Pancakes, Puzzles, and Polynomials: Cracking the Cracker Barrel

Christopher Frost, Michael Peck, David Evans
University of Virginia Computer Science Technical Report, UVA-CS-TR2004-04[*]
University of Virginia, Department of Computer Science
Charlottesville, VA
[ccf7f, map8s, evans]@cs.virginia.edu

Abstract

The Cracker Barrel peg game is a simple, one-player game commonly found on tables at pancake restaurants. In this paper, we consider the computational complexity of the Cracker Barrel problem. We show that a variation of a generalization of the problem is NP-complete.

1   The Peg Board Game

> *Leave only one — you're genius,*
> *. . . Leave four or more'n you're just plain "eg-no-ra-moose".*
> Cracker Barrel peg game instructions

Cracker Barrel is a popular restaurant chain in the United States with delicious breakfast items such as pancakes. As visitors to the Cracker Barrel restaurant, we became interested in determining the computational complexity of solving the Cracker Barrel peg game. The Cracker Barrel peg game [2], pictured in Figure 1, is a one-player game played on a board containing fifteen holes. The game begins with one hole empty and the other holes filled with pegs. The goal is to remove all but one of the pegs by jumping pegs over one another. A peg may jump over an adjacent peg only when there is a free hole on the other side of the peg. The jumped peg is removed.

Figure 1: Peg Board Game.

As anyone who has ever attempted to solve the peg game can appreciate, it is difficult to win. Since the peg board game is very inexpensive (it can be purchased from Cracker Barrel's web site [2]), it was possible to work on solving the physical peg board game before thinking about how to write a program to solve it. Attempts to solve it by hand have the feel of an NP-hard problem: it appears that one is getting close to the solution, but an incorrect move early on will leave the board in a state where it is impossible to remove the remaining pegs. We have used this problem in two introductory courses at the University of Virginia [3] [4].

2   Brute Force Solution

A straightforward approach to solving the peg board puzzle is to perform a depth-first search of all possible sequences of legal moves, this approach is shown in Figure 2. The procedure `solve-pegboard` takes a board as input and evaluates to `#f` if it is impossible to remove all but one peg on the board following the jumping rules, or to a list of moves that wins the game if there is a sequence of valid moves that leaves a single peg.

The time required to evaluate `solve-pegboard` scales exponentially with the size of the board. Solving a 4 row puzzle takes less than a second, while solving a 5 row (standard size) puzzle can take a few minutes. Note that because the search is depth first, the solution time varies substantially with the starting configuration. Since some starting positions may lead quickly to winning positions or situations where no moves are possible, some initial positions are solved quickly even for large boards.

```
(define (find-first-winner board moves)
  (if (null? moves)
      (if (is-winning-position? board)
          '() ; Winning posn, no moves needed to win
          #f) ; A losing posn, pegs, but no moves possible
      ; See if the first move is a winner
      (let ((result (solve-pegboard
                      (execute-move board (car moves)))))
        (if result ;; non-#f is a winner
            (cons (car moves) result) ; winner
            (find-first-winner board (cdr moves))))))
```

Figure 2: Solution to Peg Board Puzzle.(Complete code is available in Appendix A and at [5].)

3   Complexity of the Peg Board

To consider the complexity of the peg board puzzle, we need to describe the problem precisely. We consider a generalization of the Cracker Barrel problem (CB) with a board of any size and some initial layout of pegs:

> Given a peg board with n peg holes and some initial configuration of pegs, is there a sequence of jumps such that the board will be left with just one remaining peg?

We can prove CB is in class NP by demonstrating there is a polynomial time procedure for verifying a possible solution. This is easy: we simply execute the jumps in the solution and check that each jump is a valid move and that the final position contains only one peg. A sequence of jumps provides a clear polynomial-time verifiable certificate for the Cracker Barrel problem.

To prove CB is NP-Complete requires that we also prove the problem is NP-hard. Proving CB is NP-hard is more difficult. We simplify the problem by constraining legal jumps based on peg color (we use shapes in this presentation for easier visualization). Our variation of CB, CBH for CB-Hierarchy, is that each peg is assigned a shape: triangle, star, diamond, or circle. A hierarchy of power is created as shown in Figure 3, where whether or not pegs can jump each other is based on their shape. Triangle pegs can jump triangle, star, and circle pegs. Star pegs can jump star, triangle, diamond, and circle pegs, but not diamond pegs. Diamond pegs can jump diamond, triangle, and circle pegs. Circle pegs can only jump other circle pegs. This variation will make it easier to restrict the possible jumps in the board. The game is won when exactly one star peg remains on the board. Since the CBH problem does not obviously reduce to the original CB problem, our proof does not establish the NP-hardness of the original CB problem. To our knowledge, this is still an open problem.

