Counting the 8x8 Infinifields

Oriel Maxime, May-June 2005
orielmaxime@yahoo.com

Introduction

In this paper, we investigate the question of how many legal 8x8 Infinifield[1] fields exist. We wrote a computer program for this purpose and ran it on multiple machines over several weeks to produce an exhaustive list of such fields. We here describe the results of the run and the lessons we learned from the process.

Please see http://www.infinifield.com/ for more information about the game. Our description here is limited to the key problem and the computer program we used to solve it.

While there are plenty of distinct solutions to this problem, it is an interesting exercise to find one by trial and error. One is easily discouraged, and indeed, we initially suspected that there would only be a handful of solutions. We could not have been more wrong!

What Sets Infinifield Apart

There are many board games where creating the board is part of the game. Almost all such games limit the placement of board components to some predefined grid. The Infinifield rules, on the other hand, allow the components to be placed at arbitrary angles to each other and in three dimensions. There are still a necessarily finite number of distinct strategically equivalent boards, but there are certainly an infinite number of visually distinct boards, and this fact drew us to the game and this specific task.

The 8x8 Infinifield Problem

An Infinifield field, for our purposes, is an arrangement of 24 blocks: eight 1x1 white blocks, eight 1x2 yellow blocks, six 1x4 red blocks, and two 1x8 blue blocks. The white blocks are special; the other colors do not matter. In this investigation, we investigate fields where the blocks are restricted to (and fill) an 8x8 grid.

Blocks in a field are said to touch if they share an edge or corner point.

Besides filling the 8x8 grid, a valid Infinifield arrangement must also satisfy these two constraints:

- No two white blocks touch one another.
- No colored block touches more than two white blocks.

[1] Infinifield® is a registered trademark of Infinifield, Inc., and is used with permission throughout this paper.

An example solution will help illustrate:

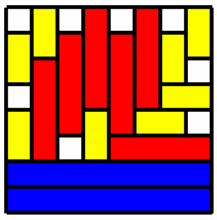

Figure 1

The first solution found by our program.

Our goal is to identify the number of distinct, legal fields. Rotations and reflections of a field are *not* considered distinct.

Solving the 8x8 Problem

The technique for solving the problem is standard to computer science: implement an exact, backtracking exhaustive-search program to find all possible solutions by trying every possible combination in a controlled way.

Such algorithms have a well-deserved reputation for bad performance, due to the potentially exponential number of combinations to consider. The most important questions in designing such an algorithm are (1) how to organize the search, and (2) how to identify partial combinations that cannot be extended to a complete solution.

The Core Dilemma

This particular problem is harder to solve by exhaustive search than it might appear. The core dilemma, as we eventually understood it, is this:

- On the one hand, one would prefer to place the largest pieces first. This ensures that no large, awkward piece is left over without a place to fit. In this approach, the white blocks would go in last, in whatever spaces are left over after placing the other blocks.
- On the other hand, the non-trivial constraints have to do with how the white blocks are allowed to touch other blocks. This suggests placing the white blocks first so that these constraints can be used eliminate bad combinations earlier in the process.

Each approach has probable merits, but we ended up using the latter. We felt that it would be easier to restrict the colored block placement given the white blocks than it would be to try to anticipate which colored block combinations left no feasible arrangement of white blocks. We may have been wrong: here's a tricky example.

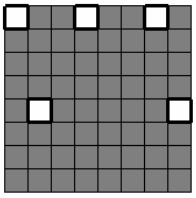

Figure 2

If the three remaining white blocks must appear somewhere in the bottom three rows of the grid, can the grid still be completed according to the rules? The answer is no, and working through the various cases highlights some of the difficulties associated with the strategy of first placing the white blocks.

Much of the problem-specific strategy is related to resolving this dilemma and effectively ruling out bad combinations of white blocks. Indeed, one key optimization was to use a separate and

streamlined program to generate all possible placements of the white blocks. A second program then used the output of the first and applied the more complicated rules required to find placements, if possible, for the colored blocks.

