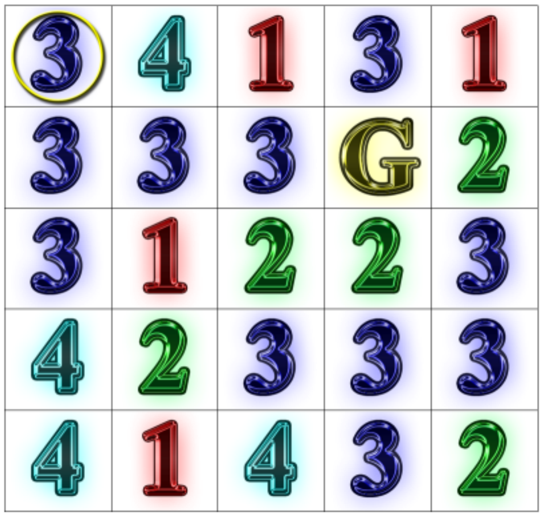
# Optimization Search



| | | | | |
|---|---|---|---|---|
| 3 | 4 | 1 | 3 | 1 |
| 3 | 3 | 3 | G | 2 |
| 3 | 1 | 2 | 2 | 3 |
| 4 | 2 | 3 | 3 | 3 |
| 4 | 1 | 4 | 3 | 2 |

**Rook Jumping Maze** is a puzzle problem of the sort that one might find in a newspaper alongside a crossword puzzle or a Sudoku puzzle. In a Rook Jumping Maze, start at the circled square in the upper-left corner and find a path to the goal square marked "G". From each numbered square, one may move that exact number of squares horizontally or vertically in a straight line. For more information on Rook Jumping Mazes, see http://cs.gettysburg.edu/~tneller/rjmaze/.

Suppose we want to build a system that generates Rook Jumping Mazes from scratch. That is, the system starts with an empty grid and must determine which number to put in each cell. The system must also determine which cell should be the start and which cell should be the goal.

One way to build a Rook Jumping Maze generator is to use optimization search: hill-climbing search, simulated annealing, or a genetic algorithm. An optimization search algorithm starts from an arbitrary state and uses an evaluation function to determine the "quality" of a potential solution.

**1. Describe a function that generates a starting state.**

> For each cell in the grid, randomly pick a number between 0 and k where k is the number of columns/rows in the grid. Randomly pick a cell to mark as the start. Randomly pick a cell to mark with a "G".

**2. Describe as many evaluation functions as you can.** An evaluation function will take a potential solution (in this case a grid of numbers, one of which is marked as a start and one that is marked with a "G") and returns a real number that reflects the quality of the solution. Be as precise as possible, but you don't have to write code.

For example, one criterion for a good Rook Jumping Maze might be that it is not impossible to solve. How would one determine whether a maze is impossible to solve? Another criterion might be the difficulty of the maze. How might one measure the difficulty of a maze? See http://modelai.gettysburg.edu/2010/rjmaze/evaluation2.html for some other ideas.

1. Impossibility: return a 0 if the maze is impossible, return 1 otherwise. To compute impossibility, run a breadth-first search backward from the "G" to the start. If the BFS fails to find a solution, then the maze is impossible. Running the search backward typically reduces the branching factor.
2. Difficulty as measured by the number of moves to solve the puzzle. Run a BFS and count the length of the solution. To make easier mazes use max_length – solution_length. To target a difficulty, use abs(target_length – solution_length).
3. Reduce the number of "black holes". A black hole is a cell that can never be reached. Use an exhaustive search such as Djikstra's algorithm. Must covert that to a real number, which is not entirely clear.
4. Reduce number of "white holes". A white hole is a cell that can never be reached when trying to solve the problem backward (from "G" to start). See #3 above.
5. Difficulty as measured by branching factor. Run BFS and measure the average number of successors generated at each iteration.
6. Difficulty as measured by the number of solutions. Run BFS but don't stop when the first solution is found.

Others exist too. The above list is not exhaustive.

Note that running a breadth-first search as the evaluation function means that every iteration of the hill climbing search must solve an NP-Complete search problem before determining whether the hill climber has gone up hill or down hill. This makes for a very slow optimization search because optimization search itself is NP-hard. However, the number of states that can be visited in BFS is bounded—in the worst case, BFS will only visit $n^2$ states where n is the number of rows/columns in the grid. That is, BFS can be considered a polynomial-time algorithm for the purposes of solving the maze. This means the optimization search is still NP-hard but with a high time complexity evaluation function.

Also consider the question: why breadth-first search instead of A*? BFS and A* have the same worst-case time complexity but A* typically has a better average time complexity. A* doesn't have a better average time complexity when solving Rook Jumping Mazes. Why?

**3. Describe a neighbor function.** This function takes a potential solution and generates a neighbor solution. This function would be used in blind hill climbing or simulated annealing to produce the next state. In a genetic algorithm this function would be the mutation operation.

> There are many potential neighbor functions. The simplest one is to randomly pick a single cell and randomly change the number in that cell.