training." But computing systems can be at least as complex as bridges or aeroplanes, and a canon of mathematical methods for modelling computing systems is therefore very much needed. "Software's Chronic Crisis" was the title of a popular and widely-cited Scientific American article from 1994 [5] – with the dramatic term "software crisis" coined a quarter of a century earlier by Fritz Bauer [16] – and, unfortunately, its message remains valid a quarter of a century later.

University computer science departments face a sociological challenge posed by the fact that computers have become everyday, deceptively easy-to-use objects. Today's students – born directly into the heart of the computer era – have grown up with the Internet, a billion dollar computer games industry, and mobile phones with more computing power than the space shuttle. They often

choose to study computer science on the basis of having a passion for using computing devices throughout their everyday lives, for everything from socialising with their friends to enjoying the latest films and music; and they often have less regard than they might to the considerations of what a university computer science programme entails, that it is far more than just *using* computers. In our experience, many of these students are easily turned off the subject when faced with a traditional course in discrete mathematics, with many of these, e.g., transferring into media or information studies. This has motivated us to reflect on our presentation of discrete mathematics, which has resulted in the following key considerations, all of which we have gleaned – and from which we have learned – from student feedback.

- *Do not rely on a service course provided by your mathematics department.* This is by no means a criticism of the mathematics department. It is simply the case that many students will not appreciate the importance of a course taken in a different department. At best, they may consider it peripheral to their studies, and at worst they will thus disengage completely.
- *Do not call it (discrete) mathematics.* A simple change of name from "discrete mathematics for computer science" to "modelling computing systems" in 2010-2011 was enough for us to witness a substantially increased level of engagement and attainment with the course, as made evident in Fig. 1. There was no other change that year to add to the cause of this effect.
- *Do not formalise early on.* The standard approach to, e.g., propositional logic is to present the formal syntax and semantics of the logic and emphasise the precise form and function of the connectives. The approach we have adopted is to stress the careful use of English, and to introduce logical symbols as mere shorthand for writing out English sentences. Formalism becomes far easier to adapt to if and once the students are comfortable with working with the concepts.
- *Exploit riddles and games.* As described later through characteristic examples, riddles and games provide an effective way to instil the rigours of computational thinking.
- *Use regular interactive small-group problem sessions.* We supplement three hours of weekly whole-class lectures with a one-hour small-group problem session (of 30-50 students) in which the emphasis is on the students carrying out computational problem-solving tasks, typically in pairs. We are confident in our thesis that this matters, as tweaking the sizes and regularity of these groups through the years coincides with peaks and dips in the attainment graphs. In particular, see the next consideration.
- *Keep these problem session groups small.* As can be seen in Fig. 1, attainment dropped between 2014 and 2017 as class sizes grew, but more than recovered in 2017-2018 despite a huge increase in the overall class size. This was due to an increase in the number of problem session groups; whilst the whole-class lectures became far less personable due to the huge numbers, the decrease in the sizes of the problem session groups resulted in much better results. Again, this being the only substantive change to delivery, we are confident in attributing the positive effect to this.

The first half of our course covers standard discrete mathematics topics: sets, propositional and predicate logics, functions and relations. Whilst it would be instructive to explore our approach to these topics, in this paper we explore our approach to teaching some of the topics from the latter part of the course. The reasons for this are two-fold. Firstly, the topics we discuss are typically not present in standard discrete mathematics courses; we make a case for why they ought to be so, for scientific reasons as well as due to the scope for presenting them in an engaging style. Secondly, our aim is to demonstrate the informal and engaging approach we take to the subject; we do so with the novel topics, leaving it to the readers' imagination as to how such techniques – e.g., the use of Smullyan- and Dudeney-style puzzles and riddles [4, 18] – can be applied to the earlier standard topics.

