Modern Introductory Computer Science

Peter B. Henderson*

Department of Computer Science
SUNY at Stony Brook
Stony Brook, New York 11794

ABSTRACT

There have been numerous testimonies to the inadequacies of our educational system [83]. For undergraduate computer science educators, major concerns regarding student preparation include poor problem solving and critical thinking skills, weak mathematics background, an inability to convey thoughts and concepts, and a lack of motivation. These problems can be addressed in the introductory computer science course by developing an integrated approach to effectively teaching discrete mathematical foundations, fundamental computer science concepts, and problem-solving skills. This paper is conceptual in nature and introduces some specific examples of possible approaches to overcoming these deficiencies and problems.

1. By Way of Introduction

Computer science is a discipline in evolution. Unlike the other sciences, its intrinsic structure is not yet well defined. This makes curriculum development both challenging and frustrating. In the context of the whole undergraduate computer science curriculum I believe that the introductory course should both define the discipline of computer science for students and lay solid foundations upon which students can build in subsequent courses. There are currently two primary schools of thought concerning this first course: focus on programming and fundamental computer science concepts, or introduce students to preliminary discrete mathematics concepts.

Over the past five years I have been experimenting with alternative approaches [HeF84, HeF86, Hen86, HeF]. The approaches I have used include: introducing software engineering principles (specifications, design, implementation, testing, etc.), concentrating on program analysis (reading), emphasizing algorithmic problem solving, and simultaneously incorporating programming, fundamental computer science and discrete mathematics concepts. This paper is based upon the lessons I have learned from teaching the introductory computer science course at SUNY Stony Brook; it addresses relevant aspects of materials development (course material and courseware) to support introductory computer science instruction and presents an overview of the Algorithm Discovery Project[1].

2. In Search of Ourselves

To understand the role of an introductory computer science course we must examine its objectives with respect to the whole undergraduate curriculum. For any computer science curriculum to be meaningful, it must be based upon some well defined objective (which unfortunately may be the least understood aspect of computer science education today). The general objective of undergraduate education is to mature students, and provide them with the knowledge and confidence necessary to succeed in life. Computer science undergraduate education prepares them for graduate school, or software development[2], maintenance, or sales careers. To achieve this, students must learn and understand the fundamental concepts of the discipline, be provided with software development experience, and be exposed to related disciplines such as mathematics, engineering, technology, etc.

To gain a better appreciation for our discipline, we should understand current trends in computer science education and relate these to other established disciplines. Although computer science is referred to as a science, in many ways it is more like an engineering discipline since it is primarily concerned with the engineering of software systems. Indeed, computer science is currently taught in a manner similar to other science disciplines (physics, biology, chemistry, etc.). That is, there are a number of lower level courses taught in the first few years (starting with the introductory course - Physics I, Introduction to Computer Science, etc.). In contrast, engineering disciplines first establish solid foundations in related areas (mathematics, physics,

*This work was supported by Department of Education FIPSE Grant G00840481 and by National Science Foundation Grant MDR-8470250.

[1]Currently funded by the National Science Foundation and a Department of Education FIPSE grant.

[2]The term *software*, rather than *program* will be used throughout to indicate an expanded scope of activities specifications, design, documentation, maintenance, user concerns, etc.

chemistry, etc.). Here students are not heavily exposed to the discipline itself until much later (typically three to four semesters). The introduction of prerequisite discrete mathematics courses into computer science curricula might represent a trend in this direction.

There are strengths and weaknesses in both approaches. Early exposure gives students a "feel" for the discipline so they can make career decisions, and it helps to motivate students. However, it frequently suffers from a horse and cart problem since solid foundations may be necessary to effectively study the discipline. Computer science is a prime example. If we teach software development (programming) too early, students have not yet acquired an understanding of, and appreciation for, the fundamental mathematical principles. That is, they can't effectively apply the relevant mathematical concepts and abstract representations required to develop good software. Nor do they understand the basic principles of the design process. Of course it is possible to develop good software design skills through programming alone. The claim is that too much emphasis is placed on programming and not enough on foundations. For example, consider the distinction between neighborhood children building a tree house, analogous to teaching programming without fundamentals, and the design and construction of a modern dwelling, like a house or apartment building (building a complex software system). The latter has to be both functional and maintainable over several lifetimes (like a software system). Most building architects may have been motivated by treehouses, but they understand that they need certain other fundamentals to construct buildings.

