Super Peg Solitaire Program
And
Algorithm Overview

By Richard Denson
for Senior Seminar COSC 4172

It never fails to amaze me to see the initial and subsequently exponential amount of work and sometimes research that goes into translating some simple physical world concept or mechanism into the world of computing. Take for instance a game you may have played while waiting for a meal at Cracker Barrel. This game is a seemingly simplistic game of “peg solitaire”. The game consists of a board in the shape of an equilateral triangle with 10 holes drilled into this board and 9 pegs occupying these holes. Usually the unoccupied hole is found at the center of the board. In this game, one must attempt to remove all pegs from the board save one. Generally speaking the board at the restaurant is not the only size board ever to be played, there are far more sizes and shapes (geometric) that exists. However, since my project was limited to the equilateral triangle design, I shall remain only within that context.

The focus of this project was a triangular board of a varying degree of sizes. This project was not coded or implemented completely and therefore you may find that some things are broken or are not fully realized. That is not to say that my project is woefully incomplete either, the emphasis was mainly on the design. My project entitled Super Peg Solitaire or SPS for short contains a graphical user interface for the user / player to interact with software. Within the program to functions that essentially provide the basis for the game; checkMove() and PegSolitaire(). The checkMove() function determines whether or not the peg is able to execute a move and the function PegSolitaire() will solve the puzzle (near optimally) automatically for various sizes of boards. I have also come up with representation that can be stored into a data structure within my program. The format and coordinate system are based on a skewed coordinate system depicted in a paper by George Bell. The design section of SPS can be found in the other two documents in my project, in this section I will only talk about the algorithm and its general realization.

I’ll begin here with data representation. First off, note that there are many possible ways to contain the data. The data for the board could be kept in a jagged array, a stack, a graph, etc. In my project, the board information will be stored in a graph, specifically an undirected graph. General valid moves that a peg is able to make are: along a column, along a row or diagonal to the column and row element. This is illustrated in figure 1a below.
Thus when we refer to a column, begin at the left-hand side and travel across the graph to the right. These column elements have been labeled using an English alphabet. For the row elements, begin at the top (or root) of the graph and move downward through the graph. Each row element is labeled with a lower-case Greek alphabet character. So, every peg can now be referenced via these coordinates.

Therefore, peg 3 can also be referred to as $(\beta, b)$. In order for a most basic move to work there must be two pegs and a nearby hole for one of the pegs to fall into.
For instance referring to figure 1b, if we wanted peg #2 at (β, a) wanted to jump peg #4 and end up in position (δ, a), peg #7 or any peg for that matter must not be present at location. It is also necessary that the peg at (γ, a) or peg #4 also be present. So for the general case, it is essential that for any peg (force) that jumps another peg (target) the force peg must be located two rows up or down and its destination is in the same or adjacent column as the target peg. When checking whether or not a move is valid:

```java
force_loc = position(row, col); //input the location of the force peg
target_loc = position(row, col); // input the location of the target peg

void calculate_dest() {
    ...
    if(force.row == target.row) {   //jump is to be made in the same row
        isAdjacent(force.col, target.col); /* make sure that the target
            column is directly adjacent to the force column
        */
        /* the force column destination will be directly adjacent (left
            or right) of the target column */
    }
    ...
    fcol  = tcol;   //ensure that the force column is the same as target’s
    frow  = fcurrentRow ± 2;   //otherwise find the new row

    //set the destination position for the force peg
}

boolean check_dest() {   /*
    if(force_dest == anotherPeg) {    if another peg is found at the
        destroy(force_dest);        specified location, get rid of the
        return false;               new coordinates and return false.
    }                                */
    return true;  //otherwise there is no peg present
}

boolean check_target() {
    chkpos = target_loc; /*
    if(chkpos == null) if a peg is not found at the target’s
        return false;  coordinates then the move is invalid
        */
    return true;
}
```

It is important to note that when we call `calculate_dest()` we are also checking for pegs to be present or missing from their respective locations. That is, every time we establish a peg’s position, we expect it to be there and every time we want to land the force peg, there should not be a peg at its destination location. The above code is rather incomplete but illustrates a brief framework for implementation.
To handle a diagonal move or jump will differ slightly but will mostly have to overcome the challenge of working indirectly with the coordinate system. As the code on the last page does not handle this, I will address it here:

```c
peg_loc = position(row, col);   //an arbitrary peg
dia_loc = position();   //diagonal peg container

void getDiagonalPeg() {
    dia.row = peg.row – 1;   //move down one row
    dia.col = columnOver(peg.col);   //move over one column to the right
    dia_loc = isDiagonal(true);   //flag the peg as diagonal
}
```

This new function defined above should pass in a peg location from which to start and another peg with empty dimensions for its position. I left the code above as is for clarity. So, to jump a peg all that is required from the user is force and target pegs as the checking and data manipulation will be handled by the software and will appear transparent to the user. The flow of normal operation would move like so: 1) grab input from the user. 2) check that the input and target pegs are present and that the force destination is vacant. 3) make the jump, “remove” the initial force peg and attach the same peg number to the new force location. 4) remove the target peg from game play. The function `checkMove()` will incorporate all of the aforementioned methods in order to determine if a jump is possible. This design should handle user input moves and only moves pegs according to the player’s will. If the player wishes to have a board solved in an optimal manner, then software will allocate a separate portion of code for doing so.

