Functional Video Games in CS1 II

From Structural Recursion to Generative and Accumulative Recursion

Marco T. Morazán

Seton Hall University, South Orange, NJ, USA
morazanm@shu.edu

Abstract. The use of video games to teach introduction courses to programming and Computer Science is a trend that is currently flourishing. One of the most successful and promising approaches uses functional video games to get students interested and engaged in programming. This approach is successful, in part, because functional video games provide a domain of interest to most Computer Science undergraduates and remove the need to reason about designing state-based programs. A plethora of examples exist that have students develop games exploiting structural recursion which resemble such classics as Space Invaders and Snake. Once students master the basics of structural recursion the time comes to move beyond structural recursion to generative and accumulative recursion. It is up to the instructor to harness the enthusiasm and appetite that students have to develop more video games. This requires finding games that require the generation of subproblems in the same class as the input problem or that require accumulators to be successfully played or solved. This article presents a road map to make the transition from structural recursion to accumulative recursion using the N-puzzle problem as motivation to capture student enthusiasm and exploit what they have learned about program design. The N-Puzzle was also chosen to demonstrate that informed heuristic search strategies, traditionally the domain of undergraduate courses in Artificial Intelligence, are within the grasp of CS1 students. With proper guidance, CS1 students can reason such an algorithm into existence instead of simply using a textbook to study such algorithms. If the work described in this article is replicated elsewhere, there is no doubt that it will be an exciting time for Computer Science education and it will elevate the relevance of functional programming in the minds of future CS professionals.

1 Introduction

Based on the teaching philosophy of *program by design* (PBD) put forth in the textbook *How to Design Programs* (HtDP) [2], the use of functional video games to teach introduction courses to programming and Computer Science is a trend that is currently flourishing. At the heart of the PBD philosophy is the *design recipe*–a series of steps that students can follow to design and write programs.

R. Peña and R. Page (Eds.): TFP 2011, LNCS 7193, pp. 146–162, 2012.

These steps include the development of data definitions based on problem analysis, the development of contracts and function headers, the development of function templates for all data definitions, the specialization of function templates to create functions, and the development and running of tests. One of the most successful and promising implementation approaches to a PBD-based course uses a hierarchy of successively richer student languages and functional video games to get students interested and engaged in programming. PLT's DrRacket [4] integrates such a hierarchy of student languages for use in conjunction with HtDP. The reader should note that students are not taught Racket, but do learn Racket-like syntax on a need-to-know basis. This approach is successful, in part, because the student languages allow for the generation of error messages that are meaningful for beginners. This approach is also successful, in part, because functional video games provide a domain of interest to most Computer Science undergraduates and remove the need to reason about designing state-based programs. A plethora of examples exist that have students develop games exploiting structural recursion which resemble such classics as Space Invaders and Snake [1,6].

At the beginning of an introduction course, the focus is on solving problems using primitive data, structures, and structural recursion. Once students master the basics of structural recursion, the time comes to explore other forms of recursion such as generative and accumulative recursion. In generative recursion, the subproblems generated are not derived from the data structure employed and are in the same class as the original problem (a typical example is quicksort). One of the important consequences of this that beginners must realize is that programs using generative recursion are not guaranteed to terminate like programs that employ structural recursion. Thus, generative recursion requires the development of termination arguments. In accumulative recursion, one or more accumulators are added as parameters to a function designed using structural or generative recursion to capture information that, otherwise, would be lost between recursive calls (a typical example is finding a path between two nodes in a cyclic graph). An important consequence of this for beginners is that they must realize that for each accumulator an accumulator invariant must developed to describe the value of the accumulator. The code students write must guarantee that the accumulator invariant holds for every recursive call. It is up to the instructor to harness the enthusiasm and appetite that students have to develop more video games to motivate these topics. This requires identifying games that can not be played nor solved by only using structural recursion. It is important to note, however, that the goal is not to make students masters at developing video games. Instead, the goal is to make students interested in generative and accumulative recursion by showing them how they are needed and/or used in a video game. Surprisingly, there are not many examples in an HtDP-based curriculum of video games that require students to go beyond structural recursion.

