An Intelligent Discovery Programming System

Haider Ramadhan
Computer Science and Artificial Intelligence
School of Cognitive and Computing Sciences,
University of Sussex,
Brighton, BN1 9QH, UK.

e-mail: haiderr@cogs.susx.ac.uk

Abstract

This paper introduces and describes an intelligent discovery programming system called DISCOVER. The system synthesizes features of Human Computer Interface (HCI) with features of an Intelligent Tutoring System (ITS). In terms of HCI, the system is capable of providing novices with an open-ended, exploratory, and free discovery programming environment (microworld) that enables them to explore, observe and discover the dynamic behavior of both individual elementary 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. In terms of an ITS, the system is capable of automatically analyzing and debugging novices' partial solutions for semantic errors during a guided discovery programming phase and provides them with intelligent feedback that guides them in the problem solving process.

Introduction

Programming is a ubiquitous and cognitively demanding task. Novice programmers have difficulties in learning to program [2,7,14,16,33,39]. Some computer scientists even say that computer programming is just too hard and too mathematical for novice programmers [2].

There are two main reasons behind these difficulties: lack of programming knowledge and lack of programming experience. Programming knowledge deals with the exact syntax and semantics of the programming language constructs being learned. This includes understanding the dynamic behavior of programming concepts such as variable declaration and binding, input and output operations, conditional statements, looping constructs and other more abstract concepts such as recursion. Programming experience deals with the skill required to connect the low-level syntax and semantics of constructs to produce properly integrated higher-level plans (programs and algorithms) [7,11]. In other words, programming experience deals with the ability to put individual programming concepts together to come up with a complete solution for a given problem. This requires acquiring a mental model of how the computer executes the program, so that the reasoning through this execution can become possible.

This paper reports on the DISCOVER system, a step towards an intelligent discovery programming environment. The system helps novices to acquire both programming knowledge and programming skill. This is accomplished in two phases:

- In the first phase, the free discovery programming phase, the system helps novices explore, observe and discover the dynamic behavior of individual programming concepts as well as of whole programs to build the underlying conceptual programming knowledge needed for problem solving tasks without requiring them to mentally simulate the intricate behavior of the machine.
- In the second phase, the guided discovery programming phase, novices compose and coordinate programming concepts and language constructs, observed and discovered in the first phase, to solve given problems under the intelligent guidance of DISCOVER, and thus transform their programming knowledge into programming skill.

Normally, conventional programming tutors and systems intended for novice programmers, such as Proust [21], Aurac [18], Talus [27], Bip [5], Laura [1], Spade [25] and the Lisp Tutor [3], combine these two phases into one phase, i.e. they attempt to help novices develop their programming knowledge and their programming skill simultaneously. In addition, these tutors and systems ignore the issue of providing novices with an environment to explore, observe and discover how each programming concept and language construct functionally and conceptually behaves and how the computer executes whole programs. Instead, these tutors and systems typically present novices with a static environment that requires them to imagine the program's execution and its dynamic behavior rather than view it on the screen. Consequently, even after using these systems, we would expect that novices still tend to exhibit some of the classical misconceptions that have been reported found in their underlying conceptual programming knowledge. These include using conditionals instead of iterations, misunderstanding the nature of variables and the value-binding process, difficulties with input and output operations and an inability to comprehend the flow of control [2,7,14,16,33,39].

Learning how to compose and combine programming concepts and language constructs to form higher level plans (programs) is a very important task in learning to program. However, understanding the semantics and the behavior of these concepts and constructs is not less important. Consequently, before requiring novices to put programming concepts, functions and constructs together to solve a given problem, we should help them first understand their dynamic behavior. Novices should be exposed to a problem-solving process only after building the correct underlying conceptual programming knowledge and a robust mental model of language execution and machine behavior. A system that achieves these goals goes beyond today's generation of programming simulators, tutors and environments. In fact, such a system is an 'intelligent discovery programming system', and hence comes the name DISCOVER to embody this philosophy.

The next section highlights the main design issues and decisions that influenced the development of DISCOVER. Section 3 presents an overview of the current DISCOVER implementation. Section 4 summarizes the main points of this paper and discusses the future plans for DISCOVER.

Main Design Issues

An Integrated System Image

A discovery programming system encourages a novice programmer to become an active learner by allowing him to form his own hypotheses, explore his own questions, and draw his own conclusions. The system develops the novice's programming knowledge as an opportunistic learner by providing him with an exploration-based, free discovery programming environment in which he can explore programming concepts, discover their dynamic behavior through observation, detect any misconceptions associated with them, and hence build the necessary underlying conceptual programming knowledge. To achieve these goals, a discovery programming system needs to support visibility [13], program visualization [28], and a concrete model of the underlying computer system with which the novice is interacting [24]. du Boulay calls this concrete model of the machine the 'notional machine', while Norman [30] calls it the 'system image'.

