## CONCRETE PROGRAMMING FOR PROBLEM SOLVING SKILLS

# Vladimir Estivill-Castro<sup>1</sup>

<sup>1</sup>School of ICT, Nathan Campus, Griffith University (AUSTRALIA)

#### **Abstract**

This paper describes a project that proposes a strategy to address the weaknesses in students abilities for abstract thinking and generic problem-solving skills. We suggest focusing on problem-solving skills outside textual programing by developing settings where descriptions of algorithmic solutions can be performed by physical and concrete constructions. It is a response to the decreasing performance in state-wide admission indicators of the cohorts arriving to university. This suggests a new way to teach is required. The proposal here is similar to new teaching trends around the work to teach computer science concepts and algorithms by implementing theories of constructivism, situated learning [11,16], active learning [15] and collaborative learning. Thus, we provide educational tools and "situated-leaning" activities that concur with others in the belief that these promote learning within an authentic context. We also follow the path that "kinesthetic learning" participation in such activities promotes learning by doing. However, the distinctive aspect of our approach is that the focus is on developing problem-solving skills using as a vehicle the situation and challenges of some computer science concepts. That is, the objective is not to teach computer science concepts (although some of these are developed). We also provide explicit instruction on problem-solving.

In this paper we describe the approach with one activity. The activity progresses from a concrete individual experience in an outdoors environment or open gym, to working with concrete building material and then with a virtual environment in the computer. Usage of the computer is delayed and the last stage is indeed a textual based programming language; however, the progression is for reinforcing the problem-solving skills and is their applicability to larger scales of the same problem

Keywords: Situated learning, problems solving, programming.

## 1 INTRODUCTION

Students graduating from Information and Communication Technology should be able to perform problem solving by expressing the solution algorithmically. The ability to solve problems with a degree of creativity is highlighted as an essential characteristic for both novice undergraduate engineers and qualified IT professionals in benchmarks published by the OECD [12]. Problem solving is crucial to the professional success of the graduates of the School of Information and Communication Technology at Griffith University, Australia. It is usually manifested by the ability to program a computer so that, from any set of input values that constitute a valid instance of the problem, the computer finds out the solution to such instance.

There is much creativity, analysis and design skills in developing solutions, since there is usually a large body of acceptable solutions. Problem solving for IT graduates means not only solving problems [10] but also expressing a method, an algorithm, in the language that defines the operation of an automaton [8]. This means not only figuring out the answer to a problem, but also describing the step-by-step process that a machine (with no intelligence or common sense) would be performing to obtain the answer. Problem-solving skills are more important than learning several programming languages, because the technologies to program computers vary constantly - computers are faster and with more memory, but they remain fundamentally state-transition machines. Thus, learning a particular programming language is ephemeral; describing solutions to problems algorithmically is and will continue to be an endurable skill. However, learning the problem-solving skills necessary for programming has proven to be hard, and so has teaching them [2]. It requires significant conceptual and abstract thinking, regularly associated with the skills for solving mathematical problems [19]. However, for IT students, there are additional challenges.

**First challenge:** the generation of an algorithm that works for all instances of the problem. Issues are complicated because computers are so fast, and it is now impossible to perceive with our senses all of what goes through them (let alone through now very large entities like the Internet and the Word Wide Web). A large literature on teaching problem solving and a discussion on analytical skills in physics, mathematics and engineering is presented by James E. Stice at

University of Texas (wwwcsi.unian.it/educa/problemsolving/stice\_ps.html) mostly deriving from the pioneering work of Donald R. Woods [7]. Other literature and its references cover many aspects of teaching problem-solving [1,20]. For IT, teaching and learning problem-solving has the added complexity of explaining and coding the solution in a programing language.

Second challenge: the abstraction of the description; namely, the language in which the solution is expressed is an abstraction of the changes of state that occur in abstractly described objects. The complications of programming computers have resulted in the software crisis. Producing software is extremely difficult, and usually does not meet user expectations, it is faulty and needs upgrades. The price of more powerful hardware drops while not much better software remains costly. New software developing tools are more sophisticated and powerful. But such advances require more concepts, from structured programming, to object-oriented and recently to agent programming, for example. Moreover, with each of these advancements, the teaching of problem solving competes with the need to train the IT-students in sophisticated environments as well as the need to introduce them to a large body of concepts. As a result, some students perceive fundamental concepts as unrelated and even irrelevant. However, the relevant knowledge and common thread is problem solving and techniques to obtain algorithms.