This article advocates the position that video games ought to be used to motivate the need to study generative and accumulative recursion in the CS1 classroom. It presents an example on how to make the transition from structural

Fig. 1. A Random 3-Puzzle Board

Fig. 2. Sample Winning 3-Puzzle Board

recursion to generative and accumulative recursion using the N-puzzle problem as motivation to capture student enthusiasm. This road map is used in the curriculum at Seton Hall University that has been previously described [6]. The primary goal is to introduce these topics in a context that exploits and reinforces lessons on program by design, structural recursion, and abstraction. Secondary goals are to expose students to ideas that they may encounter in upper-level courses such as heuristics in an Artificial Intelligence course and the use of random number generators. Section 2 briefly describes the N-puzzle game. Section 3 describes the first encounter of students with the N-puzzle game in the classroom and discusses opportunities the game presents to reinforce the lessons of program by design using structural recursion and abstraction. Section 4 discusses an initial strategy to finding a solution leading to the need for generative recursion. Section 5 discusses how the need for accumulators arises and how accumulative recursion is used in the N-puzzle game. Section 7 discusses related work and Section 8 draws some conclusions and briefly outlines future work.

2 The N-Puzzle Game

The N-puzzle game is one that is likely to be familiar to an international milieu of students and is simple enough that students can easily grasp how the game works. It consists of an $N \times N$ board with $N^2 - 1$ tiles[1] and an empty square or blank space that does not contain a tile. Each tile contains some form of symbolic or numeric data. Figure 1 displays a sample board using numeric tiles for the 3-puzzle[2] game in which the empty space is at the center of the board. Every N-puzzle game must also define a winning board. That is, a board that defines the solution to the puzzle. Figure 2 displays the traditional winning board for the numeric 3-puzzle problem.

A player can move tiles by swapping the blank space with one of its neighbors (i.e., right, left, up, or down). The goal of the game is to make a sequence of

[1] The choice of a square board is arbitrary, but facilitates developing a program.
[2] It is also common to refer to this version of the puzzle as the 8-puzzle.

```
A board is either:
    1. empty
    2. (cons number b), where b is a board

Template for functions on boards:
(define (f-on-board a-board)
    (cond   [(empty? a-board) ...]
            [else ...(first a-board)...(rest a-board)]))
```

Fig. 3. Data definition for boards and a template for functions on boards

moves that leads to the winning board. A player, of course, at some point during the game may feel stuck and the game should provide a mechanism, like a help button, to ask the computer to make the next move. The help button, of course, requires the program to first solve the puzzle before making a move towards the solution on behalf of the player.

To make the game more challenging and more interesting the game can be parameterized with a constant N. In this manner, students are free to make the board larger or smaller according to the level of the challenge they desire. A CS1 instructor should note, however, that as N increases the effective use of the help button decreases which can discourage some students.

3 The First Encounter with the N-Puzzle Game in CS1

Students that are presented with the N-puzzle game have gone through the first four parts of HtDP that cover program by design with structures, structurally recursive data types, and abstraction. They have experience designing programs that process, for example, lists and trees as well as familiarity with basic abstraction patterns that involve the use of higher-order functions such as map and filter.

When students are first presented with the N-puzzle game, they are asked what is changing while the game is played and how it can be represented. This leads to defining a board as a list of numbers[3] and to a template for functions on boards both of which are displayed in Figure 3. This brings the N-puzzle game into a realm that is familiar to the students and provides an opportunity to reinforce lessons on structural recursion.

To get students started, the first tasks they are asked to solve can be done using structural recursion and/or abstraction such as building the representation of the winning board, finding the position of the empty space, and swapping two tiles (eventually used to make moves). The solutions presented may vary with some students defining such functions using structural recursion and some

[3] The number in position i of the list corresponds the tile in row (quotient i N) and in column (remainder i N).

```
(define WIN (build-list N (lambda (n)
                            (cond [(< n (- N 1)) (+ n 1)]
                                  [else 0])))))

; get-blank-pos: board --> number
; Purpose: To find the position of the blank
(define (get-blank-pos l)
  (cond [(empty? l) (error 'get-blank-pos "Blank not found")]
        [(= (car l) BLANK) 0]
        [else (add1 (get-blank-pos (cdr l)))]))

; swap-tiles: board natnum natnum --> board
; Purpose: To swap the given tiles in the given board
(define (swap-tiles w i j)
  (build-list N (lambda (n)
                  (cond [(= n i) (list-ref w j)]
                        [(= n j) (list-ref w i)]
                        [else (list-ref w n)]))))
```

Fig. 4. Auxiliary Functions Developed Using Structural Recursion and Abstraction

students using higher-order functions. Typical solutions for the game with numeric tiles are displayed in Figure 4.