Mayer [24], Norman [30], Jones [22] and Moran [26] report that novices learn the concepts of a programming language more effectively and more easily if they are presented with a concrete model of the underlying computing machine. Similar results have been reported by Olson [31] who suggests that the main difficulty novices confront when learning to program is the assimilation of a model of the computing machine. These findings imply that the clarification of the notional machine facilitates the task of learning to program for novices. However, Jones also reports that providing a static concrete notional machine on its own is not enough and that even when presented with a notional machine, novices tend to build inaccurate models of program execution and still have difficulties in comprehending the flow of control in such dynamic concepts as looping constructs, conditional statements and procedure calls. Therefore, a discovery programming system not only needs to present novices with a notional machine but also needs to support program visualization and visibility to allow them visualize how their programs dynamically behave and how hidden and internal changes in the notional machine take place, and thus build a robust model of program execution and machine behavior.

What is needed is a a discovery programming system that, through visibility, enables novices to see crucial internal changes in otherwise hidden parts of the underlying computer system such as the memory space. This includes seeing how variables are named, how they get their values, and how the corresponding memory cells in the memory space are affected; a system that, through program visualization, enables novices to visualize how their programs' components are executed in the same order and format that they wrote them. This provides a one-to-one mapping between what they see and what the system 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, when the binding of values take place, and how their programs behave dynamically; a system that, through the provision of a concrete model of the underlying computer system, enables novices to conceptualize the properties of the machine they are interacting with and understand how different elements of the language work, and thus build a mental model of the execution for the programming language being learned.

Most existing programming tutors and systems present novice programmers with a static view of the program's execution and its dynamic behavior, ignoring the issue of providing a visible, graphic and concrete base on which novices may build their underlying conceptual programming knowledge and programming skill. Some examples of these systems include Proust [21], Talus [27], Aurac [18], Laura [1], Spade [25] and the Lisp Tutor [3]. 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.

Several programming systems have attempted to incorporate some of these design features in their implementation. However, these systems support only one or two of these features and lack an integrated image of the notional machine. Therefore, they fall short of providing a true discovery programming system or environment which integrates visibility, program visualization and a concrete model of the computing machine. Bip [5], an ITS for BASIC, through showing pointers which move around the program code as it is executed and changes in the values of variables, supports program visualization and simple visibility. Similar features were also provided by programming systems for FORTRAN [38], for PASCAL [29], and for assembly languages [37,38]. Bridge [8] also supports program visualization by highlighting program lines during execution and showing how different programming plans are connected in a graphic and diagrammatic fashion. By supporting the visual execution of whole programs, it becomes possible for these systems to help novices build a mental model of the whole program's execution. However, they lack a truly interactive environment that allows novices not only to visualize the dynamic behavior of whole programs but also to visualize the behavior of individual programming concepts and language constructs. In addition, these systems lack an integrated image of the notional machine that has been reported to help novices in conceptualizing the properties of the machine and in relating the dynamic behavior of individual programming concepts to these properties. By integrated image, I mean the incorporation of visibility and program visualization within a concrete model of the underlying computing machine.

Visual Programming systems, such as BALSA [10] and Tinker [23], also attempt to show novices the dynamic behavior of the whole programs and algorithms. However, these systems emphasize the representation of programs in graphical terms, sometimes in more than two dimensions. There is no empirical evidence yet to suggest that visual programming is inherently more effective than conventional linear and text-oriented programming for teaching novices how to program. In fact, as Green [17] claims, instead of focusing on visual programming and creating new programming notations, one can continue with the existing notation but use enhanced typography to make perceptual cues reflect the notational structure.

Case-based Reasoning

It has been advocated since before Socrates' time that people reason about the situations they find themselves in by referring to similar situations that they have experienced, heard or seen. Therefore, a discovery programming system should develop the novice's programming capabilities as a case-based learner by providing him with relevant cases (examples) to help him in tackling his own programming problems. This is different from learning-by-analogy which focuses on issues of analogical transfer: connecting the new material to be learned with the knowledge that already exists in memory [3,6,9,19,22]. When presenting a novice with a description of a problem during the guided discovery programming phase, a discovery programming system should also allow him to look at several example cases or solutions to different but similar problems. These example cases should be designed to have a close mapping onto the current problem. 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.