Before proceeding to the primary point of this paper, several other relevant aspects should be discussed. These include problem solving skills, the importance and relevance of mathematics, and the role of design in computer science education.

Although we can't always accurately define what is meant by problem-solving, many of us recognize problem-solving skills as a major deficiency of our undergraduate students. With the proper emphasis, starting from the first computer science course, our students' general problem-solving skills can be significantly increased. All too frequently we equate programming skills with general problem-solving skills; however, learning programming alone does not develop general problem-solving/thinking skills [MDV86] nor does it teach the required mathematical foundations.

In a typical curriculum, mathematics and computer science are taught independently. Students take some math courses (calculus, discrete math, etc.) and some computer science courses (programming, assembly language, data structures, compiler, etc.). The link is weakly established at best. Recently there has been an impetus to introduce discrete mathematics early in the curriculum. Many of the introductory discrete mathematics text books developed for computer science curricula tend to be very mathematically oriented, and hence squelch motivation. They also fail to appropriately integrate and relate fundamental computer science material.

Design oriented problem-solving is a significant oversight in many computer science curricula. Currently, students learn (software) design principles primarily as a side effect of their course related programming projects. Very little guidance is provided, and there is no continuous and consistent treatment of design principles in the curriculum.

3. The Best of All Worlds

Excluding minor deficiencies, most computer scienc programs meet the needs of the students. The previous section provided an overview of current trends and potentia weak points. These will be used as a forum for our discussion of introductory computer science courses in the contex of the whole curriculum.

In the "ideal" computer science curriculum student should learn fundamental computer science concept based upon mathematical foundations, be able to effectivel use these concepts and mathematical tools in the analysi and development of software, appreciate the intrinsic relationships between mathematics and computer science significantly improve general and specific problem-solvin skills, understand and be able to apply design principle understand the software development process and be able t carry it out for a significant project, gain experience wit several software systems and associated tools, appreciate th needs of the software user, be able to communicate effectivel etc. How can the overall curriculum best meet these needs?

Two approaches, the science oriented and engineering oriented paradigms, were presented earlier. For th computer science discipline a more balanced approach whic effectively integrates the three key topic areas, mathematics problem-solving, and software engineering into the compute science curriculum in a consistent and coherent way as possible. One step toward achieving this objective is to restructure the lower level computer science courses, and in particular the introductory course. The latter is the primary focu of this paper.

4. An Overview

As noted in the first section, the introductory cours should both define the discipline of computer science for students and lay solid foundations upon which students ca build in subsequent courses. This is currently achieve through courses which concentrate on computer science concepts and programming, or courses which provide an overview of the discipline. The former usually use one of th numerously available Pascal programming text books, and th latter a survey text [Bro85, GoL82].

At SUNY Stony Brook we have developed, and are currently teaching, an introductory computer science course which supports the integration of topic areas fundamental to computer science. These topic areas include: general problem solving skills, discrete mathematics, algorithmic problem solving, basic computer science concepts, and an overview of the computer science discipline. The course stresses the importance of students' understanding and appreciating the relationships between computer science and mathematics, and learning to use mathematics as an effective problem-solving tool for computer science. In addition, the foundations for creative design skills are established.

A course description and preliminary syllabus for this introductory course are presented in Appendix A. This outline of topics blandly illustrates only one dimension of the the course. The key aspects are the ways in which these concepts are threaded together to form a cohesive, integrated course, and the development of appropriate supporting material. We address each of these features in the following sections.

Please note that this course does not include computer programming in the traditional sense. It does however, stress algorithmic problem-solving and design related issues.

Because of this, as well as time constraints for a one semester course, and from past experience, software engineering is not covered. To understand and appreciate software engineering concepts, students should be exposed to something tangible, like programs. This can be achieved in the subsequent courses, but was deemed too premature for this introductory course.

