Synergizing AI and OOSE: Enhancing Interest in Computer Science Through Game-Playing and Puzzle-Solving

T.M. Rao and Sandeep Mitra

Department of Computer Science,
SUNY Brockport, Brockport NY 14420
{trao, smitra}@brockport.edu

Abstract

Writing puzzle-solving and game-playing programs can generate excitement in students. Such programs are usually assigned as projects in an AI course. These are major software projects expected to be completed by students who usually have had no prior instruction in SE concepts. Often, these are required to be implemented in languages and paradigms in which the students have minimal expertise resulting in frustration among students as well as faculty. We discuss a "problem stereotypes and solution frameworks" approach to teaching CS1/2 (most likely prerequisites to AI) which ensures that the students have exposure to a judicious amount of SE methodology. We extend this approach to teaching an AI course by identifying the *State-Space-Search* and *Two-Person-Zero-Sum-Game* stereotypes. We have developed a solution framework (consisting of Java abstract classes) that can be used to solve typical problems falling into these stereotypes. Using the framework, a student will be able to develop a puzzle-solver or a game-player program mainly by focusing on problem-specific details. Preliminary experimentation has revealed that the students found it easy to use the framework and were successful in developing puzzle-solvers. We believe that our approach based on reusable software infrastructure enables students to develop interesting programs early in their undergraduate careers.

Introduction

Almost all computer science major programs have experienced a decline in enrollment following the dot-com bust of 2000-01. At SUNY Brockport, we have experienced a decline of over 60% since 2001. Besides a reduction in the number of incoming freshmen, the attrition rate for those who initially declare a CS major is also high. Considering the reasons for this decline, we have observed that there is a perception among students that CS is a "hard" discipline (Rao et al. 2007). Our conversations with students have indicated that they find the introductory courses themselves very difficult. In earlier years, students lived with this level of difficulty because they felt that a substantial paycheck awaited them after graduation. Nowadays, they feel that this is no longer the case. The level of frustration that students endure in the introductory courses – especially in getting a program to *work* according to instructor specifications – does not seem "worth it" any more. Typically, an introductory CS course requires students to write programs of some complexity in a high-level programming language (e.g. Java). Some basic syntax and semantics are covered in class, examples of working programs are shown and the students are assigned a programming problem as homework. The *process* used to arrive at the correctly working code is rarely discussed, as a result of which the student has to essentially invent the data representations, algorithms and the mapping of these to language constructs. With no specific plan, students begin to write the code itself, and encounter obscure compiler and run-time error messages. They adopt a trial-and-error process to fix their code, and have no guarantee of getting to a successful solution by submission time. Even after all this frustration, there is no assurance of a good grade, because the "working" program may still not meet instructor's specifications. Many feel that the same amount of effort put into writing a term paper in some other discipline provides a better chance of an 'A'.

Secondly, assigned programming exercises in CS1/2, such as traversing a binary tree in post-order, are not very "exciting" either. When we asked prospective students at Open Houses and other forums what they would like to do after learning software development, one of the most frequent responses we got is "write game-playing programs." While it would be challenging and exciting to write puzzle-solving or game-playing programs, these are not feasible assignments for CS1/2. On the contrary, they are large software development projects needing, on the one hand, expertise in software design and implementation, and on the other, knowledge of game trees and heuristic functions. Typically, students learn these concepts only in upper level Software Engineering (SE) and Artificial Intelligence (AI) courses. Because of curricular restrictions on allowable prerequisites, it may not be possible to ensure that AI students have the required SE background to handle such complex projects. Thus, an AI instructor wishing to assign such "exciting" problems is handicapped by the lack of SE background in the students.

