Solution Spaces

Nadia Kasto[§], Jacqueline Whalley[§], Anne Philpott[§] and David Whalley[†]

[§]School of Computing and Mathematical Sciences
AUT University
PO Box 92006, Auckland 1142, New Zealand
{nkasto,jwhalley,aphilpot}@aut.ac.nz

[†]Educational Consultant
Auckland
New Zealand
geowah@yahoo.com

Abstract

This paper explores the idea of solution space in the context of novice programmers and code writing tasks. A definition for solution space is provided and an analysis of a series of code writing questions from a first year Java programming course's practical programming tests is provided to measure the impact of solution space size on the difficulty of a code writing question. We found that as the solution space size increases so does the difficulty of the question and that despite relatively high solutions spaces we see a very limited set of these solutions as student responses. Finally we conclude with some conjectures about the possible causes for the trends that we have observed.

Keywords: novice programmers, code writing, assessment, task complexity.

1 Introduction

"Writing high quality readable text does not come easily to most young children. Many elementary teachers express frustration at the apparent poor written products emerging from their students." (Beaglehole and Yates 2010). Similar themes have appeared in the computer-science education literature: students don't know how to design programs, and they don't know how to write programs. Soloway and Sopherer (1989) suggested that "students have difficulties in putting all the pieces together" and "many problems arise from structure composition problems". Winslow (1996) supported this view stating that, "Study after study has shown that students have no trouble generating syntactically valid statements once they understand what is needed. The difficulty is knowing where and how to combine statements to generate the desired result".

It is generally accepted by teachers that many students who are learning to write find the task easier if they are given a more open task. This premise is supported by Rogers's learner-centered model of teaching (Rogers et al. 2013). When they are allowed to write about a topic of their own choice these students quickly decide what topic they would like to write about and how they will go about it. Some other students tend to flounder in such a large space and cannot decide how to get started. On the other hand if the students are directed to write on a specific topic set by the teacher, for example a grandparent's birthday, some find that the restricted scope makes the writing task easy for them while others have difficulty engaging in a task that provides them with such limited possibilities for writing. The reasons for finding a particular writing task difficult may include: a lack of personal experience- the students may never see their grandparents-, a lack of interest in the topic, a strong desire to write about a personally more motivating topic, a perceived absence of an audience for the finished product or a lack of the vocabulary needed to engage in the topic set. In effect, some find that a large solution space provides them with many opportunities and allows them to make choices that result in effective writing. Others find a large solution space daunting and have difficulty making productive choices. What effects do differences in the solution space of programming tasks have on the ability of novice programmers to successfully complete those tasks?

In programming there are many ways to tackle a fairly small problem, and different students can produce different solutions to the same problem. In a preliminary small scale study Carbone (2007) found that when students were given open programming tasks, tasks that had many possible ways to approach the problem and hence a large solution space, some students focused on a wrong aspect of the task or pursued a wrong approach as they lost track of the big picture. It seems reasonable to assume therefore that solution space has some influence on the difficulty of a novice programming task.

In a recent study that attempted to evaluate the difficulty of questions presented in final examinations the group of academics found it difficult to agree on the difficulty of questions (Simon et al. 2012). The degree of agreement between the academics in estimating difficulty was only 40% so the inter-rater reliability was poor. This finding indicates that it is difficult for educators to be objective in their estimations of difficulty of assessment items in computer programming. There is a tendency to both under and overestimate the difficulty of these tasks. Clearly there is a need for more objective measures of difficulty for novice computer programming tasks.

2 The solution space conjecture

Our conjecture is that the difficulty of code writing tasks, for novice programmers, is related to the size (and possibly other dimensions) of the solution space for a problem. We were also interested in whether or not the number of solutions provided by students, the *students'* *solution space*, to a code writing problem is influenced in any way by the size of the *problem's solution space*.

Luxton-Reilly et al. (2013) investigated the variation in correct student solutions for problems. They defined three different types of variation: variation in structure, syntax (within a block) and presentation. For this research solution space is defined as the set of structurally or syntactically different solutions that provide a correct answer for a specific code writing problem. The addition of redundancies (for example, extra semicolons, empty if or else statements) have not been counted as additional solutions.

The notion that difficulty might be related to solution space size is perhaps not a surprising idea. Academics often consider the answers that we may get in response to a code writing assessment and write a rubric that will help accommodate those expected responses when marking the students' answers. However to our knowledge the idea that solution space size may affect difficulty has not previously been tested.

