Teaching Entering Students to Think Like Computer Scientists

Elise H. Turner Roy M. Turner
Computer Science Department University of Maine 5752 Neville Hall Orono, ME USA 04469-5257
{eht,rmt}@umit.maine.edu

ABSTRACT

This paper describes a new course developed at the University of Maine to address problems of recruitment and retention (especially of women), majors not knowing what computer science is, and students' confusion of computer science with programming. The course, called Techniques in Computer Science (TCS), is a rigorous, theoretical, non-programming introduction to computer science. It provides an overview of computer science, but also, through focusing on particular topics at an advanced (junior/senior) level, begins to teach students how computer scientists think about problems. We first taught the course in Fall 2002 and have taught it each Fall since then. Results from the first two years suggest that the course is successful in meeting the objectives. This paper describes the course and discusses our results so far with it.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms

Education

Keywords

Gender issues, Introductory Computer Science

1. INTRODUCTION

At the University of Maine, we are faced with many of the same problems in our introductory courses that face other computer science departments nationwide. In particular, we were concerned about the following problems:

- Potential majors did not find out what the major was really about until their junior or senior year [1]. Those students who wanted only to program felt deceived when they were forced to focus on more abstract concepts in upper-level courses. Others, particularly women, left the major because they believed the major would lead to a career as a programmer.
- Non-majors carried into the world the belief that computer science was only computer programming. Undecided majors, often women, who were talented in mathematics and science and interested in abstract problem solving might not try computer science because they were not particularly interested in programming [2].
- The details of programming often distracted students from more conceptual information presented in CS-I and II (see also [3]). Success with this strategy in early courses caused some students in upper-level courses to spend their time on programming assignments to the exclusion of studying the theoretical material presented in the course.
- Introductory computer science courses did not introduce students to how computer scientists address important problems in their field. After a year, many students were quite competent programmers but did not know how to think like computer scientists.

To address these problems, Techniques in Computer Science (TCS) was developed to be a rigorous, theoretical, non-programming introduction to computer science. We began teaching the course in Fall 2002 and have taught it each fall since that time. Results from the first two years suggest that the course is successful, and we expect it to be required for all of our students as part of our current curriculum revision.

In this paper, we give an overview of TCS and evaluate the course's success. In Sections 2 and 3, we give a general overview of the course, including the structure of the course and the topics covered. To give an idea of the rigor and depth of coverage of the topics, Section 3 also describes how a single topic, Hamming codes, is presented in the course. Section 4 presents results and Section 5 discusses features of the course we believe helped retain students.

2. TECHNIQUES IN COMPUTER SCIENCE

Techniques in Computer Science (TCS) was developed to introduce students to the field of computer science in a way that would focus on problem solving, instead of programming, in computer science. Students cover five areas in the course: digital logic, computer organization and architecture, programming languages, operating systems and computer networks. Each area has several topics associated with it, shown in Figure 1.

By structuring the course in this way, we have intentionally given up some breadth to allow examination of problems in more depth. For each topic, students are presented with a specific, real problem in computer science and learn the

SIGCSE '05 St. Louis, MO USA

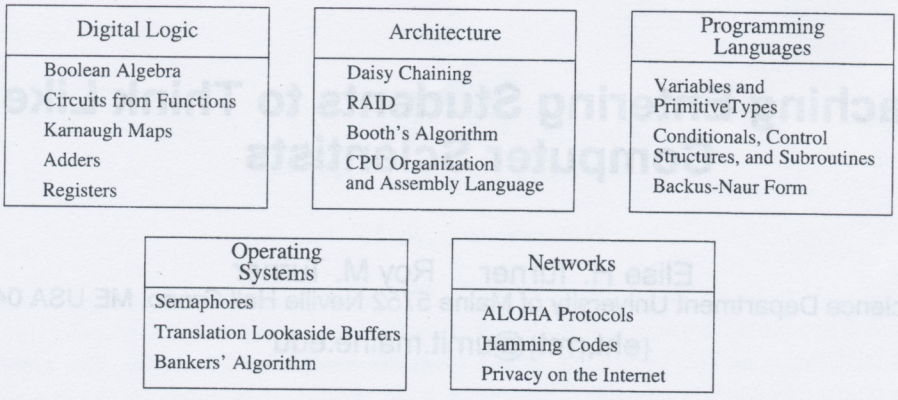

Figure 1: Areas covered, with their topics.

details of a solution to this problem, as well as the reasoning which led to that solution. Students are expected to learn the material at the level of an introductory course in that area. So, often students are exposed to specific topics at the junior or senior level. To determine if students are learning the material at this level, homework problems and exam questions are often taken from upper-level classes. We will discuss the topics in more detail in the next section.

