Artificial Intelligence: Free Flow CSP

Carsen Ball
Brett Layman

Novemeber, 2018

Abstract

We applied several heuristics to solve instances of the constraint satisfaction problem (CSP): Free Flow. Our heuristics included: most constrained variable (MCV), approximate most constrained variable (AMCV) which looks at how confined the end nodes are, and shortest path (SP) for values (an approximation of least constraining value). The heuristics were added to and compared with a basic backtracking search algorithm. We tested the backtracking algorithm and the heuristics on progressively larger and more difficult problems. We measured both the run times, and number of edges explored in the CSP search tree. We found that using a shortest path heuristic for choosing values generally lead to shorter run times. Also, our AMCV heuristic generally lead to improvements in run times. Our MCV heuristic was too computationally expensive, and lead to the worst run times, despite it generally ensuring that less of the backtracking search tree was explored.

1 Introduction

Free Flow is a CSP that involves drawing paths to connect locations of the same color on a 2D grid. The primary constraint is that paths cannot cross. So as paths are drawn, it constrains the possible paths for subsequent colors. Another constraint is that a path cannot contain zig-zags. One approach to solving CSPs is the backtracking algorithm. This algorithm performs a depth first search of the solution space, making sure that constraints are followed at each step. Backtracking alone can be very time intensive, so it's often helpful to implement heuristics for choosing a variable (in this case a color) and a value (in this case a path). One added difficulty with Free Flow in particular, is that the set of possible values is not pre-determined, as it is for other CSPs such as the map-coloring problem. In addition, the number of potential values (paths) grows exponentially with the size of the problem. This can make it difficult to implement a value heuristic.

We implemented two heuristics for choosing the next variable, and one heuristic for choosing the next value. The most constrained variable heuristic chose the color with the fewest possible paths connecting the two ends. This heuristic was rather expensive due to all of the path finding calculations it performed, so we implemented another variable heuristic designed to approximate the most constrained variable: AMCV. AMCV sums the number of colored spaces and wall spaces surrounding the two end nodes, and then prioritizes the variables with the highest sum.

Our value heuristic orders paths by length, choosing the shorter paths first. We chose this heuristic based on the assumption that shorter paths are less likely to constrain future paths (least constraining value), although we know that this is not always the case. In addition to these heuristics, we also tried capping the depth of the path finding search to speed up our algorithms.

2 Software Architecture

We based our software architecture off of a generic recursive backtracking algorithm. This algorithm is implemented in the function `recursiveBacktrack`. This function takes in a priority queue of variables (colors), and recursively calls itself until the queue is empty, updating the queue along the way. The priority queue order is determined by the heuristic used, which is determined by the type of object being used for the variables. There are three types of variable objects: `SimpleVariable`, texttttConstrainedHeuristicVariable (MCV), and `AdjacentHeuristicVariable` (AMCV). Each type overrides its `setCompareValue` function in order to determine it's priority according to the heuristic.

Once a variable is chosen, a value (path) must be selected. In order to select a path, we first find all of the possible paths for that variable by calling `getOrderedValues`. `getOrderedValues` uses a recursive method to find all of the possible paths for a color. Starting at the start position of a color, the neighbors of the start position are explored in a depth first manner. At each step a neighbor is checked to see if it passes the given constraints, if so, that neighbor is added to the path. If a path leads to the goal coordinate, the path is added to the priority queue. If a given neighbor fails to create a valid path, the recursive call returns false. This backtracks up the recursive tree and starts exploring the next neighbor of the previous step.

As with variables, the heuristic used for values is implemented by having different types of value objects. `SimplePath` uses no heuristic, while `HeuristicPath` uses the path length. As the paths are found, they are added to a priority queue. Next, the algorithm loops through the path priority queue, assigning paths to the graph, updating the variable priority queue accordingly, and making recursive calls to assign further variables. If no value assignments can be made on a particular call, then false is returned, and the path assigned previous to that call is removed from the graph. When the variable priority queue becomes empty, true is returned, indicating that all variables have been assigned and the `graphState` storing the paths is complete.

