Project: Tiling Puzzle Solver

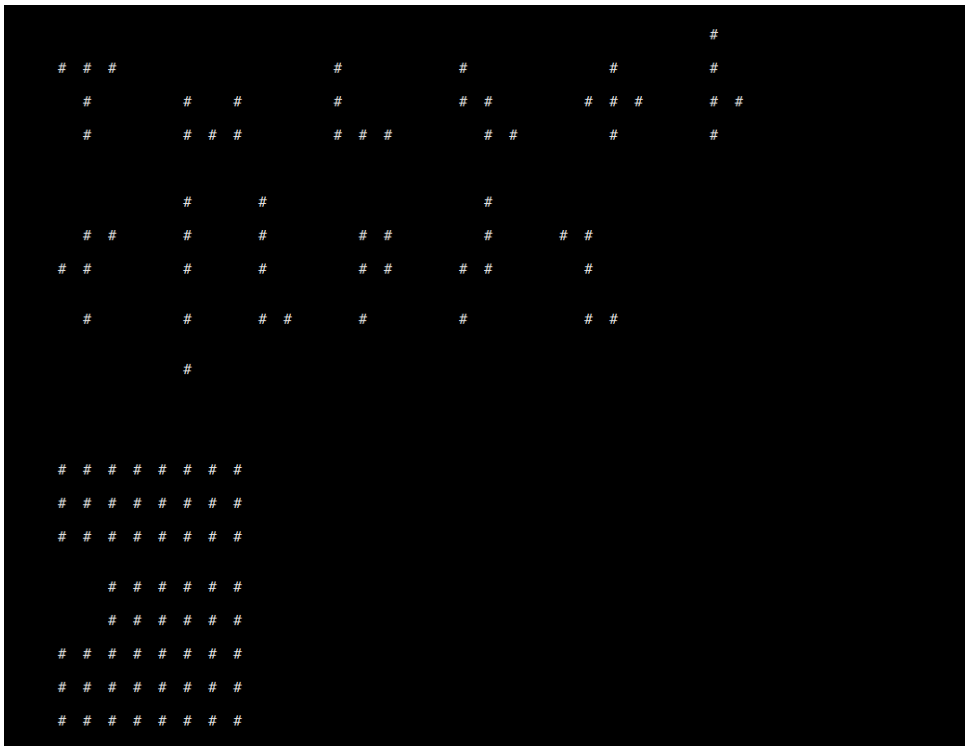
Yung Chih Chen

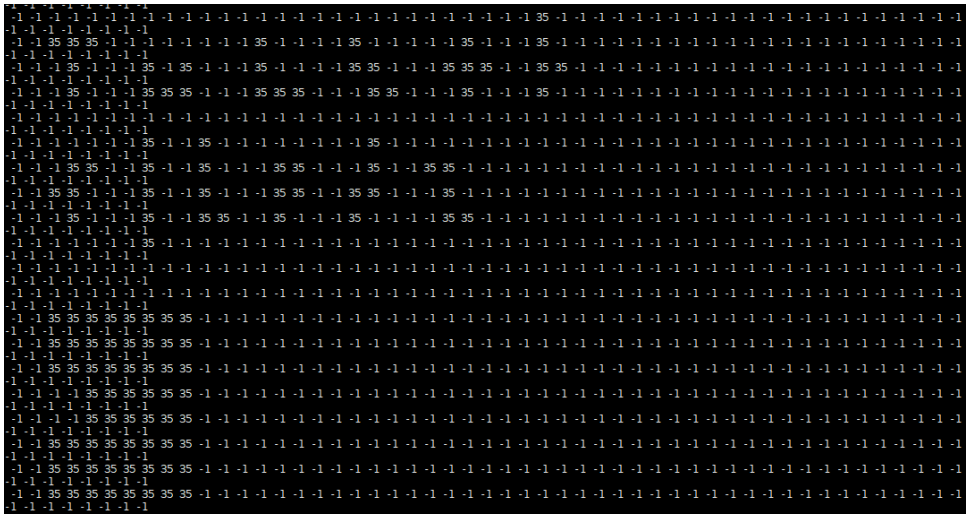
Problem Description:

This project entails implementing a program that will solve arbitrary tiling puzzles. The purpose of this project is to introduce a rich branch of discrete mathematics / combinatorics that involves tiling, symmetry, counting, solution space searching, branch-and-bound pruning, etc. Rotations and reflections need to be considered in this project.