In selecting topics for the areas, the following criteria were used, in order of importance [4]:

1. Does the topic present a challenging technique that will be interesting to the students?
2. Is the topic teachable in an introductory course within a fifty-minute lecture?
3. Does the topic allow us to introduce a way of thinking that is prevalent in computer science?
4. Does the topic cover an aspect of the area that is not covered by other topics?
5. Is the topic central to the area?

No course with a goal of teaching students to think like computer scientists would be complete without teaching the students about abstraction. To do this, we pointed out abstraction throughout the course. For example, in the introduction to the course, we point out that the areas take us from working close to the machine level to higher and higher abstractions. In the introduction to digital logic, we describe how gates are abstractions of actual circuits. We discuss programming language design in terms of providing an abstraction that allows users to think about their problem at the appropriate level.

Because we are focusing on several specific techniques within each area, we could not cover all areas of computer science. To add more breadth to the course, guest lecturers from our faculty are brought in to talk about their research areas if they have not already been covered. A half-lecture each is devoted to a quick overview of software engineering, artificial intelligence, theoretical computer science and graphics. These introductions simply make students aware of the kinds of problems that are solved in these areas, and give them an opportunity to meet our faculty. We expect students to be introduced to techniques from software engineering and algorithm design and analysis in their introductory programming class. In the future, we plan to add artificial intelligence and computer ethics as areas in TCS.[1]

The course is structured to support students trying to master the difficult material covered in this class and to help them develop study habits that will serve them well throughout their college careers. In addition to class meetings, there is also a recitation with a teaching assistant each week, and students are encouraged to meet with the instructor in his or her office.

In addition, every effort is made to make each lecture as self-contained as possible. Because there are no prerequisites, and because the material covered is so different from most students' previous experiences with computers, we assume that students are seeing everything for the first time. Although we are able to focus on specific techniques or narrow concepts and examine them in depth, the lectures necessarily devote some time to giving a context for the problem. This context may be background information needed to understand the problem or its solution. For example, to prepare students to learn about adders, we also teach them about representing numbers in a computer and about binary addition. This also gives us an opportunity to introduce broader issues. For example, to set the stage for a discussion of daisy chaining, we must first tell students that buses allow components to communicate and explain why that communication must be arbitrated.

Homework assignments are given at the end of each topic lecture and cover a single topic. A small number of homework problems drill students on specific skills (e.g., binary arithmetic). Most homework problems and exam questions, however, are similar to or taken directly from assignments in junior and senior level introductions to the area. There are 4–6 questions, usually multi-part, in each homework assignment. Special care is taken to ensure that at least one or two homework problems in each assignment challenge the students to go beyond work in the classroom. Students are explicitly told that the homework is to help them understand the material, and they are encouraged to form study groups and work together on homework problems.

3. COVERING TOPICS IN TCS

In this section we illustrate how most topics were presented, using Hamming codes as an example. But first it is

[1]We will find space for these areas by replacing in-class review sessions at the end of each area with recitations.

important to briefly discuss the two areas which have topics that do not obviously fit that pattern: digital logic and programming languages.

The topics covered in digital logic, when possible, adhere to our intention of introducing students to a particular technique for solving a problem in depth. This is clearly the case with Karnaugh maps (How can we minimize circuits?) and registers (How do we store data in a computer?). Although the lecture on algebraic equivalence gives students the background and homework experience to find equivalent circuits and prove their equivalence by algebraic substitution, this topic is less narrow than most others in the class. However, we felt that this was a natural way to give students an opportunity to use proofs in the class and to think about formal reasoning. The topic of moving from functions to circuits is introductory material that must be covered if the rest of the area is to be understood. Since gates and an overview of Boolean operations is covered in the introductory lecture, we do not have time to cover creating circuits there. The lecture on adders may seem to be too narrow. However, we use adders to make the point that circuits can be better constructed if we understand the problem and use our insights to inform the solution. We also must introduce number representation and computer addition in the adders lecture, so, although adders can be covered in a few minutes in an upper-level architecture course, we require a complete lecture to present them in TCS.