3 The data set

The data for this work was gathered from a first semester Java programming course. The course was designed with the assumption that the students have no prior knowledge of computer programming. The course adopts a back to basics procedural approach (similar to that suggested by Reges (2006)) except that the learning is supported by an in-house micro-world called *Robot World* in the BlueJ IDE. Robot World was inspired by 'Karel the Robot' (Pattis 1981). For the majority of the course students do not write their own classes but instead learn to decompose their programs into methods. The advantages of using micro-worlds as a tool for teaching novice programmers are well documented. These advantages are that they:

- reduce the complexity of a language by providing a subset of a conventional language
- enable students to visualise the execution of the program, giving immediate feedback and assisting them in the debugging process (McIver and Conway, 1996)
- increase the focus on problem solving and algorithm design (Kölling, 1999).
- facilitate learning better than text-based (non-visual) systems (Dougherty, 2007).

It is for these reasons that the traditional back to basics approach was extended to include the micro world in a simplified learning IDE as the teaching environment for this course.

The eight code writing questions analysed in this study were selected from a series of summative practical programming tests held throughout the first semester of a first year Java programming course. Sixty student responses were analysed for each question. These students had given ethical consent for their data to be used and were representative of the entire cohort.

The questions analysed are provided in Appendix A. These questions were selected from a larger body of questions. These were questions which contained concepts that had been taught to the students but which were presented in a problem they had not seen before although they had seen examples that were variations (Thompson 2010) on the problem. An example scenario is provided below.

Question 5 asked the students to work out the length of two corridors and print out the length of the longer of the two corridors provided. The corridors could be of any length and may even be the same length. The students were provided with images of one possible starting scenario for the robot (Figure 1).

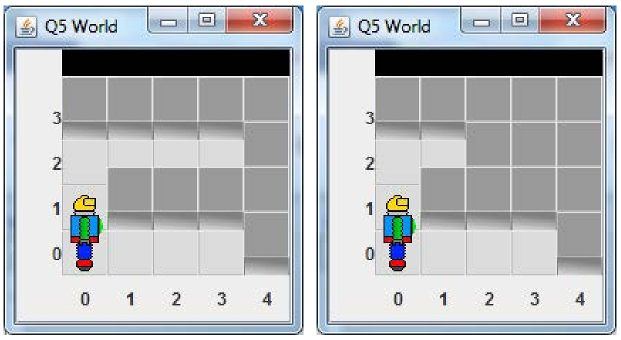

Figure 1: Question 5 the starting scenarios

The students were at a stage where they had been taught and had practiced programming code that uses the robot world methods, primitive data types, variables, mathematical operators and logical operators. In addition the following concepts relevant to this question had been taught:

- iteration –while loops only
- selection – simple if/else statements
- summation and counting algorithms

This question was given to the students in a practical programming test which followed a computer lab where the students had worked on a problem that required them to calculate the length of a single corridor. The same code had been discussed in a lecture prior to the laboratory session. In previous labs the students had been given tasks that required them to write code that compared two integers and print out the one with the highest value.

4 Determining the solution space

Solution space can be defined as the set of possible structural and syntactical permutations that provide a working solution without any discrimination of solutions due to the quality of the solution.

Two instructors developed a set of solutions to a set of first semester novice programming tasks. These sets of solutions formed the minimum solution spaces. It should be noted that each set is not necessarily the full set of all possible solutions as identifying the set of all possible solutions for a code writing task is an extremely complex problem and it becomes more problematic as the size and/or complexity of the code increases.

Even a relatively simple selection statement can generate several possible solutions. For this reason we define our solution space as at least a certain number of solutions; there may be other solutions which have not been identified.

The following discussion illuminates the way in which we have determined solution space size with an exemplar. Question 4 asks the students to write code that allows the robot to navigate through a spiral maze until they find a beeper at which point the robot should stop. Robots can only turn left. The students at this point have only learnt about while loops so the problem's solution space only consists of solutions which contain a while loop. The solutions identified by the instructors which form the problem's solution space are given in Table 2. This problem's solution space is comprised of least three candidate solutions and therefore has a size of at least three.

Solutions
`while(isGroundClearAtRobot()) { while(isSpaceInFrontOfRobotClear()){ moveRobotForwards(); } turnRobotLeft(); }`
`while (isSpaceInFrontOfRobotClear()){ moveRobotForwards(); if (isRobotFacingWall()){ turnRobotLeft(); } }`
`while (!isRobotFacingWall()) { moveRobotForwards(); while(!isItemOnGroundAtRobot()) && (isRobotFacingWall()) { turnRobotLeft(); } }`

Table 2: Solution space for Question 4

5 Results

Figure 2 shows that there is an obvious trend, for the questions we have examined, between solution space and question difficulty. The smaller the solution space the easier the students found the question.

