Competition-style programming problems for Computer Science Unplugged activities

Janina Voigt, Tim Bell

University of Canterbury, Christchurch, New Zealand
jvo24@student.canterbury.ac.nz, tim.bell@canterbury.ac.nz

Bengt Aspvall

Blekinge Institute of Technology, Ronneby, Sweden
Bengt.Aspvall@bth.se

Abstract

Computer Science as a subject is struggling to attract young people in many countries around the world, including New Zealand and the US. Often students are taught programming, but don’t appreciate the richness of the whole area of Computer Science. In contrast, Computer Science Unplugged is a set of activities designed to teach school students about Computer Science concepts and awaken their interest in this subject area without using computers at all. The activities are designed to teach children about Computer Science without using computers, based on kinesthetic activities.

In this chapter we investigate combining the unplugged approach with competition-style programming problems to reinforce the concepts students learn using the Computer Science Unplugged activities, and to connect programming with Computer Science concepts. Such a combination of programming and activities has many advantages including allowing students to be introduced to Computer Science concepts while learning how to program, and adding physical activity to otherwise sedentary work. The activities are accessible to teachers without a specialist Computer Science background and can be used as part of programming competitions that stimulate students’ interest in the area while also having a strong teaching element.
1 Introduction

The need for skilled workers with Computer Science degrees continues to grow rapidly. According to the US Bureau of Labor Statistics, 5 out of the 30 fastest growing professions in the US until 2016 are related to Computer Science [16]. Software engineers and network systems and data communications analysts both make the top 5 list, with a projected growth of around 50 percent from 2006 to 2016.

Not only are Computer Science and IT professions amongst the fastest growing, they are also very highly paid. In New Zealand, a survey of 73,000 jobs on the “TradeMe” website found that IT professions had the highest salaries, with four out of the five highest paid jobs being in the area of IT [20].

Despite the fast growth of the area and the high wages offered, enrollments in Computer Science courses at university level are dropping [23]. Between 2000 and 2004, the percentage of students majoring in Computer Science dropped by 60 percent in many western countries. The drop in enrollments has been even sharper for girls, with the percentage of females majoring in Computer Science dropping by 80 percent between 2000 and 2004. While Computer Science used to be popular with females in the 1980s, the proportion of women to men in Computer Science today is at its lowest since the 1970s.

All this has produced an acute shortage of skilled staff in the Computer Science area. According to the New Zealand government’s Career Services website [18], there is an extreme shortage of skilled workers in the area of computing, with many advertised jobs left unfilled. The area is also listed on Immigration New Zealand’s long-term skill shortage list [24].

There have been many attempts to address these issues with initiatives ranging from the reform of undergraduate Computer Science courses to high school outreach programs. Such outreach programs may involve simple classroom visits by university students [14], after school programs [7], programming competitions [10], or school camps [11]. Many outreach programs have reported successes in terms of getting more high school students interested in Computer Science.

One approach which has been used for both outreach and teaching in schools is Computer Science Unplugged, a set of activities aimed at teaching school students about Computer Science concepts without using a computer [2, 3]. It has been used for over a decade as part of outreach programs to get school students interested in Computer Science and to address the issues faced by the subject [14].

In this chapter we look at programming problems designed to supplement Computer Science Unplugged activities to give high school students programming practice and help them gain a deeper understanding of some of the most fundamental concepts in Computer Science, while also engaging in physical activity away from the computer. These problems take advantage of the competitive and team-work aspects of programming competitions that generate interest for participants, and even spectators. Using a group of local high school students with inter-
est in Computer Science and programming, we have evaluated these programming topics and found that they help gain students a deeper understanding of Computer Science concepts.

2 Background

In this section, we will look at two successful ways to generate interest in Computer Science: Computer Science Unplugged and programming competitions. Later we will combine them to see if we can get the best of both worlds.