The initial encounter with the N-puzzle game also provides an opportunity to perform data analysis that leads to the realization that more than structural recursion is required to implement the help button. Students are asked what does it mean to find a solution when the player requests the computer to make the next move. After some discussion, it becomes clear that finding a solution is finding a sequence of moves from the current board to the winning board. Students, in general, can grasp without too much trouble the idea that finding such a sequence of moves for board b means finding a solution for one of the possible successors of b, $child_b$, obtained by making a single move and adding the move that takes b to $child_b$. The question then becomes which move will be chosen to generate the child of b that is to be explored.

Observe that such a strategy is no longer in the domain of structural recursion. Structural recursion guarantees that the size of the subproblems (i.e., finding a solution starting from $child_b$) are smaller than the problem of finding a solution to the original problem (i.e., finding a solution starting at b) and are derived from the structure of the input. This is not the case, because in general some sequences starting at b are infinite as are sequences starting at any $child_b$ and $child_b$ is not used to build b. The question then becomes how do you solve problems that generate subproblems that are not guaranteed to be smaller than the original problem and are not part of the structure of the original problem. At this point, students have entered the realm of generative recursion by simply trying to implement a video game.

4 Finding a Solution

After using the N-puzzle game to discover the need for generative recursion, students are given several examples on how to design programs based on generative recursion. Examples outlined in HtDP include quicksort, fractals, binary search, Newton's method, and backtracking algorithms such as traversing a graph to find a path from node A to node C. Of these, the most relevant to finding a solution to an N-puzzle are the backtracking algorithms, because traversing a graph with cycles can lead to a path of infinite length precisely in the same manner that some sequences of moves are infinite in the N-puzzle problem. HtDP presents a solution to find a path from node A to node C in an acyclic graph using a depth-first traversal and postpones finding the solution for a graph with cycles to motivate accumulative recursion.

In the N-puzzle game, of course, we are for the most part unable to restrict our sequences of moves to those that are finite. Students, in general, are not aware at this point of this and can be led to develop a solution that seems reasonable. Class discussion is focused on how to choose a successor of the current board to find a solution. This presents the opportunity to introduce beginning students to heuristics. A heuristic can be used to choose which child of b is chosen to explore for a solution. It is important to remark to students that a heuristic is a rule that estimates how many moves away the current board is from the winning board and that is used hoping it will lead to a solution. At this point, most students will have no way to judge this statement and simply trust the professor. This trust opens the door for reinforcing lessons on the importance of testing and careful design in programming. As the reader knows, this approach is destined to immediate failure, but also to triumph after the process of iterative refinement is started.

There is a simple heuristic students can understand and implement for the N-puzzle problem. The heuristic chooses to explore the child of b that has the smallest Manhattan distance. The Manhattan distance of a board is the sum of how far away each tile is from its correct position. For example, the Manhattan distance of the board in Figure 2 is 0 given that all tiles are in the correct position. In Figure 1, tile 1 is in the right position and contributes 0 to the Manhattan distance while the blank space, in position 4 and whose correct position is 8, contributes 2 to the Manhattan distance. The code to compute the Manhattan distance of a board is displayed in Figure 5. Observe that the code only requires arithmetic and structural recursion on natural numbers which provides the opportunity to reinforce material students have already seen and to make this material relevant to their interests in video games.

Armed with the power of a heuristic, students can now delve into designing an N-puzzle solver to implement the help button. The basic idea is that given a board their program needs to return a non-empty list of boards, called a sequence, that contains all the boards in the sequence of moves from the given board to the winning board. These ideas lead quite naturally to the design of a depth-first search algorithm. If the given board is the same as the winning board,

```
; manhattan-distance: board --> number
; Purpose: To compute the Manhattan distance of the given board
(define (manhattan-distance b)
  (local
    [; distance: number number --> number
     ; Purpose: To compute the distance between the two tile positions
     (define (distance curr corr)
       (+ (abs (- (quotient curr (sqrt N)) (quotient corr (sqrt N))))
          (abs (- (remainder curr (sqrt N)) (remainder corr (sqrt N))))))
     ; adder: number --> number
     ; Purpose: To add all the distances of each tile
     (define (adder pos)
       (cond [(= pos 0) 0]
             [else (+ (distance (sub1 pos)
                                (correct-pos (list-ref b (sub1 pos))))
                      (adder (sub1 pos)))]))
     ; correct-pos: number --> number
     ; Purpose: To determine the correct position of the given tile
     (define (correct-pos n)
       (cond [(= n 0) (sub1 N)]
             [else (sub1 n)]))]
    (adder N)))
```

Fig. 5. Code for computing the Manhattan distance of a board

then the solution is trivial: a list containing the given board. Otherwise, create a list from the given board and the solution generated starting from the best child of the given board. The function to generate the children of a given board can either be done using structural recursion or map. It only entails swapping the blank space with its neighbors. Finding the best board from a list of boards also only requires structural recursion. A sample implementation is displayed in Figure 6.