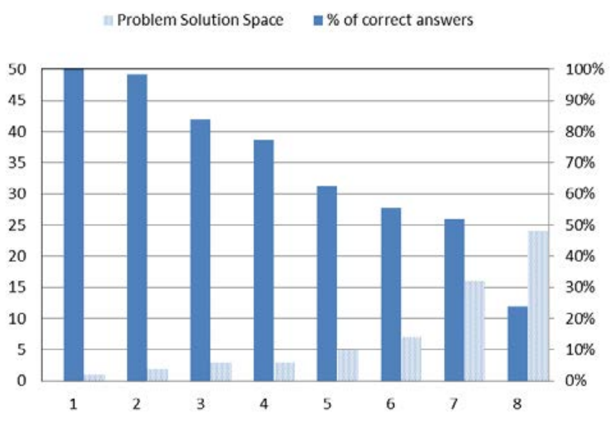

Figure 2: Solution Space Size (y axis left) and % correct answers (y axes right) by question

The questions were selected to provide a progression of programming concepts as they were delivered through the course so move from code that is a simple sequence of instructions to the robot, to selection and then to iteration. As a consequence the questions become more conceptually difficult.

Figure 3 shows the solution space size of each problem and the solution space size of the students' answers. For the last three problems the students' solution space stays relatively constant but the difficulty increases, and it increases at a rate that appears to be related to the rate of increase in the actual solution space. Difficulty maybe affected by what the students don't know. Because the students are novices presumably their knowledge is limited and therefore they are unaware of many of the possible solutions. Unlike writing in a natural language, where a substantial proportion of the students seem to benefit from the opportunities provided by a more open/larger solution space, in computing it is quite obvious that fewer students can cope with a situation where they have a big solution space. Moreover, in writing regardless of their level of writing and ability to structure their writing many students find open tasks with a larger solution space easier. In contrast in computing students tend to find it more difficult to solve programming problems that have a greater solution space.

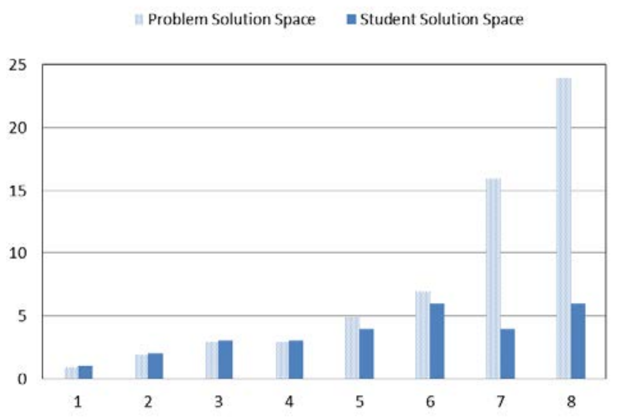

Figure 3: Size of problem vs. size of student solution space

6 Conclusions and future work

The Dreyfus Model of Skill Acquisition (Hunt 2008) suggests that novices copy solutions so if the teaching style provides patterns for solutions to a particular style of code writing problem then it is possible that the task maybe easier for the students regardless of the solution space size. Moreover, the students' available solution space is likely to be influenced by factors such as the instructor's teaching focus, previously seen code and the wording of the question itself.

For novice programmers the difficulty of a programming task tends to increase as the solution space increases. This relationship between difficulty and solution space could be used to estimate the difficulty of tasks set for students in computing labs or tests. A difficulty metric based on minimum solution space size should provide academics with a more consistent and reliable way of determining the probable difficulty of computing tasks designed for novice programmers. There is no doubt that a difficulty measure that is more accurate than the 40% agreement about difficulty levels (Simon et al. 2012) achieved using the judgement of academics familiar with the teaching of novice programmers is desirable.

In natural language metrics the measures of difficulty have usually been grouped so that the results are presented in meaningful categories such as equivalent grade levels or difficulty levels. Computing tasks for novice programmers could also be grouped into categories of difficulty to provide a quick and easy estimation of the difficulty of a task. For example, for a first semester of programming a minimum solution space size of 1-4 = easy, 5-7 = medium and > 7 hard would probably be appropriate. This of course could be adjusted for subsequent courses and or standards for a course.

One of the limitations of this preliminary work is the need to increase the clarity and repeatability of the minimum solution space size calculation. It may be that a comparison of problem characteristics to typical solutions space sizes could shed some light on useful heuristics.

7 References

Beaglehole, V.J. and Yates, G.C.R. (2010): The full stop effect: Using readability statistics with young writers. *Journal of Literacy and Technology*, **11**(4), 53-82.

Carbone, A. (2007): Principles for designing programming tasks: How task characteristics influence students learning of programming. Melbourne: Monash University.

