Alex Thayn
Final Project Part 2

The Complexity of Flood Filling Games

Main Idea

Flood-it is a combinatorial game where each tile of the game board is assigned one of the *c* possible colors. Two adjacent tiles are connected if they have the same color. The main goal is to flood the entire board so it contains only one color in the least number of moves as possible. The paper shows that a greedy approach is not an optimal solution, but that the optimal approach is **NP**-hard for c ≥ 3.

A variant of the game is Free-Flood-It where the player chooses not only the color but also the location to flood. Free-Flood-It with c = 2 is solvable in **P**. The height of the board also affects the complexity, and Flood-It is in **NP**-hard for rectangular boards with a height of at least 3, and any number of colors c. The solution is in **P** for boards of height 2.

The greedy approach to this game might be tempting, but the two most obvious greedy approaches can be pretty bad. The first greedy algorithm is where the player picks the color that results in the largest number of tiles gained. The second greedy approach is where the player chooses the color dominating the perimeter of the currently flooded region. The paper illustrates an example where the optimal solution only takes 3 moves but both greedy approaches require 10 moves to solve the board. I included the example from the paper (pg. 4) below that demonstrates this problem.

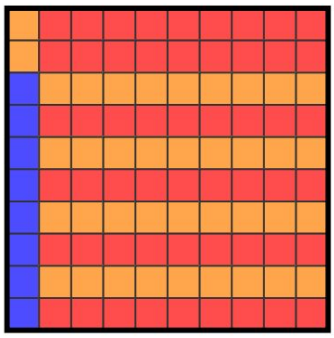


Fig. 3: A 10×10 board where a greedy approach is bad.

They introduce the c-Flood-It problem which takes input of an *n* x *n* board B of c colors and outputs the minimum number of moves *m*(B) required to flood B. The c-Free-Flood-It version is similar but adds the ability for the player to choose which tile to flood fill from. They prove that both these variations of the game are in **NP**-hard, for c ≥ 4 colors by reducing from the *shortest common supersequence* problem (SCS). They used a more specialized reduction for the c = 3, case. I won't go into that proof for my project because I want to focus on the (c,2)-Flood-It problem which is in **P** and has a running time of O(n).

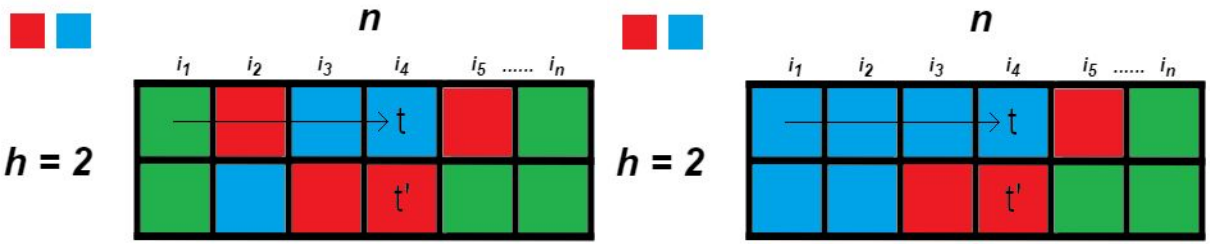
The Algorithm

The notation (c,h)-Flood-It defines a fixed height h x n board with c colors. When h ≥ 3 the problem **NP**-complete. For h = 2 and c ≥ 1, the solution to a (c,2)-Flood-It board is in **P**. Theorem 4 from the paper proposes a solution to c-Flood-It on a 2 x n board using the smallest number of moves possible.

Start: Let *i* be the leftmost marked column such that *i* contains a marked tile t that has not been flooded. Let t' be the other tile in column *i* (Example is shown in the diagram above). With this definition, we have two cases.

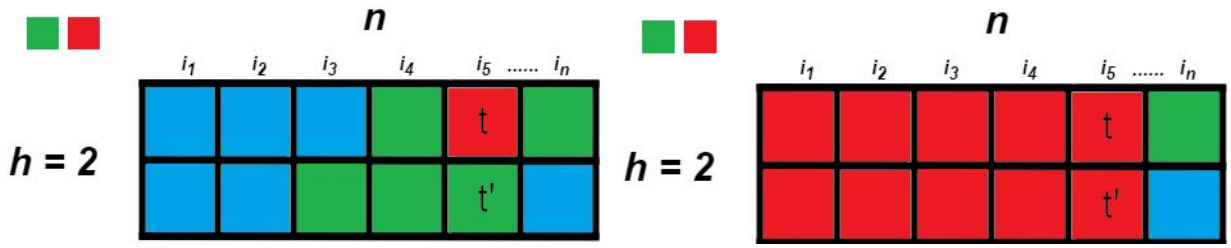
Case 1: (*t'* **is unmarked**) Let *m* and *m'* be the lengths of the shortest paths to *t* and *t'*, respectively.