The benefits of using the N-puzzle to reinforce lessons from structural recursion, to motivate generative recursion, and to capture the interest of students are likely to be self-evident to any instructor at the helm of a CS1 class. Clearly, this video game also provides the opportunity to introduce CS1 students quite naturally to depth-first search and to heuristics-based programming which is quite uncommon as far as the author knows. There are, however, two more benefits that deserve to be mentioned. These are reinforcing the value of testing and the value of iterative refinement. The instructor can strategically provide initial boards to test the game and the help button. The code in Figure 6 does, indeed, find a solution for some test boards while at the same time reveal that it fails to return a solution for some test boards. This leads to an exploration of why the program, which seems quite reasonable to most students, fails to return a solution for some boards and how it can be improved to guarantee that a solution is always returned (for a legal board).

```
; find-solution-dfs: board --> (listof boards)
; Purpose: To find a solution to the given board using DFS
(define (find-solution-dfs b)
  (cond [(equal? b WIN) (list b)]
        [else
         (local [(define children (generate-children b))]
           (cons b (find-solution-dfs (best-child children))))]))

; generate-children: board --> non-empty-list-of-boards
; Purpose: To generate a list of the children of the given board
(define (generate-children b)
  (local [(define blank-pos (get-blank-sq-num b))]
    (map (lambda (p)
           (swap-tiles b blank-pos p))
         (blank-neighs blank-pos))))

; best-child: non-empty-list-of-boards --> board
; Purpose: To find the board with the board with the smallest Manhattan
;          distance in the given non-empty list of boards
(define (best-child lob)
  (cond [(empty? (rest lob)) (car lob)]
        [else
         (local [(define best-of-rest (best-child (rest lob)))]
           (cond [(< (manhattan-distance (car lob))
                     (manhattan-distance best-of-rest))
                  (car lob)]
                 [else best-of-rest]))]))
```

Fig. 6. Code for depth-first search for a solution without backtracking

5 The Need to Remember Leads to Accumulators

The exploration of why the program fails to return a solution to some boards leads to the discussion of a situation like the one depicted in Figure 7. If the current board is the one in the root of the tree, it has two children both of which have a Manhattan distance of 18. The algorithm chooses the right child as the board to explore. This board has three children that, from left to right, have Manhattan distances of 20, 20, and 16. The rightmost child is chosen for exploration as it has the smallest Manhattan distance. At this point, all students can see the problem. The algorithm cycles through the same set of boards never choosing a different board to escape the cycle. In other words, students understand why there is an infinite recursion and why it is impossible to argue that the algorithm terminates as is required by the design recipe for programs based on generative recursion. Some readers may argue that developing a termination argument ought to always be done before implementing an algorithm. At Seton Hall, we have discovered that this is not always true. Much of it depends on the CS-maturity that students bring to the classroom. In our CS1 course, it is

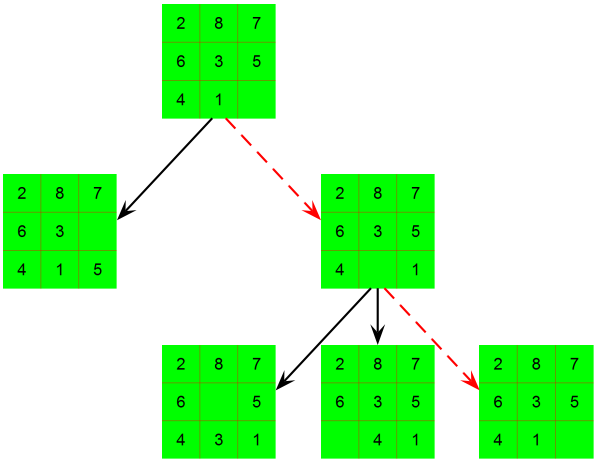

Fig. 7. Illustration of why a depth-first path (dashed) does not lead to a solution

assumed that students have little or no background in Computer Science when they start. For such students, theoretical termination arguments do not easily flow. Understanding why an implemented algorithm fails, on the other hand, presents an excellent learning experience and brings home the importance of developing termination arguments. We conjecture that as students gain experience it becomes easier for them to visualize, before implementation, termination arguments.

After understanding why the algorithm does not always terminate, students are guided to think that a solution to this problem requires that all sequences starting at the given board must be explored instead of choosing to only explore the sequence of best children. This requires that all paths explored so far be remembered. Through this analysis, students have entered the realm of accumulative recursion and this is used as motivation to return to HtDP and study how to design programs that exploit this new kind of recursion.