3   Games and Winning Strategies

There is a long-standing tradition in disciplines like physics to teach modelling through little artefacts. The fundamental ideas of computational modelling and thinking as well can better be learned from idealised examples and exercises than from many real world computer applications. Our approach employs a large collection of logical puzzles and mathematical games that require no prior knowledge about computers and computing systems; these can be more fun and sometimes more challenging than, e.g., analysing a device driver or a criminal record database. Also, computational modelling and thinking is about much more than just computers.

In fact, games play a far more important role in our approach: they provide a novel approach to understanding computer software and systems. When a computer runs a program, for example, it is in a sense playing a game against the user who is providing the input to the program. The program represents a strategy which the computer is using in this game, and the computer wins the game if it correctly computes the result. In this game, the user is the adversary of the computer and is naturally trying to confound the computer, which itself is attempting to defend its claim that it is computing correctly, that is, that the program it is running is a winning strategy. (In software engineering, this game appears in the guise of *testing*.) Similarly, the controller of a software system that interacts with its environment plays a game against the environment: the controller tries to maintain the system's correctness properties, while the environment tries to confound them.

This view suggests an approach to addressing three basic problems in the design of computing systems:

1. **Specification** refers to the problem of precisely identifying the task to be solved, as well as what exactly constitutes a solution. This problem corresponds to the problem of defining a winning strategy.
2. **Implementation** or **synthesis** refers to the problem of devising a solution to the task which respects the specification. This problem corresponds to the problem of implementing a winning strategy.

```
repeat as necessary:
    x := x mod y;
    if x=0 then return y;
    y := y mod x;
    if y=0 then return x
```
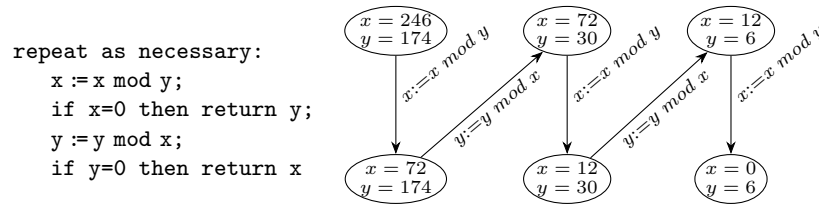

Fig. 2. Computing the greatest common divisor.

3. **Verification** refers to the problem of demonstrating that the devised solution does indeed respect the specification. This problem corresponds to the problem of proving that a given strategy is in fact a winning strategy.

This analogy between the fundamental concepts in software engineering on the one hand, and games and strategies on the other, provides a mode of computational thinking which comes naturally to the human mind, and can be readily exploited to explain and understand software engineering concepts and their applications. It also motivates our thesis that game theory provides a paradigm for understanding the nature of computation.

4   Labelled Transition Systems

Labelled transition systems have always featured in the computer science curriculum, but traditionally (and increasingly historically) only in the context of finite automata within the study of formal languages. In our course we introduce them as general modelling devices, starting with an intuitively-clear and familiar use. Fig. 2 presents Euclid's algorithm for computing the greatest common divisor of two numbers x and y, alongside a labelled transition system depicting the algorithm being hand-turned on the values 246 and 174.

In general, a computation – or more generally a process – can be represented by a labelled transition system (LTS), which consists of a directed graph, where the vertices represent states, and the edges represent transitions from state to state, and are labelled by events. An LTS is typically presented pictorially as in Fig. 2, with the states represented by circles and the transitions by arrows between states labelled by actions.

As a further example, consider the lamp process depicted in Fig. 3. The lamp has a string to pull for turning the light on and off, and a reset button which resets the circuit if a built-in circuit breaker breaks when the light is on. At any moment in time the lamp can be in one of three states:

- OFF – the light is off (and the circuit breaker is set);
- ON – the light is on (and the circuit breaker is set); and
- BROKEN – the circuit breaker is broken (and the light is off).
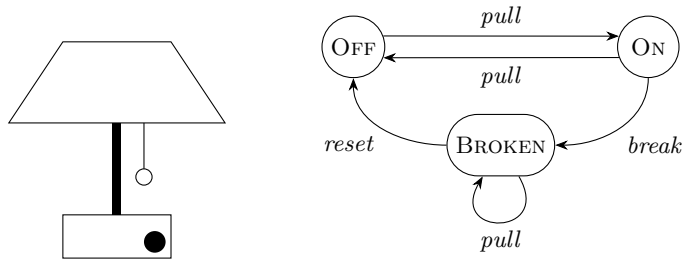
Fig. 3. The lamp process.

