

# Solucionador de Sudoku utilizando algoritmo genético

José Davi Viana Francelino  
Thiago de Oliveira Cordeiro

dezembro, 2023

## Resumo

Este artigo propõe a utilização de algoritmos genéticos para a resolução do jogo Sudoku, juntamente com uma análise comparativa entre os resultados obtidos por um algoritmo exato e uma heurística implementados por meio de modelagem em grafos em versões anteriores deste estudo. A aplicação dos métodos mencionados foi realizada utilizando a linguagem de programação C++. As versões desenvolvidas neste projeto abordam a resolução do Sudoku genérico  $n \times n$ . Vale ressaltar que o Sudoku representa um problema NP-completo, caracterizado como um desafio de raciocínio lógico que está intrinsecamente ligado ao conceito de quadrado latino.

**Palavras-chaves:** algoritmo genético. coloração de grafos. sudoku.

## Introdução

O Sudoku é um popular quebra-cabeça que desafia os jogadores a preencher uma grade  $n \times n$ , com dígitos de 1 a  $n$ , com  $n = k^2$ , para algum inteiro positivo  $k$ , onde algumas posições vêm previamente preenchidas, de modo que cada linha, coluna e subgrade  $k \times k$  contenham todos os dígitos exatamente uma vez. Uma abordagem comumente utilizada para solucionar o Sudoku envolve métodos exatos, como busca exaustiva ou backtracking.

Contudo, uma alternativa interessante é a aplicação de meta-heurísticas, como algoritmos genéticos, inspirado em princípios da computação evolucionária, para resolver o Sudoku. Enquanto os métodos exatos podem ser computacionalmente intensivos, as meta-heurísticas oferecem uma abordagem mais eficiente para encontrar soluções próximas do ótimo. Nesse contexto, os algoritmos genéticos exploram conceitos de evolução e seleção natural para encontrar soluções promissoras.


Este trabalho se concentra na implementação de um algoritmo genético para solucionar o Sudoku. Em vez de se basear em busca exaustiva ou backtracking, a ênfase está na aplicação de uma abordagem de busca mais ampla e adaptativa, inspirada no processo evolutivo, para encontrar soluções viáveis e eficazes para o desafio do Sudoku.

## 1 Descrição do problema

O problema abordado neste artigo consiste em gerar um solucionador de instâncias do jogo Sudoku. Por sua vez, esse jogo foi concebido por Howard Garns, um arquiteto e criador de quebra-cabeças. O Sudoku foi, possivelmente, baseado em uma estrutura matemática denominada "quadrado latino", desenvolvida pelo matemático suíço Leonhard Euler [3].

O objetivo do jogo Sudoku é encontrar uma configuração final válida, que atende as regras do jogo, para um dado tabuleiro. Isso implica em estender a grade inicial de forma a preencher todas as células em branco com números de 1 a  $n$ , garantindo que não haja repetições de números em nenhuma linha, coluna ou bloco. O Sudoku é comumente jogado em um tabuleiro  $9 \times 9$ , que é subdividido em blocos menores de  $3 \times 3$ . No início do jogo, algumas células do tabuleiro já estão preenchidas, enquanto outras permanecem em branco, desafiando os jogadores a resolver o quebra-cabeça atentando as regras do jogo. A [Figura 1] ilustra como é o jogo Sudoku e uma possível solução para a instância exibida na figura.

5						2	3	
				4	8	9	7	
	8				3			
1			3	8	9	7	4	
				5	6			
8	9							
	4						6	7
6	2		8	3				1
	5	1		6			2	



5	1	4	6	9	7	2	3	8
2	3	6	1	4	8	9	7	5
7	8	9	5	2	3	6	1	4
1	6	5	3	8	9	7	4	2
4	7	3	2	5	6	1	8	9
8	9	2	4	7	1	3	5	6
3	4	8	9	1	2	5	6	7
6	2	7	8	3	5	4	9	1
9	5	1	7	6	4	8	2	3

Figura 1 – Exemplo de uma instância de Sudoku  $9 \times 9$ .