One of the functions students can develop while exploring how to design programs that use accumulative recursion is a function to present a player with an initial board to solve in the N-puzzle problem. This presents an interesting task, because not all possible orderings of tiles in a board are valid boards in the N-puzzle game. In the 3-puzzle game, for example, the ordering that has 1, 2, 3 in the 0^{th} row, 4, 5, 6, in the 1^{st} row, and 8, 7, 0 in the 2^{nd} row is an invalid board. The challenge, therefore, is to design a strategy to compute an initial board that does not simply randomly assign tiles to positions in the board. After some discussion, a natural strategy to follow is to start from the winning board and randomly make k valid moves. This strategy is a good one to choose in CS1 for three reasons. The first is that it provides students at this

```
; make-moves: natnum board --> board
; Purpose: To create a board by making the given number of moves
;          in the given board.
; ACCUMULATOR INVARIANT: b is the board created by making
                         (- NUM-INIT-MOVES nummoves) moves from WIN
(define (make-moves nummoves b)
  (cond [(= nummoves 0) b]
        [else (make-moves (sub1 nummoves) (make-a-move b))]))

; make-a-move: board --> board
; Purpose: To make a random move in the given board
(define (make-a-move b)
  (local [(define blank-index (get-blank-index b))
          (define neighs-indices (neighs-of blank-index))
          (define move-to-index
            (list-ref neighs-indices (random (length neighs-indices))))]
    (swap-tiles w move-to-index blank-index)))

(define NUM-INIT-MOVES 200)
(define INIT-BOARD (make-moves NUM-INIT-MOVES WIN))
```

Fig. 8. An implementation for creating an initial N-puzzle board

early stage in their studies with an example of where the use of randomness is useful. The second is that it requires an accumulator to "remember" the board created so far. That is, after every random move a new board is created and the new board needs to be used to make any further moves. The third is that it brings accumulative recursion into the domain of structural recursion on natural numbers–a familiar world for students that have followed an HtDP-based curriculum. This approach, for example, is implemented by students as displayed in Figure 8. The function make-moves is written using the design recipe for structural recursion on natural numbers to which an accumulator has been added. As per the design recipe for designing functions using accumulative recursion, students must develop an accumulator invariant. Although the invariant in this case may seem straightforward to an experienced programmer, its development by beginning students usually requires some coaching. One effective strategy is to have students trace an example, before writing any code, to help them visualize what characteristics of the parameters remain unchanged for every recursive call. For the function make-moves in Figure 8, for example, the parameter b is an accumulator. Initially, students are led to reason that b is the board obtained by making a number of random moves starting from WIN. This reasoning is then refined to precisely define the number of moves: (- NUM-INIT-MOVES nummoves). Students can now argue that for every recursive call a move is made and the number of moves is reduced by 1. Thus, they can conclude that the accumulator invariant holds for every recursive call and that when nummoves is 0 the initial board has been computed.

5.1 Developing a Breadth-First Solution

The heuristic-based depth-first N-puzzle solver assumed a solution can be found by always exploring the best successor of the current board. This assumption is removed and all possible sequences starting at a given board are explored. This requires that a list of all sequences generated so far be maintained. It is important during the exploration of this idea in the classroom to have students realize that this list of sequences must be maintained in order by length. Otherwise, the strategy may degenerate into a depth-first search that leads to an infinite recursion. The experienced reader will recognize that such a list is, in essence, a queue. It presents an opportunity to develop an interface for queues, but our success with having students reason about queues is mixed. CS1 students need to work on several queue-based solutions to different problems to internalize what a queue is. Therefore, we usually only mention queues in passing and allow students to structure their reasoning using a list of sequences ordered by length.

The implementation builds on the work done for the heuristic-based depth-first N-puzzle solver. The function find-solution-bfs takes as input a board, b, and returns a sequence from b to WIN. To accomplish this, a helper function, search-paths, is called that takes as input an accumulator that stores the list of all sequences generated so far. Initially, this list of sequences contains a single list that contains b. The function search-paths is a combination of generative recursion and accumulative recursion. Each time the function is called, it checks if the first board in the first sequence is WIN and, if so, it returns the first sequence. Otherwise, the successors of the first board in the first sequence are generated and a new sequence is generated for each successor by adding it to the front of the first sequence. To maintain the accumulator invariant, the list of sequences that does not include the first sequence is appended with the new sequences generated for the recursive call. A sample implementation is displayed in Figure 9.