In any state the string can be pulled, causing a transition into the appropriate new state (from the state BROKEN, the new state is the same state BROKEN). In the state ON, the circuit breaker may break, causing a transition into the state BROKEN in which the reset button has popped out; from this state, the reset button may be pushed, causing a transition into the state OFF. (Note: discussions of design decisions naturally arise with the decision to always reset into the OFF state, regardless of the number of string pulls carried out in the BROKEN state. This provides a useful excursion into the problems that arise in the requirements analysis phase of software engineering.)

These two examples demonstrate the simple, but effective, use of LTSs as a means of modelling computing problems and real world objects.

### 4.1   Introducing LTSs with puzzles

Whilst the definition of a labelled transition system is surprisingly straightforward for such a powerful formalism, getting students to engage with it requires some ingenuity. Fortunately, this is equally straightforward by resorting to well-known recreational puzzles.

4.2   The Man-Wolf-Goat-Cabbage Riddle

The following riddle was posed by Alcuin of York in the 8th century, and more recently tackled by Homer Simpson in a 2009 episode of The Simpsons titled Gone Maggie Gone.

> *A man needs to cross a river with a wolf, a goat and a cabbage. His boat is only large enough to carry himself and one of his three possessions, so he must transport these items one at a time. However, if he leaves the wolf and the goat together unattended, then the wolf will eat the goat; similarly, if he leaves the goat and the cabbage together unattended, then the goat will eat the cabbage. How can the man get across safely with his three items?*

The puzzle can be solved by modelling it as an LTS as depicted in Fig. 4. A state
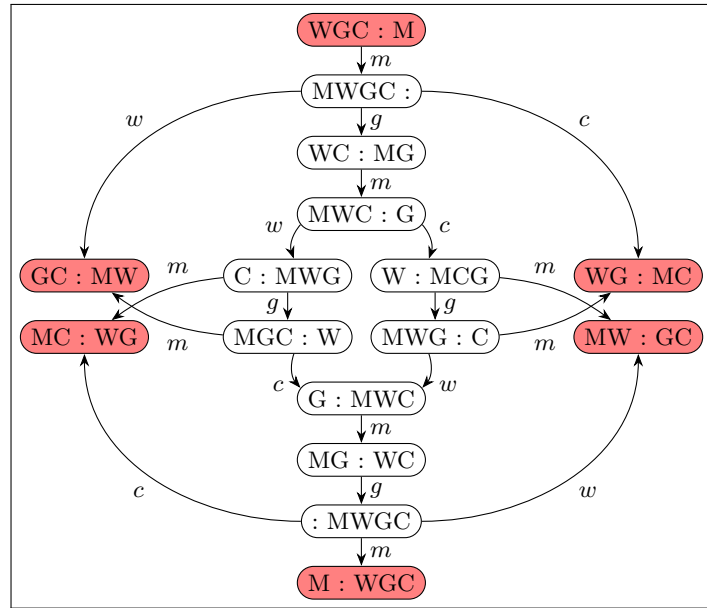
Fig. 4. The Man-Wolf-Goat-Cabbage LTS.

of the LTS will represent the current position (left or right bank) of the four
entities (man, wolf, goat, cabbage); and there will be four actions representing
the four possible actions that the man can take:

- m = the man crosses the river on his own;
- w = the man crosses the river with the wolf;
- g = the man crosses the river with the goat; and
- c = the man crosses the river with the cabbage.