Our Approach

To reduce the stress level of incoming students and also to enable students to write simple GUI-based game and puzzle solvers during their undergraduate studies, we have proposed and used the "Problem Stereotypes and Solution Frameworks" (Mitra, Rao and Bullinger 2005, Rao et al. 2007) approach to teaching CS1/2. A problem stereotype is a *category* of problems that can be solved using similar techniques. A solution framework is a typical solution to a problem, parts of which can be *reused* to solve other problems of this stereotype. The students are introduced to a stereotype by discussing many problems that belong to it. Solution frameworks then show how a typical problem of this stereotype is coded. Homework problems come from the same stereotype. In the early stages the students are given a complete set of artifacts such as Use Cases, CRC Cards (Wirfs-Brock and McKean 2002), test data, and solution code for a representative problem. The students are required to create a model of the problem before writing code. They are then taught systematic techniques of translating these models into program code. In the beginning courses, the student's responsibility would be to produce documents and code that closely mirror the example problem artifacts. After some experience, more responsibilities (design, test data generation, etc.) are left to students.

Since AI would have at least a CS2 prerequisite, introducing SE concepts in CS1/2 would certainly be useful to the AI course. This paper discusses extending this approach to teaching AI. We have identified two problem stereotypes in the AI domain: *State Space Search* and *Two Person Zero Sum Game*. We created solution frameworks for these stereotypes by developing a library of reusable code implementing behavior common to all applications of that stereotype. Following the practice used in CS1/2, students are given a sample solution to a selected puzzle/game using the reusable components of framework. Other similar puzzles/games constitute class projects. Students need to understand the manner in which the framework is used, and cast their own solutions in terms of the framework's facilities. Consequently, they only have to code the *application-specific* parts of the project problem. Through this approach, students have become more productive, thus experiencing a greater sense of achievement.

Considering our approach from the SE standpoint, we note that *classifying* a set of problems into a stereotype, and *creating* a reusable solution framework including common behavior (nowadays, usually in an object-oriented (OO) manner) is a topic for the SE course (Booch 2007, Fowler 2003, Wirfs-Brock and McKean 2002). Collaboration between AI and SE can be beneficial to both courses: SE students develop software for *application-independent* aspects (i.e., the reusable parts) such as data structures common to all puzzles/games, control flow, user interfaces, etc. AI students program aspects particular to the problem itself – such as rules deciding legality of a game move, heuristic functions, etc.

Related Work

Several educators have considered using games as programming assignments in their courses. Becker (Becker 2001), who used Minesweeper and Asteroid games as assignments, observes that "computer science is often described as one of the most demanding disciplines on the campus" and (game programs) are "well within the realm of do-able by first year students and they are fun." Goschnick (Goschnick and Balbo 2005) reports the use of a 'game board library' in Java that enabled students to develop software for games such as Snakes and Ladders and Ludo. Others (Faltin 1999, Sindre, Line, and Valvag 2003) have reported positive classroom experiences but do not provide the details of exact programming environments in which the students worked, how much of the infrastructure (supporting code) was provided to them and what the students' responsibilities were. Our paper describes a problem stereotype approach which attempts to classify problems based on common features and develops reusable solution frameworks that can support implementing many puzzles and board games within that stereotype. Our approach enables the student to quickly develop a complete new game within the same 'family', or adapt an existing game to be 'smarter' (e.g., by redefining the heuristic).

The *State Space Search* Stereotype and Solution Framework

Consider the Water Jug problem used often to illustrate problem solving in AI (Luger 1998, Russel and Norvig 2003). A version of the problem reads as follows: "You have two jugs A and B (with no markings) of capacities 4 and 3 gallons respectively. Given an unlimited supply of water, get exactly 2 gallons in A." This problem can be solved by a systematic search of the *solution space*, and therefore, it falls into the *State Space Search* (Luger 1998, Rao 2003, Russel and Norvig 2003) stereotype. We will use this example to illustrate our approach. After instructing students in the theoretical concepts of state space search (reinforced by sufficient pencil-and-paper exercises), the solution to the Water Jug example is used to show the students how to create a *model* of this problem consistent with the stereotype – i.e., a model that can be mapped to code that uses the provided framework:

1. Identify what constitutes a *state* in this problem and the data structures to represent it.
2. Identify what the *initial state* is and what the *final state(s)* is (are).
3. Identify what the *operators* are and how they transform one state to another. Design the *applyOperator* algorithms to determine the next state for each current state/operator combination.