Intelligent Coaching

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

Many of the automatic program debugging systems, including Proust, Talus, Aurac, Bip, and Laura, cannot debug partial code segments and wait until the entire program code is completed before attempting any debugging analysis. As a result, these systems lack any rich interaction with novices during program construction and require them to possess a high level of both programming knowledge and programming skill. Automatic program debuggers embodied and incorporated in a discovery programming system should be capable of debugging partial solutions as they are provided by novices. This feature is mandatory for a discovery system to be able to monitor novices' progress in putting programming concepts and language constructs together and decide when to interrupt and what to say. Novices should only be expected to have partial programming 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. In DISCOVER, this knowledge is expected to be gained during the free discovery programming phase. It is the task of the discovery programming system, through its guided discovery programming phase, to help novices transform their programming knowledge into programming skill.

Several programming systems and tutors support the debugging of partial solutions, among which are the Lisp Tutor, Bridge, and Gil [36]. Unlike Bridge, which requires the novice to specifically request the automatic analysis of his program (passive-like mode), a discovery programming system, like the Lisp Tutor and Gil, should support active, automatic debugging for it to be able to monitor each and every step that novices may take while moving on a solution path, determine when novices show evidence of misconceptions and decide when to interrupt and what to say. This requires support for possible immediate feedback on both failure and success. However, immediate feedback should not spoil the spirit of discovery learning, and should not impose on novices the rigidity found in the early version of the Lisp Tutor, for example. This can be achieved by supporting a more flexible style of tutorial interaction that:

1. Increases the grain size of automatic debugging to a complete expression or statement, not just a single symbol,

2. Permits the user to enter the code in any order, not just left-to-right, and

3. Allows the user to backtrack and delete previously entered code.

By supporting these features, a discovery programming system combines the virtues of both discovery learning and immediate feedback. A further discussion on these principle design issues can be found in Ramadhan [35].

A new implementation of the Lisp Tutor, called a student-controlled tutor [4], attempts to ease the rigidity found in the classic version of the tutor. In addition to providing novices with the three features mentioned above, the new tutor also enables them to control the timing of the feedback. However, these features are supported only in the program editing mode during which the tutor has no interaction with the novice. This transition from tutor-controlled interaction to student-controlled interaction makes the new tutor, like Bridge, a passive system that waits for the novice to request automatic analysis of his code, and thus loses the rich interaction with him. In addition, when the novice asks for automatic debugging of his code, the tutor checks over the code in the same sequence as the original version: top-down, left-to-right. Feedback is given on the first error found and the rest of the code is just ignored. Although the new implementation is an improvement, it takes away a very important feature from the tutor: the ability to monitor the novice's progress on the solution path, determine when he shows evidence of misconceptions within their proper and immediate context and decide when to guide him and what to say during interactive tutoring.

An Overview of DISCOVER

Prior to describing DISCOVER's environment itself, it is worthwhile to describe a model of the discovery learning framework embodied in it. The model is shown in figure 1. The student is a novice programmer with little or no programming knowledge or experience. The environment is the domain that the novice programmer is trying to explore and discover, plus the user interface that enables him to interact smoothly with the environment. In this case,

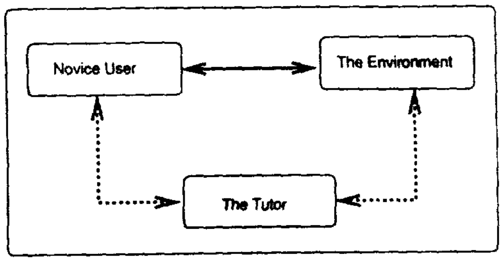


Figure 1: A model for a discovery environment

the environment comprises a concept-based, pseudo-code programming language translator, a visual and dynamic model of the underlying computing machine and the user interface.

The teacher embodies the intelligent aspects of the environment. The teacher in a guided discovery phase, presents the novice with a description of the problem specification and monitors the interaction between the student and the environment and decides when to intervene in this interaction and what to say. During the free discovery programming phase, the links between the teacher (the domain expert) and the student are disconnected. In this mode, the student (novice programmer) explores programming concepts and language constructs, discovers and observes their dynamic behavior, detects any misconceptions, and thus build his underlying conceptual programming knowledge in a 'glass-box', open-ended and learning-by-doing environment. It is hoped that by augmenting the free discovery programming phase with the guided discovery programming phase, it will become possible to transform a novice's programming knowledge into profound and efficient programming experience - properly rooted in his own actions and hypotheses.

The Programming Language

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. The following are the concepts and constructs of the DISCOVER's programming language, which can be selected from a pull-down menu.

1. CREATE is the declaration concept that declares and names memory cells in the memory space. For example:

   CREATE num1 num2 average

   declares memory cells (variables) num1, num2 and average.

2. PUT is the assignment concept that performs the storing and the assignment functions. For example:

   PUT 2*num1 IN num3

   multiplies the contents of memory cell 'num1' by 2 and stores the result in memory cell 'num2'.

