IMPLEMENTING THE INTELLIGENT SYSTEMS KNOWLEDGE UNITS OF COMPUTING CURRICULA 2001

Ingrid Russell[1] and Todd Neller[2]

Abstract *- Computing Curricula 2001 (CC-2001) presents a set of curricular recommendations for undergraduate computer science programs. CC-2001 presents a computer science body of knowledge and identifies a list of core topics/units within each component body of knowledge that a computer science program should require. While some of these core units span hours that warrant or are equivalent to a full course, the core units for other areas are significantly less. This paper presents our experiences with integrating Intelligent Systems (IS) core units of CC-2001 into the undergraduate curriculum through the more traditional core courses such as discrete mathematics, data structures, and algorithms, thus eliminating the need to require a full course in the area in departments with various constraints that prevent this from being possible.*

Index Terms - artificial intelligence, computer science education, computing curricula 2001, curriculum development, data structures, intelligent systems.

INTRODUCTION

The field of Artificial Intelligence (AI) provides tools and techniques for solving problems that have been difficult to solve by other means. Typical AI problems involve the exploration of a large search space to find a solution. In most cases the search space is so large that a full search for an optimal solution is not possible. In addition, such problems involve a need to have some knowledge about the environment so inferences and decisions can be made. AI tools include heuristic search methods and planning algorithms, as well as formalisms for representing knowledge and reasoning.

Computing Curricula 2001 (CC-2001) [1] identifies Intelligent Systems (IS) core areas that each computer science student should be familiar with, thus highlighting that such topics are now fundamental in the computer science curriculum. In the context of CC-2001, *Intelligent Systems* (IS) is a synonym for *Artificial Intelligence* (AI). Throughout the paper, we will use these terms interchangeably.

Specifically, CC-2001 recommends ten hours in the area of Intelligent Systems (IS). The IS core topics are (1) Fundamental issues in intelligent systems, (2) Search and constraint satisfaction, and (3) Knowledge representation and reasoning. For background reading on these topics, we recommend Russell and Norvig's text [2]. The relevant corresponding chapters are (1) chapters 1, 26, (2) chapters 3-6, and (3) chapters 7-9, 13-14.

While a select number of universities have been able to offer the fairly traditional curriculum that requires several courses in various areas of computer science including AI, a number of smaller programs in mostly teaching-oriented liberal arts colleges are not able to do so and may have to deal with a condensed curriculum model. This is due to factors including student backgrounds and preparedness, overcrowded curriculum, and limited resources. At such institutions, if offered at all, AI has been offered as an elective only occasionally. With CC-2001 requirements, such colleges have to be creative in finding a way to integrate these units in a condensed curriculum. This can be accomplished though the merging of topics from different core areas of computer science into one course. Alternatively, topics can be integrated into traditional courses typically included in each curriculum.

This paper presents ways to integrate the Intelligent Systems (IS) core units into the traditional introductory courses on discrete structures, data structures, and algorithms. For example, search concepts and techniques can be covered in a data structures course when discussing tree data structures, or in a traditional algorithms course where coverage can be expanded to include a discussion of heuristic search and searching huge graphs and trees that are only implicitly defined. Such coverage can easily be incorporated into the data structures course without sacrificing any of the major topics typically covered in such a course.

We will consider each of the three major IS core topics in the sections below, providing suggestions for productive educational experiences that meet the teaching objectives of these topics while strengthening the core curriculum. With emphasis on learning from experience, we provide descriptions of in-class and homework exercises to aid in the integration of the Intelligent Systems core areas.

FUNDAMENTAL ISSUES IN AI

CC-2001 recommends 1 core hour addressing fundamental issues in AI. These include philosophical questions as well as a discussion of optimal reasoning versus human-like reasoning and optimal behavior versus human-like behavior. Since CC-2001 also recommends 16 core hours in the area of Social and Professional (SP) Issues, the 1 hour on AI issues can be easily integrated alongside the social and professional issues topics. Professional and ethical issues dis-

[1] Ingrid Russell, University of Hartford, Department of Computer Science, West Hartford, CT 06117, irussell@hartford.edu, (860) 768-4191, (860) 768-5244 (fax).
[2] Todd W. Neller, Gettysburg College, Department of Computer Science, Gettysburg, PA 17325, tneller@gettysburg.edu, (717) 337-6643.

