# 

Homework 4: Sudoku

Solving Sudoku Using Genetic Algorithms

Srikar Chava, Brian, Pavan Sai Ram Korumilli

UWB CSS 343

Boxiang Pan

# **Table Of Contents**

[**Table Of Contents**](#_9iqykh665s04) **2**

[**Compilation and Running Instructions**](#_3qxr6gbdvprs) **3**

[**Introduction**](#_aem6niteg342) **3**

[**Assumptions**](#_o4fm33o3xy2m) **4**

[**Class Design**](#_838kzc837chj) **4**

[Genetic Algorithm (GeneticAlgorithm.h & GeneticAlgorithm.cpp)](#_kzo93uudua2b) 4

[Puzzle (Puzzle.h)](#_wgwoc5cd755) 5

[Fitness (Fitness.h)](#_z8sfclm9w72s) 5

[Sudoku Fitness (SudokuFitness.h & SudokuFitness.cpp)](#_31qvnpb2vgmk) 5

[Puzzle Factory (PuzzleFactory.h)](#_pqgv3hrzgenc) 5

[Sudoku Factory (SudokuFactory.h & SusokuFactory.cpp)](#_jsoinkukwvpo) 5

[Population (Population.h)](#_4vm6f3rx47xq) 5

[Reproduction (Reproduction.h)](#_9kt2drvwr7uo) 5

[Sudoku Offspring (SudokuOffSpring.h & SudokuOffSpring.cpp)](#_rehuy0cjy7vk) 5

[**Implementation and Testing**](#_q3xqd0lzd8gx) **6**

[**Outputs**](#_kgt72jn3bt0e) **6**

[Linux Program Output](#_ged7fcofovyg) 6

[**Analysis**](#_v87s13qyw0uh) **9**

[**Conclusion**](#_itarwuabp92d) **9**

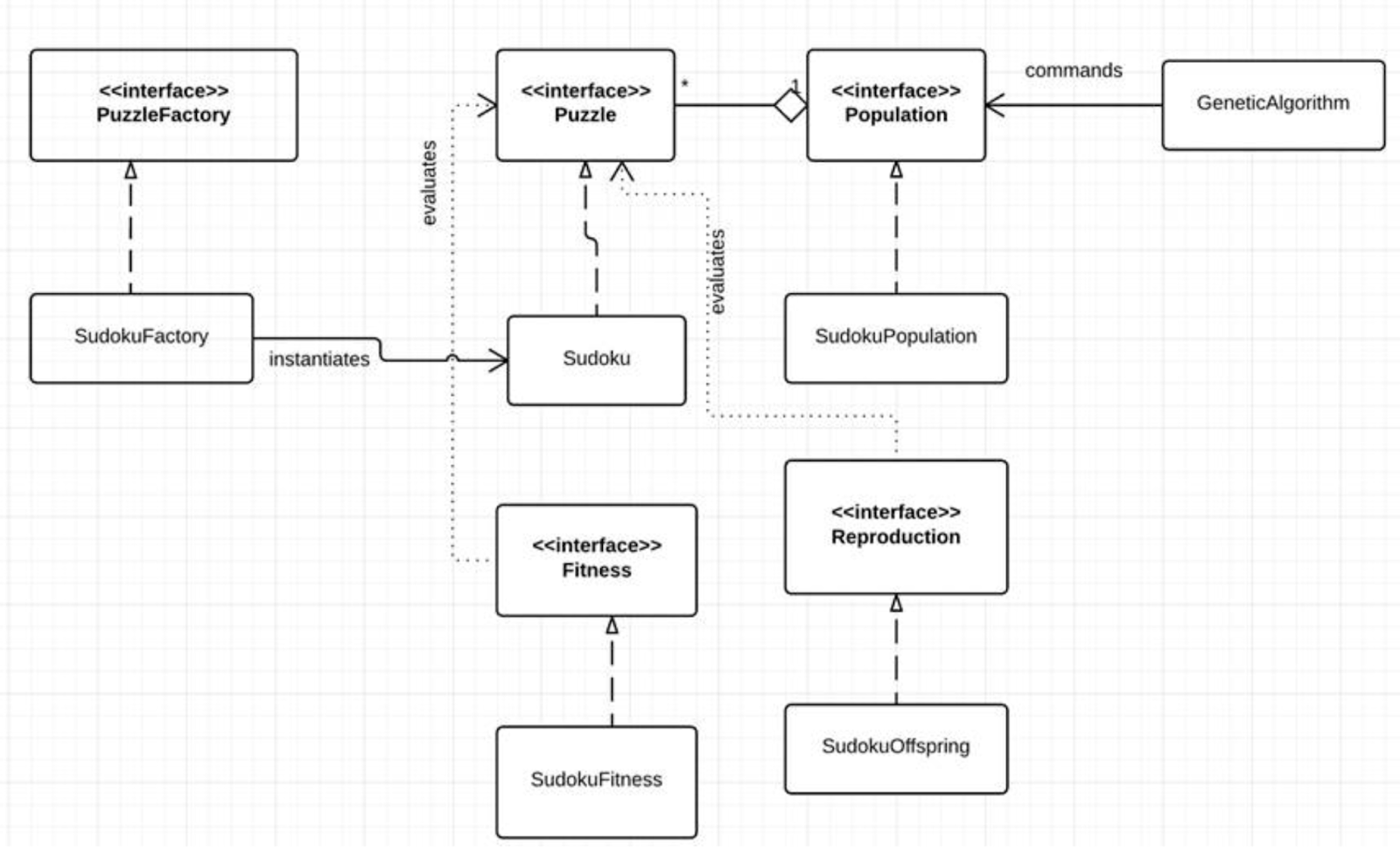
# **Compilation and Running Instructions**