3. WRITE OUT is the output concept that displays and prints values in the output space. For example:

   WRITE OUT (num1+num2)*2

   displays the result of the formula in the output space.

4. READ IN is the input concept that performs the function of reading a value from the user and binding it to some memory cell. For example:

   READ IN num2

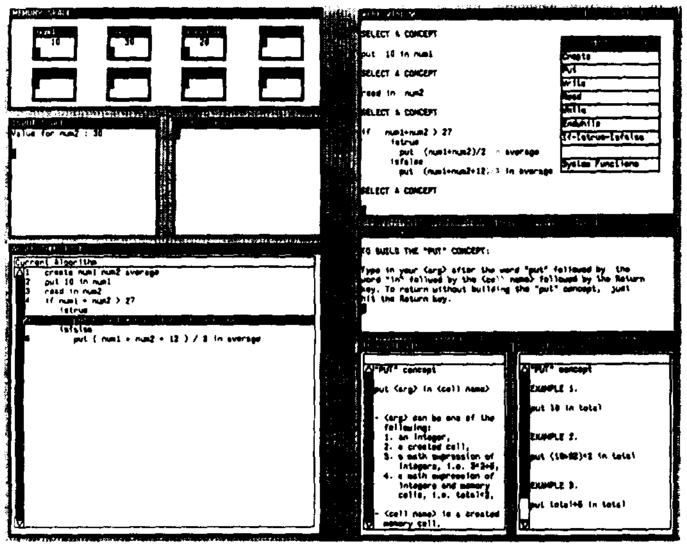
   prompts the user to enter a value in the input space and then stores that value in memory cell 'num2'.

5. WHILE - ENDWHILE is the loop and iteration concept. If the condition in the WHILE statement is true, then subsequently entered statements will be executed and the control will flow back to the WHILE statement upon the selection of the ENDWHILE phrase. Otherwise, subsequently entered statements will not be executed. The normal execution will resume only after the ENDWHILE phrase is selected to close the loop. For example, in the following code each statement in the body of the loop will be executed as it is entered.

```
PUT 10 IN num1
WHILE num1 > 5
        PUT num-1 IN num1
        WRITE OUT num1
ENDWHILE
```

6. IF-ISTRUE-ISFALSE is the logical condition concept that performs logical comparison and testing. If the condition is true, the system will execute the ISTRUE statement. Otherwise, the ISFALSE statement will be executed. There is no IF-ISTRUE statement without an ISFALSE case. This will force the user to think about what should be done if the condition is not satisfied (false). If the user does not want to specify any alternative action, than he has to explicitly select the function 'DoNothing'. The example below illustrates this point.

```
PUT (numl+num2)/2 IN average
If average < 5
  ISTRUE
            PUT average+2 IN average
  ISFALSE
            DoNothing
```



```
SELECT A CONCEPT

put  10 in num1

SELECT A CONCEPT

read in  num2

SELECT A CONCEPT

if   num1+num2 ) 27
  istrue
    put  (num1+num2)/2  in average
  isfalse
    put  (num1+num2+12)/3 in average

SELECT A CONCEPT
```

```
Value for num2 : 30
```

```
Current Algorithm
1     create num1 num2 average
2     put 10 in num1
3     read in num2
4     if num1 + num2 > 27
         istrue
         isfalse
6           put ( num1 + num2 + 12 ) / 3 in average
```

TO BUILD THE "PUT" CONCEPT:

Type in your <arg> after the word "put" followed by the word "in" followed by the <cell name> followed by the Return key. To return without building the "put" concept, just hit the Return key.

```
"PUT" concept

put <arg> in <cell name>

- <arg> can be one of the
  following:
  1. an integer,
  2. a created cell,
  3. a math expression of
     integers, i.e. 3*2+5,
  4. a math expression of
     integers and memory
     cells, i.e. total*2,

- <cell name> is a created
  memory cell.
```

```
"PUT" concept

EXAMPLE 1.

put 10 in total

EXAMPLE 2.

put (10+20)*2 in total

EXAMPLE 3.

put total+5 in total
```

Figure 2: DISCOVER's user interface

The User Interface

DISCOVER views the interface as a conversational dialog [20] between the novice programmer and the environment, focusing on issues that occur within the semantic, syntactic and lexical levels of the dialog. The novice programmer moves in a predictable manner from one point of the dialog to the next, using request-response interactions and selections from pull-down menus.

The interface of the DISCOVER system was designed to facilitate the comprehension of large quantities and kinds of information and activity options that normally characterize discovery systems. Since the task of learning to program in a discovery system involves exploring individual programming concepts, understanding their dynamic behavior, putting them together to solve problems and utilizing relevant cases and example solutions, it is important to isolate these various components of programming, when designing the interface, so that their acquisition and comprehension can be made possible and efficient.

