Presenting computer science concepts to high school students

Tim Bell
University of Canterbury, New Zealand
tim.bell@canterbury.ac.nz

Caitlin Duncan
University of Canterbury, New Zealand
caitlin.duncan@pg.canterbury.ac.nz

Sam Jarman
University of Canterbury, New Zealand
saj77@uclive.ac.nz

Heidi Newton
Victoria University of Wellington, NZ
heidirosenewton@gmail.com

May 15, 2014

Abstract

Computer science at high school often focusses on programming, but a broader view of other areas of computer science has key benefits for both writing programs that are more efficient and making more theoretical concepts more accessible to those who do not find programming intrinsically interesting. With the introduction of computer science at high schools, a lack of coherent resources for teachers and students prompted the development of the NZ Computer Science Field Guide, an open-source, on-line textbook.

This paper describes the design of the Field Guide, which has fourteen chapters about various topics of computer science. The design includes written text, videos, classroom activities and interactive applications. The need for a broad view of computer science is discussed, and programming exercises to go with the topics are suggested.

Keywords

Computer science; high school; curriculum; constructivism; open source

1 Introduction

Computer science at high school level often focusses on programming, but new approaches, including new curricula in the UK (Furber, 2012), Australia (Falkner et al., 2014) and New Zealand (Bell et al., 2010), are being developed that offer a broader view of the subject, allowing students to delve into topics such as algorithm efficiency, encryption, human computer interaction and computer vision. This broader view has two key benefits: first, it shows programmers how to write programs that are more effective, and second, for those who don't find programming intrinsically interesting, it shows the kinds of things that are done with programming, providing the motivation to learn programming.

For example, in programming competitions students are tasked with problems to solve that must run within a time limit. Writing a program to solve a problem may not be so difficult, but to do it efficiently and effectively can involve bringing to bear ideas from computer science (such as algorithmic complexity and tractability), and a good understanding of computer science principles can enable students to improve their competition code, allowing it to execute faster and fit under time limits set by the judges. Conversely, ideas from computer science (such as data compression or formal languages) can provide rich domains for programming exercises, and give students more experience thinking about the richness of approaches available to the computer scientist, such as hashing (an idea which can be applied to areas as diverse as searching algorithms, error detection by hash totals, and password encryption by secure hashing).

Although there are literally thousands of resources available that touch on areas of computer science that might be relevant to high school level students (Murugesh et al., 2010), these resources vary greatly in suitability, and tend to occur as one-off examples that can't be used as a coherent body. To address this, we have developed the "NZ Computer Science Field Guide" (referred to here as the NZ CSFG, available at `http://csfieldguide.org.nz`), an open-source, interactive, online "textbook" that introduces a wide range of topics in computer science, without necessarily expecting students to be competent programmers before tackling the range of topics covered. It is a pilot for a wider range of computer science field guides intended for international use in a variety of contexts. The index of the NZ CSFG is shown in Figure 1, showing the range of topics covered.

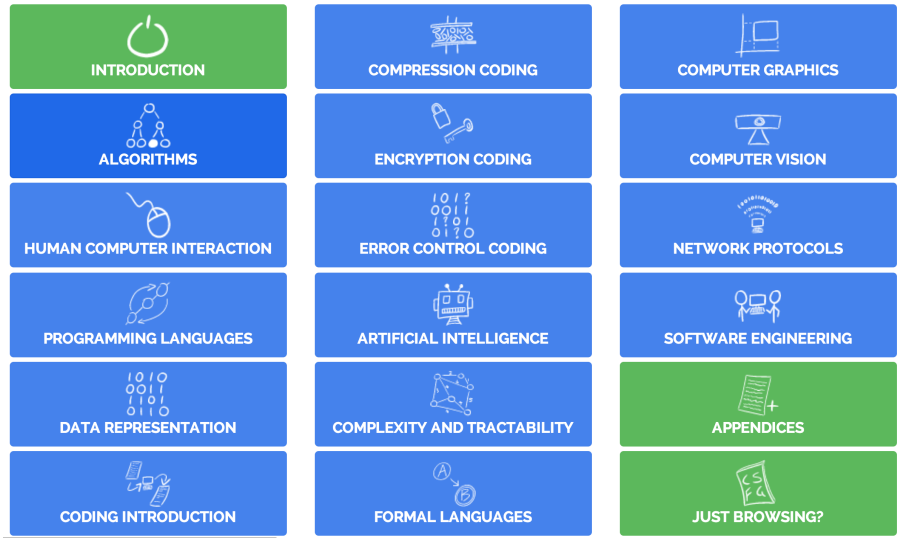

Figure 1: The front page of the CS Field Guide

The NZ CSFG has initially been developed to support the new computer science standards that became available in New Zealand high schools in 2011 (Bell et al., 2010), but it is intended to be flexible enough to support curricula for other countries, and other initiatives aimed at high school aged students, such as computing clubs and programming competitions. Because it is open source, in principle educators can adapt it to suit their situation. Because it is online it can be accessed by interested students as long as they have internet access, and an offline version is planned so that it can be delivered through other media as well. For example, while the NZ CSFG was initially prepared to support teaching computer science in New Zealand high schools, it was also being used in parallel to support a pilot for a Computer Science Club (`http://computersciencеclub.org`) for students aged around 10 to 15 years old. The club is based on a badge system where students can attain different levels of badges in topics in computer science, with many of the badge topics being associated with topics in the CSFG.

