7 Colors : The Game

J. Peignier & E. Varloot

Due 8th March 2016

Contents

1 Introduction

This whole article is about the implementation of the *7 colors game*, particularly about the basic functions necessary to the functioning of the game and about several AIs that we have to implement.

1.1 Rules

Game Start The game is played on a 30x30 grid. There are 9 colors. 2 of them designate the players' territory. At the start of the game, the upper-right unit square is colored by the first player's (refered to as P1 in what follows) color. The square on the bottom-left is colored with the second player's (refered to as P2 in what follows) color. All other squares are colored randomly with one of the 7 remaining colors.

In this whole document, n is the number of tiles on one side of the grid, and m is the number of colors (non-associated with a player).

Game Play Each player takes turn choosing a color among the 7 neutral. All groups of adjacent tiles with that color and also adjacent to the player's territory are then conquered by the player ; this means that their color turns into the player's color. The turn of that player then ends. The game ends when there are no longer any neutral squares. The winner is the player that has the largest territory at the end of the game.

1.2 Project

In the course of this project, we were asked to implement the game on our computers in the C language, to create several simple AI's and to test their power. The project was phrased as a sequence of questions that led us to the result when answered in order.

We found the project particularly interesting, insofar as this game can serve as an introduction to a lot of turn-base strategy (TBS) games, which are sure more complicated, but in which we could re-use some concepts we used here.

2 Questions & Answers

In this part, we will answer the questions from the project.

2.1 Part 2

Q2.1.: The point of this question was to create the initial grid. We filled all the unit squares randomly with neutral colors, then gave the top-right corner to P1 and the bottom-left corner to P2. This is easier to do than skipping the special squares and filling them in later, and the complexity remains the same : $O(n^2)$. The colors are represented by characters. The 7 neutral are $'A'$ to $'G'$. The players' colors are respectively $'*'$ and $'-'$.
(We chose these characters for the players rather than the suggested ones because these one were easier to see and thus easier to differentiate from the other colors.)

A	A	A	A	A	A	A	A	*
A	D	B	D	G	E	B	F	C
A	A	A	A	A	A	A	A	A
B	C	F	B	G	D	E	F	A
A	A	A	A	A	A	A	A	A
A	D	E	A	B	G	C	B	C
A	A	A	A	A	A	A	A	A
B	C	F	G	B	D	E	D	A
-	A	A	A	A	A	A	A	A

Figure 1: A snake grid ; the worst case for the update algorithm, if P1 plays A on the first turn

Q2.2.: The point of this question consists in implementing a method to update the grid when a player is done playing. The algorithm takes as arguments the player that is currently playing and the color he called. It checks the color of each square in the grid ; if the square has the called color and if it's adjacent to the player's territory, its color turns into the player's color. If the algorithm changed at least one square, it starts again. The algorithm ends when it performed a search through the whole grid without performing any checks, e.g. without changing any tile. We can double check the algorithm by asking it to stop each time it finished and update.

We believe that the worst case for this algorithm is a snake grid ; our algorithm will indeed change one tile at a time on the snake, but each change will take $O(n^2)$ because it is done in one run in the grid. As the snake has approximately $O(n^2)$ tiles (proportionally to the surface of the grid), you would need $O(n^4)$ operations in that case.

Q2.3: *Please take note that we didn't implement the mentioned algorithm here.*
This question can help solving the complexity issues that would arise from the previous algorithm if we were to use a larger grid. There are different possible approaches to this problem, the first is to implement an algorithm which would perform a depth-first search with the origin of the territory as the root, and that considers squares as nodes in a graph ; neighbors would be symbolised by the existence of an edge. The algorithm will then look at each node in the territory only once and will not leave the perimeter of the new territory.

Another approach would be to draw the new border of the player and to color all existing squares inside his territory with the right color. This might seem unacceptable since those squares aren't necessarily the player's color, or the called color. However those squares cannot be accessed by the other player. As long as the player doesn't always call colors that won't raise his score, those squares will eventually belong to his territory. Delaying doesn't change anything from an omniscient point of view.

This approach however might change how the not so omniscient AIs react. To implement this, we start at the origin of the player's territory. You use a variable that marks which direction the algorithm is looking at (initially West for P1, East for P2). The algorithm always chooses the path that turns right as often as it can turn , and that makes it travel on allowed squares (player's or called color). The algorithm thus travels the grid as a maze until it comes back to its origin twice (this is a safety measure, because there are cases where our method wouldn't allow us to go through the whole territory of the player). Then it colors everything that is contained inside the border.

For spider (*Q6.1*) this is the algorithm we use to calculate the border, (however we don't color at the end).

Any territory only reachable by one player is considered surrounded. That is to say the filling in takes in account when the algorithm reaches one of the borders of the grid.

2.2 Part 3

Q3.1: We also want to give the choice to human players to play, either against an AI or against other human players. To achieve that, we'll just create a loop, which will contain one turn for each player, always in the same order. That loop will be repeated again and again to emulate the game. On each turn, you call the corresponding player, and just ask him to pick a color. Then we call the previous function to update the world.