The initial state is MWGC : (meaning all are on the left bank of the river),
and we wish to find a sequence of actions which will lead to the state : MWGC
(meaning all are on the right bank of the river). However, we want to avoid
going through any of the six dangerous states WGC:M, GC:MW, WG:MC,
MC:WG, MW:GC and M:WGC. There are several possibilities (all involving
at least 7 crossings), for example: g, m, w, g, c, m, g.

4.3   The Water Jugs Riddle

In the 1995 film Die Hard: With a Vengeance, New York detective John McClane
(played by Bruce Willis) and Harlem dry cleaner Zeus Carver (played by Samuel
L. Jackson) had to solve the following problem in order to prevent a bomb from
exploding at a public fountain.

$$(i,j) \xrightarrow{\;fillA\;} (5,j) \qquad\qquad\qquad\quad if \quad i=0$$

$$(i,j) \xrightarrow{\;fillB\;} (i,3) \qquad\qquad\qquad\quad if \quad j=0$$

$$(i,j) \xrightarrow{\;emptyA\;} (0,j) \qquad\qquad\qquad\quad if \quad i>0$$

$$(i,j) \xrightarrow{\;emptyB\;} (i,0) \qquad\qquad\qquad\quad if \quad j>0$$

$$(i,j) \xrightarrow{\;AtoB\;} \big(\max(0,i{+}j{-}3),\min(3,i{+}j)\big) \qquad if \quad i>0 \; and \; j<3$$

$$(i,j) \xrightarrow{\;BtoA\;} \big(\min(5,i{+}j),\max(0,i{+}j{-}5)\big) \qquad if \quad i<5 \; and \; j>0$$

Fig. 5. Water jug riddle moves.

Given only a five-gallon jug and a three-gallon jug, neither with any markings on them, fill the larger jug with exactly *four gallons of water from the fountain, and place it onto a scale in order to stop the bomb's timer and prevent disaster.*

This riddle – and many others like it – was posed by Abbot Albert in the 13th Century, and can be solved using an LTS. A state of the system underlying this riddle consists of a pair of integers (i,j) with $0{\leq}i{\leq}5$ and $0{\leq}j{\leq}3$, representing the volume of water in the 5-gallon and 3-gallon jugs A and B, respectively. The initial state is $(0,0)$ and the final state you wish to reach is $(4,0)$.

There are six moves possible from a given state (i,j) as listed in Fig. 5. Drawing out the LTS (admittedly a daunting task in this instance yet a useful exercise), we get the following 7-step solution:

$$(0,0) \xrightarrow{\;fillA\;} (5,0) \xrightarrow{\;AtoB\;} (2,3) \xrightarrow{\;emptyB\;} (2,0) \xrightarrow{\;AtoB\;} (0,2)$$

$$\xrightarrow{\;fillA\;} (5,2) \xrightarrow{\;AtoB\;} (4,3) \xrightarrow{\;emptyB\;} (4,0).$$

These simple riddles and puzzles allow students to easily grasp and understand the powerful concept of labelled transition systems. After seeing only a few examples, they are able to model straightforward systems by themselves using LTSs. Once an intuitive understanding has been established, the task of understanding the mathematics behind LTSs becomes less foreboding.

5 Verification via Games

Having introduced a formalism for representing and simulating (the behaviour of) a system, the next question to explore is: *Is the system correct?* In its most basic form, this amounts to determining if the system matches its specification,
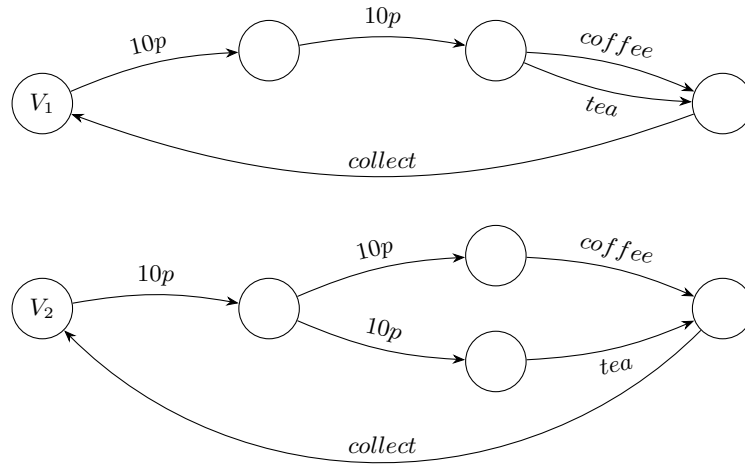
Fig. 6. Two Vending Machines.