Ademais, nas versões deste trabalho de algoritmo exato e algoritmo heurístico, modelou-se o Sudoku como um grafo, onde pode-se pensar em cada célula do tabuleiro como um vértice, onde dados dois vértices  $v$  e  $w$ , eles são adjacentes se e somente se  $v$  e  $w$  pertencem a mesma linha, mesma coluna ou mesmo bloco/subgrade. Deste modo, como existe uma aresta conectando cada vértice aos outros contidos no bloco, coluna e linha, podemos realizar a aplicação da coloração de vértices. Para cada dois vértices adjacentes, eles não são coloridos com a mesma cor. Cabe destacar que as cores foram representadas por números. Foi selecionado a modelagem por meio de grafo simples, porém também poderia ser modelado por hipergrafos. Na figura abaixo é ilustrado a modelagem do grafo, mostrando um vértice e seus adjacentes no tabuleiro:

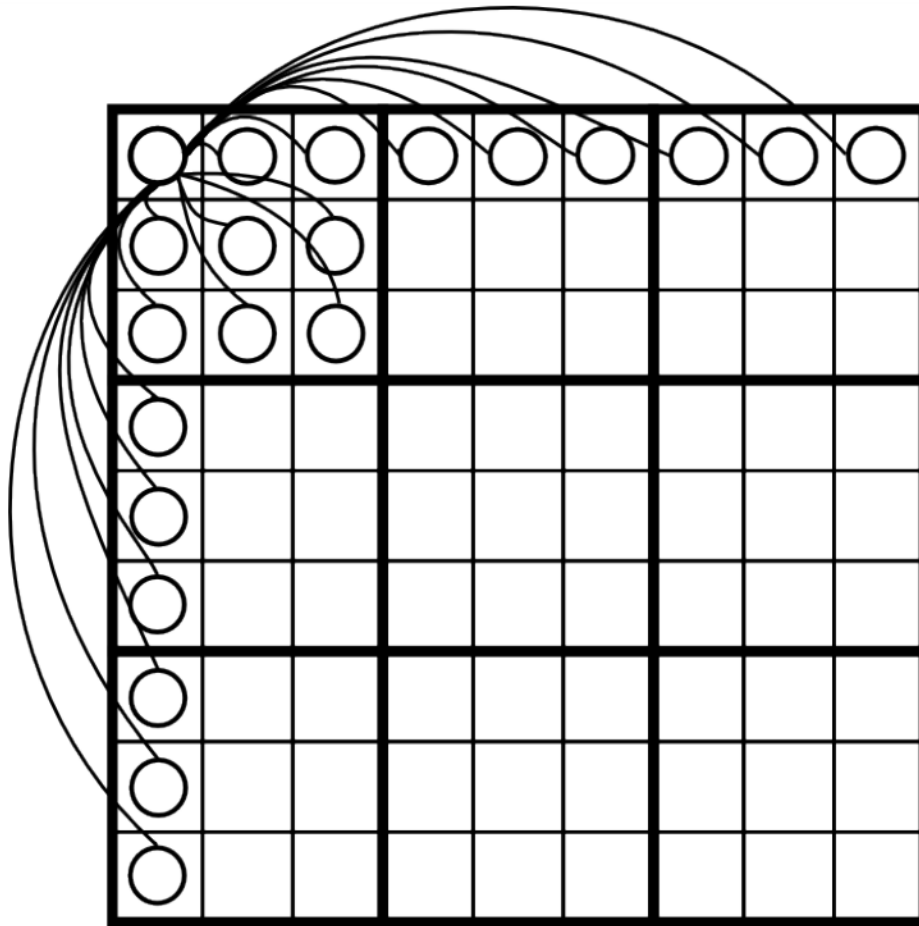


Figura 2 – Modelagem de um vértice do Sudoku  $9 \times 9$  em grafos.

## 2 *Heurística e Meta-Heurística*

Heurísticas são estratégias simplificadas e regras práticas usadas para resolver problemas de maneira eficiente, embora não disponibilizem garantias teóricas de que a solução ótima exata seja alcançada. Elas são frequentemente aplicadas em situações onde os métodos exatos são computacionalmente custosos ou inviáveis. Ao oferecerem atalhos ou diretrizes aproximadas, as heurísticas permitem tomar decisões rápidas, explorando o conhecimento prévio do problema ou características específicas do domínio.

Já as meta-heurísticas podem ser definidas como estratégias de alto nível usadas para otimizar processos de busca em espaços complexos e de grandes dimensões. Elas transcendem as limitações das heurísticas convencionais, fornecendo um arcabouço flexível e adaptativo para explorar soluções potencialmente ótimas em problemas difíceis. Ao invés de se concentrarem em encontrar soluções exatas, as meta-heurísticas utilizam conceitos inspirados na natureza, como evolução, para iterativamente explorar e refinar conjuntos de soluções, buscando atingir uma ótima solução ou próxima a ela.

