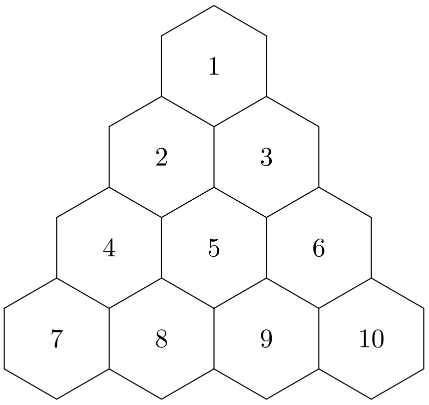
Matt Parker's Maths Puzzles: Peg Solitaire Implementation Notes

Colm Baston

Introduction

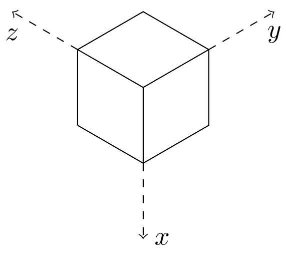
This fortnight's MPMP[1] asks us to find a strategy to complete a game of peg solitaire[2] using the fewest moves. The game takes place on a triangular board where a move may be in any of six directions, so the board is a hexagonal tiling. Let S refer to the size; here is a board where $S = 4$:

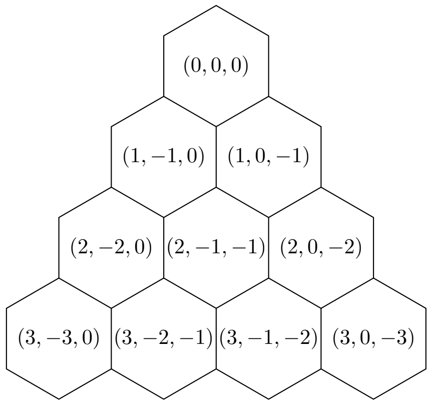


The aim of this short paper is to demonstrate, in a language-agnostic way, a robust system for programming with hexagonal tilings, namely cube coordinates, as well as some of the power of recursive functions. Some mathematical and programming knowledge is assumed.

Cube Coordinates

The cube coordinate system is an elegant way to index hexagons in a hexagonal tiling. It postulates a three-dimensional space packed with unit cubes, one centred at each $(x, y, z) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$. Now focus only on those cubes that lie on the plane $x + y + z = 0$, imagining that you are viewing orthogonal to the plane. The cubes' outlines form a regular hexagonal tiling:





The axes are rotated so that our triangle of hexagons lies at a convenient position, with x indicating the row in which the tile lies, and y increasing from $-x$ to 0 going left-to-right within a row. This way, it's easy to generate all triples of the triangle, and it's easy to check whether an arbitrary integer triple represents a tile of the triangle: just check that $-x \leq y \leq 0 \leq x < S$ and, of course, that $x + y + z = 0$. The lexicographic ordering of triples in the triangle respects the ordering of the original indices, and these indices can be recovered using one of $(\sum_{i=0}^{x} i) + 1 - z$ or $\frac{x(x+1)}{2} + 1 - z$.

But none of the above justifies the use of cube coordinates over some other system. Where cube coordinates really shine is in specifying transformations on tiles:

i. Translating from a tile to one of its six adjacent tiles corresponds to offsetting its coordinates by one of the six permutations of $\{-1, 0, 1\}$.
ii. Performing a left-right reflection of a point is the function $(x, y, z) \mapsto (x, z, y)$.
iii. Rotating a point clockwise 1/6th of a revolution about point $(0, 0, 0)$ is the function $(x, y, z) \mapsto (-z, -x, -y)$.

More general versions of these, and other, transformations can be obtained, but these are sufficient for this puzzle.

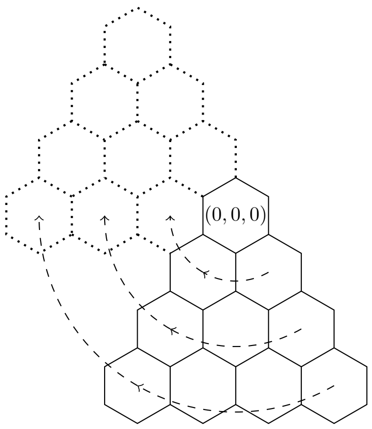
[1] `https://www.think-maths.co.uk/coin-puzzle`
[2] `https://wikipedia.org/wiki/Peg_solitaire`

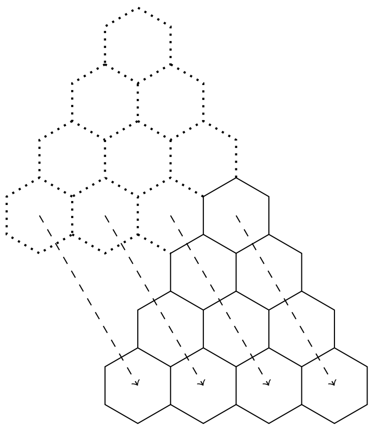
Symmetry

In the peg solitaire puzzle, we may choose one of the tiles from which to remove the couner before the start of the game. It is noted that there are really only three choices, not 10: tiles 2 and 3 are considered identical because they are reflections of one another, a tile being on the left or the right having no special meaning in the game; similarly, tiles 2 and 3 can be rotated into positions 6 and 9, or 8 and 4, so none of these has any distinction the others don't.