> **Case 1a:** (*m* ≤ *m*'). Flood using the sequence of colors found along the shortest path to t, then go to the beginning of the procedure.
>
> - For the example below m = 2 and m' = 2

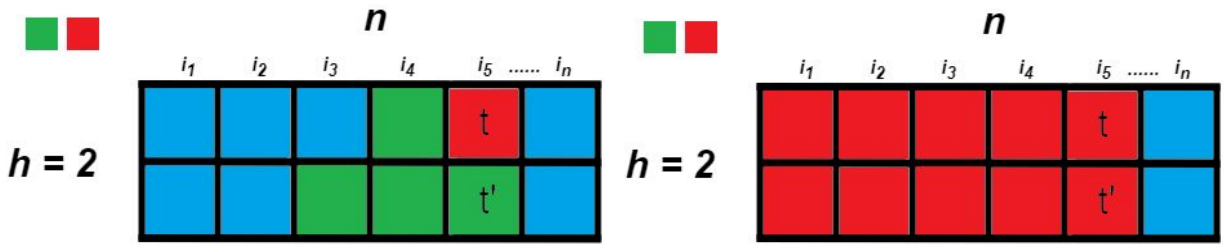


> **Case 1b:** (m > m'). Flood using the sequence of color found along the shortest path to *t'* and then flood *t.* The go back to the beginning of the procedure.

- For the example below $m = 2$ and $m' = 1$



Case 2: (*t'* **is marked).** Flood using the sequence of colors found along the shortest of the shortest paths to *t* or *t'.* Then flood the remaining tile in column *i* and go back to the beginning of the procedure.



Correctness

Case 1: The case where *t* is a marked tile and *t'* is an unmarked tile.

> **Case 1a:** For this case $m \leq m'$ and we are supposed to take the shortest path to t which involves m moves. If we were to flood *t'* first and then *t* it would take at least $m + 1$ moves, and because *t'* is not a marked tile we don't need to spend an extra move on flooding it.
>
> **Case 1b:** For this case $m > m'$ so we flood the sequence of colors along the path to *t'* and then flood *t* immediately after. This is optimal because to flood just *t* would take at least *m'* moves so if we flood *t'* too it doesn't cause any extra work to be done, but we are further to the solution.

Case 2: This involves the case where both *t* and *t'* are marked tiles. In this case, we select the shortest path of the two and flood it, then flood the remaining tile in the column. This is an

optimal approach because both tiles need to be flooded eventually, so we select the shortest path to that column and then make one more flood move to flood the remaining tile.

Resource Requirements

The running time for this algorithm is O(n). We can use a dynamic programming solution to find the shortest path to any tile *t* on a 2 x n board, which can be computed in linear time based on the distance between the flooded region and *t*. The shortest path is calculated starting from the rightmost end of the flooded region to the marked column where *t* is located.

If we look at the worst case solution where c = 2n so each tile is a different color, it would take 2n-1 moves to flood the board. Therefore the big-O running time for this worst case problem is O(n).

Implementation

For the implementation, I wanted to make a graphical solution. I found a pygame tutorial for the flood-it game here: Flood-It! Tutorial and I modified the code to fit the algorithm by creating a 2x14 grid and wrote the solution to be able to interact with the UI. The full code is included along with the submission of this project, or at my github: https://github.com/alexthayn/AlgorithmsFinal.

My solution followed the algorithm presented above. I created a function that found all the marked tiles on a board. Then it found the shortest path from the rightmost region of the flooded area to the marked tile itself. I interacted with the game interface to update the UI so we can see the progression of the solution.

To find the shortest path I started by creating a “cost path matrix” that represented the costs required to move from the source tile to the target tile. The code snippet below shows the function that finds each marked tile on the board and returns them in order from leftmost to rightmost.

