The Tile Game

Teo Gelles

December 12, 2012

Contents

1 Abstract

Last year, the Matlab Programming Contest released a problem that they called The Tile Game. The rules were rather simple, given a 2x2 grid of square "tiles", find the configuration with the best possible score. The score was calculated based on the numbers on the edges of the tiles. To find it, sum the numbers on the outer edges of the tiles, plus the differences between the numbers on touching edges. The goal of this project was to develop a Matlab-based solution to this problem, a goal which was succesfully reached.

2 Introduction

The goal of this project, as previously stated, was to come up with an effective solution to the so-called Tile Game. What makes this problem interesting is that it is very difficult for a human alone to solve, but for a computer can be relatively trivial. The simple nature of the problem means that not a significant amount of previous computer experience is necessary to understand or solve it, and yet a natural ability with programming experience is a definite boon. Finally, working through this project allows the programmer to practice important and basic Matlab skills.

3 Background/Theory

This project ended up requiring few high-level programming concepts. That being said, there are some ideas that must be understood to appreciate the problem. Most important to the project, and particularly its failings, is the primary method of measuring algorithm performance in computer science, called Big-O Notation. To put it simply, when given a data set of size n, the Big-O Notation of an algorithm performing on the set will be approximately equal to how many times the algorithm goes over the set. For example, if the algorithm goes over the entire set twice, the Big-O Notaiton will be O(2n), generally approximated to simply O(n). If, however, the algorithm need only go over a set number of data members no matter the size of the entire set, say 20 items, the algorithm would be O(20), or rather O(1). From the Big-O Notation of an algorithm, the efficiency can be predicted across almost all computer systems, regardless of processor capabilties or speed.

4 Project Design

The solution to this project is about as simple as any solution could be: try every possibility, and pick the best one. The success of this solution is predicated on the ability of a computer to perform calculations far more quickly than any human. If a normal person were asked to try all solutions to a given configuration in The Tile Game, it would take them several hours at least. However, even

the most basic pc can perform the necessary calculations in a trivial amount of time.

Given a grid of four squares, there is a limited set of possible configurations. Firstly is tile placement; Since there are four spots, and four tiles to place in the spots, there will be 4x3x2x1 = 24 possible ways to place the tiles. The other factor in the entire tile configuration is the rotation. Each tile can be rotated in four ways, so with four tiles there are 4 x 4 x 4 x 4 = 256 ways that the tiles can be rotated. All together, this gives us 24 x 256 = 6144 possible combinations in total, a large number for a human, but very small for a computer.

The central solving part of this program makes extensive use of for loops. Specifically, there is a for loop for each possible tile placement and rotation, making 8 loops in total, with 5 running 4 times, 1 running 3 times, 1 running 2 times, and 1 running just once. Each time a diffirent tile configuration is found, its score is calculated and compared against the current best score, if it is better, it becomes the new best score. To perform these actions the program uses three functions, a calcsol function which takes a cell matrix representation of a tile grid and calculates the score, a turn function which rotates a specified tile by 90 degrees, and a swap function which changes the position of two given tiles.

5 Results

Strictly speaking, the project was a success. The program is able to take any set of four tiles, and from them find the best possible configuration based on score. It does this in a matter of seconds, and prints out the results in a user-readable fashion. However, this by no means that the program is perfect. Although it is capable of finding the correct solution, it does so in the least efficient method possible: trying every possibility. The lack of a more powerful solution may not be of significance in the exact case given by the problem, but if any extension is to be implemented the efficiency may become more of an issue.

6 Conclusion

To fully understand the primary issue with the solution to this project, efficiency, it is best to implement a Big-O analysis. Given n tiles, each with m edges, the program will test every possible placement and rotation of the tiles. In Big-O Notation, this would be equivilent to $O(n!m^n)$ which is a particularly poor runtime. When n and m are both 4, the runtime is barely noticeable, but with only a few more tiles or edges the time taken by the program will become significant. This is the basis for the most obivous, and most important extension to this project: generalization. Though the current solution could be easily implemented to solve any number of tiles and edges, it will do so in a very inefficient way. What better solutions exist?

