Teaching Mathematics and Its Applications (2010) **29**, 155–163
doi:10.1093/teamat/hrq007 Advance Access publication 28 April 2010

Discrete Mathematics and its applications

ALAN OXLEY*
Computer and Information Sciences Department,
Universiti Teknologi PETRONAS, Bandar Seri Iskandar, 31750 Tronoh, Malaysia

**Email: alanoxley@petronas.com.my*

The article gives ideas that lecturers of undergraduate Discrete Mathematics courses can use in order to make the subject more interesting for students and encourage them to undertake further studies in the subject. It is possible to teach Discrete Mathematics with little or no reference to computing. However, students are more likely to be interested in a subject if they are able to appreciate its use. There is, therefore, a strong case for teaching Discrete Mathematics in context. Lecturers are faced with the task of conveying mathematical material, some of which is new to students and some of which they will have met before. Lecturers must attempt to foster mathematical dexterity. All of this takes time. Teaching the subject in context can be achieved using little, or no, additional time. Of the wide range of Computer Science subjects, Artificial Intelligence and Software Agents are particularly rich in problems that are easy to understand and for which mathematics is needed in order to formally describe the problem as well as to solve it.

Downloaded from teamat.oxfordjournals.org at VIVA Community Colleges on March 1, 2011

1. Introduction

Discrete Mathematics is a core subject in any Computer Science degree. It is included in the degree in order to give students grounding in the type of mathematics that will be of general use over the course of their studies. Most of the other courses making up the degree use mathematics to a greater or lesser extent. There are one or two courses where there is extensive use of mathematics, such as courses on 'formal methods' and 'software specification' (Alagar & Periyasamy, 1998). Several Computer Science courses are concerned with Software Engineering. This describes the whole life cycle of software and the steps that should be taken to ensure that the software produced is properly engineered (Pressman, 2009). There are several areas where mathematics is used in this subject. For example, when designing a program a number of standard diagram types are used. One of these is a graph, comprising vertices and edges. Propositional and other types of logic are used extensively in Artificial Intelligence (AI) and Software Agents. The latter subject stems from AI, yet is different.

AI is usually concerned with the design of large computer programs to tackle highly complex problems such as machine vision and natural language processing. Software Agents, on the other hand, is concerned with relatively small programs that are autonomous, such as the program controlling a robot. It is also concerned with groups of these small programs residing on different computers, or robots, and collaborating in order to achieve certain goals (Wooldridge, 2009). Examples of AI and Software Agent applications abound. One involves robotic vehicles collecting precious material from

© The Author 2010. Published by Oxford University Press on behalf of The Institute of Mathematics and its Applications. All rights reserved. For permissions, please email: journals.permissions@oxfordjournals.org

the surface of Mars (Steels, 1990). Another example is where a space probe makes decisions for itself, rather than relying on a ground crew (NASA's report on Deep Space 1, 2001). A further example is where we have a robotic arm that stacks and unstacks blocks (Fikes & Nilsson, 1971). Several applications involve the use of a square grid. In a 'vacuum world', we can explore the logic of a robotic vacuum cleaner (Russell & Norvig, 1995). In 'Tileworld' (Pollack, 1990) we can explore a video game, in which a software agent is programmed to push tiles into holes. The game is complex in that holes randomly appear and disappear. Yet another application of Software Agents is a simulated robot submarine called HOMER (Vere & Bickmore, 1990). Experimental programming languages have been written specifically for programming software agents. One of these is Concurrent MetateM (Fisher, 1994). This makes use of a temporal logic and it is interesting to note the type of problems that can be tackled.

The purpose of this article is threefold. First, it shows some applications of Discrete Mathematics in Computer Science. This is of especial help to those Discrete Mathematics lecturers who are mathematicians and are unfamiliar with Computer Science. Second, it aims to describe applications that are likely to engender interest in undergraduates studying the subject. Third, it provides an opportunity for Discrete Mathematics lecturers to update their curriculum to include applications that are of current importance. Software Agent applications have grown in popularity and in recent years have been used extensively in researching new types of distributed computing architectures, such as grid computing and service-oriented architecture. The remainder of the article is arranged as sections, with each section describing a topic in Discrete Mathematics. The topics covered are: software architecture, programming, finite-state machines, propositional logic and other logics.