The command line arguments are: ¡file name¿ ¡variable heuristic¿ ¡value heuristic¿. For the variable heuristic argument use: "None", "Constrained" (MCV), or "Adjacent" (AMCV). For the value heuristic argument use "True" for shortest path, or "False" for no heuristic.

3 Results

In this section we present and discuss the results. Due to execution time, not all possible heuristic combinations were used for the larger maps, and we could not produce a solution for the 12x12 map. Time is given in seconds. The time finding paths metric is the time that was used to calculate the constrained heuristic values.

3.1 Presentation Of Results

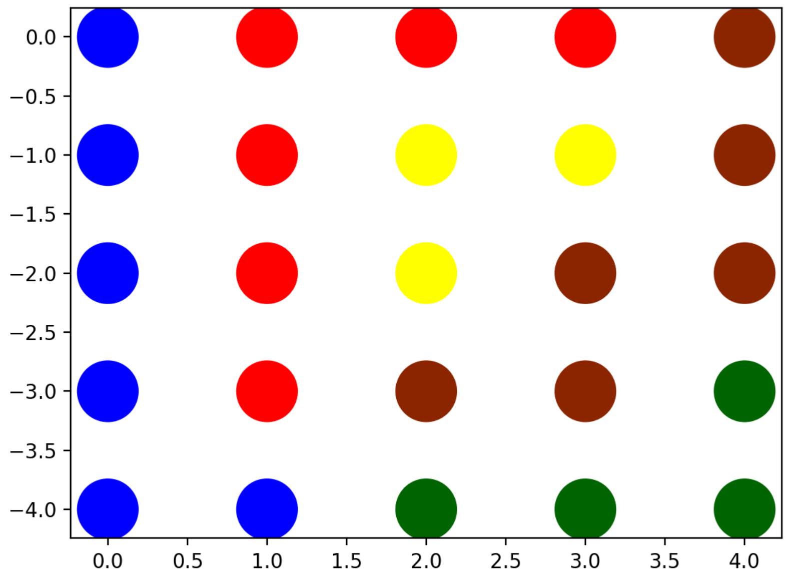

Figure 1: 5x5 Maze Solution

5x5 Maze Results			
Heuristics Used	Run Time	Path Finding Time	Paths Assigned
No Heuristics	0.001	0.000	14
No color choice, shortest path applied first	0.001	0.000	5
Constrained, no path choice given	0.002	0.001	6
Constrained, shortest path applied first	0.002	0.002	5
Adjacent, no path choice given	0.001	0.000	7
Adjacent, shortest path applied first	0.001	0.000	5

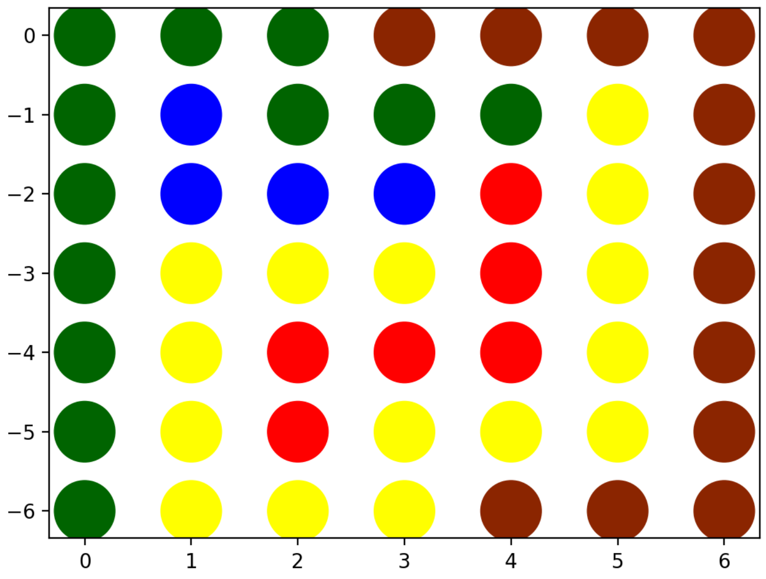

Figure 2: 7x7 Maze Solution

7x7 Maze Results			
Heuristics Used	Run Time	Path Finding Time	Paths Assigned
No Heuristics	0.161	0.000	483
No color choice, shortest path applied first	0.098	0.000	276
Constrained, no path choice given	0.745	0.592	171
Constrained, shortest path applied first	0.586	0.466	64
Adjacent, no path choice given	0.080	0.000	147
Adjacent, shortest path applied first	0.057	0.000	27