1. △ > ★

2. ★ = ◆ > △ > ●

Figure 3: Non-transitive jumping hierarchy. If $x > y$ then x can jump over y, but y cannot jump over x.

3.1 Reducing 3SAT to CBH

We prove that the CBH problem is NP-hard by showing a reduction from any 3SAT problem to a CBH problem. The 3SAT problem is known to be NP-hard [1]. A 3SAT expression consists of multiple clauses joined by logical ANDs. Each clause consists of three terms joined by logical ORs. The 3SAT problem asks if an assignment of values to the terms exists that makes the expression true. Since the clauses are separated by ANDs, an assignment must be found that makes each clause true to make the entire expression true.

We first show how to represent an instance of the 3SAT problem as an instance of the CBH problem. Then, we show that there is a satisfying assignment for the 3SAT problem if and only if the corresponding CBH puzzle can be solved. If and only if there is a sequence of valid jumps that leaves the board with exactly one star peg, then the 3SAT instance is satisfiable.

To represent an instance of the 3SAT problem as an instance of the CBH problem, we place the 3SAT clauses (with their terms) as rows on the CBH pegboard, and all of the 3SAT terms and their complements (to represent all possible assignments of values to terms) as the columns. We divide the peg board into four types of tiles:

1. *Restricted cross tiles* allow a star peg to the left of the tile to cross over to the right of the tile if and only if a triangle peg at the top of the tile has already crossed to the bottom of the tile.

2. *Unrestricted cross tiles* are set up to always allow the star and triangle pegs to cross at any time.

3. *Decision tiles* start with a triangle peg at the top and allow that peg to exit at either the bottom left or right (corresponding to assigning a true or false value to a variable).

4. *Finalization tiles* reduce the number of star pegs to one if and only if at least one peg from each clause crossed from the left side of the pegboard to the right side.

In the next section, we explain how to create each type of tile.

To create a peg board corresponding to a given 3SAT problem, we need an empty peg board large enough to contain a rectangle of size (*NumClauses* × 3 terms/clause × 10 rows/term + 3 rows) × (2 ∗

NumTerms × 6 cols/term + 4 cols + 1 col). For the left-most column, place a star peg in the fifth peg hole from each tile's top. If the corresponding term is true, a peg can cross to the right side of the board. In the top row, place decision tiles for each variable. The value of the variable is either true or false, corresponding to different vertical paths through the board. Fill up the right-most column with finalization tiles, which will determine if each clause was satisfied. The arrangement of the decision and finalization tiles depends only on the number of clauses. The actual terms determine where restricted and unrestricted cross tiles are placed. A restricted tile appears in a clause at the intersection of the clause variable and corresponding decision path. The transformation process involves a linear increase in the problem size, and all steps require constant time. Thus, this algorithm satisfies the required property of transforming a 3SAT instance to a CBH instance in polynomial time.

Figure 4 shows a representation of an instance of the 3SAT problem, $(x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee x_3)$, as an instance of CBH. Since x_1 appears in clause 1, the square corresponding to the decision that makes x_1 true contains a restricted cross tiles, and all other tiles in the row are unrestricted cross tiles.
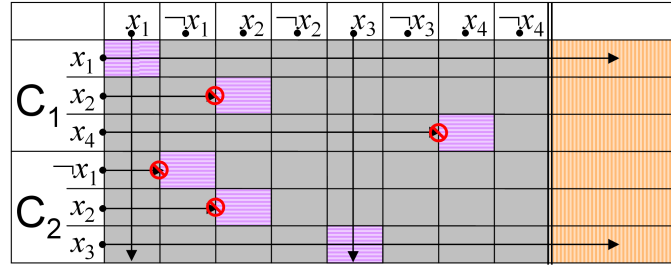

Figure 4: Reducing 3SAT to CBH. The horizontally striped tiles are restricted cross tiles. Solid grey tiles are unrestricted cross tiles. Decision tiles are along the top, and the vertically-striped region at the far right contains finalization tiles.

3.2   Decision Tile

A decision tile decides whether a term or its complement is true, allowing one triangle peg to enter the column for the term to be true. Each decision tile is constructed as depicted in Figure 5. The triangle peg can jump the circle pegs towards either the left of the right, but not both, and proceed down one of the vertical paths.
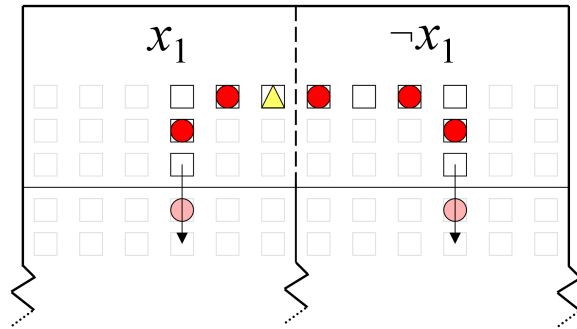

Figure 5: Decision Tile.