Figure 2 shows a snapshot of DISCOVER's user interface which appears to a user as a collection of eight windows. The functions of these windows are as follows:

1. *Work Window.* Is a dialog window in which the novice builds an individual programming concept after selecting it from the *Programming Concepts menu.*

2. *Message Window.* Displays error and tutorial-like messages during the free discovery programming phase, and intelligent feedback on both failure and success along with hints during the guided discovery programming phase.

3. *Template Window.* Displays all the information a novice may need to build individual concepts correctly according to the syntax of DISCOVER's language. During the guided phase, this window displays a description of the problem to be solved by a novice.

4. *Example Window.* Displays short examples of correctly built programming concepts selected for exploration during the free discovery phase. During the guided phase, this window displays relevant example cases and solutions to problems selected by DISCOVER.

5. *Memory Space Window.* Displays memory cells of the underlying computing machine.

6. *Input Space Window.* Displays a message requiring novices to enter a value whenever a READ IN concept is executed. The value is entered and shown in this window and not in the *Work Window.*

7. *Output Space Window.* Displays the result of executing a WRITE OUT concept.

8. *Algorithm Space Window.* Displays a sequence of programming concepts that the novice correctly builds in the *Work Window,* and shows him how it behaves dynamically through visual execution. This window also displays complete and pre-stored example programs that can be

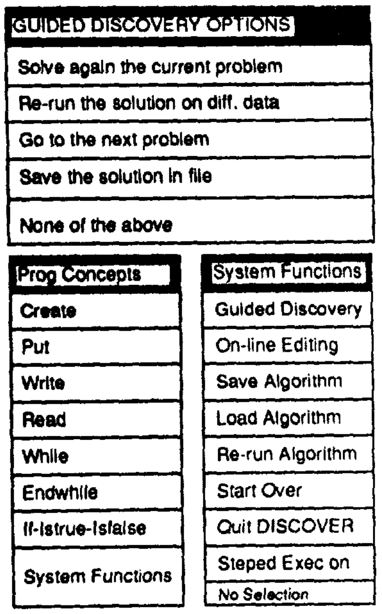


Figure 3: DISCOVER's main menus

loaded by the novice to build, through visual execution, a mental model of the whole program's execution. During on-line editing of the program, this window is used to allow the novice to carry out the three available editing options: addition, deletion and modification.

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 computing machine mentioned in section 2. In addition to these windows, there are three main menus (shown in figure 3): *Programming Concepts Menu*, *System Functions Menu* and *Guided Discovery Options Menu*.

The *Programming Concepts Menu* shows the six programming concepts of the DISCOVER's programming language described in section 3.1. The *System Functions Menu* shows the nine system options a novice can select to enter the guided discovery programming phase; edit, save or re-run the program code displayed in the *Algorithm Space*; load a prestored program in the *Algorithm Space* and visualize its dynamic behavior; switch between stepped and slow-motion visual executions; exit the DISCOVER system; or just continue with the free exploration without making any selection from this menu. The *Guided Discovery Options Menu* shows the five options a novice can select to solve the current problem all over again, re-run the solution for the current problem (provided by the novice) on different data values, move on to the next problem, save the solution for the current problem on a disk, or just skip these options and go back to do free discovery programming.

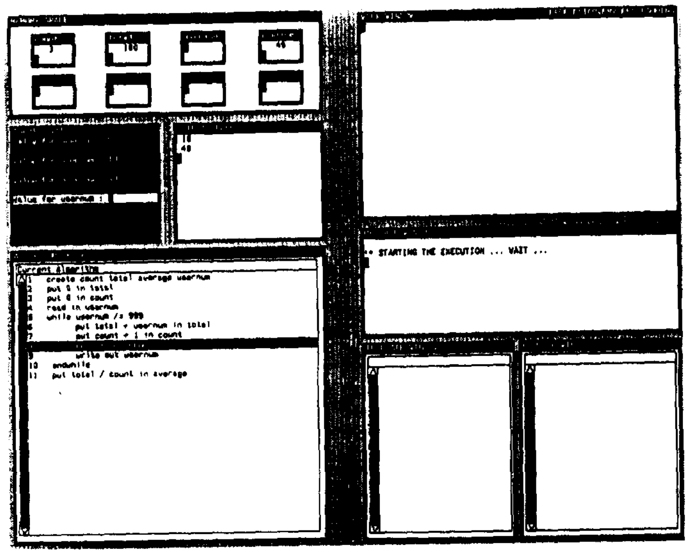
In the next two sections, two 'tours' through DISCOVER from an end-user's perspective will illustrate how these various components of the interface interact and function, the first for the free discovery programming phase and the second for the guided discovery programming phase.