2.1 Computer Science Unplugged

Computer Science Unplugged is a set of activities designed to teach children about Computer Science concepts using puzzles and magic tricks [2, 3]. They are designed for a wide range of age groups, from primary to high school. Hands-on kinesthetic activities allow children to get involved and learn about concepts such as binary numbers, sorting, information theory, models of computation, and text compression. For example, a treasure hunt across several “islands” is used to teach students about finite state automata (FSA), where the islands correspond to states and the transitions are boat trips between the islands.

Computer Science Unplugged activities deliberately use no computers to make them accessible even to people who feel uncomfortable using computers, or those with limited access to computers.

The material has been translated into a number of different languages, including Korean, Japanese, Italian, Swedish and Chinese [e.g. 9, 22]. They were recommended as a supplement to conventional standards teaching in the 2003 ACM computing curriculum [6], and the curriculum is now being used in many schools.

The activities have been successfully used in classrooms and outreach programs. Children seem to love the fact that they can get involved and try the activities themselves. For example, Lambert et al. used Computer Science Unplugged activities as part of an outreach program and found that it increased students’ interest in Computer Science and confidence in their cognitive and mathematics skills [14]. Carmichael used the activities as part of a camp designed to get girls interested in Computer Science. She found that Computer Science Unplugged activities increased students understanding of the concept and of its relation to Computer Science. Carmichael noted that “after the CS Unplugged activity for this topic was finished, they seemed to feel more comfortable with the connection, particularly because of the discussion included in the activity.” [5]

Hart et al. used Unplugged activities as part of a workshop to inform and inspire mathematics high school teachers with an interest in Computer Science [12].
The feedback from the teachers was very positive overall with all participants stating that the session stimulated their interest and would improve some aspect of their teaching.

Taub et al. used Computer Science Unplugged to try to change middle school students’ views about Computer Science and found that the activities helped with that to a certain extent [21]. However, they noted that students found it difficult to connect the activities with central Computer Science concepts. This supports the idea using programming activities to make this connection.

2.2 Programming Competitions

Programming competitions are widely used to promote interest in Computer Science, both in high schools and at university level.

The annual ACM International Collegiate Programming Contest (ICPC) is perhaps the most famous one, requiring a team of students to solve as many programming problems as possible in a set amount of time. “The contest pits teams of three university students against eight or more complex, real-world problems, with a grueling five-hour deadline. Huddled around a single computer, competitors race against the clock in a battle of logic, strategy and mental endurance” [1]. The programming problems often require sophisticated algorithms and strategies in order to be efficient enough for the test data provided.

Paxton describes how he used programming competition style problems to teach a first year course entitled “Practical Applications of Data Structures and Algorithms” in Leipzig, Germany [17]. He found that the course was overall well received by the students, who participated in programming competitions and solved homework problems to gain credit. Even the exam for the course involved solving three problems that had previously been used in programming competitions.

Ladd and Harcourt also successfully used programming competitions to engage first year university students [13]. They used robot simulation software capable of interaction and animation and got students to program robots to do winter sports such as skiing and biathlon. Overall, they found that such competitions engaged and motivated students, making learning to program more interesting and fun.

Some criticisms have been put forward regarding this approach to programming [4, 10]. Bowring, for example, states that this competition paradigm is “at odds with the pedagogical goals of modern Computer Science and software engineering degree programs” [4]. He reasons that trying to solve as many problems as possible in a set amount of time encourages hacking and rather than looking for fast completion time, programs should be judged on quality. Fitzgerald agrees, stating that in the ICPC-style programming contest held for high school students by her college “90% of what we emphasize in our beginning programming and software engineering courses was discouraged” [10].
Another criticism of traditional programming contests is that their competitive nature, while motivating for some students, often alienates minorities. Fitzgerald for example noticed that participation levels from minorities and female students were very low in the ICPC-style programming contest organised by her college [10].

As a consequence, some alternative programming contests have emerged that focus on more than just the correctness and completion time for programs. Bowring describes how the College of Charleston student chapter of the ACM hosted a successful competition in 2007 where judging criteria for programs included technical as well as artistic characteristics [4].

A similar approach was taken by Sherell and McCauley, who ran a programming competition for high school students encouraging the use of a software process over “hacking” [19]. Teams were given a project description and had one month to design, implement and document a solution.