Um dos exemplos de meta-heurísticas são os algoritmos genéticos. Eles são inspirados na seleção natural e na genética. Simulam processos evolutivos para resolver problemas complexos de otimização e busca. A evolução se concretiza através da geração sucessiva de indivíduos que apresentam aprimoramentos em relação aos seus predecessores.

Tal característica teve como inspiração a evolução das espécies (AKEMI, 2018). Desse modo, o algoritmo genético se destaca como uma estratégia eficaz para resolver problemas complexos. E foi esta a meta-heurística selecionada neste trabalho.

### 3 Complexidade do problema

Em [2], é demonstrado que o problema do Sudoku pertence a classe dos problemas NP-difícil.

### 4 Estado da arte

Há algumas abordagens que já foram apresentadas para a resolução de sudokus genéricos a partir de meta-heurísticas, algoritmos exatos e aproximativos. Sendo alguns destes:

Primeiramente, será mostrado algumas meta-heurísticas:

- *Algoritmo genético*: utiliza uma abordagem inspirada na evolução biológica. Começa com uma população de soluções (tabuleiros) aleatórias, avalia sua adequação por meio de critérios de fitness (quão próximo está da solução final) e aplica operadores genéticos (como crossover e mutação) para criar novas soluções. Repete o processo até encontrar a solução ou atingir um critério de parada (AKEMI, 2018).
- *Otimização por Enxame de Partículas*: baseia-se no comportamento de um enxame de partículas. Cada solução (partícula) se move no espaço de busca (possíveis configurações do Sudoku) e é influenciada pela melhor solução encontrada por ela mesma e pelo enxame. As partículas se movem em direção às soluções mais promissoras, buscando otimizar o tabuleiro. [6]
- *Busca Tabu*: usa memórias de curto e longo prazo para explorar o espaço de busca do sudoku. Evita revisitar soluções já exploradas recentemente (lista tabu) e usa estratégias de diversificação e intensificação para explorar e focar em áreas promissoras do espaço de busca. [7]

Acerca de algoritmos exatos que podem ser utilizados na resolução de sudokus, pode-se citar os seguintes:

- Algoritmo de *backtracking*: implementado por Skiena, o qual faz uso da técnica de Forward Checking e a heurística de Minimum Remaining Values. Neste algoritmo, há a interrupção da busca pela solução, ou seja a poda da sub-árvore de busca, quando uma variável ainda não testada não possui mais valores possíveis em seu domínio. Dentro do cenário do Sudoku, quando o algoritmo de backtracking identifica uma célula não preenchida na qual não é mais possível inserir nenhum valor, a busca atual é finalizada. [4]
- Algoritmo proposto por Norvig baseado em propagação de restrições. Neste algoritmo, é usado uma técnica que efetua a remoção de valores dos domínios das variáveis que, com certeza, não podem mais ser atribuídos. Durante a execução do algoritmo, que se dá por meio da busca em profundidade (backtracking padrão). É realizada

a exclusão de candidatos das células do tabuleiro à medida que se infere que esses valores não podem mais ser inseridos. [6]

- Algoritmo de backtracking implementado fazendo uso da técnica de manipulação de Bits. Neste algoritmo, foi-se usado como base o algoritmo de propagação de restrição do Norvig e neste algoritmo os candidatos são representados por meio de vetores de bits, utilizando manipulação de bits para realizar de forma eficiente operações como inserção, remoção e verificação de candidato. [2]

Agora, serão expostas algumas heurísticas e algoritmos aproximativos presentes na literatura:

- Forward Checking: este método sugere que a busca por uma solução seja encerrada, ou que ocorra a poda da sub-árvore de busca, no momento em que alguma variável se torna inviável, ou seja, quando seu conjunto de valores possíveis se torna vazio. Isso significa que não há mais opções viáveis para essa variável. No cenário do Sudoku, sempre que o algoritmo de backtracking se depara com uma célula não preenchida na qual não haja mais possibilidade de inserir qualquer valor, a busca em andamento é finalizada. [1]
- Minimum Remaining Values (MRV): essa heurística sugere que é preferível selecionar variáveis com um quantitativo menor de opções disponíveis, ou seja, escolher a variável mais restrita. Dentro do contexto do Sudoku, isso se traduz na seleção da célula que possui o menor número de candidatos toda vez que se faz necessário atribuir um valor a uma nova célula que ainda não foi preenchida. [1]
- Heurística do Valor Menos Restritivo: essa heurística sugere que dada uma variável, o recomendado é escolher os seus valores a partir do menos restritivo, ou seja, proporciona maior flexibilidade para as próximas atribuições. No contexto do Sudoku, envolve a seleção prioritária de células com o menor número de candidatos possíveis. Isso é feito para simplificar a resolução, uma vez que preencher células com menos restrições pode levar a menos conflitos e, portanto, a uma busca mais eficiente por uma solução válida. [1]

## 5 Banco de instância

Inicialmente, para gerar as instâncias do Sudoku, foi utilizado [um gerador](#) de instâncias. No entanto, não foram encontrados geradores de puzzles maiores do que 9x9 e os puzzles gerados por ele, mesmo no nível mais difícil, eram resolvidos de forma correta pelo algoritmo.

Portanto, usou-se instâncias extraídas do site [sudoku.com](#) para os puzzles 9x9 e [sudoku-pluzzes-online](#) para os puzzles 16x16 e 25x25.

Por conta do processo manual de transformar as imagens dos puzzles em uma entrada que o algoritmo entenda, não foi possível utilizar tantas instâncias do puzzle, principalmente para os Sudoku maiores. Foram gerados aleatoriamente 3 puzzles de cada dificuldade (easy, medium, hard, expert, evil) para o 9x9, 1 de cada dificuldade (beginner, confirmed, expert) para o 16x16 e 1 de cada dificuldade beginner e expert para o 25x25.

## 6 Implementação da meta-heurística

Utilizando a linguagem de programação C++, implementamos um algoritmo meta-heurístico (anexo I) para resolver o Sudoku. O algoritmo utiliza a meta-heurística Algoritmo Genético para solucionar o problema abordado neste trabalho.

### 6.1 Algoritmo genético

Os Algoritmos Genéticos começam representando potenciais soluções para o problema em questão como "indivíduos". Esses indivíduos são codificados em estruturas chamadas cromossomos, que, por sua vez, consistem em genes. A escolha adequada da codificação é essencial e pode variar dependendo do problema.

Uma população inicial de indivíduos é gerada aleatoriamente ou usando heurísticas específicas. Cada indivíduo possui um conjunto de características representadas pelos seus genes.

Cada indivíduo na população é avaliado quanto à sua aptidão para resolver o problema. A função de aptidão atribui um valor numérico indicando o quão bem o indivíduo atende aos requisitos do problema. Esse valor é crucial para a seleção de pais para reprodução.

Indivíduos são selecionados para a reprodução com base em sua aptidão (fitness). Métodos comuns incluem a roleta viciada, torneios e seleção proporcional à aptidão. A ideia é dar mais chances de reprodução aos indivíduos mais aptos, imitando o processo de seleção natural.

Os pais selecionados são combinados para produzir descendentes. O crossover envolve a troca de informações genéticas entre os pais, criando novos indivíduos que carregam características de ambos. O ponto de crossover é crucial e pode variar dependendo da aplicação.

Para introduzir diversidade genética na população, alguns genes dos descendentes podem ser alterados aleatoriamente. A mutação é crucial para evitar a convergência prematura para uma solução subótima e manter a exploração do espaço de busca.

Os descendentes gerados substituem parte da população original. A escolha dos indivíduos que serão substituídos pode ser determinada pela aptidão, garantindo uma evolução contínua da população em direção a soluções mais eficientes.

O processo de seleção, crossover, mutação e substituição é repetido por várias gerações até que um critério de parada seja atingido. Isso pode ser um número fixo de gerações, um limiar de aptidão ou qualquer condição definida pelo usuário.

Ao final das iterações, a melhor solução encontrada ao longo das gerações é considerada a solução ótima ou uma boa aproximação dela. Uma análise detalhada dos resultados e do desempenho do algoritmo é crucial para entender sua eficácia e fazer ajustes, se necessário.

Para aplicar o algoritmo genético no Sudoku, utilizamos como aptidão a quantidade de quebras as condições do Sudoku, priorizando as soluções com a menor quantidade de quebras.

