Nathan Williams (nlw2sx)
November 15, 2015

# CS 4102 Algorithms Term Project: Tiling Puzzle Solver

## Discussion

It is conceivable that as we continue through this age of big data, searching massive problem spaces will grow in importance as one of the greatest challenges of modern computing. Of course, there is a small subset of cases where a brute force approach is sufficient for any of these problems; however, as the size of the problem inevitably grows, brute force quickly fails because there is not enough time to make that solution practical. Consequently, we are left to find alternative approaches.

One such problem is the tiling puzzle. My strategy for solving the problem was as follows:

1. Create a graphics module to aid in visualization of the problem
2. Implement a brute force algorithm
3. Identify situations where I could “cut branches” and “backtrack”
4. Redesign my approach based on gathered insights

All work was done using Python 2.7. The graphic interface, powered by TKinter, was designed by distinguishing the puzzle into a hierarchy of moving parts. The Game was essentially the master core of the system, which managed the Board and the TileBucket. The TileBucket handled all of the Tiles. Further explanation of the *game.py* module can be found in the Documentation.

The brute force algorithm was based on the idea that there are only so many ways that $n$ tiles could be ordered in a list; in fact, there are $n!$ ways. If one were to place the tiles in a well-defined and consistent manner, then in order to find all solutions, one need only test every permutation of the tiles. Tiles were placed using a top-left rule, where the left-most block of the top row of the tile is aligned with the next open block on the board, starting from the top and moving left to right. Although this pure and simple method was functional, it did so in $O(n!)$ time.

Notice the implications of the placement rule, though. Since the tiles are placed in order, if the *i*-th tile in the list cannot be placed in the next open position after the tiles before it have been placed, then no permutation that include tiles 0 through *i* can be a solution. Thus, in such a case, the rest of the tile list need not be permuted and that branch may be cut. We move directly on to the next tile that might fit in the *i*-th position.

As a result of being able to watch my program solve the problems via brute force, I realized that sometimes a discontiguous region would form on the board surrounded by either tiles or the boundary of the board. Due to the placement scheme, these regions would not necessarily be addressed in the next attempted placement. Consider the case shown in Figure 1 where one of these holes has formed, but will not be noticed by the current algorithm for quite a while.

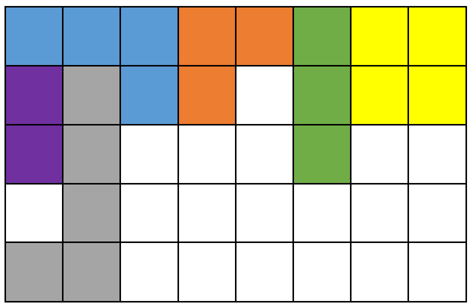
Figure 1. Discontiguous region formation

Based on the placement scheme described above, the tiles are placed in the following order: blue, orange, green, yellow, purple, and gray. The lone space in the first column of the fourth row is presumably unfillable with the assumption that there are no single block tiles in the tile bucket. Although this branch can be cut now, this space will not addressed by the algorithm until the second and third rows have been filled. It would be possible to check the spaces on the board surround a tile that has just been placed for small discontiguous regions where no or few tiles could be placed. This becomes difficult and resource intensive, though, as we consider discontiguous regions of variable size and shape. This observation may not have been practical for the brute force method, but it did partially inspire the revised approach.

I had gotten me pretty far, but I knew I could not increase the efficiency to a point I was happy with regardless of how clever I was with my branch cutting. The branching factor was simply too high. In the brute force method, the root spawned $n$ branches, each of which would spawn $n - 1$ branches, and so on.

In search of muse, I looked to existing algorithms from which I might extract generalizations. When I came across the Dancing Links algorithm, the word “constraint” popped out immediately. My brute force algorithm was very tile-centric, while the board spaces were what had to be filled in order to find a solution. I realized that I could instead treat the board spaces as “constraints” and maintain lists of possible tile placements that might satisfy those constraints. The constraint with the least number of candidate moves could then be chosen as the branching point. Minimizing the number of branches is beneficial because when we are able to cut the branch, more possibilities are eliminated.

We can further this way of thinking by also treating each tile as a “constraint” and maintaining lists of possible board placements for that tile. If the tile constraint with lowest number of candidates has even fewer candidates than the favored board constraint, then we can branch with that tile constraint instead. Keep in mind, though, that there may be extra tiles in the tile bucket. As such, if the puzzle can be solved without the tile that has been chosen to branch with, then we still have to branch with board after eliminating the selected tile from both constraint lists.

Finally, I realized I was wasting significant amounts of time with repeated solutions. I could reduce the number of times that I run into identical solutions by creating tile groups from the tiles. Instead of solving using the tiles, then, I used the tile groups. This was a particularly helpful insight considering the fact that many of these puzzles have identical tiles.

## Results

For a table of runtime results for many test puzzles, see Table 1 of the Appendix. All puzzle solves were executed with tile-flipping allowed and multithreading turned off. For additional details regarding each of the puzzles, see the Documentation.

## Appendix

Table 1. Runtime Summary

| Puzzle name | Number of solutions | CPU runtime for 1 solution (seconds) | CPU time for all solutions (seconds) |
|---|---|---|---|
| Checkerboard | 991 | 0.812 | 5464.017 |
| Custom1 | 32 | 0.047 | 0.328 |
| Partial cross | 122 | 1.078 | 111.300 |
| [3x20] | 2 | 6.015 | 109.397 |
| [4x15] | 368 | 3.309 | 3352.548 |
| [5x12] | 1010 | 1.969 | 9593.264 |
| [6x10] | 2336 | 1.437 | 18811.666 |
| [8x8] middle missing | 65 | 11.721 | 1156.806 |
| Simple | 1 | 0.000 | 0.000 |
| Test 1 | 1 | 0.031 | 0.126 |
| Test 2 | 1 | 0.047 | 0.110 |
| Thirteen holes | 2 | 0.828 | 10.002 |
| Trivial | 1 | 0.000 | 0.000 |