The Microsoft Imagine Cup is another competition that focuses on more than just the programming ability of contestants. For this competition, students need to create a product to help solve some of the world’s toughest problems and are judged not only on their technical ability but also on their presentation and business case.

The approach presented in this chapter broadens the possibilities for what can happen at a programming competition, to include social interaction, physical activity and problem solving away from the computer.

The programming activities we have developed are similar in style to the ACM programming contest problems. ACM programming contest problems are well-specified, describing in detail the required input and output for the program and the domain of valid values [1]. This is helpful because it means that solutions can be marked easily. ACM programming contest problems also come with a set of test input and output data for contestants to test their programs. Once the program is submitted, a large number of black-box tests will be run on the programs to test them thoroughly.

There are a number of online judging systems that could be used to set up a programming competition based around the programming problems we have designed. Such judging systems include Mooshak [15], a web-based scalable programming contest system, and DomJudge [8]. Of course, the set of problems need not be tied to competitions, and could be used as assignments or in self-paced courses.

3 Problem Design

We have designed and tested programming competition style problems relating to two Computer Science Unplugged activities. In the following section, we will first
3.1 Computer Science Unplugged Activities

As a basis for our programming problems we have selected two Computer Science Unplugged activities that we have found to be very popular with students of all ages: the “Parity Magic Trick” and the “Treasure Island” activity.

3.1.1 Parity Magic Trick

The Parity Magic Trick activity demonstrates the concept of error checking using parity bits. When data is stored or transmitted, it can be accidentally modified by mistake along the way, corrupting the data. It is important for us to be able to identify if and where errors have occurred. Adding parity bits to data ensures that there is always an even number of one bits in each row and column of data. If a bit gets changed along the way, the parity bits allow us to detect the mistake and in some cases even to correct it.

In Computer Science Unplugged, this concept is taught through a magic trick. It is a very popular and works well at the start of an outreach session for getting students engaged.

For the magic trick, one student is asked to put up a 5 by 5 square of magnetic cards on the board (or cards on a table). The cards are black on one side and white on the other, and the student can choose either colour for each card however they want. The teacher then decides to add another row and column to “make things a bit harder”. By adding this row and column, they ensure that there is always and even number of black cards in each row and column. This additional row and column of cards are called the parity bits. The teacher then turns away from the board and asks the student to flip one of the cards. By finding the row and column that now has an odd number of black cards they can determine which one was flipped [3]. This process can be seen in Figure 1.
The children seem to enjoy this particular activity very much. At first they are mystified by how the teacher knew which card to flip. Once the trick is explained, they are eager to try it out themselves.

This activity lends itself very well to competition-style programming problems since it is relatively easy to understand but has scope for more complicated extensions, including considering what happens when more than one card is flipped.

### 3.1.2 Treasure Islands

This activity involves a “treasure hunt” from island to island in search of Treasure Island, and can be used to teach students about finite state automata.

The activity includes a set of seven islands, each of which has two outgoing ship routes called A and B which lead to another island. A map of ships and islands can be seen in Figure 2, although students are given a version without the labeled arrows. Starting at Pirates’ Island, students are asked to take the ship routes to get to Treasure Island.
At each island, they have the choice of taking either ship A or ship B. An island is usually represented by an adult or another student holding up a card with the island’s name. Students can ask the person for either ship A or ship B (but not both) and are told which island to go to next. On their way to Treasure Island, students are given a map to complete to record the ship routes they use.

This activity is usually very well received by the students, who enjoy running around and competing with each other to see who can get to Treasure Island first.

It is ideal as a basis for competition-style programming problems because of the wide range of concepts and algorithms that can be introduced or discovered by the students in the process. The map of islands and ship routes can be seen as a graph that simple graph algorithms can be applied to it. Students can learn about breadth and depth-first search to find Treasure Island, as well as Dijkstra’s algorithm to find the shortest path given the length of each ship route. The problems can also be made easier and harder by allowing or disallowing cycles in the graph.