Problem-specific Techniques

To decrease the run-time of the program to something manageable, we used the following problem-specific techniques:

- We modeled the problem in terms of Thin Rectangles, an abstraction representing rectangles (upper left corner and lower right corner) which are one unit in either height or width. For example, there are 56 horizontal and 56 vertical 1x2 Thin Rectangles that fit on the 8x8 grid. The task then is to pick, of the 272 possible Thin Rectangle regions of an 8x8 grid, some subset such that:
    - Each square on the 8x8 grid is contained in exactly one selected Thin Rectangle.
    - The fixed number of Thin Rectangles of each size is respected (i.e. exactly six distinct 1x4 rectangles are selected).
    - The constraints with regard to white (1x1) Thin Rectangles are respected.
- One usually solves this type of problem by placing the largest pieces (most awkward) pieces first. We focus on the white blocks first, as described above. After all eight white blocks are placed in a trial solution, we then continue with the blocks in decreasing order of size (blue, red, and finally yellow).
- If, within the 8x8 grid, there is a square that touches three or more white blocks, then any block covering that square will violate a constraint. Therefore, while placing the white blocks, we ensure that each space in the grid touches at most two of them. Adding this check significantly reduced the number of white block configurations that the programs needed to examine.
- Imagine the 8x8 field as a checkerboard, with 32 red squares and 32 black squares. Since all of the colored blocks cover the same number of red and black squares no matter how they are placed, it follows that, in placing the white blocks on the field, exactly four must rest on red squares and four must rest on black squares. Otherwise, the 8x8 field cannot be filled (this is known as a parity argument). Adding this check also significantly reduced the number of white block configurations.
- The same checkerboard argument made above also applies to the smaller regions created when one of the blue blocks divides the 8x8 field. This allows us to eliminate otherwise valid placements of the blue blocks. Note that this check (at least, in this simple form) would not apply to, say, a 4x16 field. Specifically, the program verifies that there are as many white blocks on red squares as there are white blocks on black squares within subregions created by a blue block. This significantly reduced the amount of backtracking the program performed.
- The library we used for this problem (see below) supports the definition of symmetry constraints. We used this to define constraints that prune potential solutions guaranteed to result in rotations or reflections of already discovered solutions. While we can only save a factor of eight in the runtime by employing this technique, it was easy to implement and prevents us from having to code a post-processing step.

General Purpose Techniques

Several general-purpose strategic decisions are also of note:

- We used Haskell for our implementation, a very high-level language that also has an aggressively optimizing compiler. See http://www.haskell.org/ for more information about Haskell, and http://www.haskell.org/ghc/ for more information about the Glasgow Haskell Compiler. The high-level nature of the language resulted in less than 500 lines of code for the entire problem-specific module (including comments, pretty layout, testing assertions, and debugging features). This compactness allowed us to focus on the task of representing the problem constraints rather than the details of managing the backtracking process, and made it quite easy to prototype and compare different models to one another.
- We started by using a proven library for solving backtracking problems. This allowed us to focus on the problem-specific tasks, and provided several general features we leveraged, such as symmetry breaking and 1-level probing. The debugging features built into the library also helped us explore the search tree and discover additional problem-specific optimizations.
- We implemented checkpoint-restart logic, so that a failed or aborted run could be continued rather than having to start again from scratch. This allowed the run to continue over several weeks, and even allowed the program to be optimized and replaced with a newer version without losing past progress.
- We manually implemented parallelism by using several computers and adjusting the start states so that the machines were exploring different sets of possible solutions.

The importance of using a high-level language and a proven library can't be overemphasized. The resulting agility allowed us to prototype, try different approaches, abandon bad ideas and tweak good ones, all with very little effort.

Results

The program ran on at least three computers over its lifetime, and in various incarnations, consumed more than six CPU-weeks. The average configuration for a computer running the program was a Pentium 4 with 512 MB RAM and a single 2GHZ CPU running Windows XP and Haskell 6.4. We estimate that the final, tuned program could generate all solutions from the beginning in two CPU-weeks.

The program generated exactly 633,586 different fields. Clearly, this is far too many to manually check, but all spot-checks of the results suggest that there are no duplicates and that each is a legal 8x8 Infinifield.

Sample Solutions

One hundred sample solutions are displayed in Figures 3.1-3.3 (at the end of the report). You can see how the white squares closest to the upper-left corner slowly move across and down as the program progresses, indicative of the organizational strategy of examining possible placements of the white squares first.

Note that many of the sample solutions have the two blue bars in the second and third row (or some rotation thereof). Since the two resulting subproblems can be solved independently, this placement produces a wealth of different overall solutions.

Also, note that there are comparatively few solutions where no white square is in a corner. There are even fewer solutions where the four corners of the grid are neither white nor blue. The reader is invited to try to find such a solution before looking at the last sample.

Acknowledgements

Thanks to the Infinifield team for the very interesting game and the related set of combinatorial problems. Thanks to the Glasgow Haskell team and the Haskell community at large for a wonderful programming language. Thanks to Terry, just because.