This is a simple implementation, in which we can see some problems :

- The loop is infinite, as we didn't impose any condition to stop the game
- We can't just put what we need to call human players in the loop. The process is indeed the same when two AIs or a human and an AI play against each other : one loop will make the first player have his turn, and then the other one. We have to make functions that will be called in that loop, one corresponding to a human player, and one for each AI that will be implemented.

Q3.2: The surface conquered by each player increases at each turn. But it is always possible, at each turn, for at least one of the players, to increase strictly his territory by playing a color that can bring new tiles to him. (If it's not the case, both players can't win any more cases. That means that the whole board has been conquered by both players, which means the game ended). Considered that the territory of each player is strictly growing, there is a time when the surface conquered by a player reaches 50 percent of the grid. At this point, the other player cannot win anymore, and thus the game can end here.

That's why we implemented one function to compute the score of a given player, and another which tests if one of the scores is greater than 50 percents of the grid, and which is used as a condition in the while loop in which the game is running.

If one of the players can no longer improve his score, e.g. if he is stuck (despite not having reached 50 percents of the grid), he will never be able to enlarge his territory again, he is therefore certain to lose. We could stop the game and announce his opponent as a winner.

2.3 Part 4

Q4.1: In order to create an AI that plays random colors, we pick a random char between $'A'$ and $'G'$ and designate it as the color the player is picking.

Q4.2: To create the improved version we use rejection sampling. That is to say that as long as the designated color doesn't improve the player's score (e.g. that color didn't change the game at all), we re-use the random algorithm to pick another color between $'A'$ and $'G'$ (a same "useless" color can be called any number of times). Once the player's score has changed, his turn ends. The results of rejection sampling are the same as drawing a single random color among the acceptable colors. It however has one weakness. If the player is stuck and can't enlarge his territory no matter what the chosen color is, the rejection sampling cannot stop (because the algorithm is stuck in an infinite loop). To prevent this from happening, we added a condition to the algorithm that determines if it is stuck before starting the rejection sampling.

Another possible solution to avoid the infinite loop would have been to use the stuck condition to stop the game and annonce the winner before the improved player's turn (see question 3.2). This was our first idea and would have been in our opinion a more elegant solution to our problem. However it was harder to implement.

2.4 Part 5

Q5.1: At each turn, the greedy player chooses the colors that will grant him the biggest number of additional squares on that turn. To implement it we created a new player/color called temporary (character $'+'$).The algorithm tries out one after the other, all 7 neutral colors. It emulates the player's move by changing the squares the player would gain to the temporary color : the algorithm chooses a color, converts to the temporary color all the tiles the player would get if he played that color ; it then reads the score the player would get, and remembers the color that obtained the best score. Once it has finished with a color, it turns the squares it changed back to their original color. Once it finished, the greedy AI plays the color related to the best score given by the algorithm obtained.

When the algorithm finds a color which brings a raise which is as big as the raise that it already memorized, it can either keep in memory the color it already memorized (with a probability of $\frac{1}{2}$) or change and keep the new color (with a probability of $\frac{1}{2}$). The probability law isn't uniform for 3 or more equivalent choices ; however this strategy makes the player less predictable and therefore lowers the number of boards on which he is unefficient (see *Q7.1*).

Q5.2: To make the game fair, we can have the players play each grid in both positions. Each battle is therefore divided in two rounds.

P1 plays first and in the top-right corner during the first battle, then it's P2's turn to play that role on the same grid during the second round. To do this we made a copy of the grid before the first round ; after the first round, we transfer the saved board from the copy to the game board ; the second round can then start, after the initial positions of the players have been exchanged.

This way, if one configuration clearly allows one player to outmatch the other, there can be a revenge round were the second player gets a chance to teach respect to his foe.

Q5.3: To determine which AI the best, we implemented a championship system. A championship consists of 100 matches played on different grids, each match consists of the 2 rounds described in the previous paragraph. The improved random AI (built at *Q4.2*) got 0 victory, and the other got 200 (it won each match, and each revenge). Therefore, the greedy AI is by far the best strategy among the strategies we have seen so far.

2.5 Part 6

Q6.1: The spider strategy tries to extend the border (the same way a spider spreads its web) of the player applying it. We therefore use temporary again to determine which color results in the best border size.

We use the algorithm described in question *Q2.3* to determine the size of the border for the territory composed of the player's tiles and the tiles with the temporary color, for each possible move. We choose the best border among those that raise the player's score.

Theoritically, the player is stuck if each choice of a color that raises his score will also lower the size of his border. By forcing the score to strictly increase we therefore sligtly changed the AI's reaction. However no strategy that can result in the player's suicide is valide ; our intervention was therefore necessary to avoid a infinite loop where both players are stuck.

Between two equivalently tempting borders, the choice is made in the same manner as with greedy : randomly.