Data Processing:

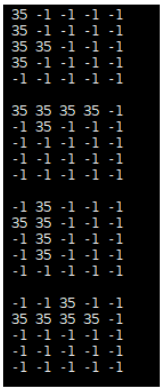
To begin with, the program will read an input file and transfer it to a large matrix in ASCII code. The diagram below is an example.





Since negative value does not exist in ASII code. We recognize blank as the negative value. Then, the program split this large matrix into a number of small matrixes. Each small matrix represents a piece and the largest one is the goal matrix. These pieces can rotate. Thus, a piece has four orientations and we use an index to distinguish them. On the other hand, the shapes of some pieces are identical after rotating. Thus, if a piece is symmetrical, the number of orientations of this piece will reduce.

The diagram below is an example. “orientation number” represents one kind of rotations.

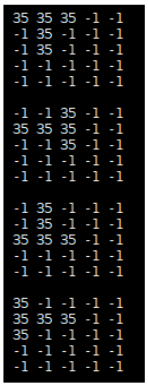


orientation 0

orientation 1

orientation 2

orientation 3



orientation 0

orientation 1

orientation 2

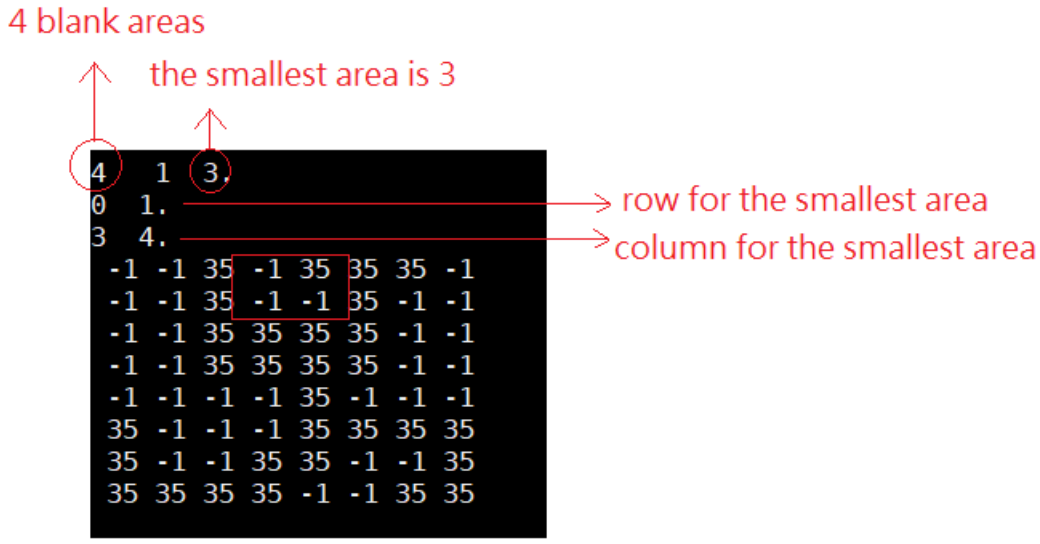
orientation 3

Algorithm:

This is an NP-hard problem and there is no polynomial time algorithm that can handle this problem. To solve it, we build a recursive, non-deterministic, depth-first, backtracking algorithm that cover all possible solutions. All of pieces will be placed in a board in every orientation. If a certain piece is unsuitable, the program will track back to the previous piece. In order to avoid unnecessary branches, the program will check if the placement of a piece is suitable before this piece is placed.

To speed up and eliminate a large number of branches, I adopt two additional algorithms. First, if there are multiple blank area, the smallest blank will be tilled at first. If no solution for the smallest area, it will track back to avoid tilling the rest of area. This strategy will tremendously remove branches.