3.2 Programming Activities

This section describes two sets of programming problems based on the above “unplugged” activities: the first based on parity error checking and the second related to the Treasure Island activity. The sets contain nine and ten problems respectively, ranging in difficulty from easy to hard.

Each of the problems comes with a problem description, test input and expected output to allow students to test their programs.
We have implemented our own solutions to all of the programming problems in both Python and Java to test their correctness and difficulty, and to ensure that there are no unforeseen complexities with the tasks. In addition, we have also produced a teacher’s edition, which includes a description of the skills needed to solve the problem, a sample solution and an explanation of the how to solve the problem.

The parity error checking problems concentrate on the identification of errors using parity bits and on adding parity bits to existing data. In some of the harder problems, multiple errors have to be dealt with.

The Treasure Island programming problems address a range of graph problems, including searching problems that can be solved using depth-first or breadth-first search, and the shortest path problem.

### 3.3 Progression of Problem Difficulties

Each of the two activity sets contains problems of varying difficulty, from easy to difficult. We have made sure that there is a substantial progression in difficulty, giving students a clear path from the easy to the harder problems.

The parity problems start off dealing with single rows of bits rather than full rectangles. This means that no nested loops are needed for the implementation of these problems. Furthermore, some of the easier problems use bits, that is 0s and 1s, while the more difficult problems sometimes use other symbols to represent data.

In New Zealand schools, counted loops (such as for-loops) are usually taught before uncounted loops (such as while-loops), because the former tend to be easier to understand. Therefore, the easy problems can all be done using counted loops, and uncounted loops are needed only for some of the harder problems. This ensures that some of the problems are approachable enough even for novice programmers who have not yet learned about the more advanced programming concepts.

Parity bits can be used to identify and correct errors. In a two-dimensional parity check, if only one error has occurred, it can always be corrected. However, this cannot be done if there is more than one error in our simplified approach to error correction. Having multiple errors makes parity problems a lot more complicated. Therefore, the easier activities only deal with single errors, while multiple errors may occur in harder problems.

The Treasure Island activities deal with a range of graph problems including searching and shortest path. These problems can be a lot easier to solve if the graph used contains no cycles. Therefore, the easy and medium difficulty activities do not include cycles for such graph problems – these challenges are addressed only in the harder problems.
The search problems also progress from easier to harder by introducing more and more specific details. At the start, students are simply required to write a program to find a path to Treasure Island. Later they need to find the shortest path, all paths, or the shortest path going through another particular island.

The Treasure Island activities are very useful for introducing elementary graph algorithms such as depth-first search, breadth-first search and Dijkstra’s algorithm. They provide a simple and easy to understand scenario in which these algorithms can be learned and applied.

4 Evaluation

To evaluate the effectiveness of the programming challenges described, we evaluated them as part of sessions with high school students interested in Computer Science. The experiment setup is described in this section.

4.1 Participants

For the purposes of the experiment, we selected seven high school students with a particular interest in Computer Science and a good amount of programming experience. All seven students were in a Computer Science “STAR” course offered by the University of Canterbury, and were around 17 years old. This course allows high school students to complete the first year of university level Computer Science while in their last year of high school. As such, all seven students had a moderate amount of experience with Java, the language taught in this course.

While the students had some general programming knowledge from high school courses, they had used Java for only about three months at the time of the experiment. They had knowledge of loops and arrays, which were required for the programming activities. Overall, their knowledge of Java was comparable to the expected knowledge for an average student who has done about two years of programming at high school, which is the level of experience we would want to aim such a challenge at.

There was a range of abilities within the group of students, similar to that expected in a normal high school class. A couple of students clearly had a lot of programming experience and were accomplished problem solvers. They were given the harder problems to work on. Other students had done little previous programming and were still coming to terms with the concepts of loops and arrays. These students were given the easier problems to work on.

The experiment was done as part of one of the lessons for the STAR course. Students benefited from taking part in the experiment because they were able to gain extra programming practice.
4.2 Task

We performed the experiment over a two hour period. The students did the parity Unplugged activity physically (with cards), then were tested on their understanding. They were then given several programming problems related to parity (most students only completed one problem), and after that were given a post-test and questionnaire to evaluate the effectiveness of this approach.