Q6.2: We then implement a greedy player that will try to take in account the move he's playing, but also his following move (this is done by turning the conquered tiles of the first move into a temporary color, and the ones of the second move into another temporary color), to determine what's the best move for him. We do it for 2 turns, but we could do it on p turns by implementing an alpha-beta algorithm. As this player will try every combination of two consecutive moves (approximately m^2 combinations), and update the board for each combination to see what he would get, with a function with a complexity in $O(n^4)$. Thus the total complexity is in $O(n^4m^2)$

2.6 Part 7

Q7.1: In order to ensure the fairness of the boards in their initial state, we made them symetrical. (Therefore P1 and P2's position don't influence the efficiency of their strategies.)

A strategy will be considered useless if the probability for the advanced random player to win against it is superior to 50%. In other words, if 1 out of 2 non-suicidal strategies win against that stategy.

All strategies are therefore considered improved (same definition as for random). We remind the reader that we implemented our AI being sure to avoid any infinite loops.

For obvious reasons improved-random is always unefficient.

The idea behind this grid is to slow down the greedy algorithm so as to give time to the random player to reach the jackpot. The greedy player always has to choose between conquering one tile or two, and he will always choose to gain two, from the beginning. The same goes for spider. None of them will pick C on the beginning, nor will they pick D, E, F or G after. On this grid the winner is the first player to cross the colored border between the areas that they will have conquered and the jackpot zone, and choses the jackpot color (the jackpot zone covers over 50% of the grid ; in figure 1, it is the big group of A in the top-left corner ; in this example, it doesn't over 50% of the grid, but it does when there are more colors ; we used fewer colors to simplify the representation) . A player using the greedy or spider strategy won't cross until they crossed one of the game sides with the bicolor path (A,B). Which would take over 10 turns in our example.

A random strategy has a 1/2 chances to win in the first two turns. It just has to choose C on the first turn. If it plays A on the first turn it can play C or D on turn 2 and A on turn 3 there is therefore a 7/3 chance to win on turn 3 if it didn't win on turn 2. Therefore these two strategies are unefficient on this board.

For n-greedy (with a sufficent small n), we can use the same board with a border of thickness n that prevents the algorithm from noticing the jackpot-zone. And we have to be sure that the 5 colors C to G are distributed in this border so as to lower the maximum score obtainable. (see exemple for our improved-gready). The objective is, for the number of turns it takes for greedy to reach the jackpot, to remain inferior to the number of turns it takes on average for a random strategy.

Q7.2:For our final AI we decided to try a mix of improved-greedy with spider. The algorithm compares of the potential gain for the border with spider with the territorial gain for improved-greedy.

According to what is considered as best the AI choses what to play. The problem is to determine where to draw the line.

A	A	A	A	A	A	C	*
A	A	A	A	A	D	A	A
A	A	A	A	A	E	B	B
A	A	A	A	D	A	A	A
A	A	A	D	E	B	B	B
A	D	E	A	B	A	A	A
C	A	B	A	B	A	B	B
-	A	B	A	B	A	B	B

Figure 2: Greedy & spider worst grid example, on a 8×8 grid and with 5 colors

A	A	A	A	A	A	F	C	*
A	A	A	A	A	F	D	A	A
A	A	A	A	A	D	E	B	B
A	A	A	A	A	E	F	A	A
A	A	A	A	F	D	B	B	B
A	F	D	E	D	E	A	A	A
F	D	E	F	B	A	B	B	B
C	A	B	A	B	A	B	A	A
-	A	B	A	B	A	B	A	A

Figure 3: Double-greedy worst grid example, on a 9×9 grid

3 Conclusions

Here we emulate a fight between every combination of AIs, the fight being constituted with 1200 matches of 2 rounds.
The number of victories is counted on rounds. In row i and column j, we put the number of victories of player i against player j.

The ranking (or tier-list) appears to be as it follows :

1. Mix and Double Greedy
2. Greedy
3. Improved Random
4. Spider

As expected, the random strategy is no good way to win against a greedy player (this strongly depends on the map you're playing in, but on the average board, you will almost always lose). But surprisingly, the spider strategy is no match for the random player ; any strategy which is outmatched by a random strategy cannot be considered a good way to win.

However, the idea of a spider strategy isn't necessarily bad, but it's too complicated and not efficient enough when it's alone ; that's why it gets beaten a lot by other players. But this strategy can be combined with others, in order to implement a player which beats every other one. This is what we did with the mix, which seems to be tougher than the other, even a little tougher than the double greedy. Though it is strongly based on the latter, it was implemented in order to go for a spider strategy when a spider move proves to be better than a greedy move.

	Improved Random	Spider	Greedy	Double greedy	Mix
Improved Random		1609	3	1	0
Spider	791		15	0	2
Greedy	2397	2385		458	452
Double Greedy	2399	2400	1942		1183
Mix	2400	2398	1948	1217	

Nevertheless, the choice between the two strategies applied by mix is based on a comparison between what both strategies would get, and but each strategy is given a weight ; the choice is done by comparing the gain multiplied by the weight. We could test other values for the weight of each strategy, in order to improve even more this mixed AI.

4 References

[1] https://openclassrooms.com/courses/les-pointeurs-sur-fonctions-1