# Board Games on Hexagonal Grids

## 1 Introduction

This document provides tips for implementing a hexagonal grid for use in a variety of board games, several of which are described and supplemented with coding suggestions. Ideally, the hexagonal grid should be implemented as a class, with subclasses for different overall shapes such as diamond, triangle or more advanced forms (e.g. the star shape of a Chinese Checkers board). These grids must then interact with pieces, which you may also choose to represent as objects of various classes. All of those details are up to you. This document just covers the basics of hexagonal grid representations and a few select games or puzzles (i.e. one-person games).

This document may be used in different versions of the course IT-3105 (AI Programming) such that only one or two of the games discussed below will be relevant in a particular semester. Read **carefully** the descriptions of games pertaining to your current semester, as these may include important requirements, such as a mandatory range of board sizes that your code must support.

## 2 Representing a Hexagonal Grid

Hexagonal grids consist of equilateral triangles, which, when combined, produce a hexagonal structure wherein all interior nodes have six neighbors. For the purposes of this course, the board's overall shape will be either a perfect diamond or a perfect (equilateral) triangle, as shown in Figures 1, and 2. In the bottom right-hand corners of each figure, notice that the neighborhood indices are **not** the same for diamonds and triangles. If you choose to implement each cell as an object, then one of its properties can be a neighbor list, thus avoiding redundant calculations (of a cell's neighborhood).

Using these or similar representations, your code can work with a simple square array of cells that it treats as a diamond or triangle by adhering to the appropriate neighborhood relationships. In this document, the *size* of a hexagonal grid refers to the number of rows (or columns) of that square array; the grids of Figures 1, and 2 both have size 4. For all of the games discussed below, the board size should be a parameter such that a range of boards, from very small to very large, can provide different *levels* of play.

## 3 Pieces on the Board

A typical board game includes pieces that are placed, moved, removed, hopped over and/or combined during play. It is often convenient to represent each piece as an object with a board location and an owner (in multi-player games).

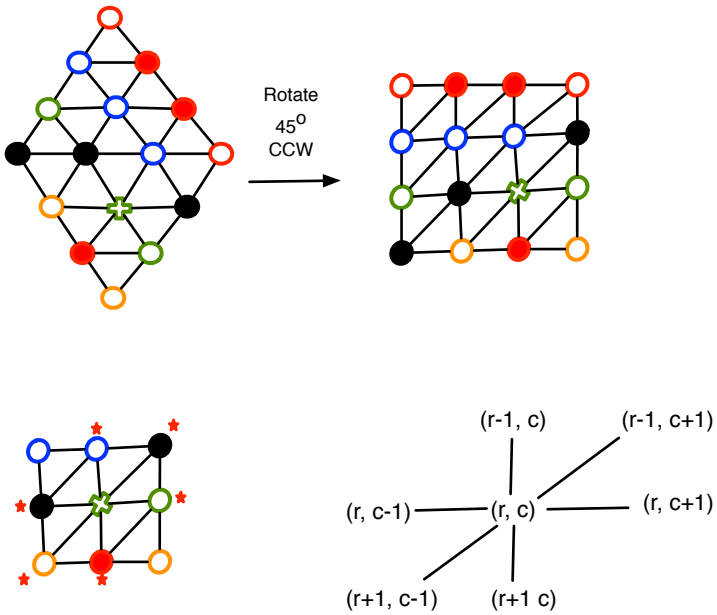


Figure 1: Representing a diamond-shaped hexagonal grid as an array. (Upper left) The grid has a perfect diamond shape, with hexagonal neighborhood structure; all neighbors are connected by edges. (Upper right) When rotated 45° counter clockwise, the board becomes a perfect square with each interior cell still having the same 6 neighbors, but visually, the hexagonal patterns may become obscured. (Lower left) Cutout of the neighborhood surrounding the node marked as a cross in the upper diagrams, with all 6 neighbors marked with small stars. (Lower right) In the coordinate system of the perfect square (whose origin (0,0) is in the upper left) the cross has indices (r,c) for *row* and *column*, and the indices of all 6 neighbors are shown. Filled circles denote grid cells occupied by a game piece, such as a peg or marble.