At the School of ICT in Griffith University, the teaching of problem solving for programming has become progressively more challenging as the abilities of the intake of students into the programs has continuously decreased. After the global softening of the demand for ICT degrees shortly after the dot.com slowdown, departments and schools across Australia re-adjusted intake, re-organized staff and re-structured to confront the fact that fewer students applied for admission to their programs. In Queensland, the effect of this is also that admission is awarded to students with weaker entry scores in the state-wide university Queensland Tertiary Admission System (QTAC). While this trend has been reduced in other institutions, the School of ICT in Griffith University continues to admit weaker students even in the period 2003-2008. Figure 1 shows data from QTAC (www.gtac.edu.au) of the ICT programs offered by Griffith University (codes 2\*\*\*\*\*), Queensland University of Technology (codes 4\*\*\*\*\*) and the University of Queensland (codes 7\*\*\*\*\*), which are the three universities in the South-East region serving the major urban centers of Brisbane and the Gold Coast. The plot shows OP Median entry scores. The larger the OP score, the further down the state-wide rank of the student performance for admission into the university system. Therefore, the upper trend for the School of ICT's programs reflects weaker performance by the cohort of students that were admitted and enrolled relative to the previous cohort's performance. Note that some programs, like 26422 have a major deterioration from the already weak OP entry score of 13 in the academic year 2003-2004 to the extremely low OP Score of 18 in the academic year 2007-2008.

Analysis of OP scores [9] reveals that there is strong correlation between better OP scores and better GPA outcomes in the first year of study. OP also correlates with attrition, with weaker OP scores increasing the likelihood of students failing subjects and eventually withdrawing from programs. Moreover, ICT schools typically have above average attrition rates.

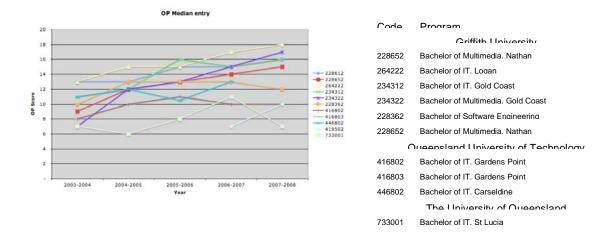


Figure 1: Weaker students are accepted and enroll at the School of ICT at Griffith University.

This poses a challenge. We are to educate much weaker students with less preparation for problem solving and enable them to perform problem solving, analytical thinking and eventually programing computers as part of their tertiary studies in preparation for their professional activity. This paper proposes that students with such weak entry scores need to be trained in algorithmic problem solving from concrete physical instruments, and gradually move into the abstract concepts of programming languages. For this, we propose to develop a series of learning activities [21] learning project at Griffith University

#### 2 PROPOSED STRATEGY

The first element of our strategy is to develop learning activities performed by students in teams. Activities¹ relate to knowledge introduced in at least 6 but possibly all 8 first-year courses of the Bachelor of IT. While such knowledge is not a prerequisite, the activities introduce it, reinforce it, or illustrate it by providing additional context. The activities integrate concepts so students can establish relationships between courses. Thus, we will produce questionnaires for students to reflect upon the relationship of material across courses, for an appreciation of the application of abstract concepts. For example, activities on graph algorithms will incorporate concepts like accumulators, counters and absolute addressing from 1007ICT Introduction to Computer Systems and Networks but also notions like graphs from 1002ICT Discrete Structures 1. Activities whose foundations are state-automata, logic gates and Turing machines will be linking with 1004ICT Foundations of Computing and Communication and 1007ICT again. Complexity of software development and problem solving will take from 1410ICT Introduction to Information Systems and 1420ICT Systems Analysis and Design. Problem solving for programming is taken from 1001ICT Programming 1 and 1005ICT Programming 2. Activities will also be monitored across vertical paths of the program.

The second element of the strategy is that activities are extra curricular and students participate in them by invitation and voluntary. We will organize events in orientation week, end of semester 1 and end of semester 2 where the students will engage in the activities within the framework of a competition (in a similar fashion to the *Science an Engineering Challenge*<sup>2</sup>) with interesting prizes. Some of the activities may become required assessment in some of the courses mentioned earlier; however, this would be if adopted by lecturers or conveners of some of the courses.