For the Water Jug problem we have:

State: [A, B], a 2-element array of current jug contents
Initial State: [0, 0], Final State: [2,_]
Operators: fill (X): Fill X until X is full dump (X): Empty the contents of X pour (X, Y): Pour from X to Y until Y becomes full or X becomes empty (where X and Y represent either Jug A or Jug B)

The students are then shown how this model maps to code using the solution framework. Reusable framework components include Java abstract classes and interfaces implementing data structures and algorithms common to any problem solved using the state space search technique. Briefly, consider some of the more important classes:

Configuration: This abstract class enables the developer to define application-specific state data. For example, a WaterJugConfiguration class encapsulates a 2-element integer array of contents. Our approach requires the framework user to override the three abstract methods in this class as described below:

- applyOperator(Operator op): This method computes the configuration resulting from applying 'op' to the current configuration.
- isMatch(Configuration otherConfig): This is a boolean method that decides if the current configuration matches the 'otherConfig' parameter. This method is used to prevent repeating configurations in our lists of search states, and check if a goal state is reached.
- getDistanceToGoal(): This method computes the heuristic distance from the current configuration to the goal configuration. This heuristic is used to steer the search towards the goal.

State: Our implementation distinguishes between a generic state and an application-specific configuration. This abstract class encapsulates a configuration, but has additional information to facilitate specific search strategies (e.g., A* search) and compute the final solution path. Therefore, data such as who this state's parent is, the operator applied to the parent to produce this state, distance from the start state (the A* g-value), the estimated distance to the goal state (the A* h-value), etc. are present. Important abstract methods in this class that the user needs to override are:

- isMatch(State otherState)
- traceToRoot(): This method is applied to the final goal state reached and computes the solution path to the start state.

Operator: This abstract class defines a generic operator, and should be extended to encapsulate data specific to the application's operators. All methods needed to manipulate this data must also be written by the user.

Solver: This abstract class defines the method "solve()", which currently implements Breadth-First (BFS) and A* search algorithms. The application-specific solver must be customized properly to indicate which search strategy to use. This customization also requires setting up instance variables defining the start and goal configurations, and a Java Vector of all possible operators. Internally, this class maintains lists of open and closed states needed in the search. It also includes code for the "expand" method – i.e., code that considers each state in the open list, applies all application-specific operators to it and generates the next set of states. We have found the OO approach to be greatly beneficial in enabling the movement of common structures/behaviors (e.g., expand(state)) to the reusable components, thus reducing the programming tasks of the student programmer to the greatest extent possible.

These classes are used as follows for the Water Jug example:

WaterJugConfiguration (inherits from Configuration): Instance variables include a 2-element integer array of jug contents. Therefore, the "isMatch (otherConfig)" method is relatively easy – it simply compares the current object and parameter's instance variables for equality. The "applyOperator(op)" method is more challenging. It requires recognizing the exact nature of the operator parameters, seeing if it is applicable to the current configuration (e.g., you cannot pour out of an empty jug) and returning the new configuration resulting from operator application. This method is usually complex, and we encourage students to take a top-down approach, which often results in several private helper methods (e.g., a "fill()" or "pour()" method). We have used a simple heuristic: the distance to goal [gA, gB] from the current state [sA, sB] is defined as the sum of |gA- sA| + |gB -sB|.

WaterJugState (inherits from State): This is mainly a factory class. It shows how a state may be created from its parent state, and a new configuration. It also indicates the "cost" of moving from the parent state to the current – in this example, the cost is assumed to be 1.

WaterJugOperator (inherits from Operator): A Water Jug operator has a name (e.g. fill, dump), a source jug, and for the 'pour' operator, a destination jug. Instance variables record this data, and accessor/mutator methods are also written.