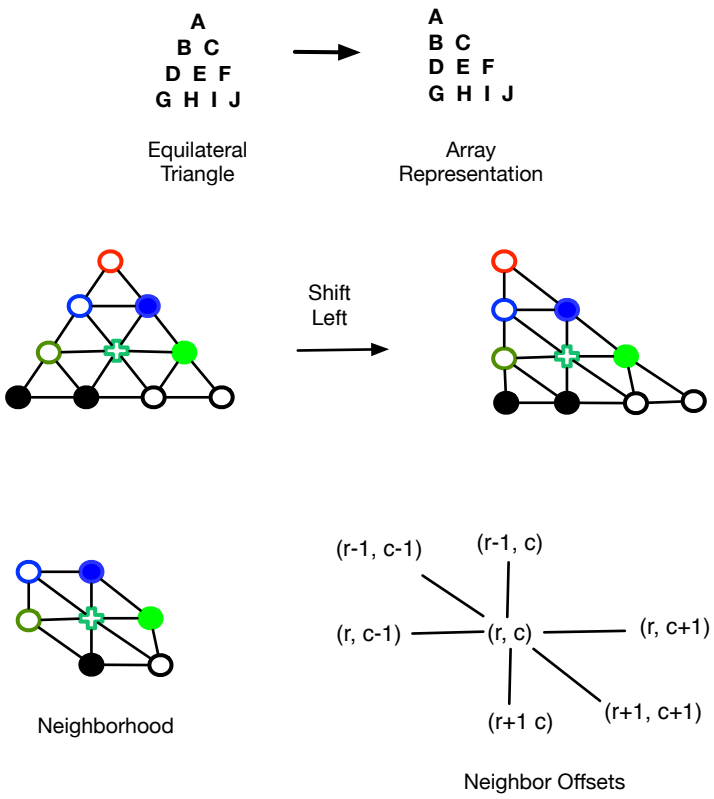


Figure 2: Representing of a triangle-shaped hexagonal grid as an array (Top) Simple illustration of the transformation from an equilateral triangle to an array, in which only the lower diagonal half contains relevant grid cells. (Middle Left) The grid has a perfect equilateral triangular shape, with hexagonal neighborhood structure. (Middle Right) Shifting the board left produces a lower-diagonal array. (Bottom Left) The neighborhood of one grid cell - the center cell of the triangle. (Bottom Right) In the coordinate system of the diagonal array (whose origin (0,0) is in the upper left) the cross has indices (r,c) for *row* and *column*, and the indices of all 6 neighbors are shown. Filled circles denote grid cells occupied by a game piece, such as a peg or marble.

In the games discussed below, the handling of pieces varies considerably. In HEX (a.k.a. Con-tac-tix), the game begins with an empty board and players take turns placing one of their pieces in an empty cell; once placed, a piece never moves. In Peg Solitaire (a.k.a. Hi-Q), the board begins with all cells (except one or a few) filled with pieces (all of the same type). A piece can then jump over a neighboring piece (if the cell just beyond it is empty). The latter piece is then removed from the board. In Chinese Checkers, a game begins with all pieces on the board. A piece can then move to a neighbor cell or jump a neighboring piece, but a jumped piece remains on the board. In short, once set of piece-handling rules definitely does not fit all board games: subclassing is inevitable.

### 3.1 Displaying a Hexagonal Board

You will need to generate some useful visualization of a hexagonal game board, whether in a special graphics window or just on the command line. Either way, your diagram needs to show the structure (diamond, triangle, etc.) of the board. Thus, the top of a displayed diamond or triangular board must be a single cell, not a whole row. It also needs to clearly differentiate between an empty cell and a filled cell. If the game involves two players, then their pieces should have distinct colors or symbols so that patterns stand out. In general, a quick glance at the diagram should reveal the complete game situation.

During any phase of system operation, it must be possible to view the progress of individual games via this graphic. This does not include individual rollout simulations (in Monte Carlo Tree Search), but does include each *actual game*, i.e. episode, during training and each game of a tournament.

This display feature must be easy to turn on/off by the user. During the demonstration session, we will want to watch a few actual games in progress....but not all games.

