Solving Sudoku Puzzles

Learning Goals

- Familiarize yourself with basic design pattern concepts.
- Understand how and when to use standard design patterns in your own software designs.
- Incorporate a set of design patterns into a non-trivial software design.
- Implement and test a set of design patterns as a part of a non-trivial program.
- Learn about and implement a type of randomized algorithm: genetic algorithms.

In this assignment, you will use a variety of object-oriented/C++ design patterns to develop a randomized algorithm to solve a non- trivial problem. Design patterns are classes and interactions that have proven useful over time, and so have been more-or-less codified among software professionals. Some are very interesting, but don't let the "gee-whiz!" of design patterns be the driving force behind your designs: they are solutions to particular problems. Beware **the story of the object-oriented toaster**, lest you suffer the same fate (or end up building a **$300 two-slice toaster** that only a small number of software engineers will ever buy).

The Problem: Solving Sudoku Puzzles

Sudoku is a type of puzzle that originated in Japan and attracted a worldwide following, appearing in newspapers, books, and web sites around the world. It is very simple: a 9-by-9 arrangement of squares, subdivided into 3-by-3 groups. A particular puzzle comes with a subset of the squares filled in with numbers from 1 through 9. To solve the puzzle, one must only fill in all the blank squares with numbers from 1 through 9 so that no row, column, or 3-by-3 block in the puzzle has the same number twice. That's all there is to it!

Deterministic Algorithms to Solve Sudoku

There are a variety of techniques that can be used to solve sudoku puzzles. One of the simplest involves the use of what are called pencil marks. In this approach, each unsolved square in the puzzle is first initialized with a list of all possible values: the numbers one through nine. Initialization continues: for each unsolved square, its row, column, and block are scanned. Each time a square that initially has a number assigned to it is found, that square's number is eliminated from the unsolved square's pencil marks. Once this process is done, each unsolved square's initialized pencil marks now contain only those numbers that are still possible solutions.

Solving the puzzle can now commence by finding squares with only single pencil marks: these squares must have that value. Setting that value for a square may change the pencil marks for other squares in its row, column, or block, hopefully producing more squares with singleton marks. This continues until either the puzzle is solved or there are no more squares with such forced values. Unfortunately, only the simplest sudoku puzzles can be solved in this manner; more sophisticated techniques are necessary for harder puzzles.

Genetic Algorithms

In this assignment, we will be using a type of randomized algorithm called ***genetic programming*** (GA) to solve sudoku puzzles. Randomized algorithms use a non-deterministic approach to search for problem solutions, coupling randomness with a means for quickly eliminating possibilities that are likely to be inferior. This may not be the most efficient strategy to solve sudoku puzzles, but it is illustrative of the power of such algorithms, in that just about all it requires is a way to evaluate the quality of a partially solved puzzle to achieve a solution (eventually). It will be interesting to see how well GA works for this problem.

GA is also an example of a general algorithm that can be implemented independently of the type of data operated on. You are already familiar with a simple approach to this, using templates. In this assignment, you will use more sophisticated OO design patterns to produce abstract descriptions of the basic operations needed for GA and then develop concrete subclasses of those abstractions.

Genetic Programming as Simplified Evolution

It is difficult for some people to understand how a process like evolution, which is primarily viewed as being driven by an underlying random process (mutation), can produce complex organisms that work so well. However, from a computational point of view, this is not surprising at all. In fact, the area of randomized algorithms is a hot topic — these algorithms have been used successfully to solve many problems for which no deterministic solution exists. From a computational point of view, evolution is a randomized algorithm.

The basic idea behind any type of problem solving is to generate possible solutions, evaluate how good they are, and then use that information to generate what are hopefully better possible solutions. This *optimization* process is repeated until either a perfect solution is found, a solution that is "good enough" is found, or you give up. This is what you do when you try to find a word in a dictionary. Problems like finding a word in a dictionary are easy to solve because the error in the possible solution at each step tells you how to change the solution for the next step. So, we can generate a *deterministic algorithm* that solves it.

If we think of the dictionary as a landscape that we're exploring, then it is like a smooth slope, and all we need to do is go downhill (like a rolling marble) until we reach the bottom — the word we're looking for. If we overshoot the word, we've rolled up the hill on the other side, and we just roll back the way we came. Simple problems have this sort of landscape.

Unfortunately, not all problems are simple. Many have rugged landscapes with many little "valleys". We don't want to get stuck in a little valley high in the mountains — we want to reach the lowest valley there is. Clearly, if the terrain is rough, then we can't use the slope of the terrain as our guide. Instead, we need to jump around a bit. And this is what randomization does for us: it allows us to explore the landscape, and it gives us the ability to "mix" the degree of jumping (how far we jump, how jumps are related to starting point, how "jumpy" we are at any point in time) and rolling.

So, evolution is a randomized algorithm in the sense that mutation and recombination change algorithms ("recipes" for organisms) in a random manner and the resultant solutions are "evaluated" by everyday life, with only the best (or, at least, "good enough") solutions able to reproduce (be modified for the next round of optimization).

