SOMETHING FOR EVERYONE: AI LAB ASSIGNMENTS THAT SPAN LEARNING STYLES AND APTITUDES

Christopher League
Long Island University Computer Science
1 University Plaza, Brooklyn, NY 11201
christopher.league@liu.edu

Abstract

One of the great challenges of teaching is managing a wide range of educational backgrounds, learning styles, aptitudes, and time/energy constraints in the same classroom. Aiming down the middle is a poor strategy; it is unacceptable to write off the lower half of a class, and we risk extinguishing the enthusiasm of the best and brightest by moving too slowly. We present a set of workbook-style lab assignments for an undergraduate course on artificial intelligence. By designing them carefully in accordance with Bloom's taxonomy, they can span learning styles and aptitudes. With them, we hope to establish a disciplinary commons – a public repository of source code, notes, questions, and exercises.

1. Motivation

The author teaches in a small computer science department on a diverse urban campus. Most of our students are the first in their families to seek higher education, including recent immigrants with families and/or full-time jobs. One of the greatest challenges of teaching in this environment is managing a wide range of educational backgrounds, learning styles, aptitudes, and time/energy constraints in the same classroom. Like Lister and Leaney, we believe "the goal of every university teacher should be to realize the potential of each student" [8, 9], and so 'aiming down the middle' is not an appropriate strategy. Rather, we try to teach and assess at multiple levels of Bloom's cognitive taxonomy [2] simultaneously.

In Fall 2006, we prepared to teach the undergraduate course on *artificial intelligence* (AI) for the first time. In accordance with this teaching philosophy, we developed a set of workbook-style lab assignments that interleave lecture notes and software demonstrations with a series of questions, tasks, and projects. Students that need reinforcement have it available, and are assessed on basic knowledge, comprehension, and application. Students that already understood the lectures and other in-class activities (or think they did) bypass the notes, answer the easy questions, then spend their time on analysis, synthesis, and evaluation.

Lister and Leaney [8] argue that "[computer science] academics place premature emphasis on the higher levels of the taxonomy." For example, in undergraduate AI courses, we traditionally toss students a Lisp book and tell them to "implement a constraint solver," or some other impenetrable synthesis task. Such an approach does not work in our environment, if indeed it can be said to have worked anywhere.

The main contribution of this work is a set of AI lab assignments that span learning styles and aptitudes. With them, we intend to establish a public repository of source code, notes, questions, and exercises; see `http://contrapunctus.net/sail/`. We invite users and contributors who share our philosophy – a disciplinary commons in the terminology of Tenenberg and Wang [15]. This paper provides an overview of our teaching strategy for the course, and shows in particular how three of the assignments fit in with our teaching philosophy and objectives for the course.

2. Method

We begin with an overview of the structure of the course. Our semester runs 14 weeks, not including a short reading period and exam week. Within this time frame, we address the following seven topics:

1. Philosophical background, strong vs. weak AI, Turing test, chat-bots. A wall-following algorithm for a simple stimulus/response agent.

2. Machine learning by example: classification problems, decision trees, entropy, and ID3.

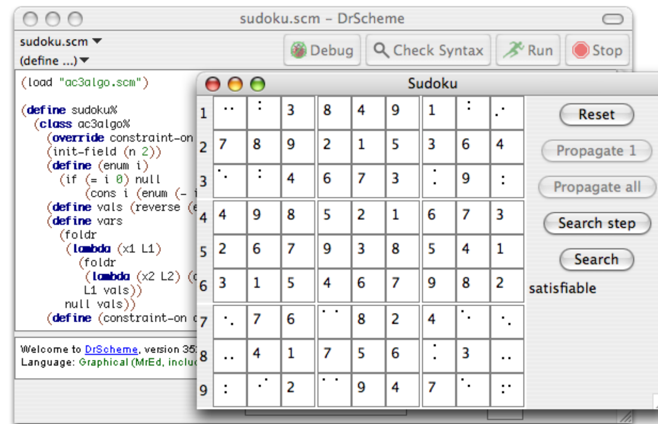

Figure 1: Interactive Sudoku solver. Dots in each square depict the set of plausible numbers; when just one dot remains, it is replaced with a numeral. The propagate buttons remove dots from co-constrained squares, while the search buttons begin a depth-first search of values for the under-constrained squares. The label reports whether the configuration is unsatisfiable, satisfiable, or solved.