It may not be clear how the area of programming languages can be presented in a non-programming introduction. Frankly, this was the area that we were most concerned about when developing the course. We wanted this section to introduce students to the constructs of programming languages and to issues in programming language design. The introduction to the area quickly covers the idea of needing to translate a programming language that is helpful to the user to code that can be executed by the computer. The bulk of the introduction is spent on discussing what kinds of tasks people need to perform using programming languages and how we might think about those tasks, using one particular task as an example. This allows us to talk about the basic constructs that are needed to support problem solving. Topics that cover the constructs allow us to talk about exactly what those constructs contribute, the design decisions that are made when the construct is included in a programming language, and how those design decisions are limited by the computer. In the lecture on variables, for example, students first see how variables are used to store values in a program. Next, students learn about the general concept of identifiers and discuss such design issues as the length of variable names and case-sensitive identifiers. Although this course, and particularly this area, were never intended as preparation for the programming course, many students who took TCS reported that they felt their understanding of programming languages from TCS gave them a framework for understanding programming language constructs which made learning about programming easier.

3.1 Teaching Hamming Codes

Hamming codes are presented as a topic in the networks area. As with all topics, we begin with a discussion on why we study Hamming codes. For Hamming codes, we tell students that these are error-correcting codes and that they are used in computer networks and internal memory.

The students have already heard of Hamming codes from studying RAID in the organization and architecture area. There, the RAID level which uses Hamming codes must be glossed over, and we remind students of this connection here. They were also introduced to parity bits when studying RAID. So, we can begin the technical portion of the lecture with a brief review of what parity bits can and cannot do. We also remind them that in RAID architectures we can correct errors because we know the location of the bit that contains the error.

Next, we set up a straw man using the idea of having a parity bit for each bit of data and then a parity bit for the parity bits. We show how the scheme would work and point out that it would more than double the size of memory. Students now see that they need to find the location of the offending bit, but should do it using as little memory as possible.

For the rest of the lecture, we present Hamming codes in the same way that we would present this topic to juniors and seniors. We summarize the goal of identifying the errors using as few bits as possible and give an overview of the basic idea of Hamming codes.

Next, we show how parity bits can be associated with data bits, using a Venn diagram for four bits of data. We then give examples of how to correct the data using the Venn diagram.

Now that the students understand the overall concept, we explain how Hamming codes work in streams of data. By this point in the course, students are comfortable enough with binary arithmetic[2] to understand how checkbits can be added to the data, and to think about the appropriate size for segments of data.

We now go further into detail, explaining how to create a Hamming code for eight bits of data. Finally, we give examples of how the Hamming code can be used to detect and correct data.

The Hamming code homework has the following questions:

1. Assume that the following bits have been sent across the network. Hamming code has been used to encode 8 data bits. Give the *correct* data. Show your work.

   (a) 101110010010

   (b) 111010011111

   (c) 100101100010

   (d) 001001001001

   (e) 111111111111

2. Show how Hamming code can be used for 6 bits of data. Give the location of the data bits within the encoded representation. Give the location of the check bits, and tell which bits they will check.

3. If only a single parity bit is used, what happens when there are two errors? If Hamming code is used, what happens when there are two errors?

[2] From performing calculations in the digital logic and architecture areas and getting used to logarithms and the amount of data that can be stored in a given number of bits from the operating systems area.

4. We said in class that the number of extra bits (in excess of data bits) required for Hamming code is on the order of log_n, where n is the number of data bits. The number of bits required is actually $log_n + x$, where x is the same for any number of data bits. What is x? Make a convincing argument for the value of x you have given.

4. EVALUATION OF THE COURSE

We began teaching TCS in Fall 2002 to test the concept of the course and to see if it should be included in the computer science major. Because it was not required of our majors, first-year students were only encouraged to take the course if they had been placed in remedial programming because they showed little aptitude in math and science. The first class also included an undeclared major, an upperclassman from a different major, and a woman in our major who was considering leaving the major because she was more interested in problem solving than programming. In Fall 2003, the course was taught for a second time. This time more students were encouraged to take the class. In addition to students who were placed in remedial programming, we also encouraged students who were drawn to computer science because of their interest in programming. This was meant to allow these students to find out if the major was right for them. Because of our success in retaining women the previous Fall [5], we also encouraged all first-year women to enroll in the course. We are currently teaching TCS for the third time. TCS was successful enough in the first two years that this year our curriculum committee recommended it for all first-year majors. The committee also expects to include it as a required course as we complete our curriculum revision this Fall. In this section, we discuss how the course has been evaluated and what we have found.