5. Problem Solving

The main theme of the course is to pull together four important pedagogical areas: problem-solving skills development, learning fundamental discrete mathematics concepts, using these concepts to acquire a better understanding of computer science, and applying these concepts as tools for solving computer based problems. For the latter, the popular problem-solving courses alone will not suffice since students frequently experience difficulty transferring such ideas to other disciplines. Several excellent text books [Pol73, Rub86, Wic74] are devoted to approaches to problem-solving; however, math and computer science text books do not introduce or discuss these approaches. With the wealth of "problems" in discrete mathematics and computer science it seems natural to discuss different problem-solving techniques as problems are presented. For example, one problem-solving approach is to work backwards from the desired goal towards the given initial state to derive a solution. A similar approach is used in discrete mathematics to solve certain problems, or as a way of understanding computer programs by working from the output statements backwards toward the input statements. Hence, general problem-solving techniques, as well as domain specific techniques (e.g., algorithmic problem-solving methods) are addressed.

One of the most interesting aspects of computer science studies is the relationship between fundamental computer science concepts and problem-solving. In many ways concepts like top-down and bottom-up development, tree and graph structures, state spaces, algorithmic processes, and production rules, are intrinsic to problem-solving. Such ideas give students new and powerful representations which they can apply to everyday problem-solving activities. It is important that students see these relationships so they can become better problem solvers, not just in computer science but in other courses and life in general. Hence we should not just teach computer science as a closed discipline, but relate it the real world and everyday activities. For this reason I strive to use real examples, rather than contrived ones. It is much easier for a student to grasp the concept of a graph when it represents some physical concept, be it an ancestor relationships, countries, or a chemical compound.

6. Discrete Mathematics

The foundations of our discipline are rooted in mathematical studies which predate the earliest electronic computers. Many ideas and concepts, including new programming paradigms find their basis in mathematics. For these and other reasons students should learn fundamental mathematical principles, be able to apply these principles to computer science, and appreciate the relationship between mathematics and computer science.

For example, we can introduce students to various paradigms of programming through their mathematical foundations. The functional programming paradigm combines functions, possibly in a recursive way, to construct or define new functions (LISP and ML are representative programming languages). These concepts can be motivated from a study of functions and functional composition in discrete mathematics. Hence, presentation of mathematical functions and their composition in a discrete mathematics setting naturally leads to a discussion of applicative and functional programming. This does not imply that we have to introduce students to LISP, but we can use a simplified pseudo functional language to illustrate these concepts.

Likewise, mathematical logic constitutes the foundations of logic programming, with Prolog being a representative example. Here the notions of mathematical relations (starting with binary relations) and logic are used to motivate a basic understanding of declarative programming. The illustration below defines a new relation, the ancestor relation, in terms of an existing relation, the parent relation.

person 1 is an ancestor of *person 2* is defined as

(*person 1* is a parent of *person 2*)

OR

((*person 1* is a parent of *person 3*) **AND**

(*person 3* is an ancestor of *person 2*))

Here a simple English description is used to formulate an understandable definition utilizing four mathematical concepts: binary relations, logic, transitive closure, and recursion. Again, the necessary mathematical foundations are introduced first, providing students with the critical link between mathematics and computer science. The potentially complex syntax and semantics of Prolog need not be understood for students to appreciate the expressive power of this class of languages.

This example also serves to reinforce the important concept of a tree (an ancestor tree), provide an explaination of backtracking strategies, and to introduce relational databases, where queries of the form "Eve is an ancestor of Joe" or "Adam is an ancestor of *person*" may be expressed and answered. The latter query yields all the descendents of Adam. With proper prodding students quickly discover that certain relationships, such as father, aunt, niece, and grandmother cannot be expressed since only one "base binary relation," the parent relation is provided. An additional binary relation, 'sex of' is required. With this new relation, students can express such relationships, and hence may formulate a simple model for a relational database system with several base relations. All of this being derived from fundamental mathematical principles.

Note that exercises based upon such ideas introduce unique opportunities for problem-solving which spans several different concepts. Traditionally in an introductory course students are taught a concept and then asked to solve problems applying that concept. Such myopic {limited domain} problem-solving activities fail to teach students how to integrate related concepts. Students become small problem-solvers and encounter difficulties with more realistic large problems. Understanding and applying the relationships between concepts is an important precursor to design oriented problem-solving.