2. Software architecture

A common way of handling a large or complex task, in any sphere of life, is to break it down into parts. (In Computer Science we call these parts 'components' or 'modules'). We do this so as to be able to manage the complexity of the task. Each of the parts is then subdivided, and so on. In computing, this activity of breaking up a task is referred to as 'problem decomposition'. Let us now consider an example. Assume that we are designing software. One component of the program involves customers making payments. We can break this component down into three components corresponding to the different ways that a customer can pay—by cash, cheque or credit card. We can think of 'payment' as the parent component and 'cash', 'cheque' and 'credit card' as the child components. This hierarchical breakdown is termed 'generalization' in software design. Let us try to do something similar in mathematics. Consider the set of natural numbers $\{0, 1, \ldots\}$. Let us regard this as a child set. It would be an interesting exercise to get students to think of a suitable parent set. Here are some possible answers:

Set A. A set with an infinite number of elements.
Set B. A set where each element is an integer.
Set C. A set where the second and subsequent elements can be found as follows: $\{x, f(x), f(f(x)), \ldots\}$
Setting $x = 1$ and $f(x) = x + 1$ in this parent set gives the child set.

There is usually no one single correct way of decomposing a problem into components. It is a high-level design activity and there are often several possible good designs. The subject of Discrete Mathematics, however, is all about mathematical notation and tools and seems to have little or nothing to do with design. We can, though, show how a mathematical entity can be represented in more than

Downloaded from teamat.oxfordjournals.org at VIVA Community Colleges on March 1, 2011

one way. Let us consider two examples. The first one follows on from our discussion of the set of natural numbers, consider a set P that belongs to $A \cap B \cap C$, where A, B and C are as described above. An alternative way of representing this new set is as follows:

P is a set of numbers
$f : P \to P$

This mapping is one-to-one but not onto

$$P' = \{y | y = f(x), x \in P\}$$
$$f : P \to P'$$

This mapping is one-to-one and onto

If $S \subseteq P$, $S \not\subset P'$, and $S' \subset S$

where $S' = \{y | y = f(x), x \in S\}$
then $S = P$

This alternative representation is referred to as Peano's axioms. Students could be asked to show that the two representations are equivalent. This is rather difficult for them. An easier approach is to give them a written explanation and ask them to try to follow it. Let us consider a second example of where there is more than one representation. In mathematics, which we learn at school, we would write $f(a) = a^3$ (or we could use mapping notation and write $f : a \to a^3$). An alternative representation is given by lambda calculus where we write $(\lambda x.x^3)a$.

3. Programming

Computer Science students must obviously learn to program computers. They will need to learn how to write programs in an object-oriented (OO) language. An OO program is made up of 'classes'. These correspond to the components that we mentioned above. In designing a class, we first think about the data that should go in the class and then we think about the operations that need to be performed on that data, which are also included in the class. In OO terminology, these are referred to as 'attributes' and 'methods', respectively. We could say to students that the function notation we learn at school, such as, *f*(*x*, *y*), is, crudely speaking, similar to the breakdown of a class into attributes and method. *x* and *y* are attributes and *f*() is a method.

When we write a computer program we make use of a programming language. We must adhere to the rules of the language, i.e. its syntax. These rules are described by a 'grammar'. Let us look at a simple grammar:

```
digit = ''0'' | ''1'' | ''2'' | . . . | ''9'' |;
digits = digit | digit,digits;
```

The first line means that 'digit' can be defined in several ways:

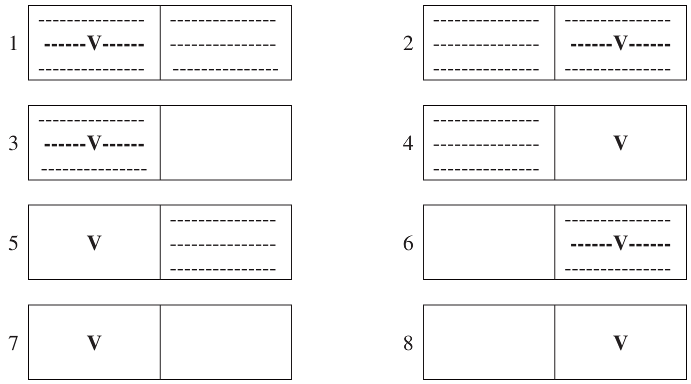
```
digit = ''0''
digit = ''1''
   . . .
digit = ''9''
```

Downloaded from teamat.oxfordjournals.org at VIVA Community Colleges on March 1, 2011

The second line enables us to build up a string of digits, for example, digits = ‘‘17’’. It would be an interesting exercise for students to understand simple grammars.