The equivalence relation which defines when two tiles are symmetric yields three equivalence classes: $\{1, 7, 10\}$, $\{2, 3, 4, 6, 8, 9\}$, and $\{5\}$. We can define this equivalence relation for our cube coordinate representation by composing the simple transformations outlined above into larger transformations that preserve the state of the game. In all, there are six of these transformations, including the identity transformation which leaves all points untouched.

Let us first define a transformation which rotates the board 1/3rd of a revolution in-place. We can start by applying our 1/6th rotation function twice: $(x, y, z) \mapsto (-z, -x, -y) \mapsto (y, z, x)$. Since this rotation function rotates about the point $(0, 0, 0)$, not the centre of the triangle, we have to perform some translation to return the triangle to its intended position:





In general, the rotated triangle will be misaligned by $S - 1$ tiles in the $(-1, 0, 1)$ direction, so we can fix this by offsetting each point by $(S - 1,\ 0,\ -S + 1)$. This gives us an in-place 1/3rd clockwise rotation function of:

$$(x, y, z) \mapsto (y + S - 1,\ z,\ x - S + 1)$$

Let's name this function *rotate*, and our left-right reflection function *reflect*. We can combine *rotate* and *reflect*, along with the identity function *id* representing no rotation or reflection, to yield all six symmetry transformations: rotate zero, one, or two times, followed by reflecting zero times or once. Two tiles are defined to be symmetrically equivalent if any of these six functions, when applied to one, results in the other:

i. *id*	iv. *reflect*
ii. *rotate*	v. $\mathit{reflect} \circ \mathit{rotate}$
iii. $\mathit{rotate} \circ \mathit{rotate}$	vi. $\mathit{reflect} \circ \mathit{rotate} \circ \mathit{rotate}$

Stated more formally, $t_1 \equiv t_2$ if and only if there exist functions $f \in \{\mathit{id},\ \mathit{reflect}\}$ and $g \in \{\mathit{id},\ \mathit{rotate},\ \mathit{rotate} \circ \mathit{rotate}\}$ such that $(f \circ g)(t_1) = t_2$. The relation is reflexive since $\mathit{id}(t) = t$ by definition. For symmetry, it is sufficient to show that, for each choice of function, we can also choose its inverse: applying *rotate* once inverts applying it twice, and vice versa, while *id*, *reflect*, $\mathit{reflect} \circ \mathit{rotate}$, and $\mathit{reflect} \circ \mathit{rotate} \circ \mathit{rotate}$ are self-inverse. For transitivity, we must show that if we compose any two of the functions, we could have chosen one which achieves the same in a single step: for example, $\mathit{rotate} \circ \mathit{rotate}$ is a shortcut for $(\mathit{reflect} \circ \mathit{rotate} \circ \mathit{rotate}) \circ (\mathit{reflect} \circ \mathit{rotate})$; there are 36 cases in total, so the rest are omitted[3]. Since these three properties do hold, our symmetry relation is an equivalence.

Generating Games

We can now programmatically generate a minimal set of initial boards, observing symmetry. First, generate all of the tiles of the triangle, considering them all to contain a counter, and calculate their symmetry classes. For each symmetry class, starting from the full set of tiles each time, choose an arbitrary tile from the class and create an initial board by removing the counter from that tile. From an initial board, we would now like to generate a set of possible games:

i. Call a tile of the triangle alive if it has a counter present, otherwise the tile is dead.
ii. A game is defined as an ordered sequence of valid moves, that is, following the rules of peg solitaire, whose final board has only one live tile remaining.
iii. A move is an ordered sequence of tiles, representing the path of a counter throughout that move.

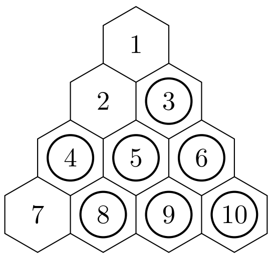
For example, [7–2, 1–4, 9–7–2, 6–1–4–6, 10–3] is a game of five moves, which, as you can see from the first move of the game, had tile 2 as its initially dead tile.