where we assume that both the system and its specification are given as states of some LTS. For example, consider the two vending machines V_1 and V_2 depicted in Fig. 6, where V_1 is taken to represent the specification of the vending machine while V_2 is taken to represent its implementation. Clearly the behaviour of V_1 is somehow different from the behaviour of V_2: after *twice* inserting a 10p coin into V_1, we are *guaranteed* to be *able* to press the coffee button; this is *not* true of V_2. The question is: *How do we formally distinguish between processes?*

5.1   The formal definition of equivalence

A traditional approach to this question relies on determining if these two states are related by a *bisimulation relation*, which is a binary relation R over its states in which whenever $(x, y) \in R$:

- if $x \xrightarrow{a} x'$ for some x' and a, then $y \xrightarrow{a} y'$ for some y' such that $(x', y') \in R$;
- if $y \xrightarrow{a} y'$ for some y' and a, then $x \xrightarrow{a} x'$ for some x' such that $(x', y') \in R$.

Simple inductive definitions already represent a major challenge for undergraduate university students; so it is no surprise that this coinductive definition of a bisimulation relation is incomprehensible even to some of the brightest postgraduate students – at least on their first encounter with it. It thus may seem incredulous to consider this to be a first-year discrete mathematics topic, even if it is a perfect application for exploring equivalence relations as taught earlier in the course. However, there is a straightforward way to explain the idea of bisimulation equivalence to first-year students – a way which they can readily
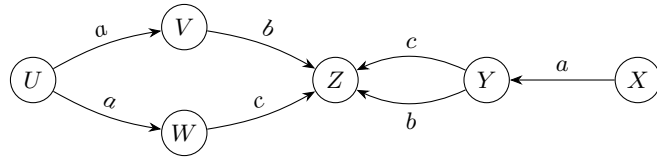
Fig. 7. A simple LTS.

grasp and are happy to explore and, indeed, play with. The approach is based on the following game.

5.2   The Copy-Cat Game

This game is played between two players, typically referred to as Alice and Bob. We start by placing tokens on two states of an LTS, and then proceed as follows.

1. Alice moves *either* of the two tokens forward along an arrow to another state; if this is impossible (that is, if there are no arrows leading out of either node on which the tokens sit), then Bob is declared to be the winner.
2. Bob must move the *other* token forward along an arrow which has *the same label* as the arrow used by Alice; if this is impossible, then the Alice is declared to be the winner.

This exchange of moves is repeated for as long as neither player gets stuck. If Bob ever gets stuck, then Alice is declared to be the winner; otherwise Bob is declared to be the winner (in particular, if the game goes on forever).

Alice, therefore, wants to show that the two states holding tokens are somehow different, in that there is something that can happen from one of the two states which cannot happen from the other. Bob, on the other hand, wants to show that the two states are the same: that whatever might happen from one of the two states can be copied by the other state.

It is easy to argue that two states should be considered equivalent exactly when Bob has a winning strategy in this game starting with the tokens on the two states in question; and indeed this is taken to be the definition of when two states are equal, specifically, when an implementation matches its specification.

As an example, consider playing the game on the LTS depicted in Fig. 7. Starting with tokens on states U and X, Alice has a winning strategy:

- Alice can make the move $U \xrightarrow{a} V$.
- Bob must respond with the move $X \xrightarrow{a} Y$.
- Alice can then make the move $Y \xrightarrow{c} Z$.
- Bob will be stuck, as there is no c-transition from V.