Students must develop an accumulator invariant as well as an argument for termination. The accumulator invariant is developed, as mentioned above, during the exploration of the idea to search all possible sequences. The argument for termination hinges on having students realize that as paths get longer the number of moves required for one or more paths to reach the winning board gets smaller. Thus, the number of moves required to reach the winning board will eventually reach 0 for some path and the algorithm returns the appropriate sequence as long as the initial board is valid.

5.2 Refining the Solution: Deriving an A*-like Algorithm

The breadth-first N-puzzle solver does find a solution for any given board, but students soon discover that the help button is very sluggish and in some cases extremely so. The problem, of course, is that exploring all possible sequences starting at a given board is a great deal of work. Students can be led to realize

```
; find-solution-bfs: board --> lseq
; Purpose: To find a solution to the given board
(define (find-solution-bfs b)
  (local
    [; search-paths: lseq --> seq
     ; Purpose: To find a solution to b by searching all possible paths
     ; ACCUMULATOR INVARIANT:
     ;  paths is a list of all seqs generated so far starting at b from
     ;  from the shortest to the longest in reversed order
     (define (search-paths paths)
       (cond [(equal? (first (first paths)) WIN) (car paths)]
             [else
               (local [(define children (generate-children
                                          (first (first paths))))
                       (define new-paths (map (lambda (c)
                                                (cons c (first paths)))
                                              children))]
                 (search-paths (append (rest paths) new-paths)))]))]
    (reverse (search-paths (list (list b))))))
```

Fig. 9. A breadth-first N-puzzle solver

that the number of sequences being searched surpasses 2^9 after 10 moves and surpasses 2^{19} after 20 moves[4]. This provides an opportunity to expose students to the problems of exponential growth. At this point, students are asked if searching all possible sequences and searching all possible sequences at the same time is necessary. This is a difficult question for them to answer. Most students will say yes to both questions, because all possible sequences must be searched. In other words, most students at this level are unlikely to realize on their own that not all sequences need to be searched and that not all sequences that ought to be searched have to be simultaneously searched.

There are two main ideas that must be planted in students' minds. The first idea is that not every sequence needs to be explored. We draw on the experience obtained from the depth-first N-puzzle solver. If any successor, s, of a given board, b, has been explored (i.e., the successors of s have been generated), then the path through b to s need not be explored. The reason is that a sequence, of equal or shorter length, to s has already been generated. The second idea is that we can choose to explore the most "promising" sequence first instead of blindly exploring all possible sequences at the same time. This leads the class discussion back to the Manhattan distance heuristic as a mechanism for deciding which sequence is the most promising. The idea to always explore the most promising sequence first is one that students in CS1 can grasp and implement.

[4] 2^9 and 2^{19} are, respectively, the number of leaves in a binary tree that describes the search space after 10 and 20 moves if all boards only had two successors.

```
(define (find-solution-a-star b)
  (local
    [(define (find-best-seq seqs)
      (cond [(empty? (rest seqs)) (first seqs)]
            [else
              (local [(define best-of-rest (find-best-seq (rest seqs)))]
                (cond [(< (manhattan-dist (first (first seqs)))
                          (manhattan-dist (first best-of-rest)))
                       (first seqs)]
                      [else best-of-rest]))]))
     (define (search-paths visited paths)
       (local [(define bstseq (find-best-seq paths))]
         (cond [(equal? (first best-path) WIN) bstseq]
               [else
                 (local
                   [(define children
                            (filter (lambda (c) (not (member c visited)))
                                    (generate-children (first bstseq))))
                    (define new-seqs (map (lambda (c) (cons c bstseq))
                                          children))]
                  (search-paths
                    (cons (first bstseq) visited)
                    (append new-seqs (rem-path bstseq paths))))])))]
  (reverse (search-paths '() (list (list b))))))
```

Fig. 10. An A* N-puzzle solver

Figure 10 displays an implementation of this strategy[5]. The function search-paths requires two accumulators each with its own invariant. The accumulator visited is a list of all the boards whose successors/children have been generated. The accumulator paths is a list of all the sequences starting at b that may need to be explored and that have no repeated boards in them. Both invariants, with some guidance, can be developed by students. The development of these invariants is likely to be the most time-consuming exercise in class. The rest of the implementation flows faster. The code finds the best sequence in paths. If the winning board has been reached by the best sequence, then the best sequence is returned. Otherwise, the program filters the successors of the last board[6] in the most promising sequence to remove boards that have already been explored. New sequences are generated using map to add each remaining successor to the most promising sequence. Notice that both of these computations are achieved by reinforcing lessons on abstraction that students have been exposed to in the near past. Finally, to maintain the two accumulator invariants, the last board of the

[5] Due to figure size limitations, all comments including contracts, purpose statements, and accumulator invariants have been omitted.