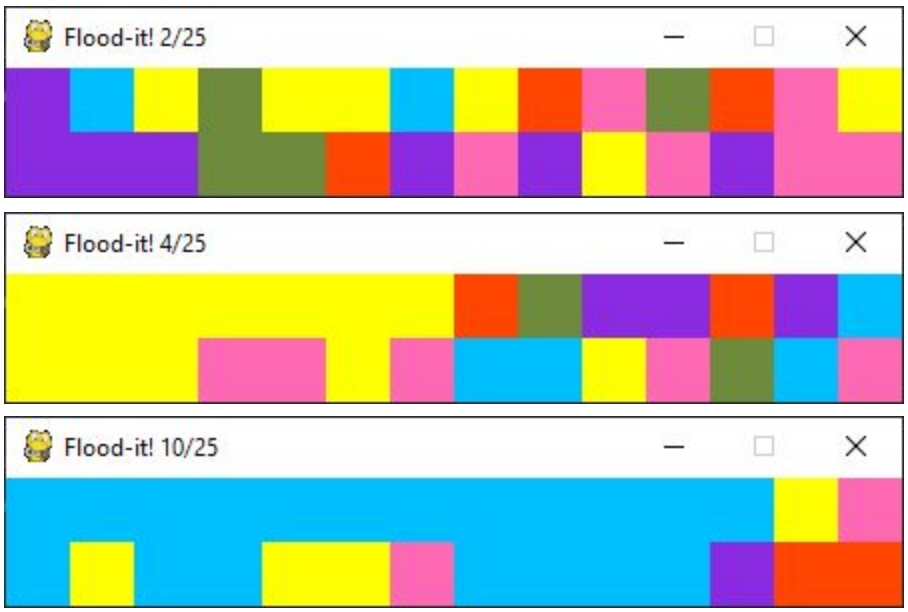
```
def findMarkedTiles(board):
    # There is a marked tile for each color on the board
    markedTiles = dict()

    for i in range(14):
        for j in range(2):
            curColor = board[i, j]
            markedTiles[curColor] = [i, j]

    # order the tiles from righmost located to leftmost located
    orderedTiles = []
    for t in markedTiles.values():
        orderedTiles.append(t)
    return sorted(orderedTiles)
```

Using this matrix the shortest path was calculated and executed. I initially wanted to implement a dynamic programming solution to finding the shortest path, but I ended up using Dijkstra's algorithm by creating a weight graph and passing in a source (the rightmost tile of the flooded region) and a target tile (the location of the marked tile). I then executed the path from source to target by flooding the path in order starting with the source color and continuing in order until the target tile is flooded. The code snippet below shows the execution of the flooding of the shortest path. It will only flood the next tile if it is a different color than the previous one in the path, otherwise, it will move to the next one in the path.

```
def executeShortestPath(location):
    # get the cost matrix
    costMatrix = createPathCostMatrix(watchlist[len(watchlist)-1]
                                      if len(watchlist) > 0 else [0, 0], location)
    graph = createGraph(costMatrix)
    source = ""
    if len(watchlist) == 0:
        source = "0-0"
    else:
        source = str(watchlist[len(watchlist)-1][0])+"-"+str(watchlist[len(watchlist)-1][1])
    target = str(location[0])+"-"+str(location[1])
    path = nx.shortest_path(graph, source=source,
                            target=target, method='dijkstra')
    prevNode = None
    for node in path:
        # flood the next tile if it is different color than the previous
        if prevNode != None and tiles[prevNode] != tiles[node]:
            print(node)
            setTile(tiles[node])
        prevNode = node
```

Example

The pygame project creates a random gameboard and animates the solution that follows the algorithm above. The number of moves required to solve the board ranges from 12-15 on average. The solution to this 2 x 1r4 board can be found in polynomial time, but if the height of the board were to increase the solution would be in NP-hard.



(Screenshots of 3 different game grids)

Alex Thayn
Final Project Part 2

Works Cited

Clifford, Raphaël, et al. “The Complexity of Flood Filling Games.” *Theory of Computing Systems*, vol. 50, no. 1, 2011, pp. 72–92., doi:10.1007/s00224-011-9339-2.

“Flood-It! Game in Python 2.7 with Pygame.” *I.am.arvin*, 18 Apr. 2012, arvinbadiola.wordpress.com/2012/04/18/flood-it-game-in-python-2-7-with-pygame/.