## 4 Example Games

What follows are three short descriptions of board games: Hex, Chinese Checkers (simplified), and Peg Solitaire. Each employs a hexagonal grid and the same set of neighborhood relationships between cells, but each uses pieces in a different way.

### 4.1 The Game of Hex

Hex, also known as *Con-tac-tix*, is played on a diamond-shaped grid with hexagonal connectivity. As shown in Figure 3, two players (black and red) alternate placing single pieces on the grid. Placed pieces can never be moved or removed. Each player ”owns” two opposite sides of the diamond and attempts to build a connected chain of pieces between those two sides; the first player to do so wins. It can be proven mathematically that ties are not possible: a filled Hex board always contains at least one chain between opposite sides. In Figure 3, black owns the northwest and southeast sides, while red owns the northeast and southwest sides. The bottom right of the figure shows black’s winning chain.

Unlike other board games (such as checkers or chess), the set of legal moves for a Hex state are trivial to calculate. Given the current player, its legal moves are stone placements in each of the unfilled cells.

You will probably have two different representations for each board state: one that includes neighbor relationships and other details that allow your code to analyze states, and another (considerably simplified) data structure, such as

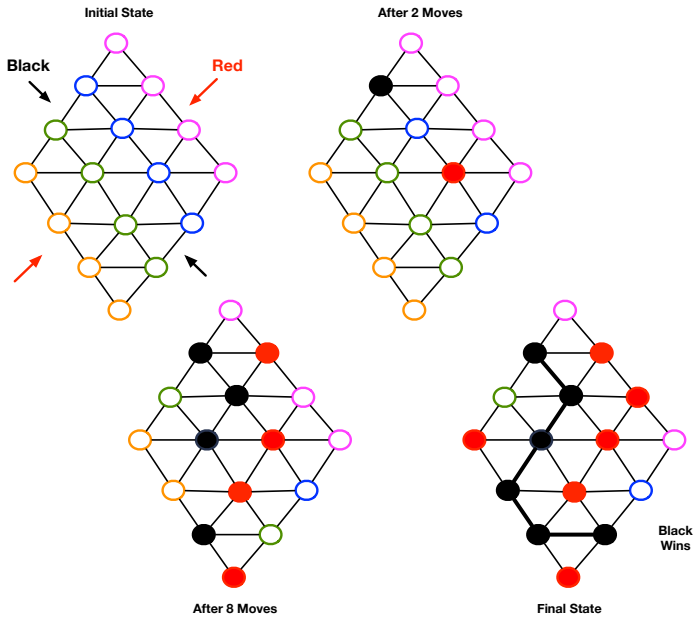


Figure 3: A 4 x 4 Hex Board in 4 different states of play. Colors of empty cells indicate their row index in a 4 x 4 array representation (as in Figure 1). Arrows indicate the borders owned by each player.

a one-dimensional array, to serve as input to a neural network. A standard representation of a board consists of 2 bits per cell, where the bits represent the three possible states of the cell: (0,0), 'empty'; (1,0), 'filled by player 1'; and (0,1), 'filled by player 2'. Each bit becomes the input for one neuron in the neural network.

One essential feature of any game state is the index of the player who will make a move from the given board configuration. So as a bare minimum, a game state should include the status of every cell on the board along with a player id of some sort. A standard id representation consists of 2 bits: (1,0) for player 1, and (0,1) for player 2.

**Your code for Hex must support boards of all sizes from 3 up to and including 10.**

## 4.2 A Basic Chinese Checker Puzzle

Chinese Checkers is normally a multi-player game, but with a simple twist it becomes a solo puzzle, with the objective of transfering all pieces from the start to the goal configuration in as few moves/turns as possible. A **single move** in Chinese Checkers can either consist of the relocation of a piece from one cell (C1) to an open neighbor cell (C2), or a series of one or more *hops*, wherein C2 is occupied but C3 is open, with C3 being C2's neighbor cell in the same direction from C2 as the direction from C1 to C2. After landing in C3, the piece may make another hop (in any direction, d) if C3's neighbor in d, C4, is occupied but C4's neighbor in d is not, and so on. However, after landing at C3, the player is not required to make additional hops, even if they are available. For additional information on movement (or other aspects of the game), consult an online resource such as Wikipedia.