Optimization of solving peg solitaire takes many forms. One such form for providing an efficient and consistent method is dubbed purge and package. This is illustrated in the book Winning Ways for Your Mathematical Ways. The authors talk about a section of the board where you may have a group of pegs, aka a package. The purge in this sense is removing all of the pegs from a given package. In Bell’s paper mentioned earlier, he utilizes the concept of a package (triangular shaped of course) which he calls a sub-board. The algorithm begins by going through each purge ‘area’ where no jumps or moves have been made. If a catalyst exists in the purge area make exactly one move, then return to looking for purges where no moves have been executed. Once this section terminates, go back to the purges in reverse order and make another move if possible. Then return to looking for zero-move purges. If it is not possible to make a move in a purge to which you have returned, move to the next area continuing in reverse order. These steps will continue until there remains only one peg. Here the algorithm terminates. The board sizes of which we are concerning ourselves vary from smallest possibly solvable to very large board sizes. My algorithm follows closely to Bell’s paper. Begin by finding or determining areas of the game board which will become packaged. Obviously the minimal number of moves will be less than the number of pegs on a game board. Second of all, the package in which we will initially move will be the one with the vacant hole or presence of the null peg.
This null peg is simply a term for a node on the graph flagged so the program ignores the peg object. This is shown in figure 2.

The program will attempt to purge an entire package if possible and then move on to the next package. The sum of the smallest number of moves in all packages on the board should yield the optimal solution. My algorithm will also utilize some of the code from the user move checking to rule out fruitless moves. Similarly every time a move is made within a given section, the package will be recalculated for the next best move. If no more moves are feasible, but there still exists more pegs, the algorithm will move on to the next package and the process shall continue until one peg is left on the board. As you can see, size is arbitrary here and should not affect the algorithm as it is applied for each board.

In conclusion, this project defines and provides an implementation necessary to construct a piece of software capable of allowing a user to try and solve the puzzle as well as an algorithm to attempt to solve the triangular puzzle for various sizes. This program can check to see if a move is valid and provides a function for solving peg solitaire in an optimal fashion. In the other two documents, a thorough design for such software is blueprinted and detailed.
Annotated Bibliography

Cut-the-Knot
http://www.cut-the-knot.org/proof/pegsolitaire.shtml
This website was basically the layman’s foundation for solving the puzzle. It also provides a fundamental understanding of the strategies for the completion of (solving) a wide variety of geometric shapes and sizes (still relatively small). The main lure of this page is the java-based applet, which gives one a hands-on or visual approach for the programmer to see how the algorithm might work. Towards the end of the page the author touches on packing or grouping target pegs.

Solving Triangular Peg Solitaire (George Bell)
Paper
George Bell’s paper describes in detail the theory and algorithms used for solving a triangular board. Each triangle described in the paper is of size $T_n$ with holes $n(n + 1) / 2$ present. The paper does not explain via “a computer algorithm” however, it does illustrate various mathematical theorems. This doesn’t necessarily translate into more work for the program itself, rather it allows for correctness checking of the algorithm implemented in the software.

Discrete Mathematics & Its Applications 6th edition (Kenneth Rosen)
Textbook Chapter
Information taken from this textbook was merely held in a “refreshment” context and only served to reinforce my background knowledge of graph theory and basic operations on graphs (definitions, connectivity, paths or traversals, notations).

Graph Theory 05 (Reinhard Diestel)
Paper
Information that I gleaned from Diestel’s paper began with section 2 entitled Matching Covering and Packing in which I used to expand upon the “Cut-the-Knot” website’s concepts to a more generalized and formal explanation. More than anything this paper served to tie in and go more in depth with graphical applications established in the textbook source. I was also looking for a more efficient way to express and represent the game board in my software. Considerations when to, of course, various data structures two dimensional arrays, trees, etc. however; I believed that a graph could be a more appropriate format.

Integer Programming Based Algorithms for Peg Solitaire Problem
(Masashi Kiyomi, Tomomi Matsui)
PowerPoint Presentation
This PowerPoint document outlines the paper given by the same authors, the paper itself requires purchase (which is not feasible for me at the moment). Contained within this presentation the usage of game trees and depth-first search are employed to realize their algorithm, which itself is backtracking combined
with pruning is not exactly how I was trying to solve this problem. It does however provide a relatively effective way to design an algorithm of this nature.