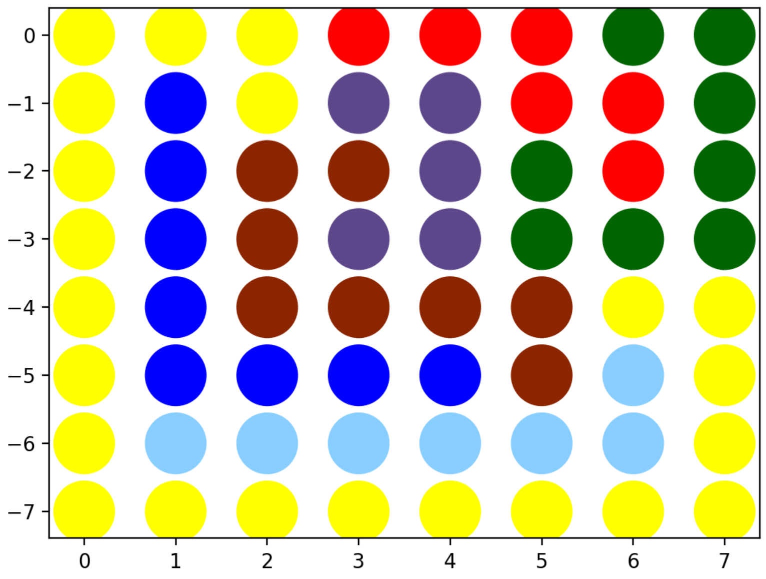

Figure 3: 8x8 Maze Solution

8x8 Maze Results			
Heuristics Used	Run Time	Path Finding Time	Paths Assigned
No Heuristics	0.949	0.000	3972
No color choice, shortest path applied first	0.257	0.000	130
Constrained, no path choice given	4.396	3.915	727
Constrained, shortest path applied first	1.640	1.540	88
Adjacent, no path choice given	0.160	0.000	111
Adjacent, shortest path applied first	0.138	0.000	22

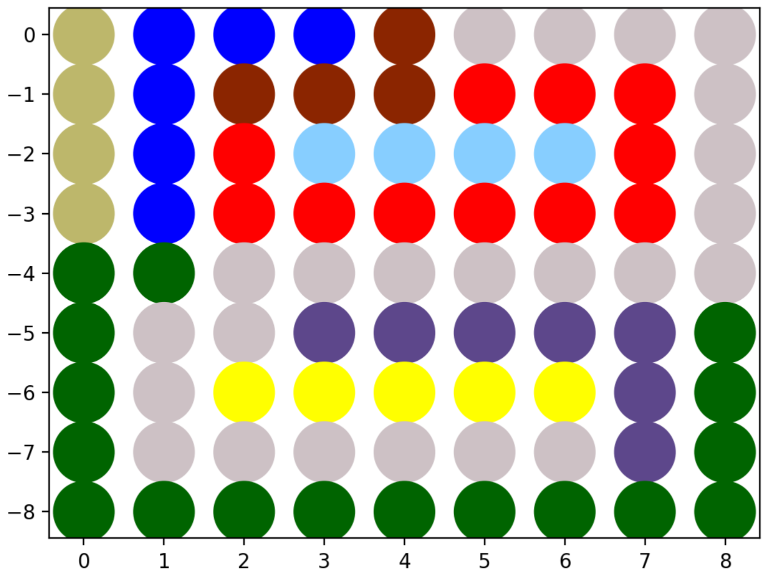

Figure 4: 9x9 Maze Solution

9x9 Maze Results			
Heuristics Used	Run Time	Path Finding Time	Paths Assigned
No Heuristics	94.549	0.000	213126
No color choice, shortest path applied first	4.612	0.000	85
Constrained, no path choice given	148.149	130.73	4754
Constrained, shortest path applied first	83.922	69.654	167
Adjacent, no path choice given	15.150	0.000	23924
Adjacent, shortest path applied first	15.970	0.000	13191