There are other collections of information available that have goals in common with the CSFG, aimed at conveying the ideas of computer science to a high school audience or through interaction and animation. Some of these have provided inspiration and ideas for the CSFG, and others are useful as follow up for students wanting more detailed information. The "Thriving in our digital world" (`http://www.cs.utexas.edu/~engage/`) course uses a similar approach to the CSFG for its teaching material and has engaging interactive presentations, but focuses on just eight topics (four of which are general, such as "Innovations"). The Virginia tech online interactive modules (`http://courses.cs.vt.edu/csonline/`) for teaching computer science cover a range of relevant topics (Balci et al., 2001), although the material again only covers a limited number of topics, and doesn't appear to have been updated since it was developed over 10 years ago. "Babbage's bag" (`http://www.i-programmer.info/babbages-bag.html`) provides a very detailed collection of technical articles on many topics in computing. It is more detailed than most high school students would need, but is valuable as a follow up on particular topics. "CS animated" (`http://www.csanimated.com/`) has interactive activities on computer science, but is more targeted at

university level students. The "Computer Science For Fun" (cs4fn.org) project provides a very readable collection of short articles aimed at a teenage audience. It is about practical applications of topics in computer science, and has been very successful in getting students interested in computer science (Myketiak et al., 2012), although it doesn't usually go into the level of detail needed to learn the topic, as it is primarily aimed at outreach. The "CS Bits & Bytes" (http://www.nsf.gov/cise/csbytes/) project takes a similar approach, with regular up-to-date articles about applications of computer science.

In this paper we discuss in more detail the value of a broader view of computer science for high schools students, and then describe the design of the Computer Science Field Guide, which is intended fill a gap for teaching computer science, and act as a tool to provide teachers and trainers with a rich resource for engaging students with this broad view of the subject. A case study is made using the chapter on algorithms to explain how design decisions were made, and we provide examples of how programming competition exercises could be formulated based on ideas in the CSFG.

2 The need for a broad view of computer science

It is not unusual for computer science in high school and programming competition environments to be regarded as being primarily about programming, and many on-line resources focus on "coding" (programming). This misses out on a much richer view of the field that explores how well the program might work, such as its efficiency, security, usability, scalability and reliability. In programming competitions, the areas of computational complexity and tractability are particularly important.

There are many definitions of what computer science is, and the approach we have taken covers a number of widely accepted definitions. A key benchmark is the ACM Computing Curricula document (Impagliazzo, 2006), which describes computer science as follows: "Computer science spans a wide range, from its theoretical and algorithmic foundations to cutting-edge developments in robotics, computer vision, intelligent systems, bioinformatics, and other exciting areas". This overview has been followed by the 2008 and 2013 Computer Science Curricula (Sahami et al., 2013) which define, respectively, 14 and 18 areas of computer science that should be covered at university level, many of which correspond to chapters in the CSFG. A crowd-sourced definition of computer science can be found on Wikipedia, which (at the time of writing, in April 2014) describes it as "the scientific and practical approach to computation and its applications", and more practically, goes on to list 16 sub-topics, 8 of which correspond to chapters in the CSFG, and most of the rest are touched on at some point.

Of course, computer science can't really be broken into some finite number of disconnected topics, and it is important to emphasise links between topics (e.g. fast algorithms mean that interfaces can respond within the times recommended through HCI principles; search algorithms are required for pattern matching in compression systems; and compression in turn improves network response times which leads to better interfaces).

Many countries are now moving to increase the amount of computer science taught at high school level. This is partly driven by the dramatic shortage of computer science graduates in western countries; teaching computer science in schools can enable students to make better career choices, and a broader view of computer science beyond just programming can attract those who are interested in the bigger picture, rather than programming as an end in itself. The analogy that "Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes" (Fellows and Parberry, 1993) illustrates the value of providing a way for beginner programmers to access the big ideas in computing. Furthermore, it is very easy to write a program that is computationally inefficient (or even intractable), and beyond simple functionality, issues such as the usability and security of programs is also important. Programming competitions can accentuate the focus on barely meeting some requirements, and a wider view of computer science can encourage a healthy view of problem solving techniques, algorithms, mathematical underpinnings, and human factors.

3 Design of the Field Guide

The NZ CSFG currently has a chapter for each of 14 areas of computer science that have initially been designed to match the new New Zealand high school standards released in 2011 (Bell et al., 2010). These areas correspond loosely to the 2008 ACM curriculum (which was the one available at the time that the school standards and NZ CSFG were designed). The ACM curriculum topics are considerably deeper than what is appropriate at high school level, so providing a broad overview of the topics is a challenge, particularly doing it in a way that students have a meaningful experience of the topic.