0-7803-7444-4/03/$17.00 © 2003 IEEE **November 5-8, 2003, Boulder, CO.**
33rd ASEE/IEEE Frontiers in Education Conference

cussion can then be extended to address AI decision-making systems and the additional risks and philosophical issues these raise. A software engineering course would be a reasonable place for this material when a full course is not possible.

KNOWLEDGE REPRESENTATION AND REASONING

The knowledge representation and reasoning core area of CC-2001 consists of the following topics with a minimum coverage of 4 hours:

- Propositional and predicate logic
- Resolution and theorem proving
- Nonmonotonic inference
- Probabilistic reasoning
- Bayes' theorem

Discrete mathematics is a fundamental course in a computer science program and CC-2001 recommends 43 core hours in Discrete Structures. CC-2001 calls for conceptual coverage of these topics rather than their programmatic implementation. All topics in the knowledge-representation and reasoning area are typically either covered in a discrete math course or are natural additions to existing topics. For example, predicate logic is listed as a topic in a core unit of Discrete Structures, and resolution is a natural addition. The same applies to probabilistic reasoning and Bayes' theorem, which are already listed as topics in a core unit of the Discrete Structures area.

The third core area in IS is Search and Constraint Satisfaction. The remainder of the paper will concentrate on the integration of this area into a data structures and algorithms course.

SEARCH AND CONSTRAINT SATISFACTION

The search and constraint satisfaction core area consists of the following topics with a minimum total coverage time of 5 hours:

- Problem spaces
- Brute-force search (breadth-first, depth-first, depth-first with iterative-deepening)
- Best-first search (generic best-first, Dijkstra's algorithm, A*, admissibility of A*)
- Two-player games (minimax search, alpha-beta pruning)
- Constraint satisfaction (backtracking and local search methods)

Search is a fundamental problem-solving mechanism in AI. Three types of AI search problems are single-agent problems, two-player games, and constraint satisfaction problems. The objective here is to cover problem-solving as search in a problem space. Given a problem, students should be able to formulate a problem space associated with it.

One of the objectives listed in CC-2001 is the understanding of the problem of combinatorial explosion and search in combinatorial problems. The implementation of the Fibonacci function can serve as an introduction to these concepts. By writing a recursive implementation of this function and studying its performance as n varies, students can find out how long it takes to compute the function for small values of n (e.g. 100). The classic n-queens problem can be used to illustrate complexity issues and combinatorial explosion. There are n^2 choose n possible queen configurations ignoring symmetries. Students can implement a backtracking solution to the n-queens problem, and run it for various values of n until they run out of time.

Brute-Force Search

Breadth-first search and depth-first search are both examples of brute-force (a.k.a. uninformed) search. Depth-first search pursues a line of node-expansion and backtracks to a previous node only when it reaches a dead end. Breadth-first search generates all nodes at depth n before generating nodes at depth n+1.

When depth-first and breadth-first searches are covered in a data structures course, they typically are applied to applications on explicitly defined graphs. Since the entire data structure is stored in memory, the concepts of searching in the large and memory limitations do not apply. However, this coverage can be easily expanded to include the type of applications AI deals with which involve searching in the large and where the graph can be infinite and only implicitly defined. This will allow for the treatment of search algorithms in an AI setting.

The Eight-Puzzle Problem

The classic eight-puzzle problem serves as a good application for illustrating these AI concepts, while at the same time giving students practice with data structure topics such as queues, stacks, and trees. In the eight-puzzle problem, a board consists of 8 tiles numbered 1 through 8 and an empty tile in a 3×3 grid. One may move any tile into an orthogonally adjacent empty tile, but may not move outside the board or diagonally.

The goal is to reach the following final goal configuration:

```
1 2 3
8   4
7 6 5
```

The problem is to find a sequence of moves that transforms the board from some starting configuration to the goal configuration. Although any starting configuration may be chosen at random, it is often instructive to work with problems with known optimal solutions. For this purpose, one may generate initial starting states with varying average optimal solution length by making n random moves from the goal configuration. Since moves are reversible, one then

knows that the optimal solution sequence has at most n moves.