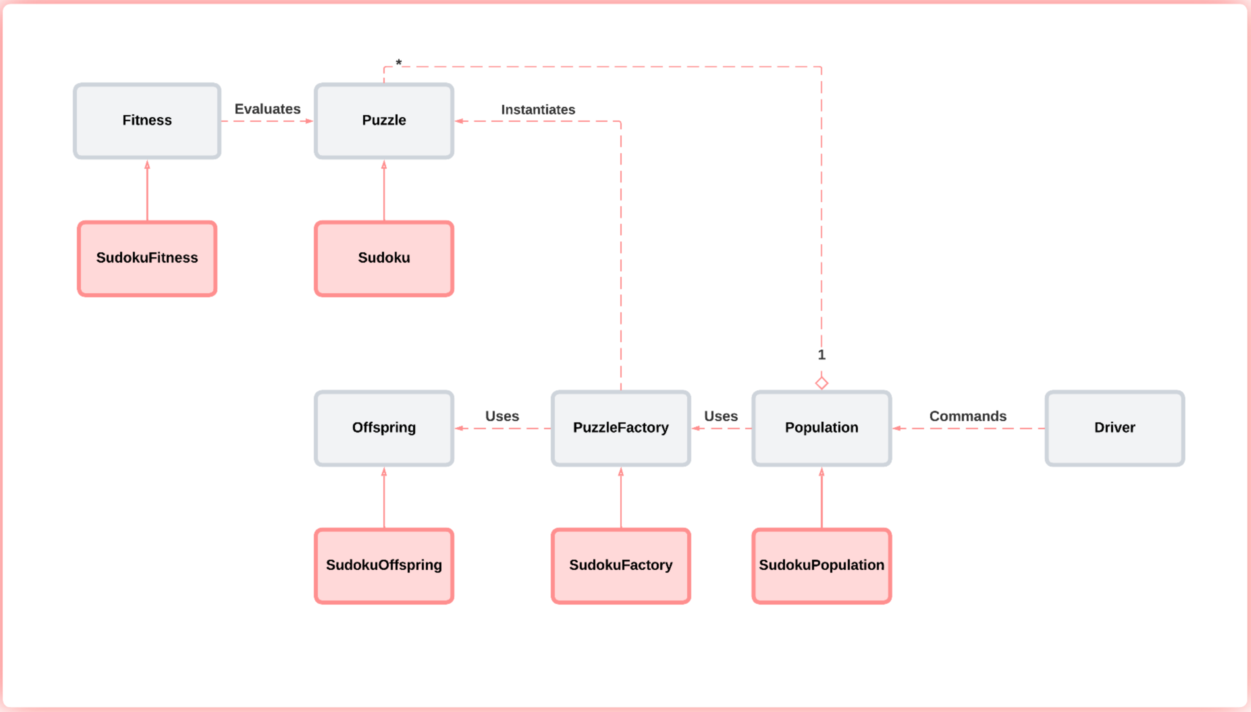
GA is one way to use an evolution-like process to find optimal algorithms. GA may in general work on representations of algorithms, but in the context of this assignment it will work on a representation of a possible sudoku puzzle solution. A potential sudoku puzzle can simply be a 9-by-9 array of digits, with all values filled in.

Clearly, if you fill in all squares in a puzzle randomly, it's unlikely you'll have a valid puzzle solution. But this is merely a starting point.

What makes this approach "genetic" is that a GA program is used that generates an initial, random *population* of solutions. In each *generation*, each of the members of the population have their fitness evaluated, and some fraction of the best solutions are selected for *reproduction*. These reproducing solutions may have their underlying representations randomly altered and may have parts of their representations swapped with other solutions. The result is a new generation of solutions based on the best of the previous generation. This process is repeated until the solution reaches some quality threshold or a maximum number of generations is exceeded.

Statement of Work

In this program, you are asked to design and implement a set of classes that define the abstractions necessary to solve puzzles of any kind using a GA. You will then produce concrete subclasses that implement these interfaces for Sudoku. The overall class relationships are described by the figure below.



Puzzle and Sudoku Classes

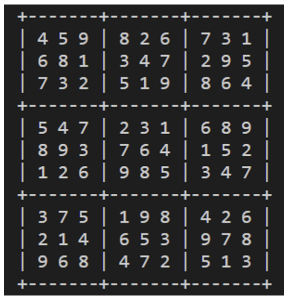
The Puzzle class defines the abstract interface for any puzzle to be used by the GA framework. A large number of puzzle (subclass) objects are held by a Population (subclass) object. Subclasses must implement stream I/O helper methods (the files **Sudoku.hpp** and **Sudoku.cpp** will define the standard stream I/O operators, **operator<<()** and

operator>>() , to call these methods). You are free to add additional pure virtual methods to this class as you flesh out your design.

Your Sudoku class must hold the representation of a Sudoku puzzle. This means that it must know the numbers of the 9 by 9 grid, as well as the fitness level of the puzzle.