4. Finite-state machines and propositional logic

The Unified Modelling Language allows us to model the architecture of a computer program using several kinds of diagrams. One of these is a State Machine Diagram. This is a type of directed graph. The vertices are the different states of an ‘object’. The directed edges (transitions) show how we can get from one state to another. As an example, let us consider an application that is used to describe certain AI techniques—‘vacuum world’ (Russell & Norvig, 1995). Imagine two equally sized squares, one on the left and one next to it on the right. The intelligent vacuum cleaner moves from square to square cleaning this simple world. One state of the small vacuum world is where the vacuum cleaner is in the left-hand square and there is dirt in both squares. Students could be asked to describe the other seven states.



Students could then start to draw a directed graph where the vertices are the eight states. We could then ask the students to draw the directed edges to complete the diagram. It should be noted that we would have to make some assumptions about how intelligent the vacuum cleaner is and also whether or not dirt is continually falling onto the squares. For example, let us assume that the cleaner knows its location and can detect dirt. Let us also assume that at random time periods dirt falls on each square. With these assumptions, students have sufficient information to complete the diagram by showing all possible transitions from one state to another. To illustrate, let us start at state 1. Table 1 describes the three transitions leaving this state.

Let us now turn our attention to propositional logic. Rather than considering the simple two-grid example described above, students could explore the logic of ‘vacuum world’ for a three by three grid.

TABLE 1. *Transitions leaving state 1 of ‘vacuum world’*

Transition	New state
Right	2
Suck; dirt falls on left square straight after it is cleaned	1
Suck; left square remains clean	5

Downloaded from teamat.oxfordjournals.org at VIVA Community Colleges on March 1, 2011

The goal is obvious—the robot is to clear up all the dirt. To consider the problem further, students need to decide how sophisticated the robot is. They need to consider three things—how sophisticated is the mechanism for moving from one square to the next, what the robot can sense about its environment and what it knows about its current location. Let us describe an example robot. First, we describe the robot movement. The robot can only travel forwards but at the time at which a decision is made about the next action to take, the robot is facing north, south, east or west. There is a mechanism that can turn the robot on its square. The mechanism is such that, in one action, it can only turn the robot to its right. Second, we describe what the robot can sense. It can only detect whether or not there is dirt in the square, in which it is currently located. Third, we describe what the robot knows about its location. It knows which square it is initially placed in and can calculate its current location following a move to another square.

The logic of this robot can be explained with the aid of predicates of the form:

Facing(d)	where *d* is the direction
Dirt	
In(row,column)	where the north–west corner is (1,1), the south–east corner is (3,3), etc.

The available actions are *forward*, *turn* and *suck*.
Students can now try to list the rules that the robot needs to execute, in priority order, in order to continuously move around the grid, cleaning it. The first group of rules will be of the form:

$$In(row,column) \wedge Dirt \rightarrow Do(suck)$$

Examples of other rules are as follows:

$In(3,1) \wedge Facing(north) \rightarrow Do(forward)$
$In(2,1) \wedge Facing(north) \rightarrow Do(forward)$
$In(1,1) \wedge Facing(north) \rightarrow Do(turn)$
$In(1,1) \wedge Facing(east) \rightarrow Do(forward)$

5. More propositional logic examples

Propositional logic is used extensively in the topics of AI and Software Agents. One of the properties of a software agent is that it is autonomous. Let us consider an example of where this autonomy might be useful. During space flights there has often been a ground crew tracking progress and deciding what to do when the unexpected happens. This is very costly and the ground crew sometimes cannot react quickly enough. An alternative is to use a software agent on the space probe that makes decisions for itself. Such a software agent was used in NASA's Deep Space 1 (NASA's report on Deep Space 1, 2001). Another example of where propositional logic is used is in a fictitious video game called Tileworld (Pollack, 1990). Imagine that the game simulates a two-dimensional grid environment on which there are software agents, tiles, obstacles and holes. An agent can move in four directions, up, down, left or right, and if it is located next to a tile, it can push it. Holes have to be filled up with tiles by the agent. An agent scores points by filling holes with tiles, with the aim being to fill as many holes as possible. In Tileworld, holes randomly appear and disappear, which makes the game much more challenging. When a hole disappears or a new one appears the agent should detect this and change its strategy on which tile to push into which hole. It is possible to program the agent in many different ways. An interesting task for students is to consider how we decide which agent is best. We can do this by calculating the utility of an agent. Students could be asked to suggest a formula that would give the

Downloaded from teamat.oxfordjournals.org at VIVA Community Colleges on March 1, 2011

utility value. An example formula would be the ratio of holes filled to holes that existed, over a fixed time period.