We began with the Computer Science Unplugged activity to demonstrate the concept of error checking using parity bits. We showed students the parity magic trick and explained the concept of error checking using parity bits and its application in Computer Science.

To assess their understanding of the concept, we gave them a test requiring them to determine (visually) the errors that had occurred in a set of bits. The test included two separate questions. In the first question, they were given six parity problems with one error having occurred in each. The second question was a little harder because there could be zero, one or two errors per problem. Students were given a short amount of time to solve the problems so we could assess both their speed and precision.

They were also asked to rate their level of understanding of the concept at the end of the test. An example of a problem with one and two parity errors and the possible ways the errors can be corrected can be seen in Figures 3 and 4.

Following the Computer Science Unplugged activity (away from the computer), we gave students an example programming problem along with a Java model answer. The answer was explained to them to ensure that they understood the way that these programming problems should be solved. In particular, we explained the use of the Scanner class to read from standard input. The testing process was also demonstrated using the BlueJ environment that students normally use.
when programming in the course. In addition, students were made aware of the modulo operator and how it could be used to determine if a number is odd or even.

Next, we gave out the programming activities. Because there was a limited amount of time to do the experiment in, some students were given the easy questions, while others were asked to attempt the medium and hard questions. We handed out the questions based on our a priori assessment of the students’ Java and problem solving skills.

We gave students an hour to solve as many of the problems as they could. Anyone who finished the set of problems they were given could start on a second set.

Following the programming activities, we gave students a test similar to the one used before they did the programming, to test their understanding of the error checking concept. The test contained the same type and number of questions and was of similar difficulty. Again, the students were given the same limited amount of time to complete the questionnaire. The idea of this test was to determine if by programming and solving the programming problems students had gained a deeper understanding of the error checking concept.

We also gave students a second questionnaire that asked them to provide some personal information such as their age, gender and previous programming experience. They were then asked to rate various aspects of the programming activities on a scale from 1 to 5.

4.3 Limitations

The number of participants for this experiment was limited due to the availability of a suitable group. Only seven students participated in the experiment, meaning that the results produced are unlikely to be statistically significant. The particular group of students who participated in the experiment is also not necessarily representative of high school computing students, since they participated in a university Computer Science course after school, showing that they are particularly interested in the topic. All of the students had previous programming experience and planned to study Computer Science at university level.

Furthermore, because all of the students showed a particular interest in Computer Science, many of them had significant previous experience with computers, programming and problem solving. This is unlikely to be the case for the majority of high school computing students.

While there was an obvious spread of programming ability and experience amongst the seven students, all of the students were male, two students were Asian, and the rest Caucasian. Thus, the group was definitely not representative of the overall population and likely not representative of high school computing classes.
Overall, it is likely that this group of students did particularly well with the programming problems because of their programming experience and ability. This means that the results recorded for this survey may not hold up in normal high school computing classes, but provide an indication of the upper bound on the performance one might expect.

5 Results

All of the students who participated in the experiment managed to either solve at least one of the parity problems, or make a good attempt at a problem. Two of the students worked on the set of easy problems, with one of them completing two problems and attempting the third, and the other completing one problem and attempting the second. These problems were similar in difficulty to the programming exercises students were normally given as part of the course.

Two students attempted the medium problems but both struggled with those problems. The main reason for that was that some of the concepts needed to solve the problems, including nested loops and arrays, had only just been introduced in class so that the students were probably not sufficiently familiar with them to apply them successfully. The problems were definitely harder than the usual exercises students were given in class. Both of the students made a good attempt at the first medium problem but neither was able to complete it.

Three students worked on the hard problems. One student made a very good attempt at the first problem. One other student solved the first problem but did not have time to attempt the last problem while the last student both solved the first problem and made a good attempt at the second problem.

Overall, given the short amount of time students had to solve the problems, all seven students did a very good job given their previous level of programming experience.

