Intelligent vs. Unintelligent Programming Systems for Novices

Haider Ramadhan

Computer Science and Artificial Intelligence,
School of Cognitive and Computing Sciences (COGS),
University of Sussex, Brighton, BN1 9QH, UK.
e-mail: haiderr@cogs.susx.ac.uk Phone: (0273) 606755

Permenant Address:
Math and Computing Dept., College of Science,
Sultan Qaboos University,
PO Box 32500 Al-Khod, Sultanate of Oman.
Phone: (968) 515551
Fax: (968) 513254

Abstract
Intelligent programming debugging and tutoring systems normally emphasize the problem solving process only and neglect the issues that deal with supporting visualization and graphics based domains and environments. Unintelligent (microworld based) programming systems, on the other hand, normally emphasize issues that deal with environments and ignore the issue of explicitly guiding novice programmers during problem solving through some sort of intelligent and automatic program diagnosis. This paper claims that to facilitate novices’ programming and problem-solving, we need to explore some alternative approaches to the design of systems for novice programming. The approach presented in this paper concentrates on integrating intelligent and unintelligent approaches to come up with an environment that helps novices develop not only problem-solving skill but also an accurate conceptual and mental model of the programming process.

1 Introduction
Most existing intelligent programming tutors, including program debugging systems, concentrate exclusively on the problem solving process (developing novices programming skill) and ignore the fact that a robust mental model of program execution and of machine behavior is an important prerequisite for problem solving. These systems neglect the issue of providing a visible and graphic environment through which novices can dynamically visualize program and machine behavior, and thus build correct programming knowledge. Some examples of these systems include Proust [10], Talus [15], Aurac [8], Laura [1], Spade [14] and the Lisp Tutor [2]. As a consequence, novices are required to mentally simulate the execution of the program which they are writing and imagine its dynamic behavior and side-effects: a task they normally fail to accomplish.

To avoid these shortcomings, several other intelligent programming tutors attempted to incorporate some of domain visualization features and tools that allow novices to understand the dynamic behavior of both programs and the notional machine. Some examples of these systems include Bridge [4], Gil [23] and BIP [3]. However, these systems also ignore the importance of providing a separate pre-problem solving, exploratory, visible and open-ended environment that would help novices to build “bug” free programming knowledge.

Unintelligent (microworld based) programming systems, on the other hand, concentrate exclusively on helping novices to build correct programming knowledge through visible, opportunistic and opened-ended environments. Systems like BALSA [5], Tinker [11], LOGO [18] and SmallStar [7] are classic examples of these environments. These systems neglect the issue of supporting those aspects of intelligent coaching through which novices can transform their program knowledge into programming skill.

These two approaches of intelligent and unintelligent programming support have already been demonstrated in individual systems. There is, in principle, no reason why they should not be combined to provide a single powerful exploratory and intelligent programming environment. It is true that there has not yet been much clear success for intelligent programming tutoring and debugging systems, except perhaps for the Lisp Tutor and Proust, and that in
2 An Overview of DISCOVER

DISCOVER synthesizes features of Microworlds with features of Intelligent Tutoring Systems (ITS). In terms of a microworld, the system provides novices with a visible, opportunistic, exploratory and open-ended programming environment. The environment enables novices to explore, observe and discover the dynamic behavior of both individual programming concepts and whole programs as well as of the underlying notional machine, and thus build the underlying conceptual knowledge associated with these concepts and a mental model of program execution and machine behavior. This is accomplished during what is called a free phase. In terms of an ITS, the system supports the immediacy of intelligent tutoring and the capability of automatic program debugging to automatically analyze novices solutions for semantic errors during an intelligent guided discovery programming phase and provides them with intelligent feedback that guides them in the problem solving process. This is accomplished during a guided phase.

Figure 1 shows a snapshot of DISCOVER’s user interface which appears to a user as a collection of eight windows. The four windows on the left side of the interface, namely the memory space, the input space, the output space and the algorithm space, represent the components of a concrete model of the underlying computer system (the notional machine) discussed in section 3.

The programming language of DISCOVER is a pseudo-code based, algorithm-like language. The language is a very simple and straightforward one, which reduces the number of abstract programming ideas and concepts by omitting procedures and recursion; avoids having too many programming tricks to be learned; and avoids requiring the learning of low-level syntax details. At present, the language has no provision for functions, procedures, recursion and complicated data structures such as records, arrays and lists, thus focusing the novice’s attention on basic programming concepts and simplifying the learning process. Programming concepts supported include CREATE, READ IN, WRITE OUT, PUT IN, IF-ISTRUE-ISFALSE and WHILE-ENDWHILE.

2.1 Unintelligent Programming

As mentioned above, the main purpose of this unintelligent free discovery programming phase is to allow novice programmers to build an accurate mental model of a program execution and the machine behavior. This includes understanding the behavior of such dynamic programming concepts as variable declaration, the value-binding process, input and output operations, conditional statements and looping constructs.