4.1 Results of Student Surveys

Each year students filled out surveys at the end of the first and last class. At the beginning of the semester in the first two years, most students had little or no programming experience. Although they felt that computer scientists should be good at solving problems, many felt that the most interesting thing they would learn in the class would be some skill used by programmers or computer operators. They also thought that the most important thing they would learn was whether or not the major was a good match for them.

By the end of the semester, students were more sophisticated about computer science. When asked what was the most interesting thing they had learned, most cited an individual topic. When asked what was the most important thing that they had learned, one of two sorts of answers appeared in most responses. Most students who responded in terms of computer science said that they thought everything in the course was important. Other students said that the most important thing they learned from the class was to be a good student.

Although the course was rated as very hard on regular student evaluations and some students mentioned this in their responses to questions on the survey, most students who were computer science majors felt the course was worth their time and effort.

4.2 Retention of Students

One goal of TCS was to retain students who are a good match with computer science. We feel the course achieves this goal.

In the first semester, fifteen students completed surveys on the first day and only nine completed the course. In the second semester eleven students completed surveys on the first day and ten completed the course. We believe are several reasons for the difference in these numbers. The first time the course was taught, the class consisted mostly of students who would be expected to drop out of computer science in their first semester. Yet, our drop-out rate was less than that of the introductory programming courses (either the remedial courses or CS I). It is also clear from student surveys that most of the students who dropped the course did so for reasons that would make them unhappy with a computer science major. Beginning of the semester surveys indicate that four of these students had previous experience using very specific skills (e.g., installing operating systems, troubleshooting without programming, Web design, certifications, very limited programming skills) and expected a B.S. in computer science to be more of the same, plus programming.[3] Students like these are often frustrated by the major in their junior and senior years. The second year that the course was taught, the New Media major was offered on campus for the first time. This major seems to attract students who are interested in becoming Web designers or computer operators. The student who dropped the course in the second year seems to have taken the course because he believed it was required for another major and dropped TCS when he realized he was in the wrong course.

Although the course was not specifically designed to retain women in the major, all women who took the course decided to become computer science majors or to enter a closely-related major. In the first year, the women who took the course included a first-year student who was an undeclared major and did not expect to major in computer science and a sophomore computer science major who was planning on leaving the major. At the end of the course, these two women planned to remain in the major. Another woman changed majors from computer science to electrical engineering technology because TCS helped her to recognize that she was more interested in working at the hardware level. These women's perspectives on the course and how it influenced their interest in remaining in the major is the subject of a technical report [5]. In the second year, three women responded to surveys on the first day. All expected to be computer science majors, all completed the class, and all expected to remain computer science majors at the end of TCS.

4.3 Preparation for Further Work in Computer Science

Students who took TCS in Fall 2002 are currently beginning juniors. Consequently, it is difficult to evaluate whether or not TCS will help students to think like computer scientists in their upper-level courses.

Because TCS was never meant to help prepare students for programming, we were surprised by the number of students who told us in casual conversation that their understanding of TCS helped them to better understand programming. As more students complete TCS and we collect more data, we

[3]One student withdrew for the semester for reasons unrelated to any courses, and another dropped the course because he thought it was too much work.

hope to determine if these students feel more comfortable because they are more mature, because they have been exposed to programming language constructs and are better able to focus on programming concepts instead of the syntax of a language, or because they have a feel for the way to think about computer science and do not ignore topics like algorithm design and software design to focus on programming in CS-I.

5. FEATURES OF THE COURSE THAT HELPED RETAIN STUDENTS

There are several features of TCS that we believe helped us to retain students in the major. Some of these are particularly important to women, but we believe they are helpful to all students.