[6] Note that sequences are reversed making the last board in the sequence the first in the list.

most promising sequence is added to visited and the new sequences are appended with sequences obtained from removing the most promising sequence from paths. The only remaining tasks students must implement is finding the most promising sequence and removing a sequence from a list of sequences. The first can be done either by using accumulative recursion with an accumulator that remembers the best sequence so far or using structural recursion. The implementation in Figure 10 displays the latter and redesigning such a function using accumulative recursion is left as an exercise to give students more practice. The second is a straightforward exercise using structural recursion[7].

The algorithm developed is in essence an A*-like algorithm [8,9]. That is, it is a combination of a breadth-first strategy and a depth-first with backtracking strategy. Such algorithms are commonly referred to as *informed heuristic search strategies* [9]. What is most noteworthy is the fact that the development flows naturally from following the steps of the design recipe and iterative refinement. Students reason the algorithm into existence instead of being told about an algorithm. Such a development challenges the tacit assumption that A*-like algorithms are too complex for beginning students to understand and, therefore, are left as material restricted to more advanced courses such as an Introduction to Artificial Intelligence. There is, of course, one important observation about the N-puzzle domain that allowed us to simplify the design. Once a board is encountered there is no need to change its predecessor, because the cost of reaching it through the sequence of a previous encounter is always as good or better than the cost through the new sequence. In a full-fledged A* algorithm, the costs of the different sequences to a board must be examined to always maintain the sequence with the least cost.

6 Facilitating Deployment in the Classroom

The most important computational components of the presented N-puzzle solver have been developed in this article. The remaining components have to deal with the development of the interface with a player. The developers of HtDP have implemented a library (or teachpack as referred to by HtDPers), called *universe*, that allows students to easily develop interactive programs such as a video game [3]. Universe envisions an animation as a series of snapshots of an evolving world. There is a clock that at every tick displays the next snapshot of the world. Students must define the elements of the world and define functions for computing the next snapshot of the world when the clock ticks or when an external event, such as a keystroke or a mouse movement, occurs. Students must also define functions for drawing the world and for detecting the end of the game/animation.

It is important to carefully gauge the amount of work that beginning students are asked to do. Although the universe library truly simplifies the development of video games, sometimes students feel overburdened by the fine details of deciding on what tile a mouse click has occurred or of drawing the N-puzzle with a help

[7] This function does not appear in Figure 10 due to space limitations for figures.

button. If such is the case, invariably students get bogged down by writing drawing and mouse processing functions which leads them to relegate to the back burner the important lessons about generative and accumulative recursion. After all, in the mind of a beginning student nothing makes sense if you can not play the game. When faced with such a problem, the best course of action may be to eliminate the need for students to develop these low-level functions. This can be achieved by writing a library/teachpack specifically for the N-puzzle problem. The teachpack ought to include all the functions necessary for drawing the puzzle with the help button and for processing mouse events as well as the interface with the universe teachpack. In this manner, students can focus on the important lessons of generative and accumulative recursion. The downside of this approach, of course, is that it reduces the opportunities to reinforce previous lessons. An instructor must decide what the right balance is for the students in the classroom.

7 Related Work

The most closely related work on teaching generative and accumulative recursion to beginners is presented in HtDP. HtDP presents generative recursion as programs that have recursive calls that do not operate on part of the input. Instead, they generate a new instance of the problem. The examples used include, among others, moving a ball across a canvas, quicksort, fractals, and the computation of the greatest common divisor (gcd) of two numbers. Of these, the only example that truly captures the imagination of students is fractals. The reason is that fractals allow for a student to personalize their solutions to problems. Problems like quicksort and gcd, although important to be exposed to, do not permit for the personality of the student to be incorporated into their programs. Fractals and the N-puzzle video game, allow students to personalize solutions to their liking and that seems to be a great motivator by giving students a creative outlet to distinguish themselves and their work. The important lesson is to strike a balance between problems that allow personalization and those that do not. Both need to be included in a CS1 course. Problems that do not allow personalization, force students to focus on the lessons of designing functions that use generative recursion. Once those lessons have been presented and practiced, it is important to give students a chance to have a little fun with problems that allow personalization like the N-puzzle problem. In the N-puzzle problem, students can personalize the board (e.g., letters, number, images, etc.), the color of the tiles, and the definition of the winning board.