3. Machine learning by evolution: optimization problems and genetic algorithms.
4. Planning using uninformed and heuristic search: breadth-first, depth-first, and A*.
5. Constraint propagation and satisfaction using AC3.
6. Adversarial search with minimax, heuristic board evaluation, games of chance.
7. Knowledge representation, logic, expert systems, common sense.

One lab assignment is associated with each topic. In class, we focus on understanding the basic techniques and concepts, and on formulating new problems in terms of these algorithms.

The primary programming language for the course is Scheme, and we use the DrScheme [4] environment. Its interactive, graphical nature engages students, and it runs the same on all the major computing platforms. Note that our introductory programming sequence (a prerequisite for AI) is based on C/C++, not Java or Scheme. In fact, this AI course is the only place that Scheme has appeared in our curriculum so far. Throughout the semester, we spend just about 3 to 4 classroom hours on explicit instruction in Scheme. The role of Scheme programming in the assignments is gradually increased, but never reaches totality. We rely on the object-oriented features of the DrScheme dialect to illustrate how, for example, the AC3 (arc consistency) algorithm for constraint propagation [10] can be directly instantiated to solve Sudoku puzzles, crosswords, or n-queens.

Figure 1 contains a screen-shot of the interactive Sudoku solver. Students can enter puzzles from newspapers or web sites, visualize the constraint propagation, and discover whether the depth- first search component needs to guess and backtrack. This same premise is used for each assignment: we provide some non-trivial interactive software as an engine for demonstration and experimentation. Then we pose a variety of questions, problems, and programming exercises that build in cognitive complexity, one step at a time.

3. Materials

Due to space constraints, we will focus on just three assignments. In each case, we describe the basic concept and provided software, then provide some sample exercises and note their classification in Bloom's taxonomy.

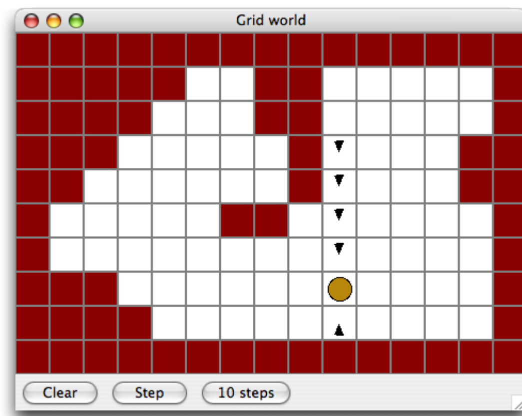

Figure 2: Wall-following robot simulation. The grid world is easily specified as a list of strings in Scheme. As it moves, the robot leaves behind arrows, making it easy to trace where it has been.

Stimulus/response wall-following. The primary objective of the first assignment is to ensure that all the students are enrolled on the course web site and comfortable with the required software. Following Nilsson's book [12], we begin by postulating a robot with eight directional sensors: s1 to the north-west, s2 to the north, and clockwise around to s8 on the west side. In response to these sensors, the robot can decide to move in one of four directions. Its goal is simply to follow the walls around the perimeter of a room.

The students are given the Scheme code to simulate this robot in a 'grid world' – see figure 2. They compose simple conditional expressions such as `(if s6 'north 'south)` that are interpreted as robot controllers by the program. They work for a while designing and testing robots, but it takes some insight to design a wall-follower correctly. A series of questions in the assignment handout will guide them:

1. Given a particular map and robot location, what are the values of its sensors [comprehension]?
2. Given a robot controller and a map configuration, which way will it go [application]?
3. Given a map of the grid world, mark all the squares from which your robot should move north. What features distinguish those squares from all others [analysis]? Repeat the exercise for moves to the east.
4. Compose and test your own controller [synthesis].
5. Look at the provided code to see how the grid world map is specified, then design your own room [synthesis].
6. What are some limitations of a stateless stimulus/response system [evaluation]?
7. Suppose we redesign the robot with an *orientation* – it can move forward only in the direction it is facing, but can also rotate 90 degrees left or right. Adapt your controller to output these three responses instead of the four directions [application]. Do you need all eight sensors now [analysis]?
8. *Rewrite the robot simulator program to support this new robot design [synthesis].

The general pattern here is that we design comprehension, application, and analysis problems that lead students gently into the primary synthesis task. We always pose more advanced and open-ended tasks (starred) near the end of the assignment, to capture the imagination of the faster learners. These often involve deeper comprehension of the provided code and significantly more programming. Most students will not get that far, but we urge the most capable ones to take up the challenge.