* **Compile with:** g++ \*.cpp;
* **Run with:** ./a.out

# **Introduction**

For this program, we were tasked to make a genetic algorithm that attempts to solve sudoku puzzles.

**Program Flow Diagram:**



# 

# **Assumptions**

The program assumes:

* The main input for the program will be through Cin when the program prompts the user to input the values for the Sudoku
* The input will be 81 consecutive numbers with no spaces in between
* The first command line argument is int populationSize
* The second command line argument is int maxGeneration
* The program will dump unless the command line arguments are passed in.
  + Ex: ./a.out 10 10
  + (10 population size, 10 max generations)
* Intended empty spaces within the Sudoku input will be represented by 0’s

# **Class Design**

This section explains all of the design principles behind each of the classes in our program.

## Genetic Algorithm (GeneticAlgorithm.h & GeneticAlgorithm.cpp)

The Genetic Algorithm class will host the input handling from main.cpp and include the function GeneticAlgorithm. This function will generate the first generation based on the initial Sudoku puzzle, Sets mutation probability to 5% for future generations, and outputs the best individual based on the howFit function and the best individual among the populations.

## Puzzle (Puzzle.h)

The Puzzle class will host the operator overloads and the function getData() and the virtual functions: read(), print(), and findFitness() from the Sudoku class. The Puzzle class also implements the operator overloads, which calls the read and print virtual functions in order to handle the input and output capabilities of the program correctly.

Sudoku (Sudoku.h & .cpp)

The Sudoku class hosts the Sudoku constructors, functions including: getFixedValue, getValue, setValue, read, findFitness, and getFitness. It also holds two 2d arrays; grid and fixedValue. Grid holds the Sudoku numbers within the board and fixedValue is a bool which indicates whether or not the value in each square should be fixed or not.

## Fitness (Fitness.h) Sudoku Fitness (SudokuFitness.h & SudokuFitness.cpp)

The Fitness classes hold the function howFit. This function calls the getFitness function on a Sudoku function. This determines the fitness level of a Sudoku. The Sudoku is dynamically casted within Sudoku Fitness.

## Puzzle Factory (PuzzleFactory.h) Sudoku Factory (SudokuFactory.h & SusokuFactory.cpp)

The Sudoku Factory class hosts the createPuzzle function that returns a puzzle object. This serves as an abstract factory design pattern for producing puzzles.

