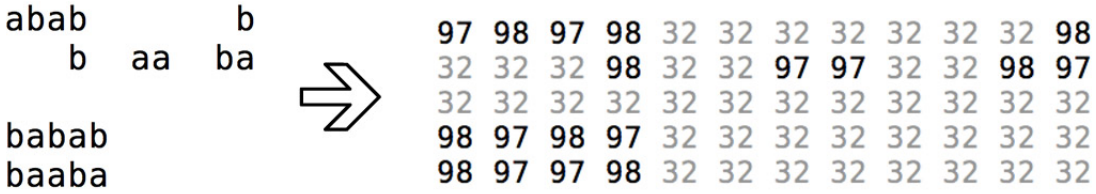
Aaren Barge (jab9rm)
CS 3102 Project Documentation

Overview

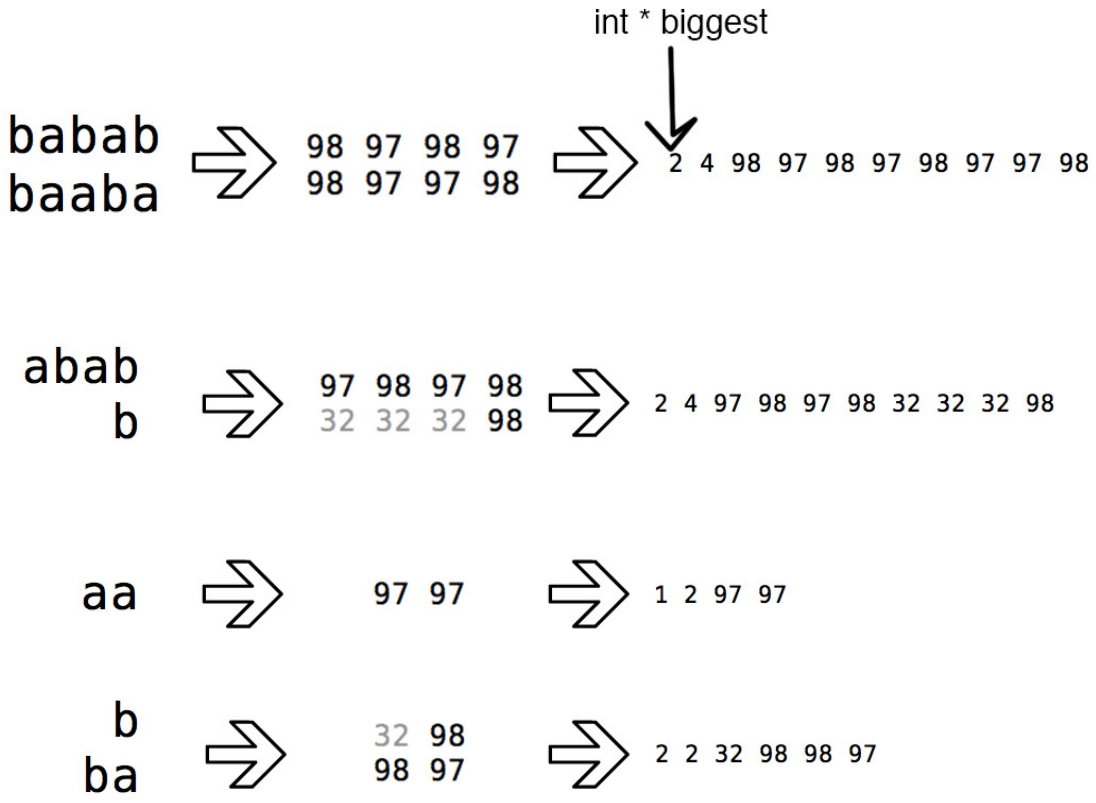
The program I've written to solve the general 2D tiling problem takes a properly formatted (as per the specifications on Robins' website) and displays the requested number of non-isomorphic solutions. In order to accomplish this, I have written three files: **Solver.c** which takes at least one and up to two parameters from the command line. The first is the file name of the input file which outlines the pieces and target configuration, the second is an optional flag which lets users toggle the rotation settings of the solver. If a "1" is provided on the command line, the solving routine allows pieces to be rotated and reflected up to isomorphism, otherwise no rotation or reflection is considered by the routine. Solver.c writes the non-isomorphic solutions to a file "c_output.txt". Additionally, I've written **drawbox.py** which is a python script that reads "c_output.txt" and draws the solutions graphically. This script takes as command line arguments the name of the file to be read, and the maximum number of solutions to be displayed. Finally, I've written **solve.sh** which is a shell script that automates the process of running the C executable and piping its output into drawbox.py as well as handing variability in the user flags. Additionally I've included a makefile that will compile the C source into ./a.out executable.

Data Representation

In order to properly represent the pieces and target configuration, I used a 2D array of integers. The values in the array are the ASCII encoded values of the characters that appear in the array. For example, an input file like the simple one that follows will be represented in full with 97 signifying 'a' and 98 signifying 'b' and 32 everywhere else.