7 Appendix

7.1 Main Program File: Final.m

```
%% Final Project: The Tile Game
% Teo Gelles

% When given four tiles with four edges, calculates the best possible
% "score" and tile structure by the rules of the tile game
tiles = cell(4, 5);
for i=1:4
    for j=1:4
        fprintf('Enter Value For Tile %d, Space %d', i, j);
        tiles{i, j} = str2num(input(':', 's'));
    end
    tiles{i, 5} = char('a'+i-1);
end
permtiles = tiles;

bestsolval = -1;
bestsolcell = tiles;

% This algorithm is sheer brute-force, it checks every possible
% combination of tiles for the best answer

% The outer four for loops represent which tiles go in which spaces
for w=1:4
    tiles = swap(permtiles, 1, w);
    for x=1:4
        if x~=w
            tiles = swap(permtiles, 2, x);
        end
        for y=1:4
            if y~=w && y~=x
                tiles = swap(permtiles, 3, y);
            end
            for z=1:4
                if z~=w && z~=x && z~=y
                    tiles = swap(permtiles, 4, z);
                end
                % The inner four for loops represent the rotations of tiles
                % in their spaces
                for a=1:4
                    for b=1:4
                        for c=1:4
                            for d=1:4
```

```
                                sol = calcsol(tiles);
                                if sol < bestsolval || bestsolval == -1
                                    bestsolval = sol;
                                    bestsolcell = tiles;
                                end
                                tiles = turn(tiles, 4);
                            end
                            tiles = turn(tiles, 3);
                        end
                        tiles = turn(tiles, 2);
                    end
                    tiles = turn(tiles, 1);
                end
            end
        end
    end
end

% Tiles are represented as text placed on a graph
fprintf('Solution: \nInitial Score: %d\nBest Score: %d\n', calcsol(permtiles), calcsol(bests
subplot(1, 2, 1);
for i=1:2
    text(.3*i-.2, .2, num2str(permtiles{i+2, 2}));
    text(.3*i-.2, .5, num2str(permtiles{i, 2}));
    text(.3*i-.1, .1, num2str(permtiles{i+2, 1}));
    text(.3*i-.1, .2, permtiles{i+2, 5});
    text(.3*i-.1, .3, num2str(permtiles{i+2, 3}));
    text(.3*i-.1, .4, num2str(permtiles{i, 1}));
    text(.3*i-.1, .5, permtiles{i, 5});
    text(.3*i-.1, .6, num2str(permtiles{i, 3}));
    text(.3*i, .2, num2str(permtiles{i+2, 4}));
    text(.3*i, .5, num2str(permtiles{i, 4}));
end
subplot(1, 2, 2);
for i=1:2
    text(.3*i-.2, .2, num2str(bestsolcell{i+2, 2}));
    text(.3*i-.2, .5, num2str(bestsolcell{i, 2}));
    text(.3*i-.1, .1, num2str(bestsolcell{i+2, 1}));
    text(.3*i-.1, .2, bestsolcell{i+2, 5});
    text(.3*i-.1, .3, num2str(bestsolcell{i+2, 3}));
    text(.3*i-.1, .4, num2str(bestsolcell{i, 1}));
    text(.3*i-.1, .5, bestsolcell{i, 5});
    text(.3*i-.1, .6, num2str(bestsolcell{i, 3}));
    text(.3*i, .2, num2str(bestsolcell{i+2, 4}));
    text(.3*i, .5, num2str(bestsolcell{i, 4}));
end
```

7.2 Function File 1: calcsol.m

```
function [ x ] = calcsol( c )
% Given a cell array, calculates the score of that array's representation
% as a tile game
x = c{1, 2} + c{1, 3} + c{2, 3} + c{2, 4} + c{3, 1} + c{3, 2} + c{4, 4} + c{4, 1};
x = x + abs(c{1, 1}-c{3, 3}) + abs(c{1, 4}-c{2, 2}) + abs(c{2, 1}-c{4, 3}) + abs(c{4, 2}-c{3
end
```

7.3 Function File 2: turn.m

```
function [ C ] = turn( C, n )
% Given a cell array with each row representing a tile,
% "rotates" the nth tile 90 degrees clockwise
w = C{n, 1};
x = C{n, 2};
y = C{n, 3};
z = C{n, 4};
C{n, 1} = z;
C{n, 2} = w;
C{n, 3} = x;
C{n, 4} = y;
end
```

7.4 Function File 3: swap.m

```
function [ C ] = swap( A, x, y )
% Given a cell array, swaps row x and row y in the array

C = A;
nx = C(x, 1:5);
ny = C(y, 1:5);
C(x, 1:5) = ny;
C(y, 1:5) = nx;
end
```

7.5 Sample Output