Machine learning of decision trees. The next topic in our progression is machine learning of classification problems from examples using Quinlan's ID3 algorithm for building decision trees [13]. The algorithm inductively finds the most significant test for distinguishing the examples by measuring the weighted average entropy after the partition. The software provided is an interactive workbench for running and querying ID3, but there is no graphical component in this case. The assignment handout reviews the concept of entropy and related formulas, then presents a table of about 25 examples of car owners with attributes such as gender, age, number of accidents, car type, and risk classification (high, medium, low) for insurance purposes. This table is used for exercises throughout:

1. Explain the concept of *entropy* in your own words [comprehension].
2. Given a decision tree diagram, write it as an if-expression in Scheme (and vice versa) [comprehension].
3. Given a decision tree and a set of examples, which examples are *misclassified* by the tree [application]?
4. Compute the entropy of an election in which 998 people vote for Lisa Simpson and 2 people vote for Eric Cartman [application]. Compared to an election that is 500 to 500, which do you expect to have higher entropy [analysis]?
5. Compose a Scheme function to implement the entropy formula for a 3-class system [synthesis].
6. Determine which is the best initial test for classifying the car insurance examples [analysis].
7. Compose a Scheme function to implement the entropy formula for an arbitrary number of classes [synthesis].
8. Apply the ID3 algorithm to the data set classifying mushrooms as poisonous or non-poisonous, from the UCI Machine Learning Repository[1] [application].
9. Justify why it is useful to partition the raw data into a training set and a test set [evaluation].
10. How low can the proportion of training data go before the accuracy of the learned decision tree falls below 97% on the test set [analysis].

The programming in this assignment is fairly minimal, but students compose Scheme function definitions to implement the entropy formula, and apply our provided definitions to manage the training and test data and produce decision trees in Scheme syntax.

Game playing. General AI courses traditionally include a unit on game playing with adversarial search. In our case, the assignment focuses on the zero-sum perfect-knowledge game Connect Four. We provide the interactive component (see figure 3) and even the minimax algorithm. The assignment handout provides a ladder for experimentation and helps students design a heuristic for evaluating board configurations with respect to one player or another. This culminates in a class where the students' heuristics compete against one another in a tournament. Here are the exercises:

1. Run the game with a *constant* heuristic (one that returns the same score for every board). Explain the actions of the computer players [comprehension].
2. Modify the heuristic so that it rewards a particular location on the board with a high score, and now observe the computer players [application]. Figure 3 demonstrates this: the heuristic heavily rewards placing your piece in the third row up of the second column (shown with a dark circle); all other board configurations are equal. We can see here that the computer player (red pieces, odd numbers) plays straight up column 1 and the bottom row of column 2, but then moves on to column 3. It will not allow the human player to capture the designated position.

[1] `http://archive.ics.uci.edu/beta/datasets/Mushroom`

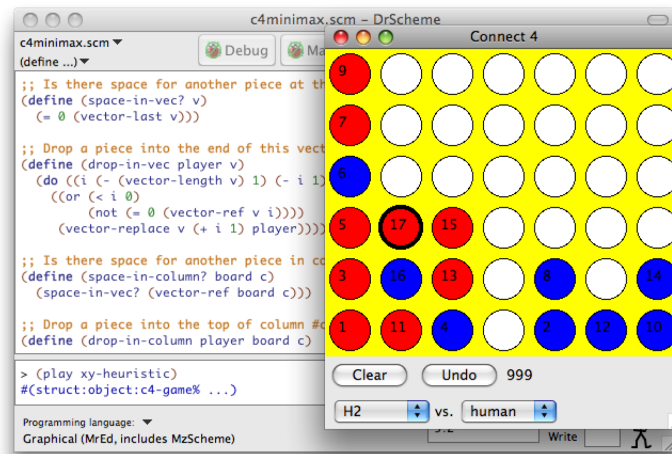

Figure 3: Connect Four game. The drop-down menus allow choice of human player, random player, or a variety of heuristics applied at different search depths. The pieces are numbered so that the history of moves is evident from any board configuration.