Figure 4 shows four of a longer sequence of states that comprise one puzzle-solving episode. In this standard setup, a small triangle of pieces is to be moved from the bottom to the top of the diamond. The dimensions of this puzzle are the following:

- size = 4. As shown in Figure 1, the diamond, when rotated, yields a 4 x 4 cell array.
- depth = 3. The initial cluster of pieces form a triangle that fully occupies the bottom 3 rows of the diamond.

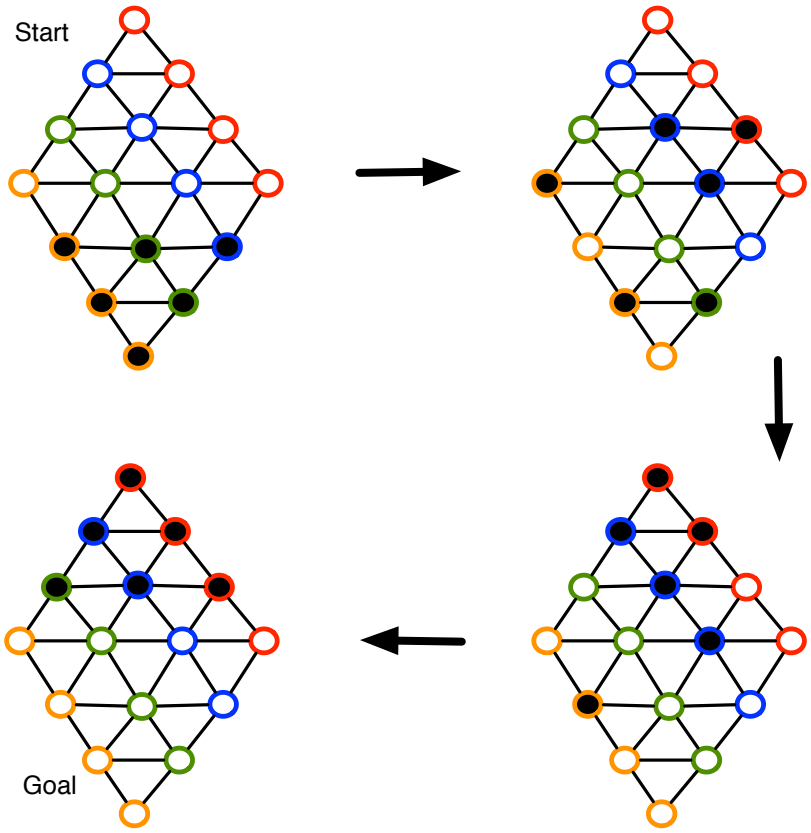


Figure 4: Excerpts from a complete series of board states for a solution to the Chinese Checker Puzzle. Filled circles denote cells occupied by a piece (often a marble or peg in physical versions of the game). This is a puzzle of size 4, depth 3.

In standard Chinese Checkers, each player has a target/goal position known as *home*, which is the far corner directly opposite that player's starting pieces. In this solo puzzle, home is the upper corner of the diamond.

Although there are no rules against hopping backwards (relative to home) in Chinese Checkers – and in many cases a multi-hop move includes one or more backward jumps – it normally makes sense to constrain moves such that the final cell ($C_f$) of a move is no further from home than is the starting cell ($C_s$) of that move. Whether $C_f$ should be required to be **closer** to home than $C_s$ is an issue that is probably best resolved by experimentation and/or detailed analysis, since certain sideways moves may enable the formation of effective chains of pieces that other pieces can hop along. Also, some endgame situations lend themselves to simple sideways moves as opposed to more cumbersome (and less efficient) alternatives.

On a Chinese Checker board with many well-distributed pieces, the move options from any given state can be plentiful. Hence, on any boards of a non-trivial size, enumerating all possible board configurations and all legal moves from them becomes an infeasible task and should be avoided. It makes more sense to generate (and then save) the legal moves from states only as those states are encountered during problem-solving search.