This example is a simplified version of the vending machine example; and a straightforward adaptation of the winning strategy for Alice will work in the game starting with the tokens on the vending machine states V_1 and V_2. We thus have an argument as to why the two vending machines are different.

5.3   Relating winning strategies to equivalence

Whilst this notion of equality between states is particularly simple, and even entertaining to explore, it coincides precisely with the complicated coinductive definition of when two states are bisimulation equivalent. Seeing this is the case is almost equally straightforward.

- Suppose we play the copy-cat game starting with the tokens on two states x and y which are related by some bisimulation relation R. It is easy to see that Bob has a winning strategy: whatever move Alice makes, by the definition of a bisimulation relation, Bob will be able to copy this move in such a way that the two tokens will end up on states x' and y' which are again related by R; and Bob can keep repeating this for as long as the game lasts, meaning that he wins the game.
- Suppose now that R is the set of pairs of states of an LTS from which Bob has a winning strategy in the copy-cat game. It is easy to see that this is a bisimulation relation: suppose that $(x, y) \in R$:
  - if $x \xrightarrow{a} x'$ for some x' and a, then taking this to be a move by Alice in the copy-cat game, we let $y \xrightarrow{a} y'$ be a response by Bob using his winning strategy; this would mean that Bob still has a winning strategy from the resulting pair of states, that is $(x', y') \in R$;
  - if $y \xrightarrow{a} y'$ for some y' and a, then taking this to be a move by Alice in the copy-cat game, we let $x \xrightarrow{a} x'$ be a response by Bob using his winning strategy; this would mean that Bob still has a winning strategy from the resulting pair of states, that is $(x', y') \in R$.

We have thus taken a concept which baffles postgraduate research students, and presented it in a way which is well within the grasp of first-year undergraduate students.

5.4   Determining who has the winning strategy

Once the notion of equivalence is understood in terms of winning strategies in the copy-cat game, the question then arises as to how to determine if two particular states are equivalent, ie, if Bob has a winning strategy starting with the tokens on the two given states. This isn't generally a simple prospect; games like chess and go are notoriously difficult to play perfectly, as you can only look ahead a few moves before getting caught up in the vast number of positions into which the game may evolve.

Here again, though, we have a straightforward way to determine when two states are equivalent. Suppose we could paint the states of an LTS in such a way that any two states which are equivalent – that is, from which Bob has a winning strategy – are painted the same colour. The following property would then hold.

If any state with some colour C has a transition leading out of it into a state with some colour C', then every *state with colour C has an identically-labelled transition leading out of it into a state coloured C'.*

That is, if two tokens are on like-coloured states (meaning that Bob has a winning strategy) then no matter what move Alice makes, Bob can respond in such a way as to keep the tokens on like-coloured states (ie, a position from which he still has a winning strategy). We refer to such a special colouring of the states a *game colouring*.