Educational theories like *Constructive Alignment* [3] suggest students should be aware that the learning outcomes are problem-solving skills and we will follow this suggestion. Therefore, the next element of the strategy is a guide on *Teaching Problem Solving* based on the theory from technical fields like engineering and mathematics [24,12] (and the Higher education Academy Engineering Subject Center on Problem Solving [www.engsc.ac.uk/er/theory/problemsolving.asp] and its corresponding supporting educational theories [12]) but adapted to information technology and problem solving for algorithm development. We will use *Constructive Alignment* [3] in order to ensure our assessment methods and our learning activities achieve the intended learning outcomes; namely improved problem-solving skills. Our intention is to evaluate under the following definition "Problem solving is the process of obtaining a satisfactory solution to a novel problem, or at least a problem which the problem solver has not seen before" [7].We may extend this to problem solving for software development and systems architecture.

The educational theory which will be the foundation of this project requires some further development. A fundamental difficulty in teaching ICT at Griffith University is the limited ability of a substantial percentage of the student cohort to think in abstract terms (we explained before the low OP scores achieved by students enrolling in the programs). This restricts the student's ability to solve problems, as well as their ability to recognize the potential of applying already known knowledge in unfamiliar new situations (which, if described in abstract terms, proves to be the same or similar to an already known problem). The planned teaching / learning methodology to be applied has some distinct characteristics:

-

<sup>&</sup>lt;sup>1</sup>Activities are independent, but thematically linked.

<sup>&</sup>lt;sup>2</sup>www.newcastle.edu.au/faculty/engineering/events/challenge

- In the activities, students are presented with concrete problems and participate in solving them.
   But the activities engage students with different levels of abstract thinking ability to successfully participate;
- through a series of increasingly difficult exercises involving guidance (cf. Social Constructivism), answering 'what if' [5,4] questions, and independent discovery, students construct and reflectively explain a generic solution, which leads to an implementation / algorithm — this approach not only ensures to test whether students managed to constructed their own understanding, but also develops their abstraction skills;
- exercises exploring the limits of the solution (through comparing the problem to other similar ones and through asking 'what if not' questions) will help the learner to develop skills to recognize concrete situations in which known abstract solutions apply.

The constructive approach and the use of teach-back [18,23], ensure that the understanding of the problem and the understanding of the solution become an observable occasion. Our design of activities (to be performed by teams of students) is around a series of puzzles for realistic problems. In these problems, the students will construct a solution, but instead of initially using a keyboard/computer (and a formal programming language as in traditional text-based programming), they will commence by constructing physical artifacts. We refer to this as concrete programming although it has similarities with puzzle-based learning [17] and with problem-based learning [6, and references]. In fact, we should be able to have students from high-school complete the activities without the need of instructing them in textual programming.

Our approach is to use concrete physical objects to build first phase solutions: here, the students will have playing cards, or other physical components (like LEGO or Konex) and they will build concrete physical solutions (but still descriptions of algorithms) to small instances of the puzzles. The students will be asked to attempt to obtain algorithms that work in general. However, these algorithms may fail in some of the many configurations. These will incorporate elements of role-play in learning. The second component of our approach is to use automated assessment: we propose to use a digital camera to capture the constructions (algorithms) of the students, and develop image processing software to interpret their physical programs; then execute them (run them) with all possible inputs and award them a score relative to the fraction of inputs in which they are correct. The third phase of our approach is programming by simulation. As with visual programming, but tackling larger problems than with the physical manipulation of 'LEGO-brick' or 'playing cards'. However, we will develop educational software that will emulate the puzzles and the concrete programming languages. They will be using simulations of cards, and tiles, and composing programs in the same way as in the physical world, but all in a Graphical User Interface (GUI). For example, RoboLab by LEGO illustrates visual programming and is used by teenagers and university students to program MindStorm Robots. The fourth stage is the transition to textual programs. To this end we will use environments oriented for the puzzle in the educational programming language MASH (developed by Dr. A. Rock). MASH is an imperative subset of the programming language Java (and by removing many of the sophistication of Java, first-year students understand every line of code they use).