## Population (Population.h) Sudoku Population (SudokuPopulation.h & SudokuPopulation.cpp)

The Sudoku Population class is a container for the number of puzzles. It holds a struct called compare that compares two puzzles by their fitness value and returns a bool value. It also holds a min heap of puzzles sorted by their fitness as well as the functions: cull, newGeneration, bestFitness, and bestIndividual. This class implements all population methods for Sudoku objects using a SudokuFitness object.

## Reproduction (Reproduction.h)

The Reproduction class utilizes the Puzzle class to make new puzzles. It uses the virtual function makeOffspring, which takes a puzzle and returns a new puzzle. This function also has a mutation that changes the offspring for each puzzle by 5%.

## Sudoku Offspring (SudokuOffSpring.h & SudokuOffSpring.cpp)

The Sudoku Offspring class hosts the functions makeOffspring, setProb. As well as private member variables valueChange, checkFixed, and probability. The probability variable sets the probability and is the basis for change in every offspring. This causes each offspring generated to be mutated, and eventually reach the point of best fitness.

# **Implementation and Testing**

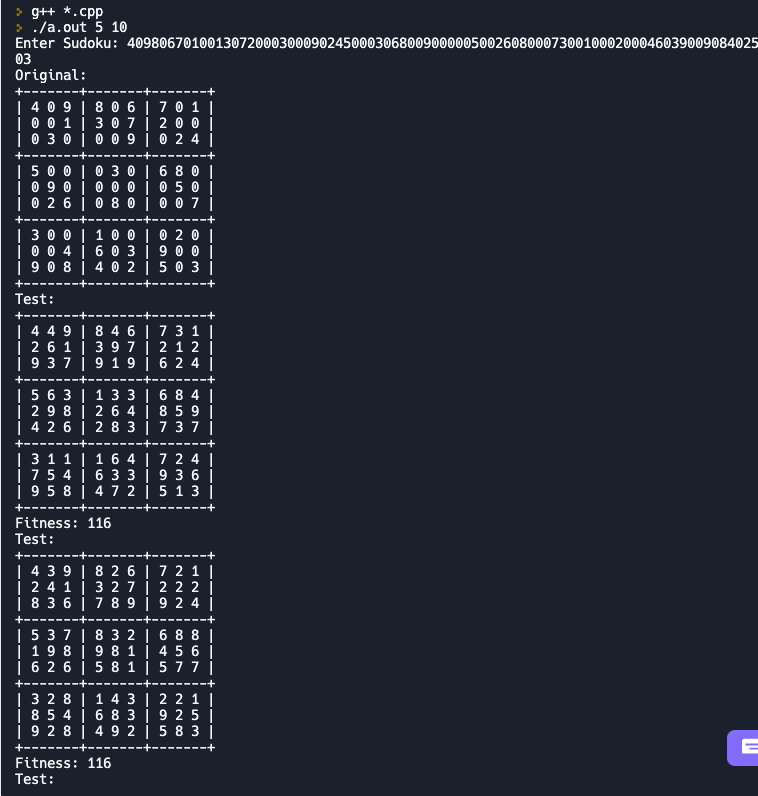
During the building phase of our program, we made sure to implement the input and output functions so that we could continually test our program. We were having early success with the implementation and testing of the Sudoku and Puzzle classes, and all of our initial tests came back fine. From here we went on to implement the SudokuOffspring, SudokuFitness, and SudokuFactory, and SudokuPopulation. We implemented these with a logic standpoint