Key features that have driven the design of the CSFG are as follows:

- Open source: teachers all over the world can access it freely, and improve it if they wish. It is intended to be a prototype for a broader range of CS Field Guides for other countries and contexts. The guide is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike licence, which means that users are welcome to take copies and modify them. The material is produced using the open-source Sphinx system (`http://sphinx-doc.org/`), which was originally designed for writing Python documentation, and works from plain text source files using the reStructuredText format. Much of the writing has been done by volunteers, and the more costly parts of the production have been supported by contributions from industry.

- Interactive: learning activities, games, videos and animations are embedded in the page as students read the book. The interactive components are intended to encourage direct engagement on the part of students, rather than something that is viewed passively.

- Focus on key concepts: Rather than teach a topic in depth, we establish what the key concepts are, and make sure they are conveyed. For example, binary representation has some obvious conversion skills that can be learned, but the key concepts are things such as the exponential increase in descriptive power with each bit added; and "algorithms" are often published as a shopping list of many different algorithms, whereas the key concepts are more around how different algorithms can have a non-linear difference in performance, and that some problems are intractable.

- Self-Paced: there is sufficient material that students can learn independently at their own pace, but also work in an environment facilitated by a teacher

- Teacher support: there is a semi-private version of the guide that has a lot more information for teachers, including solutions to all questions and hints for use in a classroom situation. The Sphinx system used enables conditional use of small units of text, so this makes multiple versions possible from a single source; the teacher guide and student version come from the same source text, and future versions for other curricula can also be created automatically by conditionally selecting appropriate sections of text.

- Engaging: drawing on our experience with Computer Science Unplugged, it should keep students engaged. This includes the use of humour (such as fictitious scenarios and tongue-in-cheek comments), "curiosities" (which give tangential examples or stories to create interest), and the use of cartoons.

- Catering to different learning styles: the use of different ways to convey the same information provides each student with multiple experiences of the topic, and some may resonate better with one student's learning style than another.

- Short video "bumpers" that provide enticing introductions to the topic: these videos, which are generally a minute or two long, provide a somewhat humorous but sound overview of the topic, raising questions and problems that are addressed in order to provoke curiosity.

- Platform independent: it should be possible to read the guide online and offline, on all operating systems (primarily in a browser), and on tablets and even smart phones. This is achieved by using the Sphinx system, which can output the material on a web site, as PDF, or as an E-book (EPUB and MOBI). The interactive activities are programmed using HTML5 and JavaScript, which will run on most web browsers and E-book readers.

- No programming required: students do not need to be competent programmers before engaging with the material. Many will be learning in parallel, but alternative ways of engaging with the concepts are provided that don't require programming ability.

Figure 2 shows the beginning of the graphics chapter as an example. The opening video involves the presenter interacting with 3D graphics (including a slapstick attack of the ubiquitous graphics teapot), and the text begins with a "Big picture" section that conveys some of the key motivation for the detailed information that follows. The following sections cover selected specific topics in graphics (in this case, transforms, and line/circle drawing algorithms) to illustrate the kind of issues that are dealt with in this topic. Each chapter concludes with a "Whole story" section, which mentions other key topics in the area of the chapter that haven't been covered but are likely to be encountered in further reading or study. Currently chapters typically only cover two or three topics, which are sufficient to illustrate the area, but could be greatly expanded in the future to present other key topics that students might want to look into.