WaterJugSolver (inherits from Solver): The constructor creates the start and goal configurations and the list of operators. It then calls the "solve()" method it inherits from its superclass, and displays the final solution. Constructing the list of operators is one of its important responsibilities, and for this purpose, a 'protected' Java Vector named 'myOperators' is provided by the superclass. The programmer defines the setup of this Vector in the "setUpOperatorVector()" method. For this problem – with

2 jugs – the operators are: fill(0), fill(1), dump(0), dump(1), pour(0, 1) and pour(1, 0). Six WaterJugOperator objects are therefore created and added to 'myOperators'. The solver also acts as a factory class, defining methods that the reusable components use to create a new configuration and a new state. However, the constructor is the main operative method, and it broadly executes the following steps:

a. startNoGUISetup("AStar"); // Choose search strategy, and no GUI input.
b. Create the appropriate start and goal configurations and assign them to the following inherited instance variables: 'startConfiguration' and 'goalConfiguration'.
c. completeNoGUISetup(); // Boiler-plated code (does the rest of the setup)
d. Solution solution = solve();
e. display(solution);

Programming Assignment: The Eight Puzzle

The Eight Puzzle problem has a 3 x 3 board with 8 numbered tiles and one blank space. A "move" consists of the blank space moving into an adjacent space occupied by a numbered tile. Thus, there are four moves: *up, down, left* and *right.* The object of the puzzle is to reach a goal configuration showing a certain arrangement of the numbered tiles from a start configuration, using only legal moves.

Following our approach, the students build a model with the following features: the configuration is a 2-D (3 x 3) array of characters; start and goal states are provided by the user; and the operators are *up, down, left* and *right*. To adapt the Water Jug solution to the Eight Puzzle, the students follow a step-by-step procedure outlined in a manual given to them.

The first thing this manual indicates is that each of the classes discussed for Water Jug will have their Eight Puzzle counterparts – therefore, be sure that for each "WaterJug*.java" file, there is a corresponding "EightPuzzle*.java" file. The "EightPuzzleConfiguration. java" file is then edited to include the 2-D array to represent a 3x3 board. The "isMatch(otherConfig)" method is again easy, and must do an element-by-element comparison of the respective 2-D arrays. Again, the "applyOperator(op)" method is the challenge. First, the blank space must be located on the board (a private helper method called "locateSpace()" is a good idea). Depending on this location, the application of the operator 'op' may be infeasible (e.g., a blank in the top row cannot move up). For feasible operators, helper methods such as "moveUp()", "moveLeft()", etc. may be written to assist "applyOperator(op)" in computing the board associated with the next configuration. After this, the students need to complete the "getDistanceToGoal(goalConfig)" method using a suitable heuristic. Commonly used heuristics include counting the number of misplaced tiles in the current state or the sum of the "Manhattan" distances for each tile from its current location to its location in the goal state.

The "EightPuzzleOperator.java" file is very easy to modify – each operator here only has a name associated with it (unlike the Water Jug operators, there are no additional data associated with them). Modifications to the "EightPuzzleState.java" file are also very minimal – since the cost associated with the transition from each state to the next is also one unit here, only names have to be changed. The constructor of the "EightPuzzleSolver" class only requires a modification of Step (b) discussed for "WaterJugSolver" constructor above. Further, the code for the factory methods (createConfiguration(), etc.) in the solver can easily be written – usually by just modifying the names in the corresponding Water Jug example.

Our AI Class Experience

We used this approach in our AI class in Spring 2007. To gauge the usefulness of our approach, we gave the Eight Puzzle in two phases. In Phase 1, we gave a linked-list implementation (used in our CS2 class) and asked students to solve the Eight Puzzle using the A* algorithm. They could use the provided linked list classes to represent the open, closed and other lists required by A*. Note that students had to write the full code (from scratch) for the A* algorithm. A few weeks later, in Phase 2, we explained the *State Space Search* stereotype, its ability to incorporate either BFS or A*, and demonstrated the framework described above. To encourage participation in using the framework to solve the same problem, students were told that the better of the two grades would count for their final grade. A survey of the students done after Phase 2 submission indicated that they reacted positively, and largely preferred to use the framework. At the end of Phase 1, only one of nine students got a fully working program, and two others were close. But after phase 2, six students created fully working programs. Written comments appreciated the availability of the reusable infrastructure. To quote: "*… reduced worries about how to search, expand the current state, define the h-value function*", "*I haven't used linked lists in a long time, so I had trouble implementing them in this problem, even though the rest of the algorithm was pretty much correct. For this reason, I liked the abstract infrastructure available*." One student commented that the linked list could have been used to eventually create the solution. He also said that understanding the classes in the infrastructure required some effort, but once understood he would rather use the infrastructure.