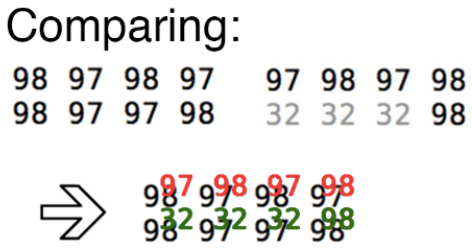
My algorithm takes this full file and searches for blocks of numbers other than 32. When it finds a block of numbers it represents each piece of the puzzle again as an integer array. However because I've implemented my program part in C, there is no built in way to store dynamically allocated 2D arrays so I created my own standard just for use within my program. The first value in the array is the height, the second value (at position 1) is the width, and the values in the array start at location 2. Here is how the above example would be translated into pieces:



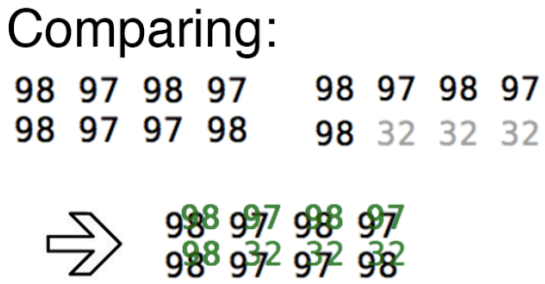
Note that the value 32 specifies that there is a "hole" in the piece at that location, because most puzzle pieces are not perfectly rectangular many will have "holes" represented with the value 32 (ASCII value of space).

Solution Finding Algorithm

My solution finding algorithm works as a brute force algorithm. It chooses a piece that it has not yet considered (the pieces are arranged in a list, so the algorithm just selects the next piece) and tries to fit that piece in every possible location in the target configuration array. This is done by iterating through the smaller array (the piece) and examining each value. If the value is positive and not equal to 32, that value must match the corresponding value in the target configuration array. For example, an unsuccessful comparison would look like this:



For the comparison to be successful, that is, the piece will fit in the target configuration at the desired location, every non negative value in the piece that is not equal to 32 must match exactly the values in the target configuration array. An example of a successful comparison is shown below:



The solution finding routine loops through every possible rotation and reflection of a piece, in every possible position in the target configuration, and when it finds a match, it alters the target configuration by negating those values the above piece occupies. Then it recursively calls the solution finding routine with this altered configuration and only those pieces that have not yet been considered. Because this problem exhibits the optimal substructure property, the solution to the smaller problem (fewer pieces and a smaller target configuration) can be combined with the knowledge of where this piece fit in the configuration to create a more complete solution.

The whole algorithm then is very similar to a Depth First Search algorithm because at the highest level (the first piece it tries to fit in the puzzle) when it finds the first place it can fit, it immediately calls the solution finding routine on the next lowest level. In order to find even the first complete solution, the algorithm must completely solve the sub problems arising from this first piece placement. Then the algorithm can continue trying other configurations of the first piece until again it finds a match, and the recursion begins again.

The algorithm is simple in theory, find a piece that fits and then find the solutions around that piece, but implementing it proved to be difficult. The most challenging part of this algorithm was finding a way to store the solutions that had been found for any level of the recursion that allowed them to be recombined at a higher level into a complete solution. I designed data structures I call the SolutionList, SolutionNode, and PointerList in order to store all this data.

The SolutionList has a pointer to a SolutionNode and a pointer to the next SolutionList. An instance of a SolutionList is a piece fitting into the puzzle at the current level of recursion. Because there are many places that a piece could fit into a puzzle at any given level or recursion, the SolutionLists can be strung together with their 'next' pointer. Within the SolutionList, there is a pointer to a SolutionNode which stores the necessary information about where and how a fit was found in the puzzle. There is an int array within SolutionNode that stores the piece ID, the rotation ID, the x coordinate, and the y coordinate of where the piece

was found to fit. Then the SolutionNode has a list of pointers that point to the next lowest level of recursion. These point to the SolutionNodes of the next piece that were found to fit the puzzle around the piece we're currently considering.

Every time that the solution finding algorithm returns a pointer to the SolutionList generated, the caller processes the top level nodes in the list and assigns the current SolutionNode's pointers to point to these lower level SolutionNodes. In this way my algorithm recursively builds an arbitrary tree where at the highest level there is a list of SolutionNode's each of which point to an arbitrary number of SolutionNodes that are consistent with it. To find a solution to the puzzle, just start at the top level SolutionList, and follow a pointer to a SolutionNode and record the identifying information, follow that node's pointer to a lower level node and record its information, follow that node's pointer to an even lower level, and continue until the node you are examining has no pointers to a lower level.

To tell if the solution found solves the puzzle we'll check that the number of node's we examined is the same as the number of pieces we wanted to fit inside the puzzle. If this is the case, and if the size of the target configuration is exactly the same as the sum of the piece sizes, then we have found one solution. Finding the rest requires recursively searching this Solution Tree with a modified DFS algorithm.

A solution to a problem would look something like this:

```
1 0 0 1
2 1 1 -2
3 2 1 3
```

Which means that piece with ID : 1 is located at (0,0) and has a rotation = 1 (no rotation). Piece with ID : 2 is located at (1,1) and has a rotation = -2 (flipped, and rotated once). Finally piece with ID : 3 is located at (2,1) and has a rotation = 3 (rotated twice, no reflection).

Because my algorithm cant determine any full solutions until nearly all of the algorithm has completed executing, I wait until the very end of the solution finding routine to call the isometric checking routine. This takes the awkwardly formatted output from the solution finding algorithm shown above, transforms it into human readable output (nearly graphical) and then determines whether each node is isometric to any others (removing those that are, and printing to a file those that aren't).

Once the file has been written with the output from the C executable, the python script reads and parses this input and draws the specified number (within reason, max = 20) of solutions to display to the user.