7. Mathematics and Algorithms

Computer science educators recognize the importance of using mathematics as a tool for computer scientists. Establishing this link is significantly harder at the introductory level where students lack both mathematical maturity and a basic understanding of the discipline itself. For example, consider the relationships between mathematical logic and algorithms. We can informally teach students about algorithms, give them a seemingly meaningful definition, and expect them to develop simple algorithms. However, without mathematics students can't truly understand algorithms [Gri81, Hoa69]. In introductory courses we can teach students the rudiments of logic. How can we establish the connection between logic and algorithms?

The key is to use logic to motivate the understanding and development of algorithms. This may seem difficult since logic is static in nature, and algorithms are dynamic, or temporal in nature. Algorithms are inherently complex due primarily to this dynamic behavior. That is, the state of information associated with the algorithm changes with time. Accurately understanding and tracking these changes makes algorithms complex for both novice students and experienced computer scientists. One way to simplify algorithms is by filtering out the dimension of time. This is effectively what Gries does in the *Science of Programming* by applying concepts from mathematical logic to the understanding and development of algorithms [Gri81].

Can these concepts be conveyed in an introductory computer science course? Yes, if presented at the proper level through the use of illustrative examples. As we shall see, the concept is intuitively appealing since it models our cognitive behavior when thinking about algorithms. Consider the following sequence of integers: 1,2,4,8,16,... . We can all determine the next integer in this sequence. How? By identifying the mathematical relationship between adjacent integers in the sequence. This mathematical relationship is finite and static, or declarative in nature; whereas the sequence itself is infinite and dynamic. This relationship represents the "essence," or invariant specification of the sequence. (Note the similarities between this concept and mathematical induction.)

In a similar way, we can represent the invariant of an algorithmic iteration. Indeed, whenever we develop an iteration we always informally use logical invariants (think about this next time you are developing an algorithm or a program). *The Science of Programming* simply formalizes this intuition. The following problem illustrates these ideas:

> Consider the problem of climbing $N >= 1$ stairs starting with both feet on the bottom step. The goal is to get to the top with both feet on the top step. Assume you are not handicaped. You can take the following two actions: Action 1) Step one step up with your right foot, and Action 2) Step one step up with your left foot. The condition "at the top" is true only when you are standing at the top of the stairs (both feet on the top step).
>
> Using these three primitives (Actions 1 and 2, and "at the top") develop two different deterministic algorithmic solutions for this problem. Express your algorithms in pseudo code.

Before proceeding with your initial impulse to "let the algorithm flow out naturally" stop and think for a minute. Let's explore the problem in some depth first. What is the initial state? The final state? List the potential intermediate states. Now, classify states based upon common characteristics. One can identify many possible classes, including: "both feet on the same step," "the right foot is one step above the left foot," "the left foot is one step above the right foot," "the right foot is two steps above the left foot," etc. Under the specified constraints of the problem only the first three classes are feasible since we are effectively walking up the stairs blindfolded, being notified only when both our feet are on the top step[3].

Here we focus attention on the characteristics of the feasible classes of states, and not the algorithmic process itself. Now the results of this analysis can be used in a unique way to derive an algorithmic solution. If $N = 0$ the initial and final states are identical. When $N > 0$ any viable algorithm must change states to make progress toward the final state. Let's identify a feasible set of classes of states for solving this problem. For example, consider "both feet on the same step" and "the right foot is one step above the left foot." For the solution currently being derived these two classes will suffice as the logical invariants. The first invariant is satisfied for both the initial state and the final state. The second is required to make progress.

Let us first concentrate on the the sequence of states necessary to advance one step (i.e., start with both feet on a step and terminate with both feet on the next, upward step). Clearly, the sequence "both feet are on the same step," "the right foot is one step above the left foot," "both feet are on the same step" will suffice, and the sequence of actions, Action 1 followed by Action 2 will achieve this as illustrated below.

```
"both feet on same step" --->
                        Step one step up with right foot
"right foot one step above left" --->
                        Step one step up with left foot
"both feet on same step" --->
```

This sequence can now be inductively applied to get to the top.

```
"both feet on same step" (Initial State)
"both feet on same step" -->   while not at the top do
"right foot above left"  -->     [Step one step up with right foot
"both feet on same step" -->     [Step one step up with left foot
"both feet on same step" (Final State)
```

One important property of invariants is that they logically mesh together in the flow of the algorithm. For example, what is logically valid at the initial state should also be valid immediately after entering the while-do statement, since no action has taken place. Also, the invariant at the end of the

[3]For this reason the intriguing "gumby solution," consisting of first moving the right foot to the top step, followed by the left foot is not appropriate.

iteration must be the same as that immediately following the iteration when the condition is false, and upon iterating when the condition is true.

