Sudoku Solver

Eric Klinginsmith and **Sebastian Sánchez** and **Benjamin**
Washington State University Vancouver
14204 NE Salmon Creek Ave.
Vancouver, WA 98686

Abstract

This document explores the performance of two separate implementations of Sudoku Puzzle agent and the resulting evaluation of the implementations. Each implementation is a distinct search method to solve any Sudoku puzzle. They will be evaluated based on how fast each solves a given set of puzzles. It was our goal to explore these methods to ultimately see how well they worked.

Introduction

A Sudoku Puzzle like the one in figure *insert number here* is any n.power.2 by n.power.2 grid sub divided into n.power.2, n by n boxes. In the typical case n is equal to 3 which makes the Puzzle a 9 by 9 grid with 9 boxes, each box being it's own 3 by 3 grid. In any case, to solve the Puzzle, distinct numbers 1 through n.power.2 must be filled into every row and column of the grid so that no two numbers are in the same row, column, or box. Our implementations solve any Sudoku Puzzle with an n

<=

7.

Implementations

SearchNxN Implementation

The SearchNxN Implementation searches for the solution by filling in one box at a time. To do this SearchNxN first selects one of the boxes to fill in. Once selected SearchNxN will generate all possible solutions for the given box. These these solutions are added to a stack. Then for every element on the stack SearchNxN selects the next box to fill in. This process is repeated until either all boxes are filled in, or the stack is empty. If all boxes are filled in, then the solution is printed to stdout. Otherwise, there is no solution and false is returned. The inspiration behind this method was taken from the common computer science problem of a three coloring. A three coloring is a problem in which when given a map with n regions to color an agent finds a way to color the entire map with only three colors so that no two regions have the same two colors. If we think of the Sudoku Puzzle as being a map where no two numbers can be in the same row, column, or box and each box is a region, then a so called three coloring of a Sudoku Puzzle would be to "color" each box so that there are no violations where a color would be one of the possible solutions to that box.

About choosing the next box to fill in *sub heading*

Before moving on we would like to explain the different ways in which SearchNxN can pick the next box to fill in. There were two methods on how to accomplish this, they are: 1) inOrder 2) mostAdjacent

inOrder The inOrder method of choosing the next box is the simplest method. It chooses the next box in a left to right top to bottom fashion. This method, although simple, has a very small time constraint. This makes inOrder very viable.

mostAdjacent The mostAdjacent method of choosing the next box is a little more complicated. This function try's to reduce the branching factor of the search by choosing the next sector that will limit other sectors the most. This is accomplished by counting the number of unfilled in sectors in the same row or column of a given sector. The sector with the most unfilled in sectors in the same row or column as it is the one that this method chooses to fill in next. This idea came about because we wanted to limit the available possibilities for other surrounding boxes.

About sole candidates

We would like to mention one other part to our implementation. This is the idea of sole candidates. In a Sudoku Puzzle there are sometimes cells were only one number can be placed there. This means that for any partially solved puzzle can have one or more sole candidates. If a sole candidate is found then our implementation will try to find another. This continues until either all the sole candidates have been found or it hits a cell where there are no possibilities. If the result is the latter then that branch of the search is invalid and therefore the search must then try a different path to find the solution.

Experiments