Etude 10 – Coroutine Puzzles

Solutions

The notation rules are as follows:

- Each coin is represented as a number. The coin starting in 0 is labelled the “0” coin and the coin starting in 1 is labeled the “1” coin.
- A coin makes a move in a “direction”.
- Each direction corresponds to an x and y modification on the grid representing the coroutine puzzle as follows (in format DIR: x modifier, y modifier):
    - N: 0, 1
    - NE: 1, 1
    - E: 1, 0
    - SE: 1, -1
    - S: 0, -1
    - SW: -1, -1
    - W: -1, 0
    - NW: -1, 1
- Each move is shown by the coin that was moved, followed by the direction or a dash “-“ if the coin could not move and had to pass.
- Note that these orders of moves are not necessarily the quickest (they are just one solution), however using our algorithm, you can easily find the quickest order of moves to make.

COROUTINE 1		COROUTINE 2	
0, S	0, S	0, E	0, NE
1, -	1, W	1, -	1, W
0, S	0, -	0, E	0, SW
1, W	1, W	1, N	1, -
0, N	0, N	0, W	0, N (WIN)
1, NE	1, E	1, SW	
0, N	0, SE	0, SW	
1, N	1, SE	1, NE	
0, S	0, N (WIN)	0, SE	
1, -		1, NW	

We found these solutions by keeping track of two main things. The first was the moves that we were making (each time we made a move we would add it to a “moves” array). The second was the decision moves that we made; that is that when we were making a move and have more than one valid move, we would record this. This was in the form of a stack which recorded for each entry an array containing: which coin was making the decision move (0 or 1), what the current state of the two coins were before making the decision move (i.e. the x and y coordinate of both, which will make sense soon) and finally the number move we were taking (i.e. the length of the moves array). We kept track of each because it meant that once we found a solution we would check to see if it was the shortest solution we found. If it was we would hold a deep copy of the moves. And then go back to the latest decision, otherwise we would simply be able to pop the latest decision off the top

of the stack and restore back to the state before we made the decision and then try the other moves on the square that we had not taken previously. This meant that we were able to keep track of the quickest way to complete the coroutine puzzle and had tested all possibilities.

Data structures

The coroutine puzzles required us to hold four main data structures:

- Coin (Object)
    - The coin needed to hold information about its location.
        - x and y coordinate
    - On top of that it needed to know about the x and y modifiers depending on which direction it moved in.
        - A dictionary holding each direction as the key and the x and y modifier (e.g. +1 for x and -1 for y) as the value for each direction. You can see the set of rules on the previous page.
- Grid (2D Array)
    - The grid was the 3 by 3 grid which held the moves possible at each location.
    - Our grid specifically was represented as a 2D array
        - Each element in the array held two lists
            - The first was a list of all possible moves when on the element.
            - The second was a dictionary of all the moves that had been played by each coin at the location where the key was the coin and the value was the list of moves that the coin had played at that grid location.
            - E.g. [[‘N, ‘NE’, ‘S’], {0: [‘S’], 1: [‘N’, ‘NE’]}]
        - This allowed us to keep track of which moves we were making at our decision points and then next time we remade the decision, we would pick a move that wasn’t in that coins “made moves” dictionary.
- Moves (Array)
    - The moves array help subarrays which were the coin moved and the direction that it moved in:
        - E.g. [[0, ‘S’], [1, ‘W’]]
- Decisions (Stack)
    - The decision stack held information about every decision (when a coin made a move and had more than one legal move) made.
        - Each element in the stack held
            - The coin that was making the decision e.g. 0 or 1
            - The current state (x and y coordinate) of each coin before the decision move had been made in the form of a dictionary where the coin was the key and the value was a list holding its x and y coordinate e.g. {0: [0, 1], 1: [1, 2]}
            - The number of the move that we were about to make e.g. the length of the moves array.
        - This would allow us to restore back to a “pre-decision” state and make the other decision to see if they would eventuate to a shorter number of moves to get too the middle.

- Other than these main things the program would need to keep track of the current most optimum series of moves so that it could compare the current solution to the current best solution each time the puzzle was solved.

Algorithm

In order to find the quickest series of rules to follow we can follow the rule set out above. Broken down we would:

- Begin by looking at the 0 coin and then seeing which moves it can make, dependant on the location of the other coin. This "checkMoves" behaviour should see which moves are valid in terms of not going off of the grid, but also in terms of not stepping on the other coin.
    - If the coin has more than one move, record this as a decision move on our stack and then choose one of the moves. Ensure that we record on the grid that this coin has made this move at this location.
    - If the coin has only one valid move, then just record the one move we can make.
    - Otherwise, record the move as a pass.
    - Make sure we aren't in a loop.
        - We can check to see if we are in a loop. We check the current coin. If it only has one or fewer possible moves (it is in a non decision state) and then the result of this move will leave the other coin in a non decision state and we haven't made any decisions since the last time we were in this position we are in a loop.
        - To check this we could write a function "checkLoop" which is called whenever a coin is in a non decision state and the result of this move will leave the other coin in a non decision state. The function will look at the stack and see the last move count of the last decision (the last element of the list at the top of the stack). It will then check between the end of the moves array and the index to see if the two non decision moves that we are being forced to take are in the moves array in the order that we must take them. If they are, we backtrack to them using the moves in the moves array and then check to see if the coins position before these two moves were taken are equal to the position of the coins currently. In other words, we check to see if we already made these moves at this position. If this is true, we are in a loop.
- Repeat the above, but with the other coin and continue making moves for each coin.
- Eventually we will come to a solution or get in a loop:
    - If we come to a solution then we check to see if we already have a solution and.
        - If not, we set this solution as the best solution and then pop the latest decision off the stack, restoring the grid and choosing another one of the decisions.
        - If we do have a solution, check to see if the current one is better (the moves array is shorter).

            - If it is, update the solution and pop off the decisions grid, restoring the grid and choosing another one of the decisions
            - If it isn't, pop off the decisions restoring the grid and choosing another one of the decisions.
        - If we come to a loop, we pop off the decisions restoring the grid and choosing another decision.
- If at any time the decisions grid is empty, we have finished covering all possible decisions so we can end the program and report the best solution.

Additionally, we could also achieve the opposite and generate a coroutine puzzle by starting one of the two coins in the middle and the other in a random position and then work them towards their starting positions (0 at 0, 0 and 1 at 2, 2), adding which moves you'd need to take to get to your starting position at each element on the grid, along the way (and then some other moves too). Eventually, you'd end up at the starting position, with a guaranteed way to get to the middle and a series of different ways to get there too.