One nice feature of the eight-puzzle problem is that it is *scalable in difficulty*. By making the grid $n \times n$ with n^2-1 tiles, we can vary the difficulty of the problem and experience the combinatorial explosion of search. There are n^2! possible eight-puzzle configurations ignoring symmetries. Analyzing the explosion mathematically, and experiencing the change in runtime are two very different learning experiences. We recommend *both* and will later recommend experimental search exercises making use of the n^2-1 – puzzle problem.

A Taste of AI: Brute-Force Search

In this section, we present experiences with additional assignments that we used in a data structures course to introduce brute-force (a.k.a. uninformed) search techniques. The instructor can vary the level of guided versus independent work as appropriate for the student skill level and budgeted hours.

Brute-force search is perhaps the best and easiest topic to integrate into a CS core curriculum (e.g. CS1/2, data structures). Among the chief benefits are (1) a strong motivating example for object-oriented design, (2) an example application for stacks and queues, (3) excellent exercise in recursive thinking, (4) a good illustration of the relationship between stack-based and recursive algorithms, and (5) an outstanding opportunity to demonstrate design tradeoffs between time, space, *and* quality of result.

We first supply students with "starter code" to both provide guiding examples and make efficient use of time. The starter code includes (1) abstract classes `Searcher` and `SearchNode` (see Appendix I), defining what fields and methods are common between search algorithms and the search nodes they manipulate, (2) skeletal unimplemented classes for uninformed searches with comments that outline the algorithms, and (3) a complete implementation of `SearchNode` classes for two simple search problems.

The first search problem is called the Bucket Problem: Given a 5-unit bucket and a 3-unit bucket, how can one measure precisely 4 units? Initially, both buckets are empty. Possible operations are to fill/empty the 1st/2nd bucket, or pour the contents of one bucket into the other until one is empty or the other is full.

This problem is particularly good for illustrating *state spaces* as one can represent the state simply as a pair of integers corresponding to the contents of the two buckets. The size of the state space is small enough to display completely. Also, like the eight-puzzle, it can be used to illustrate the problem of *repeated states* in search. It is possible to repeat states in a sequence of moves or reach the same state by different sequences of moves.

The second problem is the familiar triangular peg solitaire problem. A 5-on-a-side triangular grid of holes is filled with pegs with the exception of one of the central holes. Possible operations are *jump removals*, where a peg is linearly "jumped" over another peg to an empty space beyond. The peg that is jumped over is removed from the board. The goal is to remove all pegs except one. Since the number of pegs is reduced by one with each move, it is impossible to repeat states in a sequence of moves.

Search nodes for these two problems are implemented as subclasses of a `SearchNode` class. Two important methods of the `SearchNode` class are `isGoal`, a goal test, and `expand`, which returns a list of possible successor nodes. In providing two complete implementations of these classes, we provide a common vocabulary for discussing the code and examples to facilitate student generalization and instantiation. After explaining these implementations, we then ask students to implement a third problem that is scalable in difficulty (e.g. the n^2-1 tile puzzle). We'll refer to this as the *scalable problem.*

In *Artificial Intelligence: a modern approach* [2], Russell and Norvig generalize search according to the data structure that is used to select the next node for examination. This general iterative form may be expressed simply as follows:

(1) Insert the initial search node into the data structure.
(2) While the data structure contains search nodes:
    a. Remove a node from the data structure.
    b. If this node is a goal node, terminate with success.
    c. Expand the node and insert its successors into the data structure.
(3) Terminate with failure.

In the context of a data structures course, we assume that students have already implemented queue and stack classes. Given a skeletal implementation of the breadth-first search class, we ask students to implement breadth-first search in the iterative form above using their queue class.

(1) Enqueue the initial search node into the queue.
(2) While the queue contains search nodes:
    a. Dequeue a node from the queue.
    b. If this node is a goal node, terminate with success.
    c. Expand the node and enqueue its successors into the queue.
(3) Terminate with failure.

Students can then solve the bucket and peg solitaire problems and simple cases of the scalable problem. In all cases, students observe and explain why a shortest path solution is found.

Next, students similarly implement depth-first search using a stack as the search node data structure.

(1) Push the initial search node onto the stack.
(2) While the stack contains search nodes:
    a. Pop a node from the stack.
    b. If this node is a goal node, terminate with success.

c. Expand the node and push its successors onto the stack.