The perceptive reader will claim that this development is bottom-up and that a top-down stepwise approach is better. One can work either from the inside-out, or the outside-in as shown below.

```
"both feet on same step" (Initial State)
"both feet on same step"___  while not at the top do
                             [{make progress
                                 towards the top}
"both feet on same step" (Final State)
```

Either technique works. The important aspect is to accurately identify and specify the logical invariants, and to effectively utilize them in the derivation of the algorithm. I like to view this as working from the problem toward the algorithmic solution. With stepwise refinement we often get trapped into working from the solution toward the problem and lose important details along the way. An approach based on mathematics provides more meaningful insights into the algorithmic solution itself, thus increasing our confidence that it correct. The final algorithm simply "steps" from state to state so as to maintain the specified logical invariants.

Clearly we could have developed this algorithm without these insights; however, in doing so, cognitively we would still informally be using these invariants. For this reason logical invariants are an intuitively appealing concept. Unfortunately, for introductory students it may be difficult to motivate the need for such formal techniques. These, and similar illustrative examples will help us bridge this gap.

8. Design Oriented Problem-Solving

There has recently evolved a major interest in creative design related activities in almost every educational discipline [86]. These issues are also being addressed in computer science education [BBC86]. Although students in introductory courses have not yet acquired the basic prerequisite knowledge nor mastered the necessary skills, we can still begin to build foundations for creative design skills in introductory courses.

Design principles are fundamental to software engineering. One definition of engineering design is provided in [85].

> Engineering design is the process of devising a system, component, or process to meet desired needs. It is a decision-making process (often iterative) in which the basic sciences, mathematics, and engineering sciences are applied to convert resources optimally to meet a stated objective. Among the fundamental elements of the design process are the establishment of objectives and criteria, synthesis, analysis, construction, testing, and evaluation. Central to the process are the essential and complementary roles of synthesis and analysis.

On reading this definition one may claim that introductory computer science courses which focus on programming are teaching design. This is true to some extent, but somewhat shortsighted. Introductory students do not apply knowledge of the sciences and mathematics optimally, nor have they acquired sufficiently well developed analysis and synthesis skills. We have to provide them with the knowledge, tools, and skills required to become creative and effective software engineering designers.

An introductory course which integrates mathematics, problem-solving, and fundamental computer science concepts makes some progress toward all of the requisite features of design. Problem-solving activities help to develop analysis and synthesis skills. Here the primary emphasis should be on analysis and understanding, rather than synthesis [Hen86]. Program development assignments in introductory courses stress synthesis oriented activities at a superficial level. Accordingly, such assignments detract from learning the initially more important analysis skills. Students become rote doers, without acquiring the ability to analyze or understand the consequences of their actions. For precisely this reason the engineering disciplines, which are heavily design-oriented establish solid foundations in related areas prior to exposing students to the discipline.

One of the keys to effective design-oriented problem-solving is learning to use abstraction. This may be achieved by providing students with tools for thinking abstractly and for expressing abstractions. Such tools and techniques constitute some of the foundations which must be established early in the curriculum. For example, graphs (trees, relations, data flow, etc.), charts (structure, pie, Gantt, etc.), and diagrams (Venn, block, data flow, etc.) are representative tools. Basic mathematical concepts like sets, logic, algebra, functions, recursion, relations, counting, probability, etc., and their associated notations are important tools. All of these can be introduced in the introductory course. It is important that the relevance of these tools and concepts to computer science be defined. Also, students must realize that they will not fully appreciate and be able to apply these tools and concepts for several years.