The students will be provided with guidance and instruction for problem solving [13]. We may start with classical approaches. An extreme summary [19, www.math.utah.edu/pa/math/polya] may be provided first, see Fig. 2. Later, more elaborate instruction on problem solving will be provided. For example, students may be provided with more details on one of the step in Fig. 2. In general, much more sophistication on problem solving will be delivered than can be illustrated here; at each step elaborating on the techniques for problem solving and in particular, for algorithm analysis and design [22].

#### GLOBAL STEPS:

- 1. UNDERSTAND THE PROBLEM
- 2. DEVISE A PLAN
- 3. CARRY OUT THE PLAN
- 4. LOOK BACK AND REFLECT

- 1. Examine the solution obtained.
- 2. Can you check the result? Can you check the argument?
- 3. Can you derive the solution differently? Can you see it at a glance?
- 4. Can you use the result, or the method, for some other problem?

Figure 2: Basic/classical approach to problem solving.

We have plans for activities around the following topics: Sorting Networks, Logic Gates, Graph Generation, Graph Coverage and Traversal, Path Finding, Bin-Packing. While those names are technical terms that describe some of the computer science concepts behind them, we expect to have more appealing names (for example, there is already a Graph Coverage Traversal activity named *Dinosaur* that Estivill-Castro and Venema designed, and that Venema has used very successfully in open days, experience days, and visits by high-schools to Griffith University). As alluded to before, activities illustrate a wide range of useful problem solving skills: analogy, simplification, exploration, iteration, divide and conquer, and recursion. Instruction, illustrations and materials on methods like divide-and-conquer, reduction, and analogy will be formally provided.

## 3 ILLUSTRATION WITH ONE ACTIVITY

The activity we use as an example is named "sorting cows" but is based on Sorting Networks [14, Section 5.3.4] and has similarities with the Sorting Networks activity popularized by the Computer Science Unplugged (csunplugged.org/sorting-networks). In its first stage, the participants are introduced to the basic operations of sorting bridges (also named gates). The analogy are rail populated by cows that are connected by the gates. The gates may swap cows across rails, and the generic goal is to design and arrangement of the gates that achieves a certain objective, for example, placing the largest cow in the first rail. These simple operations are comparison operations and swap operations. The Blue (also named large up) operations ensures that the larger cow is in the lower numbered rail when two cows (items) meet at it. The green gate (also named larger down) always places the larger cow in the larger numbered rail. The red gate always swaps the cows independently of the size of the cows.

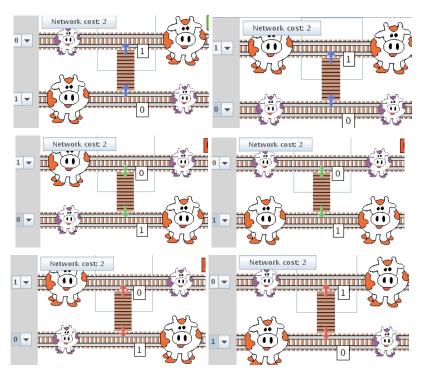


Figure 3: The basic operations.

In the first stage of the activity, the experience is completely physical. The participants walk on carpet that represent the rail and experience themselves the fact that they must wait at a gate for another participant in the connected rail, and then perform the required operation as per color (function) of the gate. They experience configurations for 3 or 4 participants and try to discover permutations which fail to reach the objective. For example, the network does not sort. They also may be asked to explore some patterns and to discuss the parameters of the problem. Instruction of problem-solving is given by analyzing first what problems may be feasible and which may be not. Variations like finding the largest and the smallest without sorting, or using only one type of gates are discussed as well as considering that the cost of bridges/gates may not be always uniform but may vary as rails are further apart. This

is what we describe as a concrete experience that is based on Piagetism and will be moved to a more informed and abstract familiarity. The idea is to strengthen the students semantic schemata (or their building processes) by associating their experience and their action with appropriate language. We expect that such language will include words like sorting, comparison, swap, cost, algorithm, permutation, and the like. It also will have more possibilities for abstractions like the notion of sorting network itself.

In the second stage of the activity, the participants still work away from computers and continue using concrete objects. In this case, they build with cloth and corresponding strings artifacts that represent the networks. They lay the rails and the bridges and experiment, evaluate and interact their concrete constructions.