(3) Terminate with failure.

Students can then apply depth-first search to the different search problems. In the case of the peg solitaire problem, the algorithm will work. We then ask students to explain why depth-first search will always work with peg solitaire problems. In many other problems (including the bucket problem), the search will go into an infinite loop cycling between states and running out of stack memory. Students are then asked to observe this behavior and place print statements to understand the failure fully.

Analyzing these algorithms, one can demonstrate that depth-first search trades off the *completeness* and *optimality* of breadth-first search for a *reduction in space complexity* from exponential ($O(b^d)$) to polynomial ($O(bd)$).

Optionally, one can have students implement depth-first search recursively and use this as a means to show how any recursive algorithm may be implemented as a stack-based algorithm.

Next, students can modify their depth-first search code to perform a depth-limited search. They can then experiment with different depths to show that a depth-limited search can be used to find the optimal solutions for the bucket problem.

Finally, students can implement iterative-deepening search by iterative application of depth-limited search. They can then be asked to observe its behavior and analyze the implications for completeness, optimality, time, and space. With iterative-deepening, we have a beautiful illustration of trade-offs between time, space, and solution quality. Whereas depth-first search trades off quality for space, iterative deepening depth-first search regains this quality by trading off time by allowing nodes to be re-expanded. Giving close attention to these three algorithms and their trade-offs is perhaps the most valuable aspect of these exercises.

Best-First Search and the Role of Heuristics

Very often AI problems involve a large search space where searching the full space is not feasible. In some cases when one is dealing with difficult problems, one needs to find a compromise between optimal solutions and being able to find a sufficiently good solution most of the time. As a result, knowledge about the unexplored region is used to guide the search, thus avoiding unproductive search. Heuristics are used to guide the search among the most promising states, thus significantly decreasing the number of nodes searched. However, not all heuristics and algorithms guarantee an optimal solution.

Whereas brute-force search techniques only consider factors such as node depth or known costs of paths, best-first search techniques also make use of heuristic estimates of the cost from any given node to a goal node. While these estimates are often difficult to make, one can easily demonstrate that even poor estimates greatly improve the performance of search. It is also very interesting to point out the necessity of *computationally efficient* heuristic estimates so that search savings are not more than offset by the overhead of computing such estimates. Exercises in the design of heuristics are rich ground for understanding tradeoffs in algorithm design.

Students should understand how heuristics aid in the search for a solution in the search space and that it is more often the case that one runs out of memory than time.

Students can then revise their previous eight-puzzle search program by implementing one of the following two heuristic functions or by using a heuristic of their own.

- Tiles-out-of-place heuristic: counts the number of tiles that are not in their correct positions.
- Manhattan distance: the sum of the *x*- and *y*-distances of each tile to its correct position.

Students can do various kinds of experimentation, such as comparing the performances of some of the heuristic search algorithms to each other or to some of the uninformed search techniques. Specifically, students can be asked to study and compare the performances based on one or more of the following criteria:

- Size of problem space examined
- Length of solution path returned by the algorithm
- The number of nodes generated
- Computing time for a solution
- Whether or not an optimal solution is found

The following section presents coverage of additional best-first programming projects that have been assigned to illustrate these AI core units while at the same time covering traditional data structures topics.

A Taste of AI: Best-First Search

The introduction of best-first search is quite straightforward in the context of the previous brute-force search exercises. Whereas breadth-first search was implemented with a queue and depth-first search was implemented with a stack, best-first search is implemented in the same general framework using a *priority queue*.

(1) Enqueue the initial search node into the priority queue.

(2) While the priority queue contains search nodes:
   a. Dequeue a node from the priority queue.
   b. If this node is a goal node, terminate with success.
   c. Expand the node and enqueue its successors into the priority queue.

(3) Terminate with failure.

It is then impressed upon students that the *cost* of a solution need not be equivalent to the depth of the goal node. A node n is prioritized in the priority queue by $f(n)$, the sum of $g(n)$, the cost to reach n, and $h(n)$, the estimated cost to reach a goal node from n. When $h(n)$ is 0, we call this uni-

form-cost search, and when additionally $g(n)$ is equal to the depth of n, we have breadth-first search.