The core algorithm is a function to generate all of the moves available from a single tile t on some board b. Of course, t must actually be a live tile of b, so return that there are no available moves if it is dead. For each of the six adjacent

[3] *omit*, verb: to leave as an exercise for the reader

directions, calculate the tile one step in that direction from t, call it o, and the tile two steps in that direction from t, call it l. Check that o is alive, that l is dead, and that l is within the bounds of the triangle; discard them if any of these checks fail, and return no moves if all six o, l pairs are discarded this way. For each remaining o, l pair, recursively calculate the moves from tile l on a copy of board b modified such that t and o are now dead, and l is now alive. If there is at least one move returned by the recursive call, prepend t to each of them, recording those as moves available from t, otherwise, if no moves are returned by the recursive call, record only that t–l is move; in either case, continue until all o, l pairs are exhausted.

But there is a problem. Consider the board after the partially-completed game [7–2, 1–4]. In the following diagram, the live tiles are circled:



The function just outlined would return that the moves available from tile 9 are 9–2–7–9 and 9–7–2–9. These are both valid moves, of course, but the algorithm is being too greedy[4]! It insists that if multiple jumps can be chained during a single move, then they must be chained. It successfully finds moves which cannot be continued with additional jumps, but misses four other perfectly valid moves: 9–2, 9–2–7, 9–7, and 9–7–2. In fact, the partially-completed game, which was on the way to being completed in only five moves, becomes impossible to complete after you perform either of the moves returned by the algorithm.

One way of fixing this flaw in the algorithm is straightforward. The first thing it should do, even before generating the o, l pairs, is to declare that t, on its own, is an available move. This enables the recursive calls to short-circuit; they will still return any chains of jumps that are possible, but they will also announce that simply stopping at their tile t is an option as well. After a recursive call finishes, any moves that it returns are guaranteed to be prepended to by its caller, making the locally-invalid singleton moves valid. At the top level, however, but not in any of the recursive calls, the singleton move that is returned is not correct, representing a move that doesn't change the state of the game at all and should be ignored; it will be the first move that was recorded, and the only one of length one. Finally, because recursive calls can no longer fail to return at least one available move, the special case returning the move t–l is redundant.

Now that we can generate all of the moves from a single tile, it is easy to generate all moves available on tile on a board state: simply enumerate all of the live tiles on the board, call the function at each, and collect the results. From that point, generating all of the games that can be completed from an initial board is just another recursive function. If there is only one live tile on the board, the game is already complete, so return the game consisting of an empty sequence of moves. Otherwise, if there are at least two, call the previous function to generate all available moves and, for each available move, recursively generate all games that can be played on the updated board after that move is performed. If no games are returned, then discard the move, otherwise prepend the move to the start of each of the games and return them.