Summary

This problem turned out to be more subtle than expected! Due to the size of the search space, we spent a lot of time tweaking and trying different things to make the search more effective. Four hours of thought and coding could easily save several days of run time.

As with all computer-generated results, the solution count of 633,586 should be considered an estimate rather than an exact figure until someone reproduces the result independently. However, it is probably safe to say that there are well over 630,000 different 8x8 legal fields. This number, although finite, would probably take a lifetime to explore, even before branching out into 4x16 fields, three-dimensional fields, or non-grid-aligned fields!

This exercise is a nice case study in language selection. A program written in an imperative language would probably have executed much more quickly, but the implementation effort and lack of modeling agility would have been problematic. The problem is too large to solve without non-trivial problem-specific techniques, and these techniques required significant experimentation and testing to discover.

The Author

Oriel Maxime can be reached at orielmaxime@yahoo.com. He is a computer system designer with a strong interest in mathematics, computer science, and puzzles and games of all kinds.

Figure 3.1 – Sample solutions 1 - 35

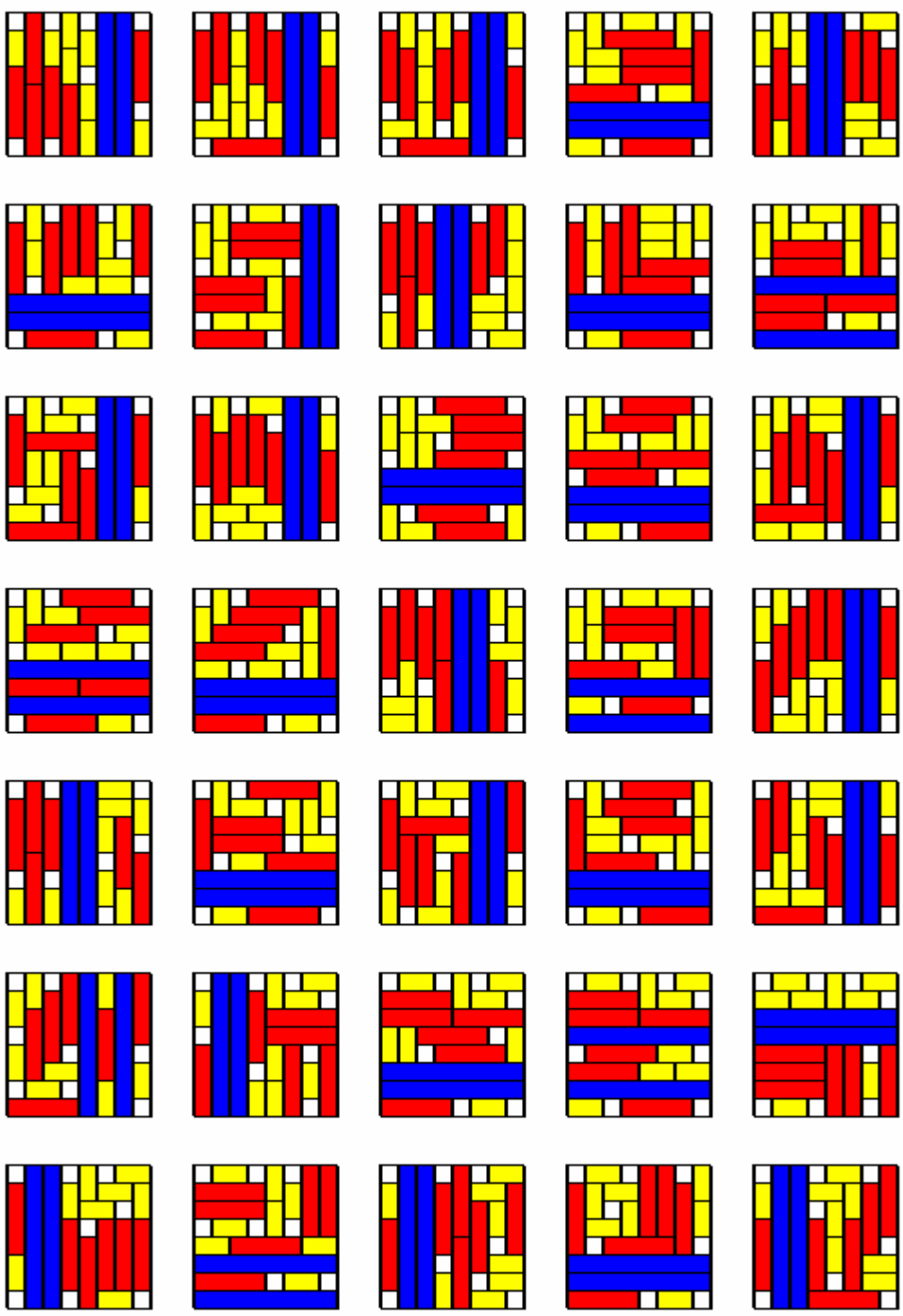

Figure 3.2 – Sample solutions 36 - 70

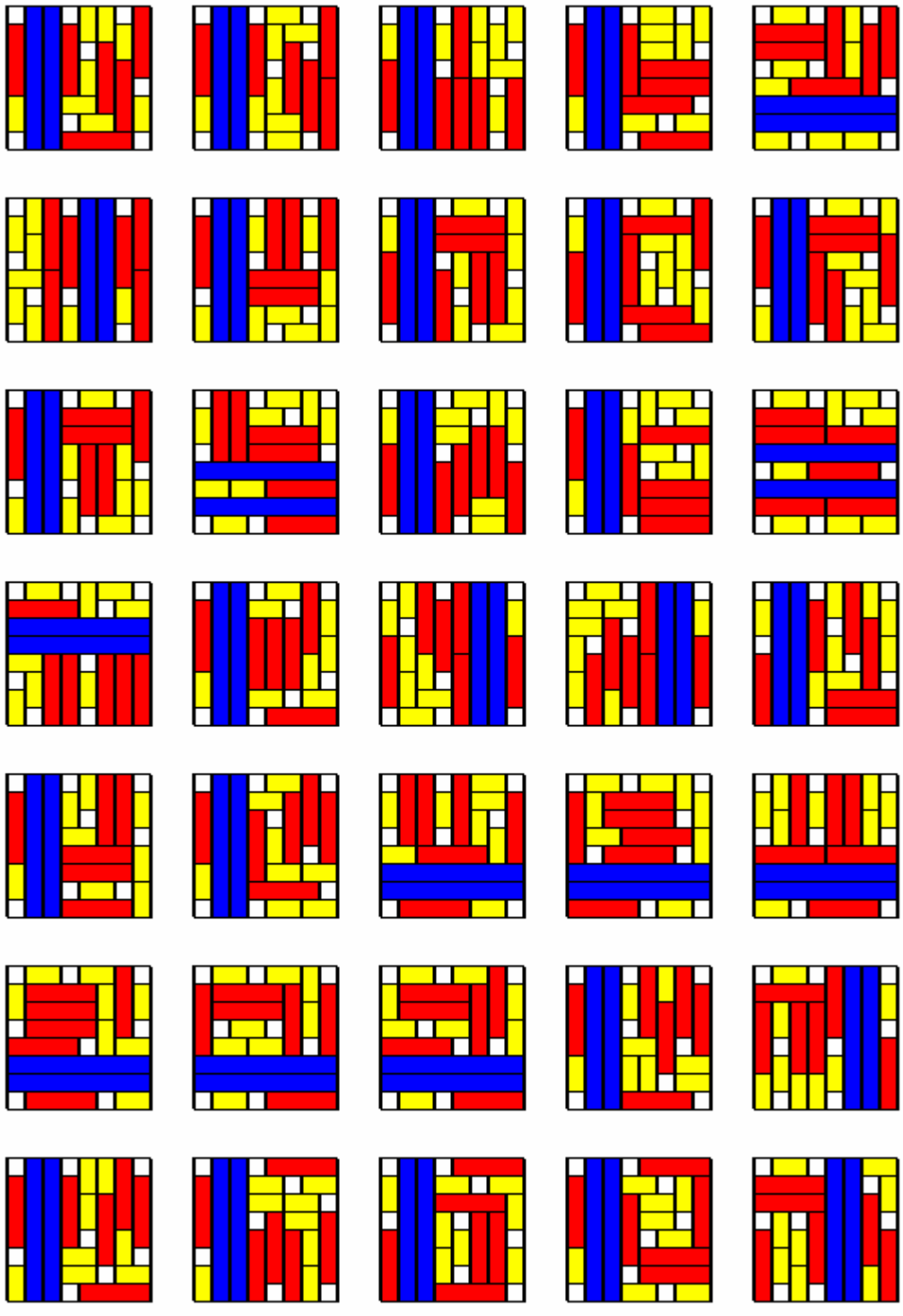

Figure 3.3 – Sample solutions 71 - 100