The pre-test and post-test each contained 12 questions and were therefore marked out of 12. Half marks were possible on the second question of the test for stating the number of mistakes that had occurred but not identifying where they occurred.

All of the students gained significantly higher marks on the post-test in comparison to the pre-test. The pre-test and post-test marks for all students and the difficulty of the problems they attempted can be seen in Figure 5. The average mark increased from 7 on the pre-test to 10 on the post-test. On average, students gained 3.1 marks between the pre-test and the post-test with the lowest gain being 1.5 and the highest 5 marks. This supports the suggestion that the programming activities gave students a better understanding of the error checking concept. Notably, the three students working on the hard activities showed the highest mark gain at an average of 4.5 marks.
We gave out the programming problems based on the experience of the students and also asked them which they would prefer to look at. Interestingly, the two students who scored highest in the pre-test picked the medium activities and struggled with those. The students who chose the hard activities were all good programmers and made good attempts at the hard problems, but did not score as well in the pre-test. In fact, one of them received zero marks in the pre-test, indicating that he did not fully understand the concept. However, he was convinced that he understood error checking and that he could perform the magic trick himself according to his pre-test survey.

On the post-test, all students were able to attempt more of the problems than they had in the pre-test. On average, the number of questions attempted increased by 2.6. Because they were given a limited amount of time to complete the survey, no student managed to attempt all of the questions in the pre-test. However, in the post-test, four students attempted all questions though they did not necessarily answer all of them correctly. This shows that their speed in answering the questions showed a definite improvement.

In addition to answering more questions, most of the students also answered fewer questions incorrectly. Only two students made more mistakes on the pre-test than on the post-test. Notably, these two students were the only two who attempted the easy programming problems. By getting more familiar with the error checking concept, they could answer more survey questions in the post-test. This meant that they were able to get as far as the last questions which included problems where two errors had occurred. The easy programming problems only deals with single errors and the students therefore did not have to think about dealing with multiple errors. Therefore, while their knowledge about error checking in general increased, they still had not learned about multiple errors and therefore answered those questions incorrectly.

For the rest of the students, the number of incorrect answers decreased or stayed the same. Two students made no mistakes in both their pre-test and their post-test. The other students decreased the number of mistakes they made by 1 to 3, showing that their understanding of error checking had clearly improved.
In the pre-test and post-test, students were also asked to rate their understanding of the error checking concept and their ability to perform the magic trick themselves. Most of them already rated their understanding of the concept highly in the pre-test, with all students picking either 4 or 5 out of 5. On the post-test, there was no change recorded for most students while one student rated his understanding one point lower than on the pre-test. Again, all the ratings were 4 and 5 out of 5.

It is interesting that students did not seem to think that their understanding of error checking increased even though there was a significant improvement of performance in the test. One reason for this may be that students mistakenly thought they completely understood the concept after the Computer Science Unplugged activity.

For the second question, students were asked to rate how well they could perform the magic trick themselves. Again, most ratings were 4 or 5 out of 5 in the pre-test, with one student rating his ability at only 2 out of 5. While four students did not believe the programming activities increased their ability to perform the magic trick, three increased their ratings by 1 or 2 points. In the post-test, all students rated their ability as 4 or 5 out of 5. All of the students whose ratings did not increase for this question already selected 5 out of 5 on the pre-test so that no improvement was possible.

After the programming activities, we also gave students a survey about the activities and Computer Science and programming in general. The survey contained seven statements that students rated on a scale from 1 to 5, where 1 meant strongly disagree and 5 meant strongly agree. The average response out of 5 for all statements can be seen in Table 1.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Average Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am an experienced programmer.</td>
<td>3.0</td>
</tr>
<tr>
<td>Computer Science is interesting.</td>
<td>4.4</td>
</tr>
<tr>
<td>I enjoy programming.</td>
<td>4.6</td>
</tr>
<tr>
<td>The programming activities were fun.</td>
<td>3.6</td>
</tr>
<tr>
<td>The programming activities were frustrating.</td>
<td>2.7</td>
</tr>
<tr>
<td>The programming activities had the right level of difficulty.</td>
<td>3.1</td>
</tr>
<tr>
<td>I would be interested in doing more of the programming activities</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Table 1: Responses to the final questionnaire (1: strongly disagree, 5: strongly agree)

All students responded very positively to the propositions “Computer Science is interesting” and “I enjoy programming”, with average scores of 4.4 and 4.6. This is to be expected because all of them were taking a voluntary after-school class in Computer Science.