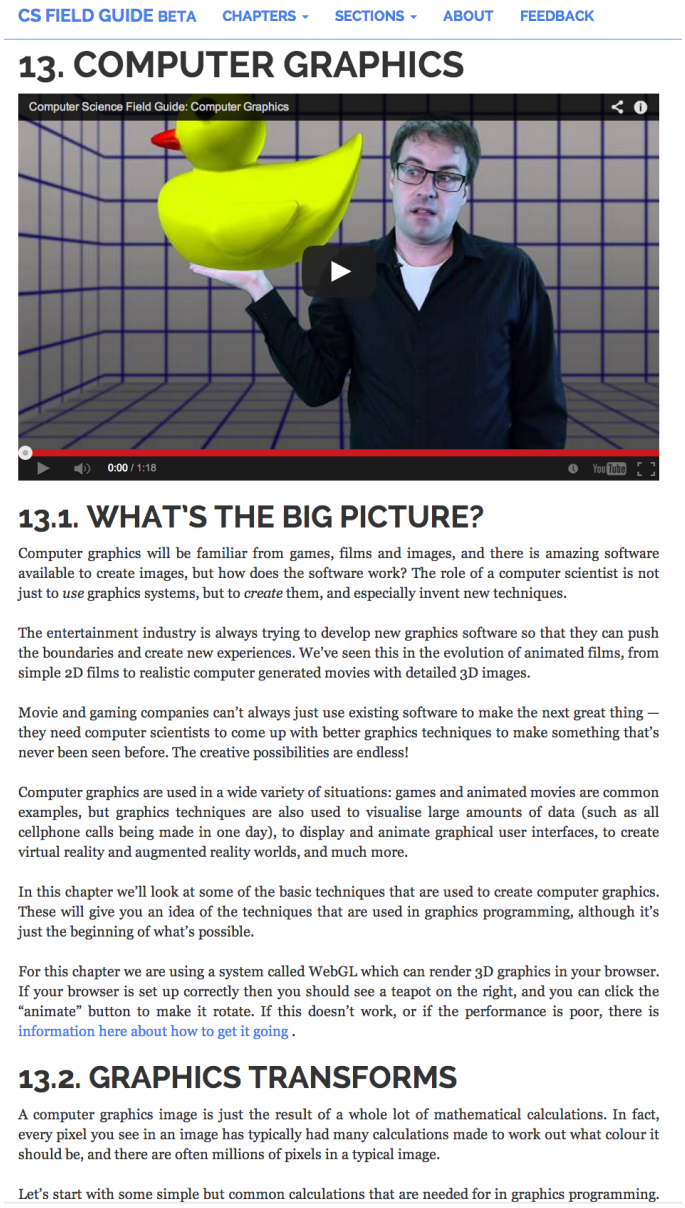

Figure 2: The beginning of the graphics chapter

Most chapters contain activities and projects that could be used for assessment purposes; "activities" tend to be smaller formative tasks, whereas projects provide a more in-depth task that is typically used for summative assessment.

The main topics covered were shown in Figure 1. As discussed above, this list largely reflects the widely-used ACM computer science curriculum for universities; of course, the topics need to be presented in a way that they are approachable for students with only rudimentary programming skills and a high-school math background, and so that each one can be covered meaningfully in just a few weeks of a class.

An important task has been to identify the key concepts that would give students an understanding of what issues the topic needs to address, rather than an exhaustive coverage of many sub-topics in the area. The main concepts identified were as follows:

- Algorithms: understanding that algorithms exist independently of any programming language, and that different algorithms for the same task not only have different running times, but that the difference may not be linear.
- Programming languages: exploring the role of compilers and interpreters in enabling a human-readable language to be run on a computer, and the idea that a computer language is implemented by a program itself.
- Human-computer interaction: critically assessing existing interfaces using well established principles, including basic psychology, and the idea that the person who implemented the interface is not in a good position to evaluate it critically.
- Data representation: representing numbers, text, images and sound using bits, particularly the relationship between the number of bits used and the quality of the representation, and the exponential increase of range with the number of bits; hexadecimal is a shorthand for binary.
- Coding: changing the representation of data to make it smaller (compression), secure (encryption) and reliable (error control). Compression concepts include lossy vs lossless compression, and the kinds of structures in data that can be exploited to reduce file sizes. Encryption covers the concept of an attack, including approaches such as brute-force and known-plaintext attacks, the cryptographic strength of keys, the key exchange problem, and areas of cryptography beyond keeping data confidential. Error control includes the idea that error detection and correction is possible, and the ability to do this with a high probability of success can be achieved by adding relatively few bits to data.
- Formal languages: efficient ways to specify and implement programming, markup, and other languages, how formal specifications are helpful in designing and communicating languages, and how to parse and process programs or documents written in such languages.
- Network communication protocols: the techniques and algorithms applied in computer networks to ensure reliable, effective and efficient communication of data between two parts of a network in the face of different kinds of threats and failures.
- Complexity and tractability: the relationship between problems and their algorithms, and the idea that many common problems don't have tractable solutions, that brute force algorithms can result in a combinatorial explosion of the running time, and that heuristic algorithms are often the best we can do in practice.
- Artificial intelligence (AI): intelligent systems and the possibility of designing systems that exhibit aspects of human intelligence, reflecting on what intelligence is, and the practical and theoretical issues surrounding this. A significant component of Artificial Intelligence is (sadly) its limitations, and understanding these can rectify popular views of AI that might be picked up from media.
- Software engineering: learning that there are systematic approaches that are applied to large software projects, typically with many team members and large amounts of program code, so that the products behave reliably and efficiently, are affordable to develop and maintain, and satisfy customer requirements.

- Computer graphics: using computers to create images and animations based on a description of a scene or collected data, including techniques such as rendering, occlusion, and transformations.
- Computer vision: processing images and recognising elements in an image, including dealing with noise, edge detection, and face detection.

We now give a more detailed description of the design of the chapter on algorithms to illustrate how the above principles are worked out in practice.

4 Case study: Algorithms chapter

The design of the chapter on algorithms is reviewed in this section to give an idea of the approach taken and what topics have been included in the CSFG — and just as importantly, which topics have been *excluded*.

The key concepts that we chose to convey through the chapter are:

- What an algorithm is and how it differs from the related concepts of programs and informal instructions.
- The concept that an algorithm has an associated cost, that this cost may be non-linear and is related to both running-time (of a program implementing the algorithm) and computational complexity.
- That two algorithms may have different costs even if they solve the same problem and that this difference in costs can be non-linear.

