Solving 'Flow Free' As A Constraint Satisfaction Problem

Jordan Pottruff

I. Introduction

This report is an overview of my methodology for solving problems in the game 'Flow Free' by treating the game as a constraint satisfaction problem (CSP). The program I developed is capable of solving such a problem using relevant algorithms and heuristics.

II. CSP Formulation

To solve Flow Free as a CSP problem, it first needs to be described in terms of variables, domains, and constraints. Each unassigned cell in a given Flow Free board is a variable. The domain for each of these variables are all of the given source colors found on that board.

In my implementation, I focus on two major constraints that every cell has. The first is what I call a *cardinality constraint*, which follows these two basic checks:

1. No cell, including source cells, have more than 2 neighbors of the same color at any time.
2. Each *complete cell* (a cell with no unassigned neighbors) has exactly 1 neighbor of the same assigned color if a source, or 2 if not a source.

My other constraint, called the *path-to-source constraint*, requires that all cells belong to a *complete path* (a path whose cells are all complete) connects to a source of the same color. If a completely filled board meets the above constraints, then it is guaranteed to be a valid solution.

III. Solver Implementation

A. Program Design

My solution uses object-oriented programming to help designate responsibilities to different components. The class *FFCell* is used to represent cells on the Flow Free game board. *FFGrid* is composed of these cells and forms the grid or "maze" for an instance of the game. The actual algorithms for the CSP are *BasicFlowFreeSolver* and *AdvancedFlowFreeSolver*, both of which inherit from the abstract class *FlowFreeSolver*, which provides abstract and concrete methods that are used by all solvers.

B. Basic Solver Design

The initial implementation of my solver, *BasicFlowFreeSolver*, relies on simple backtracking. When the algorithm chooses an unassigned variable it picks the first unassigned cell that is found in the grid, essentially making it a random decision. When a variable needs to be assigned, all values in the domain of the grid are tried.

C. Advanced Solver Design

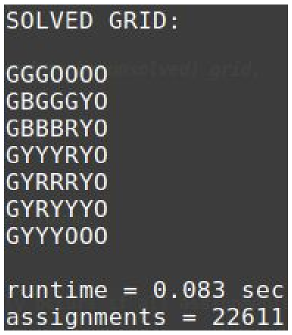
The more advanced solver implementation, *AdvancedFlowFreeSolver*, builds off simple backtracking by adding heuristics for variable preference and forward checking. To do this, each cell tracks its *domain*, which in this context means the colors that can still be assigned to it without breaking any constraint, given all other cell assignments at that time. This makes implementing the Minimum Remaining Values (MRV) heuristic very easy, as our solver now simply chooses the cell with the smallest domain.

In addition, a very rudimentary forward check is performed as part of the constraint checking. If at any point the size of any cell’s domain is zero, the last assignment needs to be backtracked. This prevents the algorithm from traversing pointless further assignments.

IV. Results

7x7 Maze

Basic

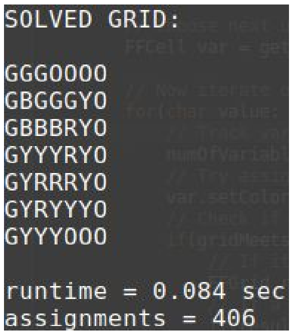


```
SOLVED GRID:

GGGOOOO
GBGGGYO
GBBBRYO
GYYYRYO
GYRRRYO
GYRYYYO
GYYYOOO

runtime = 0.083 sec
assignments = 22611
```

Advanced



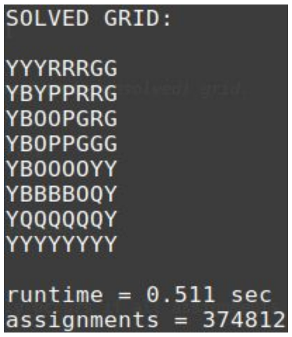
```
SOLVED GRID:

GGGOOOO
GBGGGYO
GBBBRYO
GYYYRYO
GYRRRYO
GYRYYYO
GYYYOOO

runtime = 0.084 sec
assignments = 406
```

8x8 Maze

Basic

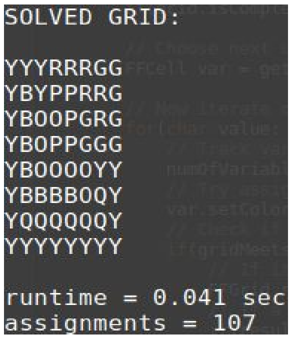


```
SOLVED GRID:

YYYRRRGG
YBYPPRRG
YBOOPGRG
YBOPPGGG
YBOOOOYY
YBBBBOQY
YQQQQQQY
YYYYYYYY

runtime = 0.511 sec
assignments = 374812
```

Advanced



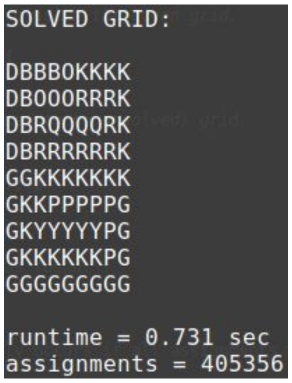
```
SOLVED GRID:

YYYRRRGG
YBYPPRRG
YBOOPGRG
YBOPPGGG
YBOOOOYY
YBBBBOQY
YQQQQQQY
YYYYYYYY

runtime = 0.041 sec
assignments = 107
```