Dougherty, J. (2007): Concept visualization in CS0 using Alice. *Journal of Computing Sciences in Colleges*, **22**(3): 145-152.

Pattis, R.E. (1981): *Karel The Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons.

Hunt, A. (2008): *Pragmatic Thinking and Learning: Refactor Your Wetware (Pragmatic Programmers).* Pragmatic Bookshelf.

Kölling, M., (1999): Teaching Object Orientation with the Blue Environment. *Journal of Object-Oriented Programming*, **12** (2): 14-23.

Luxton-Reilly, A., Denny, P., Kirk, D., Tempero, E. and Yu, S. (2013): On the Differences Between Correct Student Solutions. *Proc. of Innovation and Technology in Computer Science Education conference 2013*, ITiCSE '13, Canterbury, United Kingdom, 177-182

McIver L. and Conway, D. (1996): Seven deadly sins of introductory programming language design. *Proc.of the 1996 International Conference on Software Engineering Education and Practice*, 309–316.

Reges. S. (2006): Back to basics in CS1 and CS2. *SIGCSE Bulletin*. **38**(1): 293-297.

Rogers, C.R., Lyon, H.C., and Tausch, R. (2013): *On Becoming an Effective Teacher - Person-centered Teaching, Psychology, Philosophy, and Dialogues with Carl R. Rogers and Harold Lyon*. London, Routledge:

Simon, Sheard, J., Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., deRaddt, M., D'Souza, D., Philpott, A., Skene, J. and Warburton, G. (2012): Introductory programming: Examining the exams. *Proc. 14th Australasian Computing Education Conference*, Melbourne, Australia, **123**: 61-70.

Soloway, E. and Spohrer, J. (1989): *Studying the Novice Programmer*, Hillsdale, New Jersey, USA, Lawrence Erlbaum Associates.

Thompson, E. (2010): Using the principles of variation theory to create code writing problem sets. *Proc. of the 11th Annual Conference of The Higher Education Academy - Information and Computer Sciences*, Durham University, 11-16. http://www.ics.heacademy.ac.uk/events/download.php?file=/events/11th-annual-conf/proceedings/Proceedings_11th_Annual_Conference.pdf. Retrieved 22 August 2013.

Winslow L., (1996): Programming Pedagogy -A Psychological Overview. *SIGCSE Bulletin*, **28** (3): 17-22.

Appendix

1	For this question, the students are supplied with the method header. They are asked to complete the method body by writing *a sequence of three statements to make the robot drop the beeper it is carrying, then move the robot forward one cell and turn the robot left once.*
2	For this question, the students are supplied with the method header. They are asked to complete the method body so that *the robot turns left then if there is no wall in the way moves forward one cell.*
3	In this question, the students are provided with a robot in a cell that contains a number of beepers. The students are asked to write a method called *pickUpNBeepersCheckIfAll() that takes an integer parameter, and makes the most recently created robot pick up that number of beepers from the beeper stack at its current location. You can assume that there are enough beepers in the stack for the robot to do this safely. The method should return true if the robot has picked up all the beepers at its current location, or false if there are still beepers on the ground.*
4	Complete the method *navigateSpiral that moves the robot through a spiral maze until it reaches a beeper. The spiral will always have 6 passages but they will be varying in length.*
5	In this scenario there are interconnected two corridors, they are always connected at the same point (See Figure 1 for details). The length of each of the corridors changes randomly each time the robot world is created. A corridor number is specified by the row of the world that the corridor is in. The students are asked to: *Write a program that measures the length of both corridors, and then displays the message Corridor<m> is the longest. It is <n> long. Where:* *<m> is the number of the longest corridor.* *<n> is the length of that corridor.* *If the corridors are the same length, the message should specify corridor 0.*
6	This question asks the students to write a method called *walk() that makes the robot walk through a door to reach a beeper. The door that it must walk through could be to the east or west or straight ahead* […up…]. *A door will always be present. The robot must only pass through the location in front of the door once.*

7	In this scenario the robot starts off carrying 100 beepers, and there is also a pile of beepers at position (0, 0). *The robot should pick up those beepers and count how many there are. Then the robot should draw a square using the beepers by dropping them. The length of the sides of the square in beepers should be the number of beepers picked up from position (0, 0). For example, if the robot picks up 5 beepers then it should make a 5 by 5 square.*
8	This question asks the students to write a method called *advanceRobot()* that has two parameters a Robot and a distance to travel (the number of cells that the robot should advance). *The robot should only be able to move if it is alive and if the distance to travel is positive if it is unable to move an appropriate exception should be thrown. If the robot encounters a wall before moving the full distance it should stop rather than crashing. The method should return true only if the robot moved the full distance.*