This is now sufficient to perform an exhaustive search for all possible games. For example, here is the output of a program that generated all 45 possible games where $S = 4$:

```
[7-2, 1-4, 6-1, 4-6, 10-3, 1-6, 8-10, 10-3]
[7-2, 1-4, 6-1, 4-6, 10-3, 1-6, 8-10-3]
[7-2, 1-4, 6-1, 4-6, 10-3, 8-10, 1-6, 10-3]
[7-2, 1-4, 6-1, 9-7, 7-2, 1-4, 4-6, 10-3]
[7-2, 1-4, 6-1, 9-7, 7-2, 1-4-6, 10-3]
[7-2, 1-4, 6-1, 9-7-2, 1-4, 4-6, 10-3]
[7-2, 1-4, 6-1, 9-7-2, 1-4-6, 10-3]
[7-2, 1-4, 9-7, 6-1, 7-2, 1-4, 4-6, 10-3]
[7-2, 1-4, 9-7, 6-1, 7-2, 1-4-6, 10-3]
[7-2, 1-4, 9-7, 7-2, 6-1, 1-4, 4-6, 10-3]
[7-2, 1-4, 9-7, 7-2, 6-1, 1-4-6, 10-3]
[7-2, 1-4, 9-7, 7-2, 6-1-4, 4-6, 10-3]
[7-2, 1-4, 9-7, 7-2, 6-1-4-6, 10-3]
[7-2, 1-4, 9-7, 7-2, 6-4, 4-1, 1-6, 10-3]
[7-2, 1-4, 9-7, 7-2, 6-4, 4-1-6, 10-3]
[7-2, 1-4, 9-7, 7-2, 6-4-1, 1-6, 10-3]
[7-2, 1-4, 9-7, 7-2, 6-4-1-6, 10-3]
[7-2, 1-4, 9-7-2, 6-1, 1-4, 4-6, 10-3]
[7-2, 1-4, 9-7-2, 6-1, 1-4-6, 10-3]
[7-2, 1-4, 9-7-2, 6-1-4, 4-6, 10-3]
[7-2, 1-4, 9-7-2, 6-1-4-6, 10-3]
[7-2, 1-4, 9-7-2, 6-4, 4-1, 1-6, 10-3]
[7-2, 1-4, 9-7-2, 6-4, 4-1-6, 10-3]
[7-2, 1-4, 9-7-2, 6-4-1, 1-6, 10-3]
[7-2, 1-4, 9-7-2, 6-4-1-6, 10-3]
[7-2, 6-4, 1-6, 4-1, 10-3, 1-6, 8-10, 10-3]
[7-2, 6-4, 1-6, 4-1, 10-3, 1-6, 8-10-3]
[7-2, 6-4, 1-6, 4-1, 10-3, 8-10, 1-6, 10-3]
[7-2, 6-4, 1-6, 10-3, 4-1, 1-6, 8-10, 10-3]
[7-2, 6-4, 1-6, 10-3, 4-1, 1-6, 8-10-3]
[7-2, 6-4, 1-6, 10-3, 4-1, 8-10, 1-6, 10-3]
[7-2, 6-4, 1-6, 10-3, 4-1-6, 8-10, 10-3]
[7-2, 6-4, 1-6, 10-3, 4-1-6, 8-10-3]
[7-2, 6-4, 1-6, 10-3, 8-10, 4-1, 1-6, 10-3]
[7-2, 6-4, 1-6, 10-3, 8-10, 4-1-6, 10-3]
[7-2, 9-7, 1-4, 6-1, 7-2, 1-4, 4-6, 10-3]
[7-2, 9-7, 1-4, 6-1, 7-2, 1-4-6, 10-3]
[7-2, 9-7, 1-4, 7-2, 6-1, 1-4, 4-6, 10-3]
[7-2, 9-7, 1-4, 7-2, 6-1, 1-4-6, 10-3]
[7-2, 9-7, 1-4, 7-2, 6-1-4, 4-6, 10-3]
[7-2, 9-7, 1-4, 7-2, 6-1-4-6, 10-3]
[7-2, 9-7, 1-4, 7-2, 6-4, 4-1, 1-6, 10-3]
[7-2, 9-7, 1-4, 7-2, 6-4, 4-1-6, 10-3]
[7-2, 9-7, 1-4, 7-2, 6-4-1, 1-6, 10-3]
[7-2, 9-7, 1-4, 7-2, 6-4-1-6, 10-3]

Generated 45 games on a size 4 board.
Minimum-length game: [7-2, 1-4, 9-7-2, 6-1-4-6, 10-3]
```

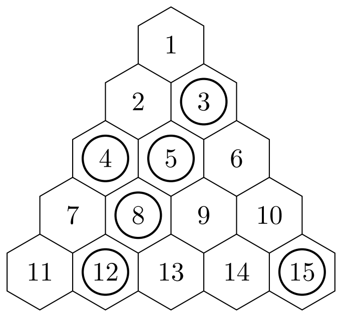
Interestingly, all of the games begin with the move 7–2, boards starting with initially dead tiles from symmetry classes 1 or 5 being impossible to complete.

Running the program when $S = 5$ generates 360,076 games in total, with the minimum length being nine: [1–4, 7–2, 6–1–4–6, 10–3, 12–5, 14–12, 11–13–6, 3–10, 15–6–4]. However, since we have only applied symmetry analysis to generating the initial boards, not generating the games, the majority of these are duplicates, with only 171,303 being unique.

Each game beginning with the move 4–1 will have a reflected game beginnning 6–1, for example, and each game beginning with the move 6–4 will have a rotated, reflected game beginning 13–4. Furthermore, two games may exhibit subgame

[4]https://wikipedia.org/wiki/Greedy_algorithm

symmetry even if they are not entirely symmetric. Consider the partially-completed game [1–4, 7–2, 6–1, 1–4, 13–6, 10–3, 11–13, 14–12], which features a symmetric board state:



Making the move 4–6 should be considered the same as making the move 4–13. Two games which begin identically, but later branch into symmetric subgames should not be distinguished. Formally, call two games subgame-symmetric if and only if either:

i. The games are entirely symmetric, that is, one of the six symmetry transformations entirely transforms one of the games into the other.

ii. The games have the same first move, and, recursively, the subgames formed by removing the first move from each of the games are subgame-symmetric.

We could filter duplicate subgame-symmetric games after the fact, but if we build this symmetry analysis into move generation we can prevent the duplicates from being generated in the first place. When calculating all available moves from a board state, before returning the moves to the game generation function, identify their symmetry classes. It is important not only that one move may be transformed to another for them to be in the same symmetry class, but also that the board states resulting from those moves coincide under the same transformation; one without the other is not sufficient, both must be true. Once the symmetry classes are determined, only return one move from each to the game generation function. This will prune subgame-symmetric branches as soon as they diverge, so we need not explore more than one of them, saving some runtime.