The Two-Person-Zero-Sum-Game Stereotype

Encouraged by the success of this approach, we have developed a *Two-Person-Zero-Sum-Game* (TPZS), a stereotype for board games. The idea is to enable students to write game-playing programs such as Tic-Tac-Toe,

Connect Four, etc. Once again, our goal was to create the *right* reusable infrastructure that is common to all such games. We took the same use-case based approach we had taken when considering the State Space Search stereotype. Our goal was to build the infrastructure to enable game development of software that can handle two versions of the game: human vs. human (computer acts as a referee) and human vs. computer. At the moment, the human vs. human version requires both players to be on the same computer. Our analysis indicated that one of the common features of all such games is the *workflow* associated with the game itself. The following sequence is common:

- A *player* takes a turn, and makes a *move*.
- This move is sent to the game *board*, which decides if the move is legal, and if so, updates its own *state* on the basis of the move contents.
- The board then checks if the game is over. If so, it then determines if there is a winner and the identity of the winning player (it is also possible for the game to end in a draw).
- The board informs the *referee* of "game termination" decision. The referee then informs both players of this decision. If the game is not over, the referee asks the other player to take a turn, thus continuing the game.

Besides the above protocol, our approach enabled us to identify common features within the players themselves. A player could either be a *human player* or a *computer player*. For any human player, it should be possible to create a suitable GUI that displays the current board configuration, accepts a move, and informs the player about the progress of the game (i.e., legality of the move made, a win/lose/draw decision, etc.). For a computer player, one of the common standard approaches is to create a limited-depth *game tree* whose nodes encapsulate a configuration of the game board. Repeated application of all possible legal moves to the nodes creates the tree to the desired level. Thereafter, the player applies the Mini-Max algorithm to the game tree, thus determining the best move to make. Note that applying this algorithm requires the board to have a *heuristic function* determining a *goodness value* from the point of view of winning the game.

We now discuss some important abstract classes provided in the solution framework. As in the State Space Search stereotype, each of these abstract classes has to be inherited from to implement a specific game.

Board: This abstract class represents a game board and provides the following abstract methods:

- isLegalMove(Move m)
- updateState(Move m)
- gameOver()
- getGoodnessValue(Symbol own, Symbol enemy)

The game developer of a specific game is required to create a specific board class that inherits from the above Board class. This class contains the most application-specific behavior. The sub-class needs to provide the code for each of the above methods.

Move: This abstract class encapsulates a *symbol* instance variable, since every move made by any player must necessarily include the player's symbol. Other application-specific move data need to be included in the sub-class.

HumanPlayer: Setting up a GUI is one of the tasks that students find most onerous. We have attempted to standardize elements of the GUI that displays features common to all games (e.g., the final decision) and thus reduce the student's burden. The abstract class "Human Player" provides facilities for the common GUI elements. The application-specific sub-class is required to set up a GUI that shows the view of this game's board, and also enable the entry of a move. But GUI elements showing the board's view and enabling user input specific to the game have to be written individually for each game.

Computer Player: As discussed above, the abstract class contains the entire infrastructure to build the game tree. The application-specific sub-class only needs to complete a method that generates a set of moves possible given a current board.

Referee: The abstract class contains logic to initiate, continue and terminate the game. It also informs players of a termination decision. The application-specific sub-class essentially sets up the whole game (i.e., the board, the two players), and determines who goes first.

In addition to the above classes, the framework contains other classes that facilitate the operation of the game workflow. For example, a class called "Player" is present, and it is the super-class of the "Human Player" and "Computer Player" classes discussed above. It contains the code to effect the player-selected move. It does this by sending the move to the "Board", which checks its legality, updates the board state, etc. Space considerations do not permit us to describe full details of the solution frameworks for our two stereotypes. Interested readers may contact the authors for further details.