To demonstrate, consider the following LTS.
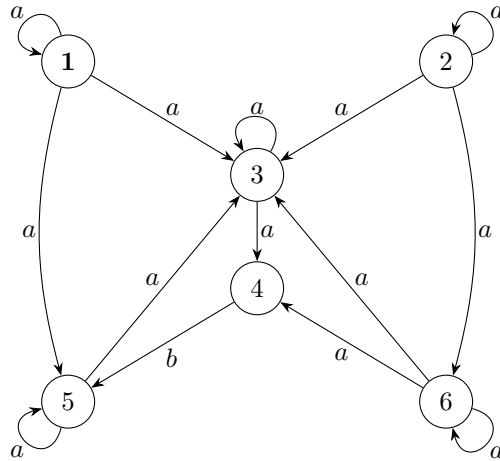
At the moment, all states are coloured white, and we might consider whether this is a valid game colouring. It becomes readily apparent that it is not, as the white state 4 can make a *b*-transition to the white state 5 whereas none of the other white states (1, 2, 3, 5 and 6) can do likewise. In fact, in any game colouring, the state 4 must have a different colour from 1, 2, 3, 5 and 6. Hence we paint it a different colour from white; say green:
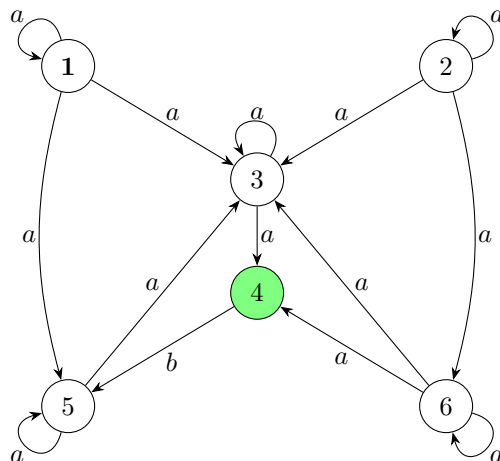
We again consider whether this is now a valid game colouring. Again it becomes apparent that it is not, as the white states 3 and 6 have a-transitions to a green state, whereas none of the other white states 1, 2 and 5 do. And in any game colouring, the states 3 and 6 must have a different colour from 1, 2 and 5. Hence we paint these a different colour from white and green; say yellow:
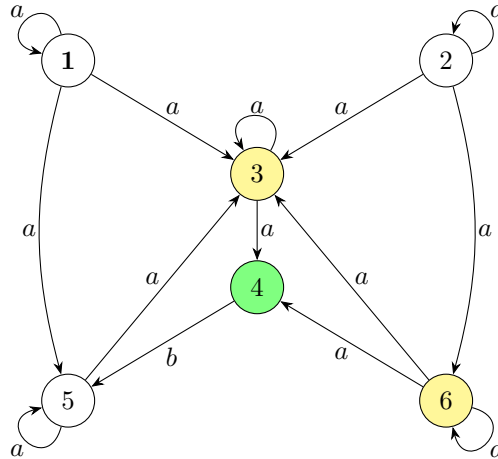
We again consider whether this is now a valid game colouring. This time we find that it is, as every state can do exactly the same thing as every other state of the same colour: every white state has an a-transition to a white state and an a-transition to a yellow state; every yellow state has an a-transition to a yellow state and an a-transition to a green state; and every green state has a b-transition to a white state.

   At this point we have a complete understanding of the game, and can say with certainty which states are equivalent to each other. This is an exercise which first-year students can happily carry out on arbitrarily-complicated LTSs, which again gives testament to the effectiveness of using games to great success in imparting difficult theoretical concepts to first-year students – in this case the concept of partition refinement.

6   Conclusion

We teach first-year Discrete Mathematics in the guise of modelling computing systems; and we find that our students quickly and easily understand the modelling of computing systems when it is done in a way which nurtures their willingness to engage. Starting with formal syntax and semantics and complicated real world examples, in our experience, makes the task very daunting, difficult and generally unpleasant for students. However, appealing to their existing understand of how the world works, using puzzles as a medium, students can quickly become comfortable using mathematical concepts such as LTSs. A

similar lesson is learnt when it comes to teaching verification: starting with the formal definition of bisimulation (or similar) is an uphill battle from the start, even for postgraduate research students. However, starting from games like the copy-cat game, such topics become immediately accessible.

We have used this approach for over a decade to teach discrete mathematics incorporating the modelling and verification of computing systems as part of our first-year undergraduate programme, resulting in the publication of the course textbook [14]. With the fine-tuning of our approach, and abiding by the considerations outlined in Section 2, we have succeeded in maximising attainment levels of the students through active and interested engagement.