Phase I: Building Programming Knowledge

As mentioned before, the main purpose of free discovery programming phase is to allow novice programmers build an accurate mental model of a program's execution and the machine's 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.

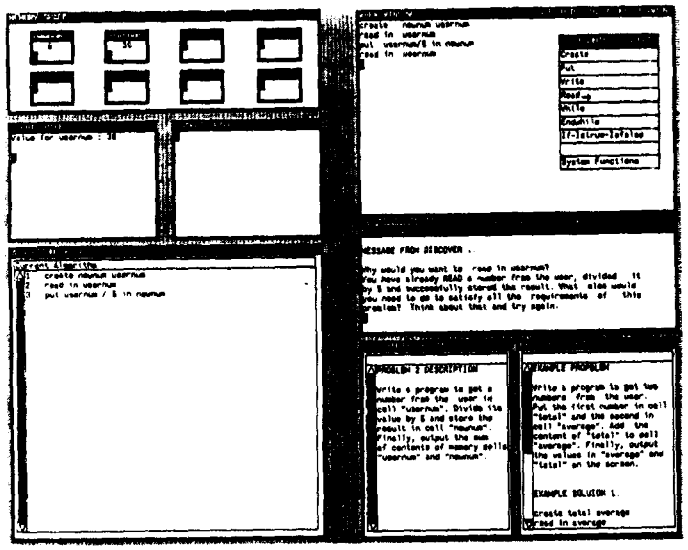
In this free phase, the novice is presented with the interface shown in figure 2. The selections in the *Programming Concepts Menu* are the beginning of phrases. Each phrase corresponds to one programming concept. Upon selecting a concept, its corresponding name is inserted into the *Work Window* and 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 *Algorithm 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



```
1   create count total average usernum
2   put 0 in total
3   put 0 in count
4   read in usernum
5   while usernum /= 999
6         put total + usernum in total
7         put count + 1 in count
8         [illegible]
9         write out usernum
10   endwhile
11   put total / count in average
```

Figure 4: The 'Ending Value Averaging Problem'



```
1   create newnum usernum
2   read in usernum
3   put usernum / 5 in newnum
```

```
create   newnum usernum
read in  usernum
put  usernum/5 in newnum
read in  usernum
```

Why would you want to read in usernum?
You have already READ a number from the user, divided it by 5 and successfully stored the result. What else would you need to do to satisfy all the requirements of this problem? Think about that and try again.

Write a program to get a number from the user in cell "usernum". Divide its value by 5 and store the result in cell "newnum". Finally, output the sum of contents of memory cells "usernum" and "newnum".

Write a program to get two numbers from the user. Put the first number in cell "total" and the second in cell "average". Add the content of "total" to cell "average". Finally, output the values in "average" and "total" on the screen.

```
create total average
read in average
```

Figure 5: DISCOVER's guided programming phase

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.*

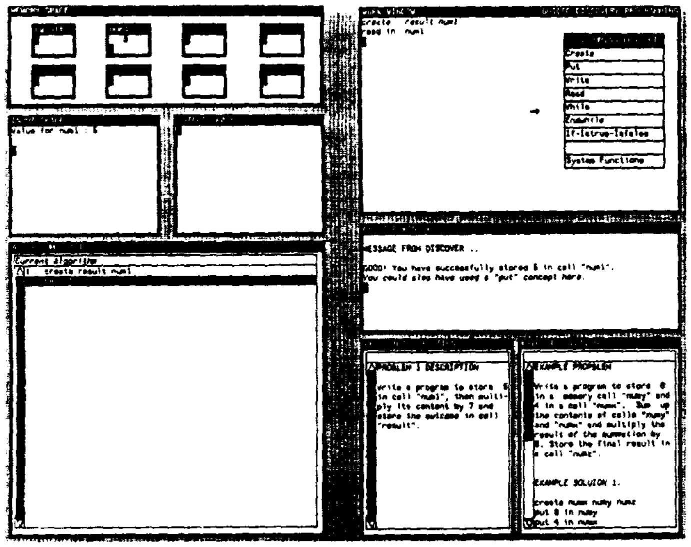
At any point, the novice can re-execute the entire code stored in the *Algorithm Space Window* either in a line-by-line stepped-execution under his own control or in slow-motion under DISCOVER's control. Visual execution of the program is supported by highlighting the line currently being executed. Figure 2 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 slow-motion 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 on-line editing, DISCOVER re-executes the entire resultant code. This is done to rebuild the state or the environment of the computation for the edited code. Any newly selected concepts will be appended at the bottom of this code in the *Algorithm Space Window.* The novice may also load a pre-stored program in the *Algorithm Space Window* and see how it is executed by DISCOVER. Figure 4 shows the 'Ending Value Averaging Problem' being executed after has been loaded by the novice. The figure also shows how the *input space window* gets fully highlighted in inverse video when a READ IN concept is executed.