4

3.3   Unrestricted Cross Tile

Each unrestricted cross tile is constructed as depicted in Figure 6. As in the restricted cross tile, the box to the top of the tile indicates where a triangle peg would arrive from above, and the box to the left of the tile indicates where a star peg would arrive from the left. The unrestricted cross tile is constructed to allow the star peg to move across to the right of the tile at any time regardless of any triangle peg movement. Also, it allows the triangle peg to move down to the bottom of the tile at any time regardless of any star peg movement. The placement of the circle, triangle, and diamond pegs inside the initial construction of the tile ensure that those movements can occur.
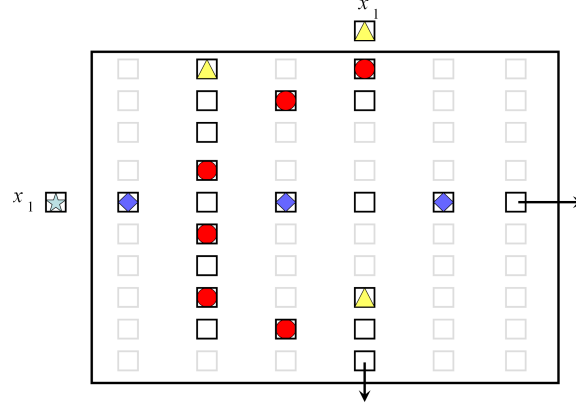

Figure 6: Unrestricted Cross Tile.

3.4   Restricted Cross Tile

A restricted cross tile must allow a star peg to cross from left to right only if a triangle peg traverses through the tile from top to bottom. Each restricted cross tile is constructed as depicted in Figure 7. The box above the tile containing a triangle peg indicates the location where a triangle peg would arrive from the above tile. The box to the left of the tile containing a star peg indicates the location where a star peg would arrive from the left tile.

When the star peg arrives from a tile to the left, our goal is to find a sequence of moves that allows the star peg to move on to tiles to the right. The star peg can jump the first blue square to its right, but cannot just the second square since the landing hole contains a red peg. Because of the jumping hierarchy, the red circle cannot jump over the blue square. Hence, the only way the star can make progress is if a peg traverses the tile vertically, removing the red circle peg. This occurs only when the corresponding term is true, and a triangle peg arrives at the top of the tile. If it is, that peg can jump down the row of circle pegs, removing the blocking peg. Hence, a star peg can cross a restricted cross tile only when the corresponding term is true.

3.5   Finalization Tile

The finalization tile must leave a single star peg on the board if and only if each clause of the 3SAT problem is true. The arrangement of the other tiles ensures that a star peg can cross the board through a row corresponding to a particular clause, if and only if that clause is satisfied.

Each finalization tile is constructed as depicted in Figure 8. It, working with the other decision tiles, allows the number of star pegs to be reduced to one if and only if each clause was able to move at least one
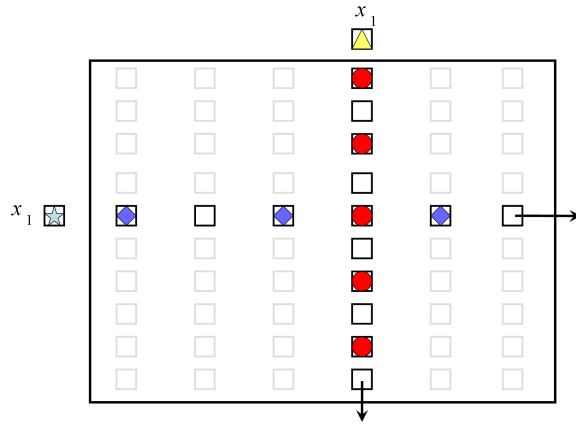
Figure 7: Restricted Cross Tile.

of its star pegs across the pegboard. Unless the top-most star peg coming from the above tile is moved, from Position 1, no star pegs created in the decision tile can jump or be jumped, thus we consider this top-most star peg. The goal for the finalization tile is to jump all other star pegs and move a star peg below the bottom of this tile, to Position 3. The top-most star peg can jump star pegs until it arrives at Position 2. However, if a star peg for this clause has reached a peg hole to the immediate left of the finalization tile, this star peg, by jumping diamond pegs in this finalization tile, can move into a position allowing the originally-top-most star peg to jump it, and continue jumping the remaining star pegs for this clause's finalization tile.