We have used this solution framework and implemented several games: Tic-Tac-Toe, Connect Four, a variation of it which we called "Make a Buck", Othello (Reversi), etc. Once the solution framework was written, we found that it was easy to implement specific games (including GUI's). Some times it took us just a couple of hours to adapt the software to play a new game. One of our graduates, currently a free-lance software developer, used our instruction manual on this stereotype and solution framework and developed a two-human-player version of the Checkers game in a couple of days. Further, with some assistance from one of the authors, he was able to develop the version with a computer player with two days of work. The heuristic used to compute the goodness value in this

version is rather overly simplistic. As a result, the game, at the time of this writing, is easily beaten!

We plan to use this stereotype in our AI class in Spring 2008. The representative problem would perhaps be Tic-Tac-Toe, and projects for the Connect Four, or the Othello/Reversi game could be assigned. Another possible assignment here could be to evaluate alternative heuristics used to determine the "goodness value" of the board. With our framework, one simply needs to replace the "goodnessValue()" method to implement another heuristic.

Conclusion and Future Work

In recent years, CS major/minor programs are facing serious challenges from low enrollment. The perception that CS is just "programming" and therefore is not intellectually exciting is a major issue to be addressed. Writing game-playing and puzzle-solving programs provides a good opportunity for students to experience the excitement in programming. In this paper, we have discussed the "Problem Stereotypes and Solution Frameworks" approach that enables the teaching of simplified SE techniques in CS1/2. A full discussion of these techniques is traditionally done in a final year SE course. We have described how this same approach can be extended to the AI course, enabling the students to program reasonably complex puzzles and games. The first AI stereotype we present, *State Space Search* for puzzle solving, was used during the Spring 2007 semester and met with reasonable success. We have developed a *Two-Person-Zero-Sum-Game* stereotype recently, and used it to create many simple board game-playing programs. We believe that our approach can be helpful to both AI and SE courses – with the latter focusing on design and development of an object-oriented solution framework for the stereotype, and the former focusing on programming details of a particular application.

References

Becker, K. 2001. Teaching with Games: The Minesweeper and Asteroids Experience. *The Journal of Computing Science in Colleges* 17(2): 23-33.

Booch, G. 2007. *Object-Oriented Analysis and Design with Applications, 3rd Edition.* Reading, Mass: Addison-Wesley.

Faltin, N. 1999. Designing Courseware on Algorithms for Active Learning with Virtual Board Games. In *ACM SIGCSE Bulletin, Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE '99)*, 135-138. New York, NY: ACM.

Fowler, M. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition.* Reading, MA: Addison-Wesley Professional.

Goschnick, S., and Balbo, S. 2005. Game-first Programming for Information Systems Students. In *Proceedings of the Second Australasian Conference on Interactive Entertainment (IE 2005)*, 71-74. Sydney, Australia: Creativity and Cognition Studios Press.

Luger, G. F. 1998. *Artificial Intelligence, III ed.* Reading, Mass: Addison Wesley.

Mitra, S., Rao, T.M., and Bullinger, T.A. 2005. Teaching Software Engineering Using a Traceability-Based Development Methodology. *The Journal of Computing Sciences in Colleges* 20(5): 249-259.

Rao, T.M. 2003. Using Java to teach AI. *The Journal of Computing Sciences in Colleges* 18(3): 114-125.

Rao, T.M., Mitra, S., Canosa, R., Marshall, S., and Bullinger, T. 2007. Problem Stereotypes and Solution Frameworks – A Design First Approach for the Introductory Computer Science Sequence. *The Journal of Computing Science in Colleges* 22(6): 56-64.

Russel, P. and Norvig, P. 2003. *Artificial Intelligence, A Modern Approach*. Upper Saddle River, NJ: Prentice Hall.

Sindre, G., Line, S., and Valvag, O.V. 2003. Positive Experiences with an Open Project Assignment in an Introductory Programming Course. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, 608 - 613. Washington, DC: IEEE Computer Society.

Wirfs-Brock, R., and McKean, A. 2002. *Object Design: Roles, Responsibilities and Collaborations*. Reading, Mass: Addison-Wesley Professional.