Phase II: Building Programming Skill

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 5 presents a snapshot of DISCOVER's 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. The relevant case and its example 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 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



```
create   result num1
read in  num1
```

Value for num1 : 5

Current algorithm

```
1   create result num1
```

MESSAGE FROM DISCOVER ..

GOOD! You have successfully stored 5 in cell "num1".
You could also have used a "put" concept here.

PROBLEM 1 DESCRIPTION

Write a program to store 5 in cell "num1", then multiply its content by 7 and store the outcome in cell "result".

EXAMPLE PROBLEM

Write a program to store 8 in a memory cell "numy" and 4 in a cell "numx". Sum up the contents of cells "numy" and "numx" and multiply the result of the summation by 6. Store the final result in a cell "numz".

EXAMPLE SOLUTION 1.

```
create numx numy numz
put 8 in numy
put 4 in numx
```

Figure 6: Deriving a mapping between concepts

the novice by 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 prestored 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. By doing that, DISCOVER explicitly guides the novice in the process of putting together programming concepts to solve the given problem. A complementary paper for Ramadhan [forthcoming] 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.

DISCOVER monitors the novice's actions, not on a symbol-by-symbol basis, but 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 that still remain to be satisfied. Figure 5 gives an example of reasoning supported by DISCOVER during this guided phase. Unlike the Lisp tutor, DISCOVER provides feedback not only on failure but also on success. Figure 6 shows an example of feedback provided by DISCOVER upon a successful movement along the solution path. This figure also shows how, in this particular case, the system provides a mapping between programming concepts that have the same underlying semantic interpretation (e.g. between PUT and READ IN concepts).

Conclusions

This paper discusses a framework for designing and developing a discovery programming system. It is argued that a truly versatile discovery programming system must be able to allow novices to acquire both programming knowledge and programming skill. A prototype of such a system, called DISCOVER, has been developed. DISCOVER supports novices in an initial free discovery programming phase and in a subsequent guided discovery programming phase. In the initial phase, novices explore, observe and discover the dynamic behavior of individual programming concepts as well as of whole programs to build underlying conceptual programming knowledge. In the subsequent phase, novices compose and coordinate programming concepts and language constructs, explored, observed and discovered in the initial phase, to solve programming problems under the explicit and intelligent guidance of DISCOVER in order to transform their programming knowledge into programming skill.

The integration of visibility and program visualization within a concrete model of the underlying notional machine, coupled with the case-based reasoning and the immediacy of intelligent tutoring provide DISCOVER with the potential to teach novices basic computer programming in a dynamic and conceptually rich way. We are currently conducting well-designed and rigorous experiments to evaluate the effectiveness of DISCOVER. These experiments will concentrate on testing the effectiveness of DISCOVER's visual system image and validating the proposed approach of augmenting a free discovery programming phase with that of a guided discovery programming phase.

Acknowledgement

I am deeply grateful to Ben du Boulay for many helpful discussions, suggestions and stimulating intellectual challenges to do with my research, and also for critically reading earlier versions of this paper. I also would like to thank Derek Sleeman and Mike Sharples for their useful comments. This research is a part of an on-going PhD study supported by a full scholarship grant from the Sultan Qaboos University, Oman. DISCOVER is fully implemented in POP-11 using the POPLOG programming environment developed at the University of Sussex.

References

[1] Adam, A. and Laurent, J. LAURA, A System to Debug Student Programs. *Artificial Intelligence* 15 (15): 75-122, 1980.

[2] Anderson, J. R. Acquisition of Cognitive Skill. *Psychological Review* 89, 369-406.

[3] Anderson, J. R., Boyle, C. F. and Reiser, B. J. Intelligent Tutoring Systems. *Science* 228, 456-462.

[4] Anderson, J., Boyle, C., Corbet, A. and Lewis, M. Cognitive Modeling and Intelligent Tutoring. *Journal of Artificial Intelligence* 42, 7-49.

[5] Barr, A., Beard, M., and Atkinson, R. The Computer as a Tutorial Laboratory. *International Journal of Man-Machine Studies* 8, 567-596.

[6] Bayman, P. and Mayer, R. Instructional manipulation of user's mental models for electronic calculators. *International Journal of Man-Machine Studies*, 20, 189-199.

[7] Bonar, J. G. *Personal Programming in BASIC.* Academic Press, USA.

[8] Bonar, J. G. Intelligent Tutoring with Intermeidiate Representations. *Proceedings of the First Conference on Intelligent Tutoring Systems, ITS-88*, Montreal, Canada.