Traditional programming languages are not good vehicles for teaching design-oriented problem-solving via abstraction. This is because novice students tend to concentrate on the syntactic and semantic details of the language and not on developing good problem-solving skills [HeF86]. Other reasons include the vast solution space (for any problem numerous "correct" programs exist), a large conceptual gap between the problem to be solved and the solution expressed in the programming language, and improper, weak or non-existent criteria for program design. For the latter, the focus leans toward programming language oriented criteria such as style of presentation, identifier names, commenting, parameter passing, etc. rather than design features.

In [HeF86] an approach to teaching design-oriented algorithmic problem-solving is presented. Here students are encouraged to identify abstract primitives which naturally convey the structure of the problem. These primitives are grouped into different classes such as object (data), operational (procedural or functional), conditional (boolean), and information (data) flow primitives. Objects and their associated operational and/or conditional primitives are grouped to form data abstractions. These primitives form the basis for the design of the algorithm at a relatively high level of abstraction. The basic design criteria is to achieve a good match between the intrinsic structure of the problem and the structure of the solution. Such a "match" can be empirically measured by making small perturbations to the problem and accessing the impact on the algorithmic solution. These design-oriented concepts and techniques can be taught, in a limited way, in introductory courses. They should be reinforced in subsequent courses.

One goal of this approach is for the student to actually develop an abstract pseudo language built from the primitives identified and the traditional flow of control primitives (while-do, if-then, case, etc.) for initially expressing their designs. This pseudo language will be closer to the problem being solved and provide a more manageable size solution space. By starting with simple, real-world problems students intuitively develop guidelines for deriving meaningful abstractions and combining these to produce an understandable, concise and coherent algorithmic solution.

9. Supporting Courseware

Often a course which is primarily concept-oriented fails to provide the proper motivation. Students have trouble seeing the connection between concepts and reality. To help minimize this gap, and to both encourage exploratory learning and develop creative design instincts courseware for this introductory computer science course has been developed.

A course which attempts the integration of concept oriented topic areas should make use of simple, concrete examples to reinforce basic concepts. According to an ancient Chinese proverb[4] it is best to involve students and have them discover ideas and concepts on their own. This can be achieved through a judicious selection of pencil and paper exercises, by having students observe real world phenomena and then relating their observations to the appropriate concepts, or by providing software which students can use to explore new concepts and improve problem-solving skills.

The impetus behind the Algorithm Discovery project at Stony Brook [HeF84, HeF] is to develop a computer-aided instructional environment for teaching algorithmic problem-solving skills [HeF86]. When solving a problem, students must first understand the problem. With this instructional environment the problem is introduced by permitting students to interactively experiment with specific instances of the problem until the interactive environment is confident that the student understands the problem and has discovered all of the key insights necessary to develop an algorithmic solution.

For example, students can understand searching via binary search by playing a simple guessing game where the computer selects a number and students try to locate this number in an ordered list of numbers. The system tells them whether their guess is high, low or correct. By monitoring the guesses the system can determine if the student has "discovered" the key insight of binary search - divide and conquer. If the student fails to find this insight within a reasonable number of attempts, the system gently provides them with appropriate hints.

As noted earlier, traditional programming languages are not good vehicles for teaching design oriented problem-solving via abstraction. The instructional environment overcomes these limitations by providing students with a restricted set of data and control primitives naturally suited to solving the specified problem. This substantially limits the number of design alternatives, and permits students to develop algorithms at a more suitable abstract level. Accordingly, in the next phase, students are provided with a simple algorithmic language for expressing their algorithms. This language includes the standard flow of control primitives (while-do, repeat-until, if-then, and if-then-else) and appropriate abstract primitives (operational and conditional) necessary to solve the problem.

Algorithms are developed using a simple syntax-directed editor and are executed using a visually oriented interactive interpreter[5]. Students learn to understand the primitives provided by developing and executing algorithms for some simple algorithmic sub-tasks. When the instructional environment is satisfied that the student is sufficiently familar with the primitives provided, then the student may proceed to the development of a complete algorithmic solution, again using the editor and interpreter. To ensure that this solution is correct, the system monitors the execution of the algorithm and warns the student of any anomaly in its behavior. Initial experience with this instructional environment in the introductory computer science course has been very encouraging.