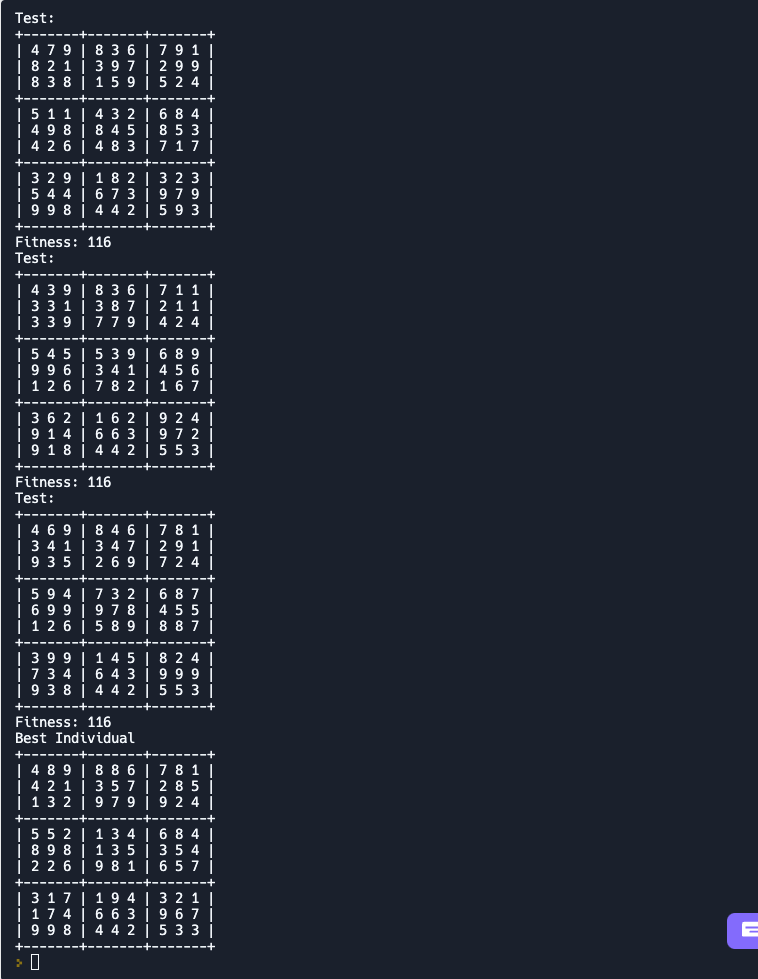
# **Outputs**

## Linux Program Output

**Ran with:** ./a.out 5 10

The output was engineered to ultimately match the ones outlined in the specifications. The program randomises the gap values in the input, or in other words, where the input has 0’s in the spaces. The formatting is done within the “print” function in Sudoku class, which gets called by an operator overload. From here the Program outputs the number of populations specified by the command line argument. The program also outputs the original puzzle, followed by the Fitness value for each grid with the randomised puzzles as well as the best individual at the end. Our fitness algorithm was on the final steps of being done but all other algorithms including best individual, offspring, and mutations were all tested and do work. The reason these do not show in the output is because of the Fitness implementation that is malfunctioning.





# **Analysis**

* Can you reliably get good solutions?
  + Since we were unable to correctly assess the Fitness function, our program is unable to reliably get good solutions. The only fix it requires, that we have identified the problem to be within, is the priority queue where Fitness is stored. We believe that this is actually bringing the higher fitness levels to the top, which actually means bad puzzles since the higher the fitness, the worse the puzzle is. Other than this error, we are confident that our program will work and be reliable.
* How many generations does it take?
  + Our program only outputs one generation because of our Fitness issue.
* How does this depend on how hard the puzzle is?
  + The puzzle difficulty does not affect the generations within our program.
* Does adding additional genetic operators improve performance?
  + We have not tested if adding genetic operators improves performance

# 

# **Conclusion**

In conclusion, this program from the start really tested our team’s persistence and problem solving skills. The overall program is being held back by one function not working but we are confident that this would be solved with just a little more time. Our team as a whole learned a numerous amount about inheritances within C++ and design patterns that are necessary within large programs like this one. We also learned a lot about how genetic algorithms are structured and composed in a very short amount of time with almost no background knowledge for all of us on this topic. What we are most proud of is how far we came to finish the program with all the external circumstances we faced. Our team showed true grit as we spent countless hours throughout the night (disregarding sleep) to finish this program. We also found continuous unit testing to be a major asset to our development cycle, as well as clear communication with topics such as “what needs to be fixed, and what has been done so far”. The debugging and unit testing nature of the project also seemed to be very great exposure for us and something that we will continue to develop and better in the future. Overall we felt the project to be very good exposure to team work on large projects. It was very time-consuming and stressful, but the sense of accomplishing tasks in a team setting was very refreshing for all of us, and we definitely walked away from this project with a lot of new understanding about Genetic Algorithms and important design patterns.