A common way of describing some aspects of AI has been using the ‘Blocks World’. There is a robot arm, several blocks of equal size and a table-top. This example application, as with others, calls for a formal representation of the problem domain in order that we can reason about that domain. In other words we need an ontology. An ontology to describe a state of Blocks World is shown in Table 2.

Students could be given a sketch of the location of blocks and then asked to write down a description of that state using this function notation. For example, imagine a state where blocks B and C are on the table, block A is on top of block B, and the robot arm is empty. One way of representing this state is to list those functions that are true, as follows:

Clear(A)
On(A, B)
OnTable(B)
OnTable(C)
Clear(C)
ArmEmpty

An alternative representation of the state is as follows:
$Clear(A) \wedge On(A, B) \wedge OnTable(B) \wedge OnTable(C) \wedge Clear(C) \wedge ArmEmpty$

The next thing we have to do is represent an action. The technique is based on earlier work (Fikes & Nilsson, 1971). We give each action a name, have a ‘pre-condition list’ of functions that must be true before the action can take place, a ‘delete list’ of functions that will be deleted from the ‘pre-condition list’ after the action takes place,and an ‘add list’ of functions that will be added to the ‘pre-condition list’ after the action takes place. As an example, let us look at the stack action that is used to get the robot arm to place the block it is holding, x, on top of block y. Formally we describe this as follows:

Stack(x, y)
pre $Clear(y) \wedge Holding(x)$
del $Clear(y) \wedge Holding(x)$
add $ArmEmpty \wedge On(x, y)$

Students could be asked to formally describe the unstack action, which is used to get the robot arm to pick up block x from on top of block y. The answer is as follows:

UnStack(x, y)
pre $On(x, y) \wedge Clear(x) \wedge ArmEmpty$
del $On(x, y) \wedge ArmEmpty$
add $Holding(x) \wedge Clear(y)$

TABLE 2. *An ontology for Blocks World*

Function	Description
On(x, y)	block x on top of block y
OnTable(x)	block x is on the table
Clear(x)	nothing is on top of block x
Holding(x)	arm is holding x
ArmEmpty	robot arm empty

Downloaded from teamat.oxfordjournals.org at VIVA Community Colleges on March 1, 2011

Students may need to be told to check that their stack and unstack actions are inverses of one-another.

Another Software Agent application is a simulated robot submarine called HOMER (Vere & Bickmore, 1990). The submarine is given instructions by a user and a dialogue develops between them. It is interesting for students to look at an example dialogue and to see how intelligent the submarine is. Another example of robots is where vehicles are roaming an area of Mars in search of precious rocks (Steels, 1990). The rocks are found in clusters. There is a mother ship that emits a signal. The signal strength diminishes the further away from the mother ship it is. The robotic vehicles must find the rocks and take them back to the mother ship. It is possible to program each robot so that it works alone. An interesting exercise is to give students the handful of rules needed to program the robot, in random order, and get students to sort them in priority order. The highest priority rule is 'avoid obstacle'. The lowest priority rule is 'roam randomly'. Having done this, we can describe to students a new scenario where the robots are communicating with one another. Students could be asked to write new rules for these robots.

6. Other logics

Concurrent METATEM is an experimental language specifically designed to aid in the programming of software agents (Fisher, 1994). Each agent is programmed with a set of rules. Each of these is expressed in terms of time. The language involves temporal logic, which makes use of certain operators. Examples of some of the operators are now shown.

$\square$important(mathematics)

means 'it is now, and will always be true that mathematics is important'

$\diamond$important(nanotechnology)

means 'sometime in the future, nanotechnology will be important'

($\neg$friends(us)) U apologize(you)

means 'we are not friends until you apologize'

$\bigcirc$apologize(you)

means 'tomorrow you apologize'.

Yesterday, today and tomorrow do not refer to the normal 24-hour clock, they refer to computer clock times. With Concurrent METATEM, we are concerned with different programs running on different computers and communicating via a computer network. It is interesting for students to look at examples of these programs. Each program comprises a header and about one or two temporal logic statements. As an example, let us consider a producer/consumer scenario where there is one producer of resources and two consumers. The program executed by the resource producer (rp) is as follows:

```
rp(ask1,ask2)[give1,give2]:
        ● ask1 → ◇give1;
        ● ask2 → ◇give2;
        start → □¬(give1 ∧ give2).
```

The ● symbol denotes 'yesterday'. The first rule means: if 'yesterday' consumer 1 asked for a resource then, sometime in the future, give it the resource.