Other instructional courseware supports the instruction of basic discrete mathematics and computer science concepts, and specifically the link between mathematics and computer science. Again, the idea is to promote understanding, through exploration. To date two instructional software packages have been developed and others are in the planning stage. One permits students to explore logical truth tables. Students input a logical expression and the system interactively displays the corresponding truth table, including relevant subexpressions. This helps to alleviate the traditional boredom associated with developing endless truth tables when learning fundamental concepts of logic. In addition, students are required to make their own logical conjectures and prove or disprove them using this software.

A second courseware package teaches students fundamental properties of binary relations, logic, transitive closure, and recursion as discussed in Section 6 (Discrete Mathematics). This package provides an intuitive view of logic programming through exploratory learning. Students actually see a representative parent relation (ancestor tree) and can initially make simple queries like "is John a parent of Sue?" The system highlights the specified names in the tree and then responds with either TRUE or FALSE. Subsequently students can actually define new relationships in terms of existing relations, including ones previously created by the student. This is achieved with a syntax-directed editor (functionally the same as the one for creating algorithms) and a simple relation definition language (see the definition of the ancestor relation in Section 6). Students can now pose queries using these new "derived" relations.

The query interpreter, which is Prolog like in nature, incorporates a mode which illustrates the query evaluation mechanism. The actual evaluation is visually traced by highlighing names and paths in the parent relation displayed on the screen, along with the results (True/False) of subordinate logical expressions (e.g., when *person 3* is bound to a specific name and the truth/falsehood of resulting "sub-query" is evaluated). Using this courseware students not only learn about logic programming, but also about the mechanics of the underlying backtracking algorithms

[4]"Tell me I forget; Show me I remember; Involve me, I understand"

[5]Visually oriented since execution, via the interpreter, actually displays changes to a visual representation of the state of the problem. Interactive since execution is graphically traced and students control the execution mode (single step, slow, fast, and forward or reverse execution).

and how these relate to the relational definitions. Plans to include the base relation "sex of" are currently being considered. Then students can define new binary relations like grandfather, nephew, mother, etc. using logic and the two base relations "parent of" and "sex of."

Other courseware packages we plan to develop include exploring functions and functional composition, simplified functional programming, mathematical induction, graphs and trees, finite state machines, recursion, and basic counting principles.

10. Conclusions

Both the area of mathematics known as discrete mathematics and problem-solving are fundamental to computer science. This paper gives a flavor of the potential for a modern introductory computer science course which effectively integrates mathematics, computer science and problem-solving in a unique way. Such a course is currently being taught at Stony Brook and initial student response has been very favorable. The course is still evolving, and new materials and courseware to support the underlying philosophy are under development. Once the course has stabilized, this material and associated courseware will be made available to other computer science educators.

11. Acknowledgements

The work presented in this paper has been supported by a Department of Education FIPSE Grant G00840481 and by a National Science Foundation Grant MDR-8470250. The following students at Stony Brook have made significant contributions to the development of the courseware described above: Steve Fu, Albert Kuo, Joyce Liu, Yu Liu, Beth Murray, Wally Olds, Steve Pecchio, Mike Winoski, and Bob Zennilman, In addition, I would like to thank David Ferguson and Alan Tucker for many fruitful discussions and contributions to this project.

12. References

[BBC86] T. Booth, T. Brubaker, T. Cain, R. Danielson, R. Hoelzeman, G. Langdon, D. Soldan and M. Varanasi, "Design Education in Computer Science and Engineering", *IEEE Computer*, **9**, 6 (June 1986), 20-27.

[Bro85] J. G. Brookshear, *Computer Science: An Overview*, Benjamin/Cummings Publishing Co., 1985.

[GoL82] L. Goldschlager and A. Lister, *Computer Science: A Modern Introduction*, Prentice Hall, 1982.

[Gri81] D. Gries, *Science of Programming*, Springer-Verlag, New York, 1981.

[HeF84] P. B. Henderson and D. L. Ferguson, "Algorithm Discovery", *Proceedings of the Sixth Annual National Educational Computing Conference*, 1984, 128-135.

[HeF86] P. B. Henderson and D. L. Ferguson, "A Conceptual Approach to Algorithmic Problem Solving", *Proceedings of the Seventh Annual National Educational Computing Conference*, June 1986, 243-249.

[Hen86] P. B. Henderson, "Anatomy of An Introductory Computer Science Course", *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education*, February 1986, 257-263.