In this free phase, the novice is presented with the interface shown in figure 1. The selections in the Programming Concepts Menu are the beginning of phrases. Each phrase corresponds to one programming concept. Upon selecting a concept, its corresponding name is inserted into the Work Window and all the novice has to do is complete it by typing in its parameter part (e.g. the names of the memory cells to be created). Once a concept has been selected, the Template Window displays the general syntactic form of that concept (the template) along with information on each component of the template, the Example Window displays some examples of the use of that concept and the Message Window displays tutorial-like instructions describing how to type in the expected parameter parts for building the selected concept. Any syntactically incorrect parameter parts provided by the novice are trapped by the Syntax-directed Editor and reported in the Message Window.

Once a concept is completed it appears in the Algo-
rithm Space Window, where the code so far explored and entered is stored. At each step, DISCOVER neatly formats the code for the novice. Immediately after the concept moves across to the Algorithm Space Window, DISCOVER visually executes it and instantly shows the changes that take place in the memory space, input space or output space windows. For example, executing a 'create' concept causes a cell from the array of cells in the Memory Space Window to be named. Any assignment, either via the 'put' or 'read in' concept, to a memory cell shows up immediately in that cell. Input and output, via 'read in' and 'write out' concepts, occur in the Input Space Window and Output Space Window. Any newly selected concepts are appended at the bottom of the code displayed in the Algorithm Space Window.

At any point, the novice can re-execute the entire code stored in the Algorithm Space Window either in a line-by-line stepped-execution under his own control or in slow-motion under DISCOVER's control. Visual execution of the program is supported by highlighting the line currently being executed. Figure 1 shows the screen after the novice has selected and completed a condition concept. The same figure also shows how a completed condition concept is moved across to the Algorithm Space Window and visually executed by DISCOVER in stepped-execution mode.

At any time during this free phase the novice may add, delete or modify statements in the Algorithm Space Window. After the novice finishes such editing, DISCOVER re-executes the entire resultant code all over again. This is done to rebuild the state of the environment of the computation for the edited code. The novice may also load a pre-stored program in the Algorithm Space Window and see how it is executed by DISCOVER.

2.2 Intelligent Programming

The main goal of this phase is to help novices transform their programming knowledge, expected to be gained during the first phase, into programming skill. Figure 4 presents a snapshot of DISCOVER's intelligent guided discovery programming phase. As shown, the Template Window has changed to the Problem Description Window and the Example Window to the Example Solutions Window. Based on the novice's past experience and performance in this guided phase, DISCOVER selects the next problem from the problems library and presents a description of its specifications in the Problem Description Window. At present, the next problem to be solved is selected by DISCOVER's domain expert, like BIP, in a predetermined order (e.g., specified by the designer). Relevant example problem and its solutions are presented in the Example Solutions Window.

The novice is to build his solution to the current problem by properly putting together programming concepts, explored and discovered in the first phase. As before, concepts can be selected by making choices from the Programming Concepts Menu. Upon selecting a concept, its corresponding name is inserted into the Work Window and the novice is expected to type in its parameter parts. Templates of a concept and examples of its use are not provided during this guided phase.

Once a concept is completed and accepted by the Syntax Directed Editor, it is passed to the intelligent component of DISCOVER (the domain expert) for automatic debugging and analysis. In doing so, DISCOVER attempts to model the steps taken by the novice in evaluating his actions and responses. DISCOVER analyzes the surface code of the completed concept (partial solution code) without much specific knowledge about the problem to be solved or about how to design and construct an algorithm (i.e. DISCOVER cannot solve the problem itself).

Much like Laura, Talus, Aurac and Bridge, DISCOVER relies on a pre-stored reference solution (the ideal student model) for a given problem and applies various heuristics and pattern matching techniques to match the solution code provided by the novice with the reference solution to spot errors and misconceptions. Unlike these systems, however, DISCOVER is capable of interactively analyzing partial solution code and providing immediate, but flexible, feedback on both success and failure. This flexibility in the feedback is achieved by increasing the grain size of

Figure 2: Immediate feedback from DISCOVER.
automatic diagnosis to a full program statement, instead of a single token or word. By doing that, DISCOVER explicitly guides the novice in the process of putting together programming concepts to solve the given problem. A complementary paper [22] describes and discusses how DISCOVER represents its knowledge of the reference solution; how it conducts the automatic matching of the novice’s solution to the reference solution; and how it handles the problem of nondeterminism in novices solutions.

Unlike the Lisp Tutor that analyzes single symbols, DISCOVER monitors novices’ actions on a complete concept-by-concept basis. As long as each concept represents a correct goal on a solution path, DISCOVER continues guiding the novice towards the final goal, reasoning about the goals already satisfied and hinting at the goals still remain to be satisfied. Figure 2 gives an example of an immediate feedback supported by DISCOVER during this guided phase.

3 Principle Design Decisions
Several design issues and decisions have influenced the development of DISCOVER. The three most important of these issues are: the provision of domain visualization, the support of example-based learning and the incorporation of intelligent debugging and tutoring. A full discussion on these design considerations can be found in [19,21].