At the start of time ● *ask1* and ● *ask2* will both be FALSE because there was no 'yesterday'.

Downloaded from teamat.oxfordjournals.org at VIVA Community Colleges on March 1, 2011

start is an operator that is TRUE at the start of time and FALSE at all other times.

The last rule ensures that the producer gives out resources one at a time.

A resource consumer (rc) could be programmed in several possible ways. Following is one way:

rc1(give1)[ask1]:
 start $\rightarrow$ ask1;
 $\bullet$ ask1 $\rightarrow$ ask1.

Students could draw a table with row 1 representing time 0, row 2 representing time 1, and so on. *rp*, *rc1* and *rc2* could each be allocated two columns, one showing the input at each time and the other, the output.

To illustrate the language in more detail, we could use the scenario of Snow White and the seven dwarves. The seven dwarves each request sweets from Snow White. In computing terms, this is similar to seven consumer computers requesting data from one provider computer. Snow White can only give a sweet to one dwarf at a time. If a dwarf asks for a sweet then eventually the dwarf will get one. Snow White will form a list of requests that she must satisfy. If a dwarf asks again and his request is already on her 'to do' list then the request is ignored. Turning our attention to the dwarves, we can give them names to indicate the frequency with which they are requesting sweets. For example, dwarf 'eager' asks for a sweet initially, waits until he has received one, and then asks again. Dwarf 'greedy' asks all the time. Dwarf 'courteous' only asks when 'eager' and 'greedy' have been given one since he last asked. Dwarf 'shy' will only ask for a sweet when no-one else is on Snow White's 'to do' list. It is interesting for students to look at examples of programs for Snow White and these dwarves. Having understood the programs, students can then be given textual descriptions of other dwarves and asked to write programs for them.

7. Conclusion

A number of ideas for making Discrete Mathematics courses more interesting have been presented. All of these are directly related to computing applications. Some Discrete Mathematics lecturers are mathematicians and they should find this work particularly useful. Using these ideas, students are able appreciate the relevance of the mathematics to their overall degree studies, and its importance. The ideas can be incorporated into courses without any additional resources, time or otherwise. Many of the ideas are based on AI and Software Agent studies. Students are able to use mathematics to formally describe the problems as well as to solve them. Software Agent applications have matured and extensive research involving their usage continues. Other ideas presented include those related to software architecture, programming and finite-state machines.

References

Alagar, V. S. & Periyasamy, K. (1998) *Specification of Software Systems (Graduate Texts in Computer Science)*. Berlin: Springer.

Fikes, R. E. & Nilsson, N. (1971) STRIPS: a new approach to the application of theorem proving to problem solving. *Artif. Intell.*, **2**, 189–208.

Fisher, M. (1994) A survey of Concurrent MetateM – the language and its applications. *Temporal Logic – Proceedings of the 1st International Conference* (D. M. Gabbay and H. J. Ohlbach eds), LNAI Vol. 827, Berlin: Springer, pp. 480–505.

NASA'S Report on Deep Space 1. (2001). (see http://nmp.jpl.nasa.gov/ds1/) [accessed 19 September 2009].

Downloaded from teamat.oxfordjournals.org at VIVA Community Colleges on March 1, 2011

Pollack, M. E. (1990) Plans as complex mental attitudes. *Intentions in Communication* (P. R. Cohen, J. Morgan, and M.E. Pollack eds), Cambridge, MA: MIT Press, pp. 77–104.

Pressman, R. S. (2009) *Software Engineering: A Practitioner's Approach*, 7th edn. New York: McGraw-Hill.

Russell, S. & Norvig, P. (1995) *Artificial Intelligence: A Modern Approach*, 2nd edn. Englewood Cliffs, NJ: Prentice-Hall.

Steels, L. (1990) Cooperation between distributed agents through self organization. *Decentralized AI – Proceedings of the 1st European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89)* (Y. Demazeau and J.-P. Müller eds), Amsterdam: Elsevier, pp. 175–196.

Vere, S. & Bickmore, T. (1990) A basic agent. *Comput. Intell.*, **6**, 41–60.

Wooldridge, M. (2009) *An Introduction to MultiAgent Systems*. Chichester, UK: John Wiley & Sons.

Alan Oxley is a professor in the Computer and Information Sciences Department at Universiti Teknologi PETRONAS, Malaysia. He has authored papers in both Mathematics and Computer Science.

Downloaded from teamat.oxfordjournals.org at VIVA Community Colleges on March 1, 2011