## 7 Análise do algoritmo

No anexo 1 é mostrado o método algoritmo genérico, ele executa um loop principal que executa um loop interno (crossover e mutação) e ordena a população. A complexidade de tempo depende da execução do loop principal, tornando-se aproximadamente  $O(g * (n^2 + n^2 * n + n \log n))$ , onde  $n$  é o tamanho do lado do tabuleiro e  $g$  é o número máximo de gerações.

No anexo 2 é apresentado o método de aptidão do algoritmo genético e esse método itera sobre cada célula do tabuleiro várias vezes, comparando os números em linha, coluna e subgrid. Sua complexidade é  $O(n^2 * n)$ , onde  $n$  é o tamanho do lado do tabuleiro (rank \* rank).

No anexo 3 é mostrado o método de mutação do algoritmo genético e este método itera sobre cada célula do tabuleiro uma vez. A complexidade de tempo é  $O(n^2)$ , onde  $n$  é o tamanho do lado do tabuleiro (rank \* rank).

Já no anexo 4 é mostrado o método de crossover, que itera sobre cada célula do tabuleiro duas vezes, para criar dois novos tabuleiros. A complexidade de tempo é  $O(n^2)$ , onde  $n$  é o tamanho do lado do tabuleiro (rank \* rank).

## 8 Resultado do algoritmo

O algoritmo meta-heurístico não conseguiu resolver nenhuma das instâncias de forma perfeita, mesmo alterando os parâmetros e mudando o funcionamento de alguns métodos.

Enquanto o algoritmo heurístico desenvolvido gerava tabuleiros de Sudoku usando mais números que o permitido, o meta-heurístico usa apenas os números permitidos, mas quebra a regra de não repetir um número na mesma linha, coluna ou bloco. Essa diferença torna difícil comparar os dois.

## 9 Links importantes

[Como jogar Sudoku: estratégia, dicas e regras.](#)

O link acima fornece um panorama geral sobre o jogo abordado neste projeto. Seu funcionamento, regras e afins.

[O que é um grafo?](#)

Este segundo link apresenta conceitos, definições e exemplos da teoria de grafos, abordando conceitos importantes para entendimento satisfatório deste trabalho tais como adjacência e matriz de adjacência.

[O problema da coloração dos vértices de um grafo](#)

No link acima é mostrado o que é o problema da coloração de vértices e também mostra heurística para solucionar.

[Repositório de implementação do algoritmo deste projeto](#)

O link acima trata-se do repositório onde o código desenvolvido durante este projeto está hospedado.

## Conclusão

Por meio da elaboração deste projeto, foi possível consolidar o conhecimento obtido na disciplina de Projetos e Análise de Algoritmos (PAA), colocando em prática conteúdos estudados na disciplina, em especial o conteúdo de análise de complexidade, complexidade de problemas, backtracking e algoritmos aproximativos, heurísticas e meta-heurísticas, além de revisar conceitos relevantes como o de grafos simples, modelagem de problemas em grafos e coloração de grafos.



# Sudoku solver using graph coloring

José Davi Viana Francelino  
Thiago de Oliveira Cordeiro

dezembro, 2023

## Abstract

This article proposes the use of genetic algorithms to solve the Sudoku game, together with a comparative analysis between the results obtained by an exact algorithm and a heuristic implemented through graph modeling in previous versions of this study. The application of the mentioned methods was carried out using the C++ programming language. The versions developed in this project address the resolution of the generic  $n \times n$  Sudoku. It is worth mentioning that Sudoku represents an NP-complete problem, characterized as a logical reasoning challenge that is intrinsically linked to the concept of Latin square.

**Key-words:** genetic algorithm. graph coloring. sudoku.

## Referências

**Figura 1.** [http://jogadamaais.blogspot.com.br/2013\\_11\\_01\\_archive.html](http://jogadamaais.blogspot.com.br/2013_11_01_archive.html). Acessado em: 2023- 10-17.

**Figura 2.** <http://vigusmao.github.io/manuscripts/sudoku.pdf>. Acessado em: 2023-10-17.

[1] Rosa, J. L. G. **Algoritmos Avançados**. <http://wiki.icmc.usp.br/images/6/62/SCC210Cap7.pdf>. Acessado em 25 de outubro de 2023.