Various heuristics can be explored, as already noted with the 8-puzzle. With an *admissible* heuristic (i.e. one that never over-estimates), best-first search is called A*-search. Iterative-deepening A* is simply a variation of iterative-deepening depth-first search where we iteratively increase an $f(n)$-limit rather than a depth-limit.

Thus, a rich array of best-first search methods may be developed incrementally by (1) the introduction of a priority queue, (2) small modifications to brute-force search code, and (3) the addition of a heuristic function h, as illustrated with the 8-puzzle.

Two-Player Games

Two-player games provide an opportunity for interesting applications following coverage of brute-force and best-first search. Among the chief benefits are (1) a motivating example for object-oriented design, (2) exercise in recursive thinking, and (3) design for hard real-time constraints.

We have found that students greatly enjoy competitive game-playing assignments. In one such mancala assignment, we provided (1) an interface to real-time game tournament software, (2) a complete implementation of a mancala game state class with a trivial heuristic evaluation function (score difference), and (3) an implementation of minimax search.

First, we have students augment minimax search with alpha-beta pruning. Then, after having students play several games of mancala, we challenge students to come up with a better heuristic evaluation function. This provides an excellent opportunity to point out how an overly complex heuristic can easily incur computational costs such that any potential benefit is offset in the face of real-time constraints.

Constraint Satisfaction

The n-queens problem is a simple constraint satisfaction problem where n pieces must be placed on an $n \times n$ grid such that none share the same row, column, or diagonal. Chronological backtracking constraint satisfaction may be viewed as a depth-first search where the start node is an empty grid, successor states are legal placements of the next unplaced queen, and a goal state is a successful placement of all n queens.

Not only can this be easily implemented using previous algorithms, but it also provides a great opportunity for illustrating the importance of state space formulation. Consider the significant difference between the following two formulations: (1) allow the next queen to be placed in any location that does not violate constraints, or (2) allow the next queen to be placed in any location *of the leftmost unoccupied column* that does not violate constraints. In the first formulation, we introduce a combinatorial explosion of ways the same states can be reached. In the second, our added constraint only permits a single path to any given state. This latter formulation also lends itself to a physical demonstration with a chess set (using pawns as queens).

CONCLUSION

We presented our experiences integrating the Intelligent Systems core units of CC-2001 into the undergraduate computer science curriculum through the more traditional introductory courses on discrete structures and data structures and algorithms. We presented projects that, while illustrating course concepts, aided in the integration of the core IS units into these courses. The projects were well received by students and helped introduce an important area in computer science. By using projects involving games, we provided additional motivation for students. While a full course in Artificial Intelligence will allow for more coverage of AI concepts, smaller programs with limited resources that are not able to offer such a course can benefit from using a similar approach to meet CC-2001 guidelines in this area.

APPENDIX I: SEARCHER AND SEARCHNODE

```
/** Searcher.java - an interface for AI
  * searcher classes. */

public interface Searcher {
    /** search - perform a search from a
     * given root, returning whether or
     * not a goal node was found. */
    boolean search(SearchNode root);

    SearchNode getGoalNode();

    int getNodeCount();

    void printGoalPath()
}

/** SearchNode - an abstract class for
  *brute-force search nodes. */

public abstract class SearchNode
  implements Cloneable {

    /** parent of node; null if root */
    SearchNode parent = null;

    /** search depth of node
    int depth = 0;

    /** Creates a SearchNode instance
     * and sets it to an initial search
     * state. */
    public SearchNode() {}

    /** isGoal - goal test for node */
```

```
    public abstract boolean isGoal();

    /** expand - return a (possibly
     * empty) Vector of this node's
     * children. */
    public abstract Vector expand();

    /** childClone - returns a clone of
     * this node that has been made a
     * child of this node and has a
     * depth one greater. */
    public SearchNode childClone()
    {
        SearchNode child
          = (SearchNode) clone();
        child.parent = this;
        child.depth = depth + 1;
        return child;
    }

    public Object clone() {
        // deep clone of node
    }
}
```

REFERENCES

[1] G. Engel and E. Roberts, Eds., *Computing Curricula 2001 Computer Science: final report December 15, 2001*, Los Alamitos, CA: IEEE Press, 2002.

[2] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach, 2nd ed.*, Upper Saddle River, NJ: Prentice-Hall, 2003.