The Sudoku stream input helper method must read a puzzle in a compact text format. The format consists of a sequence of 81 digits, as ASCII text, each corresponding to the contents of one square in the puzzle grid, in *row-major* order (i.e., starting at the upper left corner, reading across each row and then continuing with the next row, ending at the bottom right corner). Blank (variable) squares are represented with zeros; fixed squares of course have non-zero values. Your method should ignore all non-digit characters (i.e., scan the input character by character and use the first 81 ASCII characters in the set {0, 1, ..., 9} found). Your input operator should not make any assumptions about white space (i.e., it works for inputs that do and do not have white space between characters).

The Sudoku stream output helper method must produce a "human friendly" output as formatted text. Output a puzzle as 13 lines of text, with each line either being a separator or one containing the numbers in a row of the puzzle separated by single spaces. Use the '|', '+', and '-' characters to separate 3-by-3 blocks (so, each line of output should be 25 printing characters long: 9 digits, 12 spaces, and 4 '|' for "non-divider" lines and 21 '-' and 4 '+' for "divider" lines).



```
+-------+-------+-------+
| 4 5 9 | 8 2 6 | 7 3 1 |
| 6 8 1 | 3 4 7 | 2 9 5 |
| 7 3 2 | 5 1 9 | 8 6 4 |
+-------+-------+-------+
| 5 4 7 | 2 3 1 | 6 8 9 |
| 8 9 3 | 7 6 4 | 1 5 2 |
| 1 2 6 | 9 8 5 | 3 4 7 |
+-------+-------+-------+
| 3 7 5 | 1 9 8 | 4 2 6 |
| 2 1 4 | 6 5 3 | 9 7 8 |
| 9 6 8 | 4 7 2 | 5 1 3 |
+-------+-------+-------+
```

Population and SudokuPopulation Classes

A Population is a container for a number of Puzzles. It can use a Fitness strategy to evaluate the quality of a Puzzle. It can "cull" Puzzles from the population (based on their fitness). It can use a PuzzleFactory and an Offspring strategy to produce a new generation of Puzzles from the most fit members of the old generation. It performs these operations when commanded by the main GeneticAlgorithm. Population or its subclasses must implement:

- constructors (that create an initial, random set of Puzzles).
- a "cull" method (that eliminates the X% least fit members of a generation).
- a "new_generation" method (that produces a new generation with N individuals for every single individual in the old generation — presumably after the old generation has been culled, so that the population size stays constant from one generation to the next).
- a "best_fitness" method (that returns the fitness of the best Puzzle in the current generation),
- and a "best_individual" method (that returns the Puzzle with the highest fitness).

You are free to add additional pure virtual methods to this class as you flesh out your design. The SudokuPopulation class implements all Population methods for Sudoku objects.

PuzzleFactory and SudokuFactory Classes

A PuzzleFactory is an **abstract** design pattern for producing puzzles. Subclasses must implement a "create_puzzle" method, which returns a new Puzzle (subclass) object. You are free to add additional pure virtual methods to this class as you flesh out your design.

The SudokuFactory uses a SudokuOffspring object and implements "create_puzzle" to produce new Sudoku objects.

Fitness and SudokuFitness Classes

A fitness value is used by a Population to compare individual Puzzles. Fitness and its subclasses are implementations of the **strategy** design pattern. Fitness subclasses must implement a "how_fit" method, which takes a Puzzle (subclass) object and returns an int . Perhaps counter-intuitively, lower fitness values indicate more fit individuals (in other words, how_fit is indicating how far a Puzzle is from perfection).

The SudokuFitness class implements "how_fit" for Sudoku objects. To evaluate the fitness score, count the total number of conflicts in a puzzle: how many duplicate entries there are in each row, column, and 3- by-3 block. Each duplicate pair will count as two conflicts (not just one): eg. two 3s in the same row would increase the fitness by 2, three 3s in the same row would increase the fitness by 6, and two 3s in the same row *and* block would increase the fitness by 4.

Offspring and SudokuOffspring Classes

Offspring is used by PuzzleFactory to create new Puzzles from old ones. Offspring and its subclasses are also implementations of the strategy design pattern. Offspring subclasses must implement a "make_offspring" method, which takes a Puzzle and returns a new Puzzle.

The SudokuOffspring class implements "make_offspring" for Sudoku objects. In this program, you need not implement all of the different genetic operations mentioned **here** (crossover, mutation, and architecture-altering operations); you need only implement mutation. A simple implementation would make a copy of the original Sudoku, with some probability (you might start with values like 2% or 5%) of changing each variable number in the puzzle as it is copied.

The Program

Your program should take two *command line* arguments: the size of the population and the maximum number of generations. It should then read the Sudoku puzzle from cin, print that puzzle in the “human friendly” format, and create the first generation (i.e., randomly fill in the squares that are not fixed to produce a population of the specified size with that many different Puzzles). It should then begin the GA iterations. In each iteration, it should:

1. command the Population to cull 90% of its Puzzles, then
2. command the Population to reproduce a new_generation
3. halt iterations if a solution has been found or if the maximum number of generations has been reached, otherwise move on to the next iteration

At the end, it should output whether the puzzle was solved or not and the solution (along with the number of generations it took to solve) or the best puzzle found (along with its fitness value).