3. Assuming that no columns become full, how many times will a heuristic be applied when searching 1, 2, and 3 levels deep [analysis]?
4. Count the number of unique winning positions in each row, column, and diagonal [analysis].
5. Compose a function that implements a given table of scores for certain column configurations [synthesis].
6. Generalize your heuristic to consider rows and diagonals [synthesis].
7. Critique the performance of your heuristic in a variety of situations, and adjust its scoring scheme [evaluation].
8. *Improve the given minimax algorithm with alpha-beta pruning [synthesis].

The competitive nature of the tournament clearly engages students, and we allot more time for this assignment than any other.

4. Results

Obtaining significant statistics on instructional effectiveness is difficult in the best of times; with the national downturn in enrollments in computer science programs, it is nearly impossible. Anecdotally, however, most students engage very successfully with this approach. In comparison, when assigned a traditional synthesis task on its own, most students flounder and submit nothing. For the first incarnation in Fall 2006, our student survey recorded maximal scores across the board – the only time this has occurred in our experience.

5. Related work

We mentioned already the ground-breaking work of Lister and Leaney [8, 9] on applying Bloom's taxonomy to computer science, particularly for criterion-referenced grading. Our philosophy also connects well with Bruner's concept of a spiral curriculum: "A curriculum as it develops should revisit this basic ideas repeatedly, building upon them until the student has grasped the full formal apparatus that goes with them" [3, page 13].

Imberman [6] recently presented three fun AI assignments that are compatible with our approach. In fact, we also spend time on activities related to chat-bots and the Turing test, and on the sixteen puzzle in

our unit on planning. Although her paper does not specifically address our philosophical objectives about reaching a wide range of students, the proposed projects similarly do not require a tremendous amount of programming from scratch. Scheessele and Schriefer [14], McGovern and Fager [11] and Hansen et al. [5] offer ideas on game-playing agents that would certainly attract our students. (Some of our students maintain that Connect Four is 'lame' and we should play Texas hold 'em instead.)

Bloch promotes Scheme as a syntactically simple teaching language [1], even for concepts such as object-oriented programming, typically associated with more mainstream languages such as Java. In our software, the object-oriented features of DrScheme were extraordinarily helpful, and students with more instruction in Scheme could take them further. Kozak [7] criticizes Lisp and Prolog for modern AI courses because of their limited facilities for I/O and GUIs, but this clearly does not apply to DrScheme.

Acknowledgment: Thanks to Josh Tenenberg for discussion and advice on these ideas, and for pointing me to the work of Raymond Lister. Any problems or limitations in the work are mine alone.

References

[1] S. Bloch. Teach scheme, reach Java: introducing object-oriented programmming without drowning in syntax. *Journal of Computing Sciences in Colleges*, 23(3):119–119, 2008.

[2] B. S. Bloom. *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Longmans, Green, and Company, 1956.

[3] J. S. Bruner. *The Process of Education*. Vintage, 1960.

[4] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[5] D. M. Hansen, J. Bruce, and D. Harrison. Give students a clue: A course-project for undergraduate artificial intelligence. In *Proc. Conf. on Computer Science Education (SIGCSE)*, pages 44–48. ACM, 2007.

[6] S. P. Imberman. Three fun assignments for an artificial intelligence class. *Journal of Computing Sciences in Colleges*, 21(2):113–118, December 2005.

[7] M. M. Kozak. Teaching artificial intelligence using web-based applications. *Journal of Computing Sciences in Colleges*, 22(1):46–53, 2006.

[8] R. Lister and J. Leaney. First year programming: let all the flowers Bloom. In *Proc. Australasian conference on Computing Education (ACE)*, pages 221–230. Australian Computer Society, Inc., 2003.

[9] R. Lister and J. Leaney. Introductory programming, criterion-referencing, and Bloom. In *Proc. SIGCSE Technical Symp. on Computer Science Education*, pages 143–147. ACM, 2003.

[10] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[11] A. McGovern and J. Fager. Creating significant learning experiences in introductory artificial intelligence. In *Proc. Conf. on Computer Science Education (SIGCSE)*, pages 39–43. ACM, 2007.

[12] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

[13] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[14] M. R. Scheessele and T. Schriefer. Poker as a group project for artificial intelligence. In *Proc. SIGCSE Technical Symp. on Computer Science Education*, pages 548–552. ACM, 2006.

[15] J. Tenenberg and Q. Wang. Using course portfolios to create a disciplinary commons across institutions. *Journal of Computing Sciences in Colleges*, 21(3):142–149, February 2006.