Valuable background for the design of this chapter was provided by an in-depth analysis of reports that included work on algorithms submitted by students for assessment (Bell et al., 2012). This analysis identified several key areas in the algorithms topic that had a great impact on the grades achieved by students. The majority of students chose either sorting or searching algorithms to investigate (63% sorting and 19% searching) and usually these students earned a passing grade or better for the algorithms section of the report if they explained their work satisfactorily. It was found that students who chose other algorithms, or used their own programs, were much less likely to pass the algorithms section of the report. Students needed to compare the "costs" of algorithms (i.e. algorithmic complexity) to do well, and the majority of students unknowingly limited their ability to discuss the cost difference between algorithms as they choose to only compare the costs of their algorithms for relatively small input sizes, for example $n = 10, 20, 30$. Because the non-linear difference in costs for some of these algorithms only emerges when larger numbers are used, such as $n = 100$ or $n = 1000$, some students were unable to observe this relationship. About 10% of students were also unable to observe this trend as they chose to compare algorithms with the same complexity, such as Selection and Insertion sort, and so only observed a constant difference in their costs. This observation drove the selection of algorithms in the chapter; it is more important to have a small number of algorithms with different asymptotic complexities than many algorithms that had the same complexity. There were also several cases where students interpreted the "cost" of an algorithm as the length, in lines of code, of a program implementing the algorithm. This suggested students required some guidance in how to measure the cost of an algorithm.

From the study, Bubble sort and Quicksort were the most popular sorting algorithms used by students. While these have drastically different running times, which gives students the opportunity to talk about the non-linear difference in their costs, it has been argued that Bubble sort has little pedagogical value and can be confusing for students (Astrachan, 2003). Selection and Insertion sort are both suitable alternatives to Bubble sort as they provide just as strong a contrast with Quicksort and are more worthwhile for students to learn. Linear search and Binary search were the most popular searching algorithms used in student work, and these provided a suitable contrast for students to discuss. The choice of algorithms made a very clear difference to the quality of student reports. The algorithms which most often led to high grades were pairs of algorithms with significantly different complexities. The most successful pairs were Binary search vs Linear search (which provided a comparison of $O(\log n)$ vs $O(n)$) and Quicksort vs one of Bubble sort, Selection sort and Insertion sort (a comparison of $O(n \log n)$ vs $O(n^2)$).

Students are not required, or encouraged, to implement the algorithms themselves and use their own programs for measuring costs because this risks a bug in their program giving them the wrong impression of algorithmic performance. Therefore implementations of each example algorithm used are provided for students to download, although students can follow up by implementing their own versions. The concept of a "cost" as the number of comparisons an algorithm makes is emphasised through the interactives, and the downloadable programs measure it as both comparisons and time taken. Students are encouraged to test the algorithms with large inputs so they are able to observe the non-linear differences between algorithm costs.

Students learn better when they are given the opportunity to construct knowledge themselves through experience, rather than simply learning from definitions or complete instructions (Wadsworth, 1996). The chapter has therefore been designed so that students are given the opportunity to discover algorithms for themselves, and explore the differences in their costs, rather than simply explicitly telling students how each algorithm works and the differences between them.

Another key tool in supporting students construction of mental models of these concepts is the use of analogies and metaphors for the use of algorithms (Forišek and Steinová, 2012). These are used throughout the chapter but especially during the introduction section (for example, searching the library) to ensure students have begun building a mental model about algorithms before they encounter the interactives.

It has been shown that learning about two algorithms in parallel and comparing them, rather than learning them separately, contributes to students gaining a greater understanding of both the algorithms and the differences between them (Patitsas et al., 2013). Through each of the Searching and Sorting sections algorithms are presented and discussed in relation to each other, rather than viewing each as a separate entity.

There are several algorithm visualisation tools and interactive tutorials that were considered for use in the chapter. Visualisations of algorithms have been popular for teaching different algorithms but it has been noted that many are too complex for students to understand (Murugesh et al., 2010). Furthermore, many don't require interaction from the student, and so encourage passive use of the resource. It was found when examining the visualisations and interactives available that some covered more algorithms than were necessary, and used advanced language that made them unsuitable for use by school aged students. Several contained valuable information and taught the algorithms well but unfortunately were aesthetically unappealing or repetitive, which didn't engage students. Thus, we make limited use of visualisations, and have focused on making them appealing and at a level that is meaningful to high school students.

Following the pattern for the CSFG, the chapter begins with a video and a "What's the big picture?" introductory section, each of which gives an overview of the topic of algorithms and the key concepts the chapter is going to present.

The introductory videos are not intended to teach any of the chapter content, but by giving a 'big picture' view of the main topic they give context to the lessons in the chapter. The first step in the video development process was to decide which concepts were to be conveyed. Several different combinations of concepts were reviewed, including the best and worst cases for an algorithm and the differences between an algorithm and a program, but the final key concepts chosen were the following:

- That an algorithm is a set of instructions for completing a task or solving a problem, we use them in our everyday lives, and algorithms are used to tell computers how to solve problems.
- There can be many different algorithms for solving a particular problem and some of these algorithms are better than others.
- Using a better algorithm can be better than using a faster computer.

For the video, several scenarios were considered, including algorithms for working with a collection of CDs, navigating supermarket aisles and mazes, connecting up networks, boarding planes and finding routes on a map. The concept chosen used two characters, one representing a fast computer and the other a very slow computer, and had them race each other to find a book in a library. The character representing the fast computer is much faster at looking through the books and running through the library, but uses a Linear search algorithm to try and find the book, searching the entire library book by book until they find the one they were searching for. The character representing the slow computer takes a much longer time to walk

through the library and examines each book for a long time before placing it back on the shelf and moving on. This character, however, uses a Binary search algorithm, and finds the book much faster. The full video can be viewed online at `http://www.youtube.com/watch?v=FOwCCvHEfY0`, and can be viewed or downloaded at `http://vimeo.com/69609500` (all the videos in the CSFG are provided on Vimeo as well as YouTube, to make it easier for teachers to download them and play them in a classroom, as some schools limit access to online video sites.)