HtDP introduces accumulative recursion as a solution to the loss of knowledge between recursive calls. This can lead to efficiency issues in the case of programs designed using structural recursion or to problems not being solved in the case of generative recursion. The examples developed include finding a path in a graph and reversing a list. HtDP also outlines exercises that, like the work presented in this article, require students to combine skills to design programs that exploit structural, generative, and accumulative recursion. None of the problems are

video-game-based, but, in fairness, HtDP was published before the development of the universe teachpack.

To the best knowledge of the author, there have been no published attempts to have beginning students work on the N-puzzle problem nor on developing A*-like algorithms. The N-puzzle game has been used to motivate topics in Artificial Intelligence and Machine Learning [5]. In addition to using the N-puzzle in an undergraduate AI course, the authors report using the N-puzzle game in a data structures and an algorithms course. In contrast, the approach presented in this article targets beginning students.

8 Concluding Remarks

The teaching philosophy of *program by design* put forth by HtDP when applied to the design of functional video games is a powerful combination that allows CS1 students to receive a solid introduction to programming while at the same time to become enthusiastic about the field of Computer Science. The enthusiasm comes from seeing in practice that what they are learning in the classroom is directly applicable to a domain that is of interest to them. In addition, the video game domain allows students to personalize solutions which means that students are not all producing the exact same solution to problems. Contrast this to solving problems in a Mathematics, Physics, or Chemistry course and it is easy to see why students find working with video games fun, personally rewarding, and enlightening. There are examples in the literature that illustrate how to design animations and video games that require the use of primitive data, structures, and structural recursion. The work described in this article is an example of how, in the CS1 classroom, to make the transition from structural recursion to generative and accumulative recursion using a video game as motivation to capture student enthusiasm. The choice of game, the N-puzzle, was made to also demonstrate that informed heuristic search strategies, traditionally the domain of undergraduate courses in Artificial Intelligence, are within the grasp of CS1 students. Students do not simply study such an algorithm. Instead, the full power of program by design allows CS1 students to reason such an algorithm into existence. If this work is replicated elsewhere, there is no doubt that it will be an exciting time for Computer Science education and it will elevate the relevance of functional programming in the minds of future CS professionals.

Future work includes demonstrating how functional video games can be an effective pedagogical tool for motivating and teaching distributed/parallel programming to CS1 students. Functional programming has been identified as providing a clear and concise way to program parallel computers and distributed computations [7,10]. It is time for this knowledge to reach down to the CS1 classroom. The approach will assume that students have a foundation using different forms of recursion as well as abstraction and will use the universe teachpack as in the work described in this article. A second line of future work is to extend the work presented in this article to other, more complex, games such as checkers and chess. The biggest challenge in this second line of future work is identifying

heuristics that can be understood and implemented by CS1 students. Finally, a third line of future work focuses on the impact the use of video games in CS1 has on detecting plagiarism. The hypothesis underlining this line of work is that a programming medium that allows for the personalization of solutions, such as the development of video games, may make it easier for instructors to detect plagiarized code.

References

1. Felleisen, M., Findler, R.B., Fisler, K., Flatt, M., Krishnamurthi, S.: How to Design Worlds (2008), `http://world.cs.brown.edu/1/`
2. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs: An Introduction to Programming and Computing. MIT Press, Cambridge (2001)
3. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: A Functional I/O System or, Fun for Freshman Kids. In: ICFP, pp. 47–58 (2009)
4. Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: A Programming Environment for Scheme. Journal of Functional Programming 12(2), 159–182 (2002)
5. Markov, Z., Russell, I., Neller, T., Zlatareva, N.: Pedagogical Possibilities for the N-Puzzle Problem. In: Proceedings of the Frontiers in Education Conference, pp. S2F1–S2F6 (November 2006)
6. Morazán, M.T.: Functional Video Games in the CS1 Classroom. In: Page, R., Horváth, Z., Zsók, V. (eds.) TFP 2010. LNCS, vol. 6546, pp. 166–183. Springer, Heidelberg (2011)
7. Peyton-Jones, S.: Parallel Implementations of Functional Programming Languages. The Computer Journal 32(2) (1989)
8. Rich, E., Knight, K.: Artificial Intelligence. McGraw-Hill, New York (1991)
9. Russell, S.J., Norvig, P., Candy, J.F., Malik, J.M., Edwards, D.D.: Artificial intelligence: A Modern Approach. Prentice-Hall, Inc., Upper Saddle River (1996)
10. Szymanski, B.K.: Parallel Functional Languages and Compilers. Frontier Series. ACM Press, New York (1991)