Should more than one star peg cross for this clause, the remaining diamond pegs are able to jump these extra star pegs.
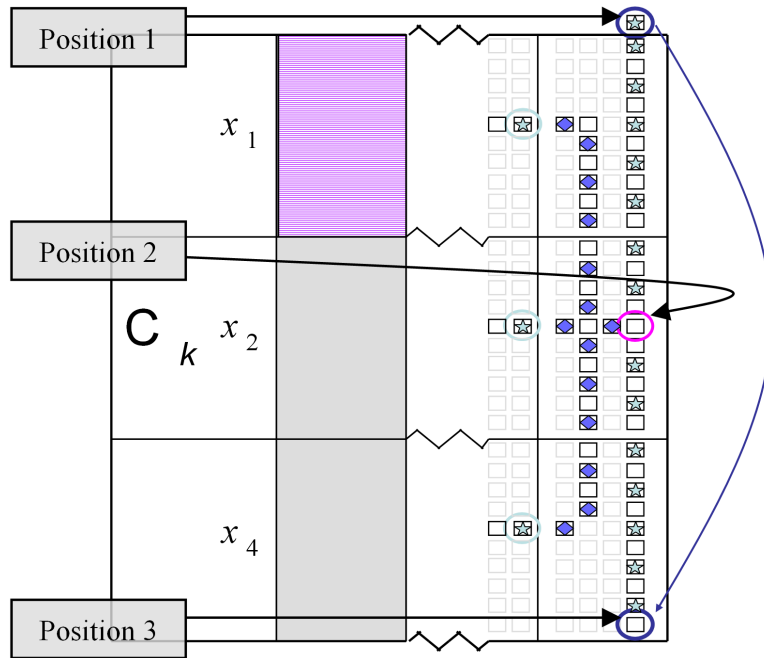

Figure 8: Finalization Tile.

4   Conclusion

The Cracker Barrel game is a fun, one-person game whose generalized problem intuitively has the feel of an NP-Complete problem to us. We have proved the variation of the Cracker Barrel problem CBH to be NP-Complete. To our knowledge, whether CB itself is NP-Hard or not is an open question.

5   Acknowledgements

6   References

1. Stephen Cook. The Complexity of Theorem Proving Procedures. Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 1971.

2. Cracker Barrel Old Country Store. `http://shop.crackerbarrel.com/`. (Search for "Peg Game").

3. The University of Virginia, CS200, Spring 2003. `http://www.cs.virginia.edu/cs200-spring2003/lectures/notes6.html`

4. The University of Virginia, CS201J, Fall 2003. `http://www.cs.virginia.edu/cs201j-fall2003/problem-sets/ps4/`

5. Pegboard-solving code and proof presentation. `http://www.cs.virginia.edu/pegboard/`.

A   Pegboard Solver

```
;;; Pegboard Solver
;;;
;;; From the University of Virginia's CS 200, Spring 2004, Notes #11
;;; http://www.cs.virginia.edu/cs200/lectures/notes11.html
;;;
;;; To solve the board:
;;;                  [1,1]
;;;               [2,1]  2,2
;;;            3,1   3,2   3,3
;;; where [x,y] means there is a peg in that hole, otherwise the hole is empty:
;;;    (define my-board (make-board 3 (list (make-position 2 2)
;;;                                         (make-position 3 1)
;;;                                         (make-position 3 2)
;;;                                         (make-position 3 3))))
;;; (solve-pegboard my-board)
;;; Which will evaluate to (((1 . 1) (2 . 1) (3 . 1))), the move
;;; which jumps peg at position 1,1 over peg 2,1 to position 3,1.


;;; A board is a pair of the number of rows and the empty squares
(define (make-board rows holes) (cons rows holes))
(define (board-holes board) (cdr board))
(define (board-rows board) (car board))

;;; make-position creates an row col coordinate that represents a position on the board
(define (make-position row col) (cons row col))
(define (get-row posn) (car posn))
(define (get-col posn) (cdr posn))

(define (same-position pos1 pos2)
  (and (= (get-row pos1) (get-row pos2)) (= (get-col pos1) (get-col pos2))))

;;; on-board? takes a board and a posn, returns true iff the posn is on the board
(define (on-board? board posn)
  (and (>= (get-row posn) 1)  (>= (get-col posn) 1)
       (<= (get-row posn) (board-rows board)) (<= (get-col posn) (get-row posn))))

;;; There are rows + (rows - 1) + ... + 1 squares (holes or pegs)
(define (board-squares board) (count-squares (board-rows board)))
(define (count-squares nrows) (if (= nrows 1) 1 (+ nrows (count-squares (- nrows 1)))))

;;; peg? returns true if the position on board has a peg in it, and false if it doesn't
(define (peg? board posn)
  (define (peg-holes? holes posn)
    (if (null? holes) #t
        (if (same-position (car holes) posn) #f
            (peg-holes? (cdr holes) posn)))))
```