**Your code for the Chinese Checker Puzzle must support diamond boards of all sizes from 3 up to and including**

**8, and all depths from 2 to 4 of the initial triangle of pieces.**

## 4.3 Peg Solitaire

As shown on Wikipedia, many versions of the game exist, where most share the same basic rules for peg hopping and removal, but they differ in terms of the board shape and connectivity: only some use hexagonal neighborhoods. In the version discussed below, the board uses hexagonal connectivity and has a diamond or triangle shape. The game is actually a puzzle to be solved by a single player.

The initial puzzle state has all cells except K (where K is typically 1) filled with a peg/marble/piece. The only legal operation is to jump one peg over a neighbor peg and then remove that neighbor from the board, leaving one more empty cell. When no more jumps are possible, the game ends. The final state constitutes a *win* when only a single peg remains on the board. In some versions of the game (though not the one covered here), the final peg must also be in the cell that was originally open at the beginning of the game. The rules for when a peg can jump its neighbor are the same as those for Chinese Checkers: if the peg in cell A is to jump over the peg in cell B, then B's neighbor cell C must be open, where the direction from B to C is the same as the direction from A to B.

Figure 5 shows a basic triangular board of size 5 at various stages of a Peg Solitaire game.

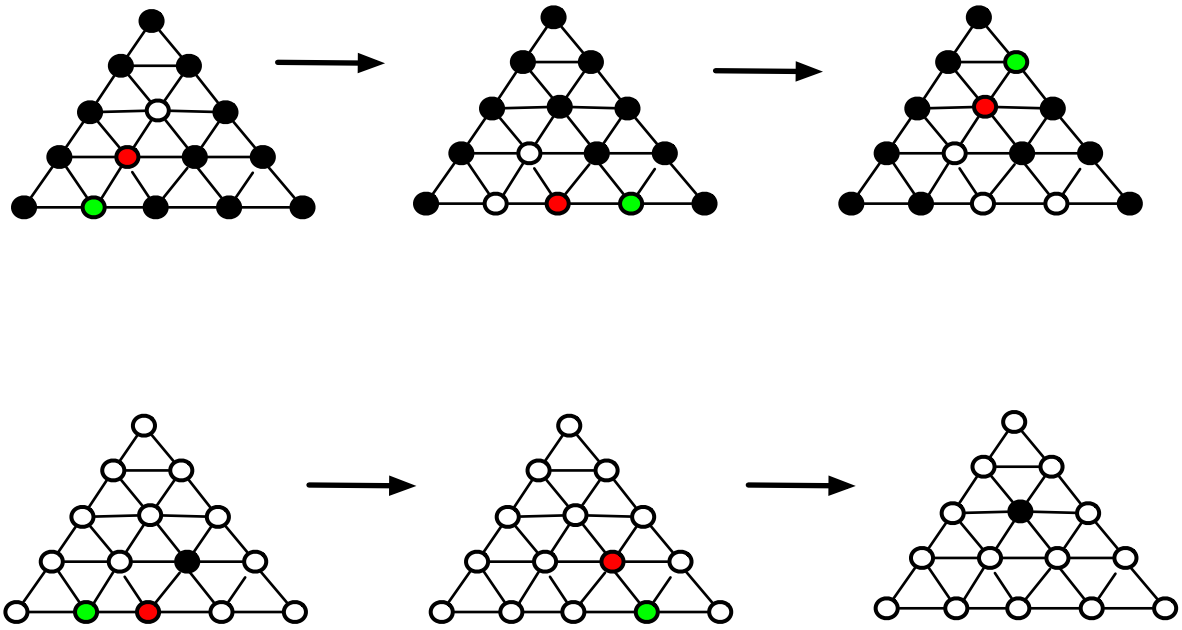

Figure 5: Several moves in a game of Peg Solitaire, where filled nodes represent pegs, green fill denotes the peg that will hop on the next move, and red denotes the peg that will be hopped over, and removed, on that move. (Top Row) Early stages of a game with the initial state in the upper left. (Bottom Row) Final three states of a game that ends in success.

**Although this game normally begins with a single empty cell, in the middle, your code must include possibilities for any number of open cells, at any location. And, as mentioned earlier, your code must handle boards of varying sizes and shapes: triangles from size 4 up to size 8, and diamonds from size 3 up to size 6.**