Of course, problem solving through recreational mathematics – which is ultimately what we are exploiting in our approach – has very many proponents, and there is a long and extensive history of books marketed towards the mathematically-inquisitive. We are by no means alone in recognising the power of applying recreational mathematics to the development of computational problem solving skills; as relevant exemplars we note Averbach and Chein's *Problem Solving Through Recreational Mathematics* [1], Backhouse's *Algorithmic Problem Solving* [2], Levitin and Levitin's *Algorithmic Puzzles* [11]; and Michalewicz and Michalewicz's *Puzzle-Based Learning* [12]. What we offer in particular is an embedding of the approach from day one of the first year of our students' undergraduate journey, in particular to engage them in a topic – discrete mathematics – that they typically struggle with, both academically and in terms of recognising its relevance in the subject. In this sense, we are closely related to the various approaches that have been developed of late for introducing school-aged audiences to computational thinking. In this vein we note the CS Unplugged[1] and the CS4Fun[2] initiatives. Indeed, much of our material has been adapted into school workshops for the Technocamps[3] initiative.

The "informal" way in which we approach the teaching of formal methods has many parallels with Morgan's *(In)Formal Methods: The Lost Art* [15]. The course described in this report is for upper-level computer science students who are already adept at writing programs who are studying software development methods, whereas our course is for first-year students and thus very much preliminary. Nonetheless, many of the findings in [15] – in particular as reflected in the student feedback – are replicated in our course, where positive feedback is provided on: the interactive and hands-on approach; the amusing exercises and assignments; the class room style teaching; the overall teaching methodology with dedicated tutors; and the means by which the relevance of the course is stressed.

As a final note, many of the considerations that we have identified as being important in teaching mathematics to computing students are reflected in [3] as being useful and thus adopted in their novel approach to teaching computing to mathematics students.

---

[1] csunplugged.org

[2] cs4fn.org

[3] technocamps.com

References

1. Averbach, B., Chein, O.: Problem Solving Through Recreational Mathematics. Dover (1980)
2. Backhouse, R.: Algorithmic Problem Solving. Wiley (2011)
3. Betteridge, J., Davenport, J.H., Freitag, M., Heijtljes, W., Kynaston, S., Sankaran, G., Traustason, G.: Teaching of computing to mathematics students. In: Proceedings of the 3rd Conference on Computing Education Practice, CEP 2019, Durham, United Kingdom, 9 January, 2019. pp. 12:1–12:4 (2019)
4. Dudeney, H.: The Canterbury Puzzles, and Other Curious Problems. Andesite Press (2015)
5. Gibbs, W.W.: Software's chronic crisis. Scientific American **271**(3), 86–95 (2004)
6. Hein, J.L.: Discrete Structures, Logic, and Computability. Jones and Bartlett, 4th edn. (2017)
7. Higher Education Statistics Agency (HESA): Recruitment data for computer science courses in the uk (2019), https://www.hesa.ac.uk
8. House of Commons Science and Technology Committee: Digital skills crisis: Second Report of Session 2016-2017 (7 June 2016)
9. Johnsonbaugh, R.: Discrete Mathematics. Pearson, 8th edn. (2017)
10. Kay, A.: A conversation with Alan Kay. ACM Queue **2**(9), 20–30 (2004)
11. Levitin, A., Levitin, M.: Algorithmic Puzzles. Oxford University Press (2011)
12. Michalewicz, Z., Michalewicz, M.: Puzzle-Based Learning. Hybrid Publishers (2010)
13. Moller, F., Crick, T.: A university-based model for supporting computer science curriculum reform. Journal of Computers in Education **5**(4), 415–434 (2018)
14. Moller, F., Struth, G.: Modelling Computing Systems. Springer-Verlag (2013)
15. Morgan, C.: (In-)Formal methods: The lost art. In: Engineering Trustworthy Software Systems. Lecture Notes in Computer Science, vol. 9506, pp. 1–79. Springer (2016)
16. Naur, P., Randell, B. (eds.): Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October 1968. NATO Scientific Affairs Division (1969)
17. Rosen, K.H.: Discrete Mathematics and Its Applications. McGraw-Hill, 8th edn. (2019)
18. Smullyan, R.: To Mock a Mockingbird: And Other Logic Puzzles. Oxford University Press (2000)