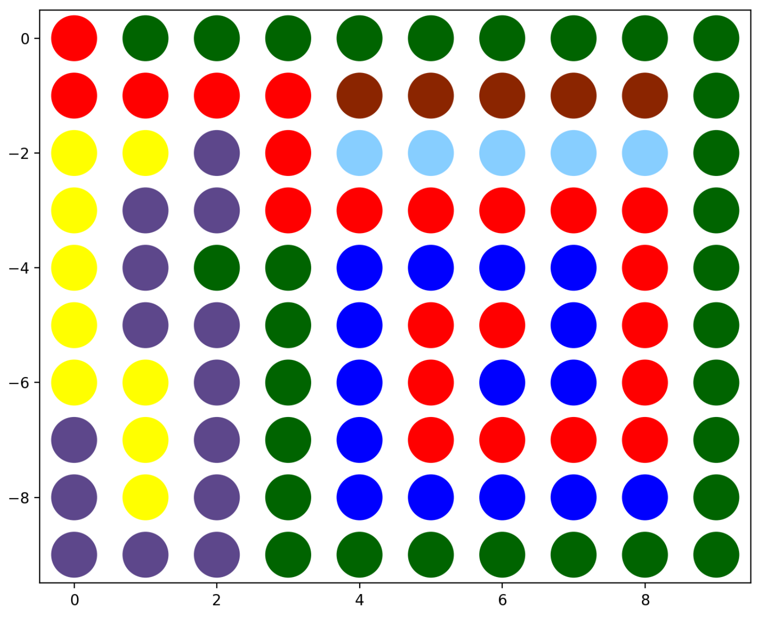

Figure 5: 10x10 Maze Solution

10x10 Maze Results			
Heuristics Used	Run Time	Path Finding Time	Paths Assigned
Adjacent, shortest path applied first	14141	0.000	15477666

3.2 Analysis Of Results

For all maps the dumb implementation applied many more paths than when using heuristics. Though many more paths were applied, the run time was not necessarily the largest when using the dumb implementation. Using the the Constrained heuristic greatly increased the run time in comparison to many of the other runs. The increase in run time was due to the fact that calculating all the possible paths for each color at every step is quite an expensive function. Though using the constrained heuristic did reduce the number of paths applied, it did not always produce the lowest number of paths applied.

For the 5x5, 7x7, and 8x8 using the adjacent heuristic had a positive effect on run time and produced the smallest number of paths applied. Interestingly on the 9x9, though some of the shorter run times were produced when using the adjacent heuristic, it produced some of the larger numbers of paths assigned. Due to the low cost of calculating the open adjacent positions at any given step, the run time was less than the other options. Having start and end points that have less available neighbors highly reduces the amount of possible paths for the color, thus less path assignments were needed to find the solution path for a given color. For the 9x9 maze, the explanation for why less number of paths were applied when using no color heuristic but using apply the shortest path first is luck. By not using color heuristics, the best possible color to apply first must have been chosen at random.

For every map, in comparison to using the same variable heuristic without a path choosing heuristic, applying the shortest path first reduced the run time and number of paths assigned. We believe applying the shortest path first helps, because in general the solution consists of shorter versus longer paths. This shows that out of the solution space, many longer paths are unnecessary to calculate.

The solution did solve for the 10x10 maze in roughly 4 hours. We did not have an infinite loop in our program, but due to the large exponential increase in time from solving the 9x9 mazes to solving the 10x10 mazes, it was assumed that solving for the 12x12 was unfeasible

4 Individual Contributions

4.1 Carsen

Produced the code to find the paths, and apply and remove paths from the map. Then integrated that functionality into the backtracking written by Brett. I adapted previous code for the node and map objects, and how to read in the map, find start and stop points of the colors. I also produced the code to print the solution plot. I wrote the results section and made a small contribution to software architecture.

4.2 Brett

Wrote the code for the backtracking algorithm, and helped to develop the heuristics used. Also helped to debug some issues with recursion and references. Wrote most of the Introduction and Software Architecture sections of the paper.