- The course was rigorous. Students were engaged in the types of challenging problem solving that our best majors, and particularly women [6], often find a very attractive aspect of the computer science. It also built students' confidence that they could handle difficult material.
- Students knew that they were in a "real" computer science course, so they could expect success in CS-T to predict success later in the major. Students knew that the content of the course and the rigor required were similar to that of other computer science courses. Students were repeatedly reminded in class, especially when homework assignments and tests were returned, that they were doing work usually expected of juniors and seniors in the major. Women and men, majors and non-majors were in the class together, demonstrating to women that they could successfully compete with men in the major (see [7]).
- The experience that most high school students have with computers was not relevant in TCS. Consequently, students who entered college with little or no computer experience, as is often the case in our state, were not disadvantaged. In addition, aggressive students who may intimidate others by flaunting their knowledge in introductory programming courses [8, 9] have few opportunities to discuss this knowledge in TCS.
- The structure of the course supported the students' learning of difficult material, which, in turn, built their confidence [10]. Although the students reported in surveys and casual conversation that they worked hard, they also believed that the material was not particularly difficult.
- Students fully appreciated that computer science was not equal to computer programming. On end of the semester surveys, four students in the first year the course was taught cited this as the most important thing that they learned in the class. For students, often women [2], who are more interested in problem solving than programming, this knowledge can make the difference between staying in the computer science major or leaving it.
- The course covered a range of areas and a wide variety of topics. This allowed students to get an accurate sense of the field and helped to satisfy a diversity of interests.

6. CONCLUSION

This paper presents preliminary results from teaching a rigorous, non-programming introduction to computer science. Results from two years of experience with the course suggest that students are interested in learning about important techniques in computer science, even when learning about them requires considerable work. Our preliminary results also suggest that by teaching students about the interesting problems in computer science, students, including women, want to continue in the major.

7. ACKNOWLEDGMENTS

The authors wish to thank Thomas Wheeler and Judith Richardson for helping to create the philosophy of the course and for many helpful discussions about the content of the course. We also would like to thank the students who participated in the course and gave us feedback through student surveys, course evaluations, and casual conversation. We would especially like to thank Cathy Emerton, Christina Logan and Rebecca Ray, the women in the Fall 2002 class who participated in writing the technical report about their experiences. Parts of two technical reports about TCS [4, 5] have been used in this paper.

8. REFERENCES

[1] T. P. Murtagh. Teaching breadth-first depth-first. *SIGSCE Bulletin*, 33(3):37–40, 2001.

[2] J. Prey and K. Treu. What do you say? Open letters to women considering a computer science major. *Inroads (The SIGCSE Bulletin)*, 34(2):18–20, 2002.

[3] A. B. Tucker, C. F. Kelemen, and K. B. Bruce. Our curriculum has become math-phobic. In *The Proceedings of the Thirty-Second SIGSCE Technical Symposium on Computer Science Education*, pages 243–247, New York, NY, 2001. Association for Computing Machinery, Inc.

[4] E. Turner, R. Turner, and T. Wheeler. A rigorous introduction to computer science without programming. Technical Report UMCS–TR–2004–1, Department of Computer Science, University of Maine, 5752 Neville Hall, Orono, ME 04469, 2004.

[5] E. H. Turner, C. Emerton, R. Ray, and C. Logan. A rigorous introduction to computer science without programming: Three women's perspectives. Technical Report UMCS–TR–2004–2, Department of Computer Science, University of Maine, 5752 Neville Hall, Orono, ME 04469, 2004.

[6] M. Barg, A. Fekete, T. Greening, O. Hollands, J. Kay, and J. H. Kingston. Problem-based learning for foundation computer science courses. *Computer Science Education*, 10(2):109–128, 2000.

[7] A. Pearl, M. E. Pollack, E. Riskin, B. Thomas, E. Wolf, and A. Wu. Becoming a computer scientist. *Communications of the ACM*, 33(11):47–57, 1990.

[8] L. J. Barker, K. Garvin-Doxas, and M. Jaskson. Defensive climate in the computer science classroom. In *The Proceedings of the Thirty-Third SIGSCE Technical Symposium on Computer Science Education*, pages 43–47, New York, NY, 2002. Association for Computing Machinery, Inc.

[9] B. C. Wilson. A study of factors promoting success in computer science including gender differences. *Computer Science Education*, 12(1–2):141–164, 2002.

