**Backtracking**

I, Evan DePosit, chose to solve the puzzle using the constraint satisfaction Backtracking Search algorithm from the Russel and Norvig Text. Initially, I had planned on using the AC-3 algorithm, but I read that AC-3 can only solve the simplest of Sudokus. AC-3 only works for simple sudokus because often you can figure out which number goes in a square by determining that it can’t be any of the other numbers that are in the same row, column, or group as it. This is what makes partially filled out sudokus quite easy. However, the “two not touch” puzzle has no initial placement of stars; therefor, one must start by guessing or searching for a solution. For this reason I decided that the backtracking search would be more fruitful

When the program is run, the board is transformed into a constraint satisfaction problem by creating two variables for each region of the board, so that each variable represents one of the stars in a region. The domains are initiated for each star to include the entire set of coordinates for its corresponding region. The constraint is a function that takes in a list of assignments and outputs True if they match the constraints of the problem and false if they do not. The constraint function, notTooClose(), checks for four things: that no assignment is the same as another, that no assignment is adjacent to another, and that there are no more than two assignments per row and column.

**Implementation**

The backtracking algorithm first checks if all the variables have assignments. If a solution has not been found, the select\_unassigned\_var() function returns the variable that has the smallest domain. The reason for this is that if the value can be found to not be a viable solution early, a large chunk can be pruned from the tree. This follows the minimum-remaining-value heuristic. Next, a value is chosen to try that is returned from the order\_domain\_values function. Although some variations of the backtracking algorithm will select a specified order to try values, this implementation does not. Instead, this function simply returns a subset of the domain of each variable that is arc consistent with the assignments made so far. Next, The inference() function restricts the domain of each variable by creating a set of coordinates that can no longer be tried. It does this by taking the new value and returning all the neighbors of that value, the value itself, and any coordinates in the same row or column as the value if two stars are in that row or column. These unavailable values are then deleted from the domains of the other variables by the order\_domain\_values() function. If the value that is tried results in failure, the unavailable values are added back to the domains of variables, the algorithm backs up, and the next value is tried after the one that resulted in failure.

A screenshot of a cell phone

Description automatically generated

**Figure 1: Backtracking Algorithm**

**Results**

I tested the backtracking algorithm on 3 easy puzzles, 1 medium puzzle, and 2 hard puzzles. The algorithm solved all but one of the hard puzzles. The one that it couldn’t solve was the heart shaped puzzle that is the bottom right of Appendix A. The algorithm returns a None object when it fails to find a solution. There is not other clues as to why it failed other than it returned None, so I’m not sure why it failed to find a solution for the last puzzle.

|  |  |
| --- | --- |
| A close up of a logo  Description automatically generated | A close up of a piece of paper  Description automatically generated |

**Figure 2: Solved Puzzles**