After the video a short introduction section reiterates and emphasises the key lessons from the video and describes the sections of the chapter. To emphasise the points that there are a number different algorithms for the same problem and that some of these algorithms are better than others, a sorting algorithm visualisation has been designed (Figure 3). The aim of this visualisation is not to teach the algorithms, although it may be of use to refer to it again after students have learnt the sorting algorithms so they can see them in action. It is intended to be engaging, to keep students interested in the chapter content, and to show again the difference in the performance of algorithms. The visualisation we have designed shows only four algorithms to avoid overwhelming students. We have also placed a strong emphasis on the aesthetic of the visualisation as it is intended to be eye catching and engaging.

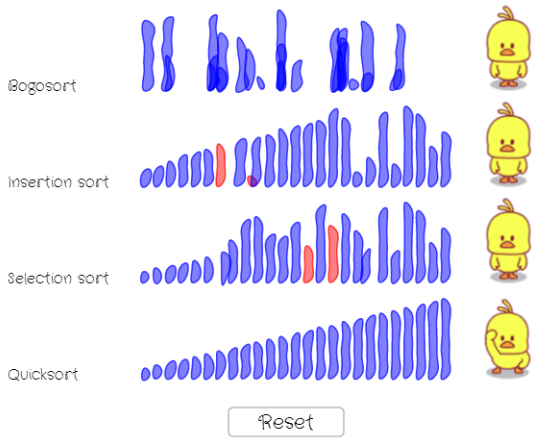

Figure 3: The sorting algorithms comparison animation

The algorithms mentioned include "bogosort", in which values are shuffled randomly until they end up in the correct order (which is very unlikely to ever happen for a large list). While this isn't a useful algorithm, it illustrates some algorithmic concepts starkly (such as relative running times, worst case time, best case time, and tractability). The other rows are sorted using Insertion sort, Selection sort and Quicksort. Once the bars are found to be in the correct order the bird beside that particular row begins a victory dance.

Teaching the details of algorithms begins by introducing Linear and Binary search. These algorithms were chosen because it is easy for students to gain a high level understanding of them and they are easy for students to perform themselves with physical objects. The non-linear difference in their complexities also makes them suitable choices for students to compare. The searching algorithms are taught using a constructivist approach through a game in which students have to search through a large number of presents in an attempt to find and collect the missing pets of two children, shown in Figure 4. The game is based on the CS Unplugged "Battleships" activity (`http://csunplugged.org/searching-algorithms`), but was changed to unwrapping presents, since blowing up battleships was likely to appeal more to male than female students.

Using the constructivist approach, students are simply told whether or not the numbers in the presents are in sorted order, and what number they are required to find. From our experience, students very quickly realise that an unsorted list is very slow to search (usually), and for a sorted list they quickly adopt a binary search related approach which enables them to find a number quickly without using too many of the small number of "lives" they are given.

There is a risk that students may try an interpolation search for the sorted list (e.g. guess that lower numbers are nearer the start); to confound this, we have adjusted the distribution of numbers to be non-

(a) (b)

Figure 4: Game for teaching search algorithms constructively. (a) An unsorted set of presents in which students must find the given number; a large number of lives is provided since a linear search will be required. (b) A sorted list; only a few lives are available, so students will need to use binary search to avoid looking at too many presents.

uniform, so this strategy will generally not behave significantly better than a conventional binary search, and will quickly discourage students from trying to guess number locations. The exact distribution of numbers was tested using simulations of likely student strategies on different number distributions, which identified patterns that would be pedagogically most valuable.

The sorting algorithms section of the chapter focuses on Selection sort, Insertion sort and Quicksort which, like the searching algorithms, were selected for their contrasting run times (for Selection or Insertion sort vs Quicksort) and the successful results achieved by student using them in past assessment (Bell et al., 2012). Selection and Insertion sort are very simple to explain and demonstrate with physical objects. Despite the complexity of implementing Quicksort it can also be simple to teach the basic method and demonstrate it with objects, which is all students require to understand it.

Although there are many animations of sorting algorithms available, these don't usually engage the viewer in the process. We have used a constructivist approach to explain the sorting algorithms using a balance scale that can compare only objects two at a time (simulating the data comparison step of conventional sorting algorithms). This is based on the CS Unplugged sorting activity (`http://csunplugged.org/sorting-algorithms`). Since a physical balance scale isn't always available, an online simulation was provided (shown in Figure 5). The simulated scale has the advantage that we can enforce having only one weight on each side of the scale. Students are guided through the sorting algorithms; for example, for selection sort, they are first asked to find the heaviest weight of the set, comparing just two at a time. Students soon find that this can be done in $n-1$ comparisons, and then $n-2$ for the second smallest, and so on. The other algorithms are also demonstrated using the approach from the CS Unplugged sorting activity. Quicksort is seeded with the idea of putting a randomly chosen weight on one side of the scale, and comparing each of the others with it. Students often come up with the idea of applying the algorithm recursively to the two groups of weights.