[10] K. Treu and A. Skinner. Ten suggestions for a gender-equitable CS classroom. *The Journal of Computing in Small Colleges*, 12(2):244–248, 1996.

Assignment 4
Due: November 21, 2012

Group assignment: You are permitted to work in teams of 1 to 4 students. No special permission will be given for larger groups. Write down an object-oriented program in C++ or Java or Ocaml or another object-oriented programming language. Submissions that do not use OO concepts will not be evaluated.

In this assignment, we will implement the Huffman encoding and decoding algorithm. Compress the files located at:
http://sydney.edu.au/engineering/it/~matty/Shakespeare/texts/tragedies/juliuscaesar
http://sydney.edu.au/engineering/it/~matty/Shakespeare/texts/tragedies/hamlet
http://sydney.edu.au/engineering/it/~matty/Shakespeare/texts/tragedies/othello
http://sydney.edu.au/engineering/it/~matty/Shakespeare/texts/tragedies/macbeth

, and then de-compress these files. Compute the ratio of the compressed files and the original files for each of these inputs.

We will execute the Makefile in your submission folder before we make the call to .huffmanencode <filename> and ./huffmandecode <filename>. **Please submit the source-code as a zipped folder named asg4.zip to the instructor and the TAs on November 21, 2012** – do not include any pre-compiled binaries. **Submit a (hard copy) report summarizing the data structures and algorithms used and their run-time complexity to the TAs. Also, include a table in your report indicating the ratio of the compressed files and the original files for each of the inputs.**

Extra Credit Project

Due November 21, 2012

November 5, 2012

Instructions: All source code must be written in C++/Java/OCaml for this extra-credit project. Each project must be completed in teams of at most 5 students. A report that clearly summarizes the algorithms, pseudocode, and results obtained should be submitted to the instructor at the beginning of the class on *November 21, 2012.* The report may include table, graphs, and plots that help communicate the results. The source code should be emailed to the instructor jha@eecs.ucf.edu before 4:30pm on November 21, 2012. The email should have the subject line: *COP3503 Extra Credit Project Submission*

Data: The Excel file at `http://www.imf.org/external/pubs/ft/weo/2012/02/weodata/WEOOct2012all.xls` or the shortened URL `http://tinyurl.com/d34p4co` contains the NGDPDPC (National Gross Domestic Product Per Capita, Current Prices) expressed in US Dollars for all the countries studied by the IMF (International Monetary Fund). We may study the NGDPDPC data for all countries from the year 2000 to 2011, and seek to determine the dependence between different economies.

1. Compute the Pearson correlation coefficient $C(x, y)$ between the NGDPPCs of every pair (x, y) of countries. You should only use the data from 2006 to 2011. See `http://en.wikipedia.org/wiki/Correlation_and_dependence#Pearson.27s_product-moment_coefficient` or the shortened URL `http://tinyurl.com/bwjj7lw` for a definition of the Pearson correlation coefficient.
2. Construct the (undirected) graph $G = (V, E)$ where V is the set of sovereign countries studied by the IMF, and E will be used to describe the correlation between the economies of two countries, as measured by the NGDPPCs. As edge $(x, y) \in E$ if and only if $|C(x, y)| > t$, where t is a constant threshold. Compute the value of t such that the edge density of G is between 0.05 and 0.50. Any such choice of t will be considered acceptable.
3. Compute the connected components of G. How many components are there in G? What does it imply?
4. The *distance* between two countries in this network G is the product of the *magnitudes* (absolute values) of the correlation coefficients. Using the graph G, compute the list of those countries whose economies have a distance of at most d from the economy of:
   (a) United States of America
   (b) United Kingdom of Great Britain and Northern Ireland
   (c) Federal Republic of Germany
   (d) Italian Republic
   (e) Hellenic Republic (Greece)
   (f) People's Republic of China
   (g) Republic of India
   (h) Federative Republic of Brazil
   (i) Russian Federation
   (j) United Mexican States

   Perform this analysis for the values of $d = 0.5, 0.05, 0.01, 0.001, 0.0001$, and 0.00001. You may choose to display your results in a table or a graph.
5. Using the graph G, compute the distance between a few countries that have been involved in a conflict over the last 10 years. Also, compute the distance between a few countries that are part of a collaborative group such as NATO, EU, NAFTA, OPEC, etc. You may choose to display your results in a table or a graph.