8

```
  (peg-holes? (board-holes board) posn))

;;; remove-peg evaluates to the board you get by removing the peg at posn from board.
(define (remove-peg board posn)
  (make-board (board-rows board) (cons posn (board-holes board))))

;;; add-peg evaluates to the board you get by adding a peg at posn to board.
(define (add-peg board posn)
  (make-board (board-rows board)
              (filter (lambda (pos) (not (same-position pos posn)))
                      (board-holes board))))

;;; move creates a list of three posn, a start (the posn that the jumping
;;; peg starts from), a jump (the posn that is being jumped over), and end
;;; (the posn that the peg will end up in)
(define (make-move start jump end) (list start jump end))
(define (get-start move) (car move))
(define (get-jump move) (cadr move))
(define (get-end move) (caddr move))

;;; execute-move evaluates to the board after making move move on board.
(define (execute-move board move)
  (add-peg (remove-peg (remove-peg board (get-start move)) (get-jump move))
           (get-end move)))

;;; generate-moves evaluates to all possible moves that move a peg into
;;; the position empty, even if they are not contained on the board.
(define (generate-moves empty)
  (map (lambda (hops)
    (let ((hop1 (car hops)) (hop2 (cdr hops)))
      (make-move (make-position (+ (get-row empty) (car hop1))
                                (+ (get-col empty) (cdr hop1)))
            (make-position (+ (get-row empty) (car hop2))
                           (+ (get-col empty) (cdr hop2)))
            empty)))
      (list
  (cons (cons 2 0) (cons 1 0))        ;; right of empty, hopping left
  (cons (cons -2 0) (cons -1 0))      ;; left of empty, hopping right
  (cons (cons 0 2) (cons 0 1))        ;; below, hopping up
  (cons (cons 0 -2) (cons 0 -1))      ;; above, hopping down
  (cons (cons 2 2) (cons 1 1))        ;; above right, hopping down-left
  (cons (cons -2 2) (cons -1 1))      ;; above left, hopping down-right
  (cons (cons 2 -2) (cons 1 -1))      ;; below right, hopping up-left
  (cons (cons -2 -2) (cons -1 -1))))) ;; below left, hopping up-right

(define (all-possible-moves board)
  (apply append (map generate-moves (board-holes board))))
```

```scheme
(define (legal-moves board)
  (filter (lambda (move) (legal-move? move board)) (all-possible-moves board)))

(define (legal-move? move board)
  ;; A move is valid if:
  ;;    o the start and end positions are on the board
  ;;    o there is a peg at the start position
  ;;    o there is a peg at the jump position
  ;;    o there is not a peg at the end position
  (and (on-board? board (get-start move)) (on-board? board (get-end move))
       (peg? board (get-start move)) (peg? board (get-jump move))
       (not (peg? board (get-end move)))))


(define (is-winning-position? board)
  ;; A board is a winning position if only one hole contains a peg
  (= (length (board-holes board)) (- (board-squares board) 1)))

(define (find-first-winner board moves)
  (if (null? moves)
      (if (is-winning-position? board)
      '() ;; Found a winning game, no moves needed to win (eval to null)
      #f)  ;; A losing position, no more moves, but too many pegs.
       ;;; See if the first move is a winner
       (let ((result (solve-pegboard (execute-move board (car moves)))))
    (if result ;; anything other than #f is a winner (null is not #f)
        (cons (car moves) result) ;; found a winner, this is the first move
        (find-first-winner board (cdr moves))))))

;;; solve-pegboard evaluates to:
;;;    #f if the board is a losing posn (there is no sequence of moves to win from here)
;;;    or a list of moves to win from this position
;;;
;;; NOTE: null is a winning result!  It means the board has one peg in it right now and
;;;    no moves are required to win.

(define (solve-pegboard board)
  (find-first-winner board (legal-moves board)))

;;; These definitions are needed for R5RS Scheme (but not for MzScheme):

(define null '())

(define (filter test lst)
  (if (null? lst) null
      (if (test (car lst))
          (cons (car lst) (filter test (cdr lst)))
          (filter test (cdr lst)))))
```