Programs implementing all the main algorithms discussed are provided for students to download in common programming languages used in schools, so that students can test their speed, confident that the implementation is correct (since the main learning outcome desired is to observe speed differences, rather than the ability to implement well-known algorithms).

The "whole story" section mentions the range of other problems and algorithms that exist, and also the "big oh" notation that students will quickly encounter if they look at other resources on algorithms. This notation isn't needed to understand the main concepts in the chapter, and so is avoided to make it accessible to students without the necessary math background, but it is important to mention it here since it is so common in this context.

Figure 5: The Sorting Interactive

5 Programming exercises based on the CS Field Guide

Creating exercises based on this kind of material has been explored previously (Voigt et al., 2009), where computer science concepts based on CS Unplugged activities were adapted for a programming competition environment. For example, the parity error correction technique (which appears in the CSFG also) can be used as the basis for a task to identify which bit(s) are identified by a parity error, with a step-up obtained by going from a single row of bits to multiple rows, and from single errors to multiple errors. For this kind of activity, in principle the most challenging version would be a full implementation of an error correction protocol, which is both authentic and motivating.

Some suggestions for exercises building on the material currently in the CSFG are:

- Algorithms: implement one of the searching or sorting algorithms; solve a sorting-based problem where there is a tight time constraint that requires the use of Quicksort rather than the $O(n^2)$ algorithms.
- Programming languages: implement a simple translator or assembler based on a small language (such as MIPS, or a subset, which is used in the chapter).
- Human-computer interaction: design a progress bar (or create information to support one) that gives an accurate estimation of completion time; or implement an experiment that measures user behaviour for response time or pointing time (Fitts's law).
- Data representation: convert numbers between binary, decimal and hexadecimal; convert binary codes to the 5-bit letter system used in the chapter; perform rounding of numbers required when (say) 24-bit colour is converted to 16-bit colour.
- Coding: encode or decode run-length encoding compression; implement the longest match search required for Ziv-Lempel compression; implement a simple substitution encryption system; write a brute-force system to attack an encrypted message; detect errors in data protected with parity bits; calculate checksums for product bar codes or ISBN numbers.
- Formal languages: write a program that implements an FSA from a transition table; use regular expressions in a program to check input; implement a simple lexer; generate random text based on a formal language (grammar or FSA).
- Network communication protocols: write a program to assemble packets that arrive out of order; implement a system that can acknowledge packets and request a re-send to assemble messages reliably.
- Complexity and tractability: implement an exhaustive evaluation of a small NP-complete problem (e.g. TSP, graph colouring, vertex cover, knapsack); implement a heuristic and award points for the answer which is closest to optimal.

- Artificial intelligence: implement a pattern-matching chatbot; perform elementary data mining by statistical analysis; implement a min-max search.
- Software engineering: implement a software metric or visualisation (lines of code, comments, digraph of flow control); write a test generator to create examples to test some software that has been supplied.
- Computer graphics: implement basic transforms; write software to combine multiple transforms into one matrix operation; implement Bresenham's line-drawing algorithm.
- Computer vision: Implement a simple filter that performs a weighted sum of surrounding pixels; perform simple face recognition given measurements of facial features in pre-processed images; perform simple edge detection by finding discontinuities in an image.

The above suggestions are simply to seed ideas, and students or instructors reading the chapters may well come across ideas that could be implemented. For each topic, the difficulty of assignments ranges from a simple simulation to a full implementation of the concepts in a way that could be used in a practical situation.

6 Conclusion

The Computer Science Field Guide has taken a new approach to making concepts from computer science accessible to high school students, encouraging a very broad view of the field rather than a depth-first approach that inevitably focuses on programming. The open nature of the guide is intended to make it easy for adaptation for new situations and curricula.

The guide includes a feedback link, which provides a tight feedback loop where any user can suggest clarifications or improvements. Over 100 suggestions have been received to date. About 23% are simple typos that can be fixed very quickly; 12% are "bouquets", acknowledging the usefulness of the resource; and the remainder are a mixture of clarifications and suggestions that may take longer to implement but are being prioritised for attention.

Currently all but one of the chapters covers the key concepts that we have aimed to convey (the final one to be written, on Network Communication Protocols, is in progress). Further work is needed to add more examples to each chapter; for example, tractability is currently explained using the Travelling Salesman Problem, but other topics such as bin-packing or graph colouring could be added as other ways to illustrate tractability. Future plans include adding more topics such as Computability, Big data, Parallel computing, and Databases. Other topics that are currently considered specialised may well become important as a basic part of computer science in the future (e.g. Quantum computing).

Other features that could be added to the guide include quizzes and student login and tracking. Also, multiple versions for different curricula are planned, and eventually a system to support translations would be useful.

The CSFG has been designed to be flexible to adapt to future needs of computer science education in high schools, and may end up being used in situations that can't even be imagined now. Fortunately the computer science community has a strong ethos of open systems, crowd sourcing and creativity that we hope will enable this project to adapt to future demands.