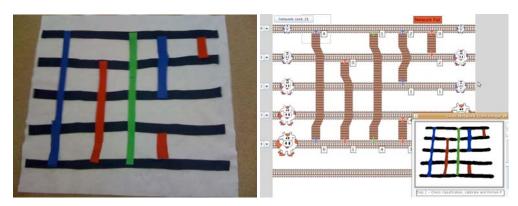


Figure 4: A sorting network constructed of cloth and capturing it into the virtual environment.

Figure 4 shows the types of networks that can be physically built with a white blanket as background, black stripes as rails, and also cloth strings as gates.

The realization that this network is a representation of computation (that is, an encoding of behavior) is achieved by our supporting tools. These networks are captured by a camera (as simple as a mobile phone camera) and fed trough an image recognition software. This image recognition software recuperates the network and can represent it in a graphical user interface that is also coupled with a simulator. The participants can see the effects of the network in any permutation they wish to test it, or on a sample test set or even more in all permutations (as long as the network has less that 9 rails). A participants network can be evaluated against many objectives. The user can chose the objective (like finding the median). They also get feedback on the cost of the network in parameters like gates used, or time used. With this, participants are converting their constructions to a graphical simulation in the computer and observing the effect on their tests.

Once this step is completed, the participants can move completely to the graphical user interface and configure far larger networks that would be possible physically, and explore far more patterns.

This is the third stage, where the experience remains concrete. However, such experience is no longer physical, but is now virtual. All the actions of the sorting network, and the rails, and the cows is now in the environment of the simulator and the graphical user interface. Large networks become impossible to build for a particular objective unless one identifies a pattern. The identification of these patterns introduces notions like subproblem. For example, from repeatedly finding the largest cow all rails by a diagonal pattern, it is possible to scale the pattern down from n rails to n1 rails. The pattern can also be seen as scaling up the basic operator from 2 to 3 rails. However, this pattern can be repeated to create a sorting network by using it repeatedly from size n, then n=1 and so on down to 2.

From here, the 4-th stage of the activity is the migration to a textual encoding of this behavior. We introduce this using a particular environment of MaSH. This environment is an API that emulates completely the GUI but enables the introduction of textual programming language concepts. We start with the level named *statements* in MaSH and can produce a direct mapping between the visual representation of sorting networks in the GUI and MaSH-statement program.

This enables the introduction of concept like identifiers, separators, and most of the lexical instruments experienced in textual programming languages.

Figure 5: The textual programming environment for a single gate.

There are several levels in this stage of the activity that correspond to the levels in the design of MaSH but which can be re-enforced with the activity. The repetition of the comparator each time in the next rail for all rails results directly into a control structure.

Figure 6: MaSH at the level of control structures.

The pattern we described earlier of using the solution to find the largest as a subproblem to sorting, enables the introduction of the notions of method (or subroutine).

```
import sortingNetwork;
void select (int place) {
    for (int i=0; i<place; i=i+1)
        blueLargerUp(i,i+1);
}
void sort (int place) {
    for (int i=place; i>0; i=i-1)
        select(place);
}
void main() {
    createRails(5); // create five rails with data in random order printOrder(); // print the data
    sort(4); //Sort
    printOrder(); // print the data
```

Figure 7: MaSH at the level of subroutines/methodsstructures.

The activity can now progress more rapidly into other programming concepts and in particular, MaSH is designed to introduce behavior first and later introduce data, and finally introduce the plethora of object-oriented concepts.

We envisage also a later stages of these activities, that can lead to artificial intelligence (like to search for cost optimal networks for specific parameters of the problem), or even to research challenges, like to establish the best network for certain cost structures and sets of operators. The activity also leads itself to consider concurrency and parallelsim and the introduction of programming concepts in this area.

#### 4 ACKNOWLEDGEMENTS

Prof. V. Estivill-Castro thanks the advise in educational theory matters by Prof.B. Bartlett and also by Assoc. Prof. P. Bernus. Estivill-Castro also acknowledges several contributions. Dr. A. Rock has developed MaSH. Dr. D.Chen is the first-year advisor at the Nathan campus and also a convener of Programming 2 (a first year course). Other programs and first-year course conveners like D. Billington, D. Tuffley, R. Hexel, and S. Venema involvement is acknowledged as they have provided suggestions to integrate and use concepts in their courses in the activities. Other course conveners are also ensuring that problem solving is taught in their subjects, among them R. Hexel, and R. Topor who are involved in observing and using the approach in courses in later years.

