Dodgy Tetric Report

Scott Lenser and James Bruce
Carnegie Mellon University, CSD
Pittsburgh, PA 15213
{slenser,jbruce}@cs.cmu.edu

November 1, 2000

Abstract

This is a *very* brief description of our winning entry in the IC 2000 programming contest. It's ben formatted this way mostly just to make it easier to read and is not expected to make much sense to someone who hasn't read about the contest already.

1 Introduction

We developed our program very incrementally. The program evolved to become much more capable as time went on. The most important part of our program is the local search routines. Their job is to take a puzzle that has been augmented with fill rows (full rows that will be removed) and produce a sequence of moves to solve the puzzle. Running this part alone gives us a hand generation capability. The next most important part is the global search routines. These routines use the local search to greedily solve a puzzle that has not been augmented. The last part is the puzzle analysis. This part pre-analyzes the puzzle which speeds up the global search routines.

The remaining sections detail the incremental development of the code with new sections for major improvements. Hopefully, this will make the code easier to understand since the history of the code is very evident in its implementation.

2 Local search

We started out with a program that given the fill rows would simply try to exhaustively search for the solution. Naturally, this couldn't solve very large puzzles but throwing out moves that fill blocks that are supposed to be empty still reduces the branching factor dramatically. Our approach still shows remnants of this initial design. We exhaustively search all possible moves in a brute force manner. The speed of our program comes from careful implementation and sophisticated search tree pruning. The next improvement was to disallow

filling a block that is above an unfilled target block (hereafter, target). Doing this helped the speed some although it later turned out this was a little too strict. We noticed that the program would screw up one side of the puzzle and then work on the other side to a great height before it finally got stuck and had to backtrack. This happened frequently with our depth first search methodology. So we implemented a window in which the search could work in. The window starts at the first row that has unfilled targets and extends for a constant number of rows after this. We disallow the search to fill blocks above this window (note that the window moves up as rows get filled). With a window size of 5 the program produces good solutions. We believe a window size of 8 is sufficient to guarantee that a solution is found if one exists. This sped up the program very nicely allowing us to solve puzzles, albeit somewhat slowly (think minutes).

We still weren't happy with the speed though. We felt we needed a global search to completely solve the puzzles in order to win. We wanted the local search to be much faster so that we could call it frequently to easily solve large puzzles part by part. We made two further major enhancements to the local search to accomplish this at around the same time. We noticed that the solver was finding many different ways to get to the same configuration. Since the differences were not important to us (same number of pieces always), we wanted to avoid these duplicates. We added repitition suppression to do this. We hash every board state we encounter and through out moves that reach a state we have already seen. We use STL to do this. Note that there is no memory bound imposed so this also causes running for too long of time to cause out of memory errors on the machine. This sped the searh considerably. The next improvement was to add connected component analysis. We find connected components of unfilled targets and use these components for additional filtering. We through out moves that result in connected components that have a number of blocks not divisible by 4. We also through out moves that result in connected components more than 2 blocks high that are not supported by a filled block. This works because none of the pieces can reach more than 2 blocks down and still be supported. Since we create connected components from the bottom of the board up, components that even have a part that can't be supported get thrown out. The mod 4 check and the support check are about equally important. We use a very efficient implementation that does entire rows at a time when creating the connected components. Connected components drastically increases the speed of the algorithm. Local search runs in well under a second most of the time. We are able to completely exhaustively search most puzzles for a solution.

3 Global search

We actually have 2 different global search routines. Both are geedy searches that solve the puzzle by repeatedly running the local search and building up a solution one line at a time. The top down routine starts at the top of the puzzle and works its way down. It calls the local search to solve the topmost

k lines (with fill rows added as needed for padding). If this succeeds, it adds a line and repeats the process (note that the previous local search is completely thrown out). If this fails, it adds a fill row above this line and tries again. Eventually this succeeds if the puzzle if solvable. It repeats this until it works its way all the way down to the bottom of the puzzle and returns the result of the final local search as the solution of the puzzle. Bottom up works similarly except starting from the bottom of the puzzle. Top down usually works better than bottom up but not always. Top down always finds a solution if one exists, bottom up sometimes fails. We solve a lot of the puzzles in under 9 seconds. The time spent in the rest of the puzzles is dominated by the time to prove that a solution does not exist for some of the local puzzles. The amount of time this takes is affected by a parameter `GLOBAL_SEARCH_STEP_LIMIT`. Larger values take more time but shorter values sometimes through out better solutions.

4 Puzzle analysis

We added a puzzle analysis procedure around the same time we were working on the global search. This procedure analyzes the puzzle and attempts to figure out places that fill rows are needed. This was intended to provide a conservative lower bound on the number of fill rows needed. It turned out to be ever so slightly more than the lower bound on some problems. So we added a switch to turn it off. In normal situations it simply makes the program run faster, particularly on sparse puzzles. The basic idea is to look for rows that cannot be supported with the blocks currently in the board. Rows like this require a fill row to be added either above or below them. The code next checks whether the row can be supported from above using a fill row above this row (this calculation turned out to be very slightly pessimistic about the possibility of certain configurations). If the row can be supported from above, it merely marks that the row needs to be connected. If the row cannot be supported from above, then a fill row must be added beneath this row. It then checks to make sure that the fill row itself can be supported. If not it adds another row. It then checks to see whether the two combined fill rows are supportable. If not it adds another row. This lets us put in 3 fill rows exactly were needed on many of the sparse puzzles without wasting time running fruitless local searches.

5 Quality optimizations

We noticed when the new puzzles came out that some people were doing better on some of the puzzles then we were able to do. We traced the problem to an assumption in our local search pruning code. We had assumed that pieces would always be connected in the puzzle augmented with fill rows. This is not actually the case, however. Pieces can have pieces on both sides of a fill row if this row is removed before the piece is dropped. We made a couple of changes to fix this problem. The first fix was to the occlusion metric. Recall that we

through out moves that occlude a target with a filled block. We had to modify this to we rule out moves that occlude a target with a filled block *which is not in a fill row*. The second change was more difficult. We changed the connected component code to consider targets connected if they are only seperated by a fill row. Unfortunately, this doesn't work as well as the old method at pruning the search space so we made whether to consider these kinds of moves an option. This version blows up on some local search problems, particularly when there are many fill rows next to each other. It can find any solution that is possible however.

6 Conclusion

Exhaustive search rulez.