[9] Bonar, J. and Soloway, E. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Human-Computer Studies in Mathematics*, 20, 293-316.

[10] Brown, M. *Algorithm Animations.* Ph.D. thesis, Brown University, USA, 1986.

[11] Chi, M. T. H., Glaser, R. and Rees, E. *Expertise in Problem Solving.* In Strenberg, R. (editor), Advances in the Psychology of Human Intelligence. Lawrence Erlbaum and Associates, Hillsdale, New Jersey.

[12] Dijkstra, E. W. How do we Tell Truths that Might Hurt? *SIGPLAN Notices.* 17(5): 13-15, May.

[13] du Boulay, J.B.H., O'Shea, T. and Monk, J. The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Man-Machine Studies*, 14, 237-249.

[14] du Boulay, J.B.H. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2, 57-63.

[15] du Boulay, J.B.H., Taylor, J. *Computers, Cognition and Development.* J. Wiley and Sons, 1987.

[16] Eisenstadt, M., Rajan, T. and Keane, M. *Novice Programming Environments.* Ablex Publishing, Brighton, UK.

[17] Green, T. Programming Languages as Information Structures. In Hoc, Green, Samurcay and Gilmore (Eds.), *Psychology of Programming*, Academic Press, London.

[18] Hasemer, T. *An Empirically-Based Debugging System for Novice Programmers.* Ph.D. thesis, The Open University, UK, 1983.

[19] Hoc, J. Analysis of beginner's problem solving strategies in programming. In Green, Payne and Van der Veer (Eds.), *The Psychology of Computer Use*, London: Academic Press.

[20] Hutchins, E., Hollan, J. and Norman, D. Direct Manipulation Interfaces. In Norman and Draper (Eds.), *User Centered System Design*, Hillsdale, NJ, USA: Erlbaum.

[21] Johnson, W. *Intention-Based Diagnosis of Errors in Novice Programmers.* Ph.D. thesis, Yale University, USA, 1985.

[22] Jones, A. How Novices Learn to Program. *Proceedings of the First IFIP Conference on Human Computer-Interaction*, INERACT-84, London, UK.

[23] Lieberman, H. Seeing what your programs are doing. *International Journal of Man-Machine Studies*, 19, 253-271.

[24] Mayer, R. E. The Psychology of How Novices Learn Computer Programming. *Computing Surveys* 13, 121-141.

[25] Miller, J., Kehler, T., Michaels, P. and Murray, W. *Intelligent Tutoring for Programming Tasks.* Technical Report, Texas Instruments, 1982.

[26] Moran, T. and Card, S. Applying cognitive psychology to computer systems: A graduate seminar. In Moran, T. (Ed.), *Eight Short Papers in User Psychology*, Palo Alto, CA, USA: Xerox.

[27] Murray, W. *Automatic Program Debugging for Intelligent Tutoring Systems.* Ph.D. thesis, Texas University, Austin, USA, 1986.

[28] Myers, B. A. The State of the Art in Visual Programming and Program Visualization. *Carnegie Mellon University Technical Report*, Computer Science Department, Carnegie Mellon University, USA.

[29] Nievergelt, J. XS0: A Self Explanatory School Computer. *SIGCE Bull* 10, 66-69

[30] Norman, D. Some Observations on Mental Models. In Gentner and Stevens (Eds.), *Mental Models,* Hillsdale, NJ, USA: Erlbaum.

[31] Olson, G. and Gugerty, L. Comprehension differences in debugging by skilled and novice programmers. In Soloway and Iyengar (Eds.), *Empirical Studies of Programmers*, Norwood, NJ, USA: Ablex.

[32] Papert, S. *Mindstorms: children, computers and powerful ideas.* New York: Basic Books.

[33] Pea, R. D. Language-Independent Conceptual 'Bugs' in Novice Programming. *Journal of Educational Computing Reseaech* 2, 25-36.

[34] Rajan, T. *Novice Programming Environments.* Ablex Publishing, Brighton, UK.

[35] Ramadhan, H. A Discovery Programming System. *Cognitive Science Research Reports,* Sussex University, Brighton, UK.

[36] Reiser, B., Kimberg, D., Lovett, M. and Ranney, M. Knowledge Representation and Explanation in GIL, an Intelligent Tutor for Programming. *Cognitive Science Laboratory Reprot*, Princeton University, NJ, USA

[37] Schweppe, E. J. Dynamic Instructional Models of Computer Organization and Programming Languages. *SIGCE Bull* 5, 26-31.

[38] Shapiro, S. C., and Witner, D. P. Interactive Visual Simulations for Beginning Programming Students. *SIGCE Bull* 6, 11-14.

[39] Soloway, E. and Ehrlich, K. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering. Special Issue: Reusability*, Sept.