Report for Programming Problem 1

Team

2019232272 - Tomás Mendes
2019227468 - Joel Oliveira

1 Algorithm description

Before starting to solve the puzzle, some pre-processing is made. An hash table is used to store the pieces that match with a sequence of two or three numbers. The keys of the hash table are the pairs/triplets, and the values the set of pieces that can match it.

```
Function Solve(counter)
  if counter = rows * columns then
    return True
  for piece in position_matches[current_match(board)] do
    if piece.used = False then
      for i = 1 to 4 do
        if piece.rotation(i) fits board.current_position then
          piece.current_rotation = i
          piece.used = True
          board.insert(counter, piece)
            if Solve(counter + 1) = True then
              return True
            piece.used = False
  return False
```

Base case: Board is full
Recursive Step: Insert next piece
Rejection Condition: There isn't a way to fill the board with the current pieces

One of the improvements made was to count the ocurrences of each digit. If a board can be filled, each number, that is not in a corner, appears an even number of times. With this in mind, we know that at max, there will be 4 numbers that occur an odd number of times. If there are at least 5 numbers that were counted odd, by default, the puzzle is impossible.

Another improvement wast at the code level. Since the shape of the board was R x C, there was no need to explicitly "pop" the last piece as it would just be replaced by the next insert in the same position.

2 Data structures

An array was used to store the pieces as they were read. The board data was stored in an one dimensional array, so that all the memory could be allocated in a single chunk (row/column was manually calculated when needed).

To store which pieces could fit a certain sequence of numbers we used an Hash Table, with the keys being the sequence of numbers to match, and the values arrays pointing to the array the actually contained the pieces.

3 Correctness

Our approach optimizes the backtracking with the preprocessing step. It reduces the number of pieces to look for whenever a step is done. It takes into account not only the right side of the last piece inserted (or the piece currently above the insert position, in the case of the first row), but also the bottom side of the piece above. This is much more explicit in the piece configuration, and finds if a given path is wrong much sooner.

The submission that was accepted could have been improved even further, by also storing at which rotations the piece fitted in that particular match sequence, instead of searching it every time we used the piece.

Firstly, all of the code written wasn in Pyhthon. A lot of optimisations were implemented in an effort to reach 200 points in Python. Unfortunately we couldn't get our code accepted. As a suggestion from the teachers, we converted our algorithm to C++. In C++ we didnt even need all the optimisations made in Python to get our code accepted in mooshak.

4 Algorithm Analysis

4.1 Time complexity

We have a set of N pieces to fit in a position. By trial and error we remove a piece from the set, and keep trying to fit it in the board.

This means that worst case scenario we try to fit the rest of the set, with (N-1) elements, N times.

$$
\begin{aligned}
T(N) &= N \cdot T(N-1) + C \\
&= N \cdot (N-1) \cdot T(N-2) \\
&\dots \\
&= \prod_{i=0}^{k-1} (N-i) \cdot T(N-k) + C, k = N \\
&= \prod_{i=0}^{N-1} (N-i) \cdot T(0) + C \\
&\Rightarrow O(\prod_{i=0}^{N-1} (N-i)) = O(N!)
\end{aligned}
\tag{1}
$$

4.2 Space complexity

An array is used to store all the pieces along with their rotation and a flag to check if the pieces have been used or not. $O((4+2) \cdot N) = O(6 \cdot N)$

Each piece can generate up to 8 entries in the Hash table has all the pairs and triplets of each piece are stored. If the pair or triplet of a piece already has an entry in the table, the piece is added to that entry but a new one is not created. The worst case scenario would be $O(8 \cdot N)$.

At max, there can be 2500 pieces. The board needs to store all the pieces. The Big-O notation would be $O(N)$ with N the total number of pieces. The total space complexity would be $O((6 \cdot N) + N + 8 \cdot N) = O(15 \cdot N)$

5 References

1. https://stackoverflow.com/questions/42701688/using-an-unordered-map-with-arrays-as-keys