3.1 Domain Visualization
DISCOVER does not view the issue of visualization as simply a question of animating program execution (program visualization [16]) nor of displaying different values of variables during execution, but of supporting a high-level, integrated, conceptual and concrete environment that allows novices to relate problem-solving with the dynamic behavior of both language and machine. To achieve this, DISCOVER integrates visibility [6], program visualization [16], and a concrete, dynamic model of the underlying computer system with which the novice user is interacting [13] (du Boulay [6] calls this concrete model of the machine the ‘notional machine’).

The system (1) through visibility, helps novices visualize hidden and internal changes in some conceptual parts of the underlying computer system such as memory space. This includes, for example, visualizing how variables are named, how they get their values, and how the corresponding memory cells in the memory space are affected; (2) through program visualization (highlighting the line currently being executed), helps novices visualize how their programs statements are executed. This provides a one-to-one mapping between what they see and what the interpreter is doing, and thus enables them through this ‘What You See Is What Happens’ feature to see how control flows from one line to another and how their programs dynamically behave; and (3) through a concrete model of the underlying computer system (dynamical notional machine), helps novices conceptualize the properties of the machine they are interacting with and thus, can build an accurate model of its behavior.

3.2 Example-based Learning
DISCOVER develops the novice’s programming capabilities as an example-based learner by providing him with relevant example problems and solutions to help him in tackling his own programming problems. A number of studies have shown the usefulness of providing novices with a goal structure (relevant example) [9,12,17]. When presenting a novice with a description of a problem during the intelligent guided discovery programming phase, DISCOVER allows him to look at several example cases and solutions to different but similar problems. Example problems are designed to have close mappings onto actual programming problems. The novice should be able to use the example solution as a model for his own solution by transforming the whole or a part of the example solution into his own solution, replacing and modifying only those individual elements of the example solution that do not satisfy the new requirements. During the free phase, these cases are examples of different uses of a concept (rather than a whole problem) selected for exploration.

3.3 Intelligent Tutoring
DISCOVER develops the novice’s programming skill by providing him with an intelligent guided discovery programming phase. In this phase, the novice composes and relates different programming concepts and language constructs to form complete algorithms for given problems. The system monitors the novice’s actions as he moves along the solution path, automatically analyzes solutions for semantic errors and misconceptions, and offers intelligent feedback whenever he deviates from a correct solution path.

To be able to monitor novices’ progress in putting together programming concepts and language constructs together and to decide when to interrupt and what to say, DISCOVER supports active, automatic debugging of partial solutions as they are provided by the novice. Novices are only expected to have partial pro-
gramming knowledge of how programming concepts and language constructs work, how they affect the underlying notional machine and how the machine executes and treats whole programs. This knowledge is expected to be gained during the unintelligent free discovery programming phase. It is the task of DISCOVER, through this intelligent guided discovery programming phase, to help novices transform their programming knowledge into programming skill.

4 Instructional Advantages

DISCOVER's current architecture makes it possible to provide novices with a great flexibility in their choice of an instructional strategy when learning to program. Particular strategies they might pursue include:

- **Opportunistic, exploratory and free discovery programming.** Novices can construct programming concepts, explore and observe their dynamic behavior (through interactive and visual simulation), see their side effects on the notional machine and detect any misconceptions they might have in their programming knowledge. Novices can thus create their own hypotheses, explore their own actions and draw their own conclusions. The system hence permits an open-ended programming by discovery strategy.

- **Intelligent and guided discovery programming.** The system presents novices with a sequence of problem-solving situations that attempt to develop their programming skill. In solving programming problems, via intelligent tutoring, novices attempt to transform their programming knowledge into programming skill. The emphasis is on having novices solve problems on their own. When novices show misconceptions, the system provides them first with a reasoning feedback that attempts to make them induce the correct step for themselves. After novices fail to infer the correct step for several times, only then, the system intervenes and spells out the expected action.

- **Learning by examples.** The system allows novices to become example-based learners by providing them with relevant example problems and solutions. Novices can see how similar problems are solved and can use these examples as models for their own solutions.

It is hoped that by allowing these multiple learning strategies, DISCOVER will play a valuable role in helping novice programmers to develop both programming knowledge and programming skill.

5 Conclusion

This paper describes DISCOVER, an intelligent discovery programming system that synthesizes features of both ITS and Microworlds. DISCOVER supports novices in an initial unintelligent free discovery programming phase and in a subsequent intelligent guided discovery programming phase. In the initial phase, novices explore, observe and discover the dynamic behavior of individual programming concepts and whole programs as well as of the notional machine to build the underlying conceptual programming knowledge. In the subsequent phase, novices compose together programming concepts and language constructs, observed and discovered in the initial phase, to solve given problems under explicit intelligent guidance of system domain expert to transform their programming knowledge into programming skill. The integration of domain visualization and example-based learning with the immediacy of intelligent tutoring and automatic debugging are expected to provide DISCOVER with a potential to teach novices basic computer programming in a dynamic and conceptually rich way.

Acknowledgement

I would like to thank Ben du Boulay, my supervisor, for insightful suggestions regarding this research, and for comments on an earlier draft of this paper. DISCOVER is fully implemented in POP-11 using the POPLOG programming environment.

References