## **REFERENCES**

- [1] J. Adams, S. Kaczmarczyk, P. Picton, and P Demian. Improving problem solving and encouraging creativity in engineering undergraduates. In *International Conference on Engineering Education ICEE-07*, Coimbra, Portugal, 2007.
- [2] J.P. Adams and S. Turner. Problem solving and creativity for undergraduate engineers: process or product? *Innovation, Good Practice and Research in Engineering Education*, page 61, 2008.
- [3] J. Biggs. Teaching for Quality Learning at University. Shire and Open University Press, UK, 1999.
- [4] S. Brown and M. I. Walter. What if not? an elaboration and second illustration. *Mathematical Teaching*, 51:9–17, Spring 1970.
- [5] S. Brown and M. I. Walter. *The art of problem posing*. Lawrence Earlbaum, Hillsdale, NJ, second edition, 1990.
- [6] C.E. Cindy E. Hmelo-Silver, R. G. Duncan, and C. A. Chinn. Scaffolding and achievement in problem-based and inquiry learning: A response to Kirschner, Sweller, and Clark (2006). *Educational Psychologist*, 4(2):99–107, 2007.
- [7] Woods R. D. Teaching problem-solving skills. *Engineering Education*, 66(3):238–243, December 1975.
- [8] N. Dale and C. Weems. *Programming and Problem Solving with C++*. Jones and Bartlett Publishers, Inc., USA, 3rd edition, 2002.
- [9] J. Day and Y. Dlugosz. Targeting university marketing activities using student demographic and performance data. In *Australasian Association for Institutional Research*, Rockhampton, Queensland, 2001.
- [10] J. M Dewar. Increasing maths majors' success and confidence through problem solving and writing. In F. Rosamond and L. Copes, editors, *The Influences of Steven I, Brown*, pages 117–121, Bloomington, Indiana, 2006. Educational Transformations, AuthorHouse.
- [11] J. J Holdsworth and S. M. Lui. Gps-enabled mobiles for learning shortest paths: a pilot study. In *FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 86–90, New York, NY, USA, 2009. ACM.
- [12] W. Houghton. Learning and Teaching Theory for Engineering Academics. Engineering Subject Centre, 2004.
- [13] M.D. Jones. *The Thinker's Toolkit: 14 Powerful Techniques for Problem Solving.* Three Rivers Press, USA, 1998.

- [14] D.E. Knuth. *The Art of Computer Programming, Vol.3: Sorting and Searching.* Addison-Wesley Publishing Co., Reading, MA, 1973.
- [15] J. Lave and E. Wenger. Situated Learning. Cambridge University Press, UK, 1991.
- [16] D. Lindquist, T. Denning, M. Kelly, R. Malani, W. G. Griswold, and B. Simon. Exploring the potential of mobile phones for active learning in the classroom. SIGCSE Bull., 39(1):384–388, 2007.
- [17] Z. Michalewicz and M. Michalewicz. Puzzle-Based Learning An Introduction to Critical Thinking, Mathematics, and Problem Solving. Hybrid Publishers Pty Ltd, Victoria, Australia, 2008.
- [18] G. Pask. Conversation, cognition and learning. Elsevier, New York, 1975.
- [19] G Polya. How to Solve It: A New Aspect of Mathematical Method. Princeton University Press, second edition, 1957.
- [20] M. Prince and B. Hoyt. Helping students make the transition from novice to expert problem-solvers. In 32 Annual Frontiers in education (FIE-02), volume 3, pages F2A7-11, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
- [21] P. A. G. Sivilotti and S. M. Pike. The suitability of kinesthetic learning activities for teaching distributed algorithms. *SIGCSE Bull.*, 39(1):362–366, 2007.
- [22] S. S. Skiena. The Algorithm Design Manual. Springer-Verlag, London, second edition, 2008.
- [23] L. S. Vygotsky. *Mind and society: The development of higher psychological processes.* Harvard University Press, Cambridge, MA, 1978.
- [24] W.A. Wickelgren. How to Solve Mathematical Problems. Dover, New York, 1995.