[HeF] P. B. Henderson and D. L. Ferguson, "Guided Algorithm Discovery", *Submitted to The Journal of Educational Computing Research*, .

[Hoa69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Comm. ACM*, **12**, (October 1969), 576-583.

[MDV86] R. E. Mayer, J. L. Dyck and W. Vilberg, "Learing to Program and Learning to Think: What's the Connection?", *Comm. ACM*, **29**, 7 (July 1986), 605-610.

[Pol73] G. Polya, *How to Solve It*, Princeton University Press, second edition 1973.

[Rub86] M. F. Rubinstein, *Tools for Thinking and Problem Solving*, Prentice-Hall, 1986.

[Wic74] W. Wickelgren, *How to Solve Problems*, W.H. Freeman and Co., 1974.

[86] "New Methods for Teaching Composition and Design: Interdisciplinary Perspectives", *Proceedings of the Seventh Annual %J Proceedings of the Seventh Annual National Educational Computing Conference*, June 1986, 71.

[83] "A Nation at Risk: The Imperative for Educational Reform", National Commission on Excellence, US Dept of Education, April 1983.

[85] "1985 Annual Report", Accreditation Board for Engineering and Technology, October 1985.

Appendix A

Syllabus for CSE-113
Foundations of Computer Science

A rigorous introduction to the conceptual and mathematical foundations of computer science. Problem-solving techniques and mathematical concepts that aid in the analysis and solution of algorithmic problems will be stressed. The course will concentrate on general problem solving principles, algorithmic problem solving, and discrete mathematics concepts including: logic, relations, graphs, counting principles, functions, sequences, induction, proof techniques, language concepts, algorithms, algorithm complexity and verification, and recursion. These concepts will be motivated within the context of computer science, and its applications. The background provided in CSE-113 will prepare students for further computer science courses where these fundamental ideas and concepts can be applied to the development of actual software systems. Mathematical maturity at the level of pre-college calculus is expected.

Mathematical Foundations

A. Sets
 1) Representation
 2) Subsets
 3) Set Operations
 4) Power Sets & Counting

B. Logic
 1) Propositions & Statements
 2) Truth Tables
 3) Logical Equivalence
 4) Logical Implication
 5) Logical Proofs

D. Functions, Sequences and Induction
 1) Recursive Functions
 2) Sequences
 3) Mathematical Induction

E. Relations and Graphs
 1) Orderings, Equivalence Relations
 2) Composition, Closure
 3) Directed & Undirected Graphs
 4) Matrices, Graphs and Relations
 5) Special Graphs (Complete, etc.)

F. Trees
 1) Properties
 2) Binary Trees, Recursive Definition
 3) Rooted trees

G. Discrete Counting Principles
 1) Basic Counting Rules
 2) Permunations and Combinations

H. Languages
 1) Regular & Context Free
 2) Specification Mechanisms
  a) Transition Diagrams
  b) Syntax Diagrams
  c) Grammars & BNF

Problem Solving

A. General Problem Solving
 1) Goals, Givens, and Rules
 2) Inference and States
 3) Actions Sequences
 4) Subgoals, Divide and Conquer
 5) Contradition and Working Backward
 6) Relationships Between Problems
 7) Effective use of Abstraction

B. Algorithmic Problem Solving
 1) Problem Understanding
 2) Structure of the Problem
 3) Identify Abstract Primitives
  a) Data, Procedural, Functional
  b) Data Abstractions
 4) Notations for Expressing Primitives
  a) English, Procedural
 4) Identify Structure of Solution
 5) Algorithm Development

Algorithms

A. Representation
 1) Pseudo Code
 2) Structured Flowcharts

B. Development
 1) Stepwise Refinement
 2) Top-Down vs. Bottom-Up

C. Modularity
 1) Procedural and Functional Abstractio
 2) Data Abstractions

D. Recursion

E. Data and its Relationships

F. Fundamental Algorithms
 1) Searching
 2) Tree Traversal
 2) Sorting
 3) Table Manipulation

D. Mathematical Concepts
 1) Algorithm Complexity
 2) Pre & Post Conditions
 3) Assertions and Invariants
 4) Correctness Proofs
 5) Principle of Mathematical Induction