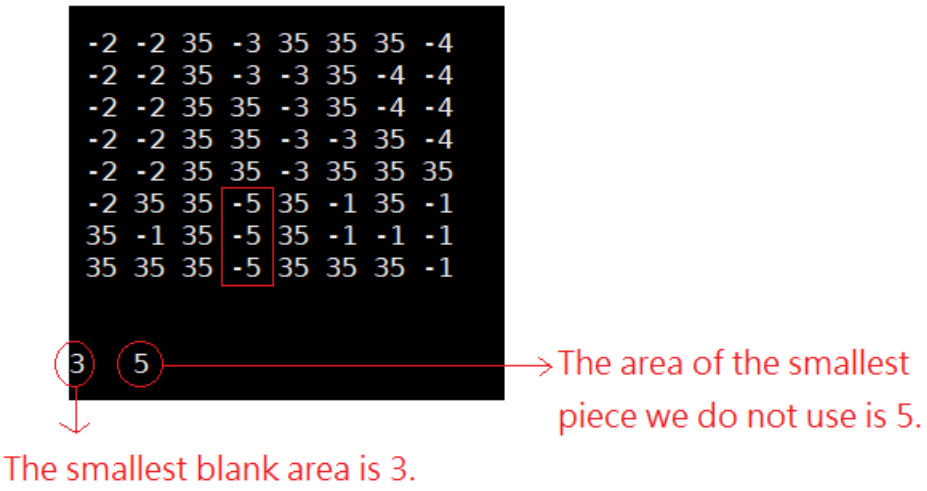
The diagram below is an example.



In this case, the program will target the smallest area, and then fill this area at first. The row of this area is 0~1, and the column is 3~4. If there is no solution in this area, the program will not fill the rest of blank. It will track back.

Second, if the tiniest blank area smaller than the tiniest piece we do not use yet, it will track back because there is no solution.

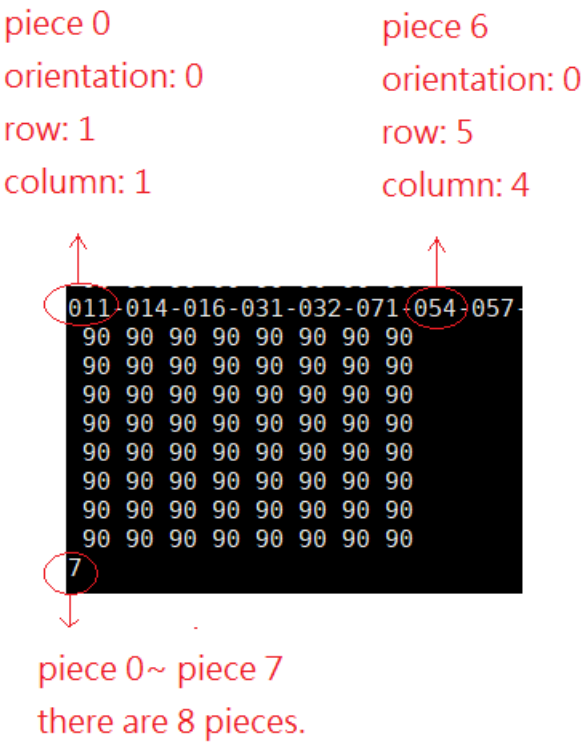
The diagram below is an example.

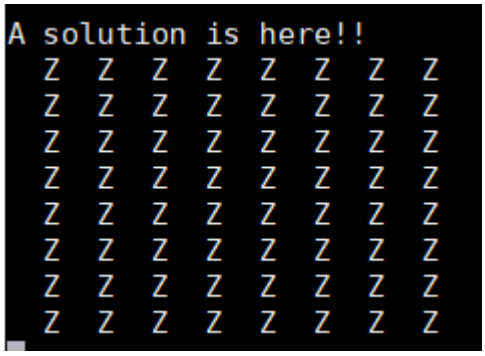


In this case, the smallest blank area is 3. The area of the smallest piece which is not used is 5. Thus, there is obviously no solution in this situation. The program will stop and track back.

When program find a solution, it will display the final aim and the combination of the pieces.

The diagram below is an example.





Performance:

Puzzle name	CPU time
IQ_creator	2 second