9x9 Maze

Basic

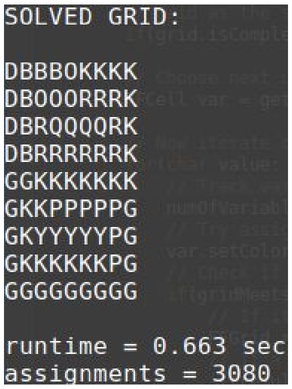


```
SOLVED GRID:

DBBBOKKKK
DBOOORRRK
DBRQQQQRK
DBRRRRRRK
GGKKKKKKK
GKKPPPPPG
GKYYYYYPG
GKKKKKKPG
GGGGGGGGG

runtime = 0.731 sec
assignments = 405356
```

Advanced



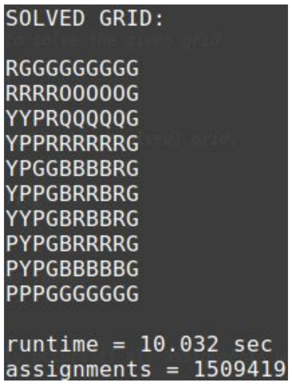
```
SOLVED GRID:

DBBBOKKKK
DBOOORRRK
DBRQQQQRK
DBRRRRRRK
GGKKKKKKK
GKKPPPPPG
GKYYYYYPG
GKKKKKKPG
GGGGGGGGG

runtime = 0.663 sec
assignments = 3080
```

10x10 Maze

Basic

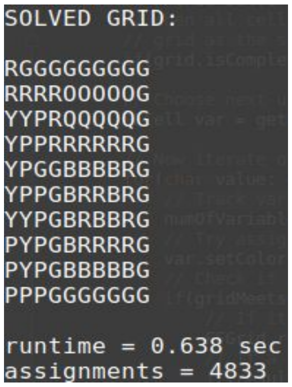


```
SOLVED GRID:

RGGGGGGGGG
RRRROOOOOG
YYPRQQQQQG
YPPRRRRRRG
YPGGBBBBRG
YPPGBRRBRG
YYPGBRBBRG
PYPGBRRRRG
PYPGBBBBBG
PPPGGGGGGG

runtime = 10.032 sec
assignments = 1509419
```

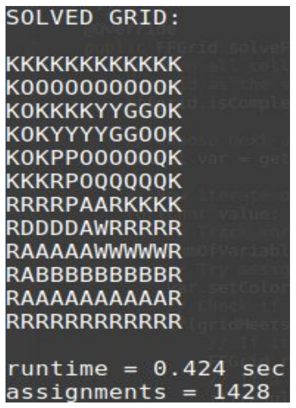
Advanced



```
SOLVED GRID:

RGGGGGGGGG
RRRROOOOOG
YYPRQQQQQG
YPPRRRRRRG
YPGGBBBBRG
YPPGBRRBRG
YYPGBRBBRG
PYPGBRRRRG
PYPGBBBBBG
PPPGGGGGGG

runtime = 0.638 sec
assignments = 4833
```

12x12 Maze	
Basic	**Advanced**
Unable to be solved in a realistic amount of time.	

```
SOLVED GRID:

KKKKKKKKKKKK
KOOOOOOOOOOK
KOKKKKYYGGOK
KOKYYYYGGOOK
KOKPPOOOOOQK
KKKRPOQQQQQK
RRRRPAARKKKK
RDDDDAWRRRRR
RAAAAAWWWWWR
RABBBBBBBBBR
RAAAAAAAAAAR
RRRRRRRRRRRR

runtime = 0.424 sec
assignments = 1428
```

Advanced Improvement Over Basic*:

Maze	Time	Assignments
7x7	0.98x	55.7x
8x8	12.5x	3,503x
9x9	1.1x	131x
10x10	15.7x	312x
12x12	N/A	N/A

* measured as ratio of basic:advanced

V. Analysis

These results indicate that, in general, the advanced implementation is superior in terms of both time and assignment compared to the basic solver. However, this advantage is not absolute nor is it always consistent.

The largest discrepancy is the 7x7 maze, which was actually solved faster by the basic solver despite it assigning far more values than the advanced one. The reason for this is that forward checking and determining the most constrained variable are expensive operations relative to the simple algorithm used by the basic solver. The cost of these additional steps are normally far exceeded by the advantages of eliminating “dead-end branches.” However, sufficiently small mazes may have branches that are actually faster to explore than to eliminate using a heuristic.

Another important point is inconsistency along maze dimensions. The 8x8 maze for example is solved 12.5 times faster and in 3,503x fewer assignments by the advanced solver over the basic one. However, the 9x9 maze shows a multiplier of only 1.1 for time and 131 for assignments. While our advanced algorithm should perform better and better relative to our basic one as the grids become larger, its important to understand that other factors affect performance as well. The ratio of source cells to total cells, as well as the position of these source cells greatly influences how quickly mazes can be solved by either algorithm.

The larger mazes, which tend to be most resistant to these discrepancies and inconsistencies, show very clearly that the costs of our additional heuristics are outweighed by their massive benefits. The biggest performance boost comes from the MRV heuristic. This heuristic often eliminates numerous cells almost instantly, as its common for certain cells to have only one possible value. Early branch eliminations like this can lead to drastic increases in performance time.

VI. Contributions

I conducted the assignment on my own. Therefore, all contributions to the program, research, and report are made by myself, Jordan Pottruff.