Most of the students also said that the programming activities were fun with five students selecting 4 out of 5 and one student each selecting 2 and 3 out of 5. It
also seemed that students did not find the activities too frustrating, with two students selecting 2 out of 5 and five students selecting 3 out of 5 on the proposition “The programming activities were frustrating”.

There was some disagreement about the level of difficulty of the activities. One student who struggled with the medium problems selected 1 out of 5 for the proposition “The programming activities had the right level of difficulty”. Four others selected 3 out of 5 with one person each selecting 4 and 5 out of 5.

The last proposition was “I would be interested in doing more of the programming activities” and was essentially the ultimate test of whether or not students enjoyed this style of learning. While one student selected only 2 out of 5, two students each selected 3 out of 5, 4 out of 5 and 5 out of 5. This shows that at least four of the students would definitely would like to do more of the programming activities.

Overall, the programming activities clearly improved students’ understanding of the error checking concept, allowing them to work through the test faster and more accurately, answering significantly more questions correctly on the post-test. Most of them seemed to enjoy the programming activities, with mostly positive responses to the propositions “The programming activities were fun” and “I would be interested in doing more of the programming activities”. All this indicates that the programming activities are a valuable addition to the Computer Science Unplugged activity.

6 Future Work

We plan to conduct more experiments to confirm the results presented in this work. For these follow-up experiments, computing students from local high schools will be used as participants to ensure that the results presented here translate to the general high school population.

In this experiment, we only tested one of the two sets of programming activities we designed. In future experiments the set of Treasure Island activities which includes a number of more difficult activities should also be evaluated.

There is also the possibility of using the activities in the first year Computer Science courses at university. There are a number of challenging activities that may be too difficult for most high school students and would be more adequate for university level students.

If the activities prove as successful as this experiment indicates, more programming problems should be developed for different Computer Science Unplugged activities. Possible topics for such activities include binary numbers, minimum spanning trees, sorting, searching and text compression. Such activities could well be used in algorithms courses at university level to reinforce concepts.

Once thoroughly tested, the programming activities and teacher’s manual for the activities can be made available as a teaching resource online. As such, they
could be used by secondary school teachers or university lecturers for homework, assignments or programming competitions.

7 Conclusion

We have designed a suite of competition-style programming problems for high school students that relate to Computer Science Unplugged activities. This has allowed us to run sessions where we first demonstrate a particular Computer Science concept using the Computer Science Unplugged activity, and then deepen the students’ understanding of this concept by asking them to solve related programming problems.

We have developed two such suites of programming problems related to error checking using parity bits and finite state automata. These programming problems can be used in a variety of ways, including school projects and programming competitions. Because of the range of problems we have designed, there are problems accessible to weaker students as well as some that will challenge even the top students.

Tests that we conducted with high school students have confirmed our expectations that giving students the chance to further explore the concepts they learn through Computer Science Unplugged deepens their understanding of the concept. Students were given two tests to determine the depth of their understanding of the concept: one test after the Computer Science Unplugged activity and one test after the subsequent programming activity. Their performance on the test after the programming activity was significantly improved.

Furthermore, students responded mostly positively to questions they were asked about the programming activities in a questionnaire at the end of the session.

All this indicates that supplementing Computer Science Unplugged activities with competition-style programming problems increases students’ understanding of Computer Science concepts. For competitions, the Unplugged activities provide an opportunity for physical or kinesthetic activity to break up long periods working at the computer. As such, we believe that this kind of programming problem is a useful addition to Computer Science courses in high school and outreach programs.

8 References