Acknowledgements

We acknowledge the hard work of the many people who have contributed to the NZ CSFG project. The field guide is part of the International CS Field Guide Project (`http://www.csfieldguide.org/`), and the idea and encouragement to develop the field guide (and associated badge system) came from Peter Denning and Rick Snodgrass. Much of the technical work has been done by Jack Morgan, and in addition to the authors of this paper, significant portions have been contributed by Janina Voigt, Rhem Munroe, David Thompson and Ian Witten. Funding for producing the guide has been provided by Google Inc.

Partial funding for the US field guide project was provided by the US National Science Foundation under Grant No. 0938809.

References

Astrachan, O. (2003). Bubble sort: An archaeological algorithmic analysis. *ACM SIGCSE Bulletin*, 35(1):1–5.

Balci, O., Gilley, W. S., Adams, R. J., Tunar, E., and Barnette, N. D. (2001). Animations to assist learning some key computer science topics. *ACM Journal on Educational Resources in Computing (JERIC)*, 1(2).

Bell, T., Andreae, P., and Lambert, L. (2010). Computer Science in New Zealand High Schools. In Clear, T. and Hamer, J., editors, *ACE '10: Proceedings of the 12th conference on Australasian Computing Education*, volume 32 of *Australian Computer Science Communications*, pages 15–22, Brisbane, Australia. Australian Computer Society, Inc.

Bell, T., Newton, H., Andreae, P., and Robins, A. (2012). The introduction of computer science to NZ high schools: an analysis of student work. In *Proceedings of the 7th Workshop in Primary and Secondary Computing Education*, WiPSCE '12, pages 5–15, New York, NY, USA. ACM.

Falkner, K., Vivian, R., and Falkner, N. (2014). The Australian Digital Technologies Curriculum: Challenge and Opportunity. *Proc. Sixteenth Australasian Computing Education Conference (ACE2014)*, pages 3–12.

Fellows, M. and Parberry, I. (1993). SIGACT trying to get children excited about CS. *Computing Research News*, page 7.

Forišek, M. and Steinová, M. (2012). Metaphors and analogies for teaching algorithms. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 15–20, New York, NY, USA. ACM.

Furber, S., editor (2012). *Shut down or restart? The way forward for computing in UK schools*. The Royal Society, London.

Impagliazzo, J. (2006). Computing curricula 2005. *ACM SIGCSE Bulletin*, (September).

Murugesh, S., Bell, T., and McGrath, A. (2010). A Review of Computer Science Resources to Support NCEA. In Mann, S. and Veerhaart, M., editors, *First annual conference of Computing and Information Technology Research and Education NZ (CITRENZ2010)*, pages 173–181.

Myketiak, C., Curzon, P., Black, J., McOwan, P. W., and Meagher, L. R. (2012). cs4fn: a flexible model for computer science outreach. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 297–302. ACM.

Patitsas, E., Craig, M., and Easterbrook, S. (2013). Comparing and contrasting different algorithms leads to increased student learning. In *Proceedings of the ninth annual international ACM conference on International computing education research*, ICER '13, pages 145–152, New York, NY, USA. ACM.

Sahami, M., Roach, S., Cuadros-Vargas, E., and LeBlanc, R. (2013). ACM/IEEE-CS computer science curriculum 2013: reviewing the Ironman report. In *Proceeding of the 44th ACM technical symposium on Computer science education*, SIGCSE '13, pages 13–14, New York, NY, USA. ACM.

Voigt, J., Bell, T., and Aspvall, B. (2009). Competition-style programming problems for Computer Science Unplugged activities. In Verdu, E., Lorenzo, R., Revilla, M., and Regueras, L., editors, *A new learning paradigm: competition supported by technology*, pages 207–234. CEDETEL, 47151, Boecillo, Spain.

Wadsworth, B. J. (1996). *Piaget's theory of cognitive and affective development: Foundations of constructivism* . Longman Publishing.

Authors

Tim Bell is a Professor in the Department of Computer Science and Software Engineering at the University of Canterbury. His main current research interest is computer science education, and he directs the Computer Science Unplugged project; in the past he has been also worked on computers and music, and data compression. He received the ETH (Zurich) ABZ International honorary medal for fundamental contributions in Computer Science education in 2013.

Caitlin Duncan holds a Bachelor of Science with Honours in Computer Science from the University of Canterbury, NZ, where she is currently a PhD student. Her research is focused on computer science and computational thinking education in primary and high school, and she has previously worked on the Algorithms chapter in the CSFG.

Sam Jarman holds a Bachelor of Science in Computer Science from, and is now a postgraduate student at the University of Canterbury, NZ. His interests include computer science education, mobile application development, human computer interaction and software engineering. He is currently working towards completing the CSFG chapters by developing content around the topic of Network Communication Protocols.

Heidi Newton holds a Bachelor of Science and Postgraduate Diploma in Science in Computer Science (University of Canterbury, NZ), and is a postgraduate student in the School of Engineering and Computer Science at Victoria University of Wellington, NZ. Her research interests include computer science education and artificial intelligence. She has contributed about half of the chapters in the CSFG.