[2] Takano, K; Freitas, R. de; Sá, V. G. P. de. **O jogo de lógica Sudoku: modelagem teórica, NP-completude, algoritmos e heurísticas**. 2015.

[3] SANTOS, R. P. dos; VASCONCELLOS, L. A. dos S. **A matemática por trás do sudoku**. C.Q.D.– Revista Eletrônica Paulista de Matemática, Bauru, v. 12, p. 26-46, jul. 2018.

- [4] Skiena, S. S. (2008). **Algorithm Design Manual**. Springer Publishing.
- [5] Norvig, P. (2006). **Solving every sudoku puzzle**. <http://norvig.com/sudoku.html>. Acessado em 09 de novembro de 2023.
- [6] Ayala, H. V. H; Coelho, L. S. **Otimização por enxame de partículas de controle multivariável em uma aplicação de robótica de manipuladores**. 2007.
- [7] Souza, S. S. F. de; R. R. **Metaheurísticas simulated annealing e busca tabu aplicadas na resolução do quebra-cabeça sudoku**. 2013.
- AKEMI, P. (2018). **Introdução a algoritmos genéticos**. <https://www.ime.usp.br/gold/-cursos/2009/mac5758/PatriciaGenetico.pdf>. Acessado em 01 de dezembro de 2023.
- McDiarmid, C. (1979). **Determining the chromatic number of a graph**. **SIAM Journal on Computing**, 8(1):1–14.

## Anexos

### Anexo 1 - Algoritmo genético

```
1 void SudokuBoard::genetic_algorithm(double mutation_rate, int
   max_generations) {
2   srand(static_cast<unsigned>(time(nullptr)));
3
4   // população inicial gerado a partir dos número presentes no
   tabuleiro
5   std::vector<SudokuBoard> population(POPULATION_SIZE);
6   population[0] = *this;
7   for (int i = 1; i < POPULATION_SIZE; ++i) {
8     population[i] = generate();
9   }
10
11   // quanto menor o fitness, melhor
12   auto compare_fitness = [](const SudokuBoard &a, const SudokuBoard
   &b) {
13     return a.fitness() <= b.fitness();
14   };
15
16   for (int generation = 0; generation < max_generations; ++
   generation) {
17     std::sort(population.begin(), population.end(), compare_fitness
   );
18
19     // caso já exista um tabuleiro perfeito na população, retorná-
   lo
20     if (population[0].fitness() == 0) {
21       std::cout << "Solution found in generation " << generation +
   1
22                 << std::endl;
23       this->nodes = population[0].nodes;
24       return;
25     }
26
27     // gera a nova população através do crossover e da mutação
28     std::vector<SudokuBoard> next_generation;
29     for (int i = 0; i < POPULATION_SIZE / 2; ++i) {
30       int parent1_index = rand() % (POPULATION_SIZE / 2);
31       int parent2_index = rand() % (POPULATION_SIZE / 2);
32       auto children =
33         crossover(population[parent1_index], population[
   parent2_index]);
34       mutate(children.first, mutation_rate);
35       mutate(children.second, mutation_rate);
36       next_generation.push_back(children.first);
37       next_generation.push_back(children.second);
38     }
39
40     population = next_generation;
41   }
42
43   // ninguém conseguiu revolver, retorna o que chegou mais próximo
44   std::sort(population.begin(), population.end(), compare_fitness);
```

```

45
46     this->nodes = population[0].nodes;
47 };

```

## Anexo 2 - Função de aptidão

```

1  int SudokuBoard::fitness() const {
2      auto size = rank * rank;
3      int conflicts = 0;
4      for (int row = 0; row < size; ++row) {
5          for (int col = 0; col < size; ++col) {
6              int num = nodes[row * size + col];
7              if (num != 0) {
8                  for (int i = 0; i < size; ++i) {
9                      if (num == nodes[row * size + i]) {
10                         // conflict in row
11                         conflicts += 1;
12                     } else if (num == nodes[i * size + col]) {
13                         // conflict in col
14                         conflicts += 1;
15                     } else {
16                         auto checkRow = (row / rank) * rank + (i / rank);
17                         auto checkCol = (col / rank) * rank + (i % rank);
18                         if (checkRow == row && checkCol == col) {
19                             continue;
20                         } else if (num == nodes[checkRow * size + checkCol]) {
21                             // conflict in block
22                             conflicts += 1;
23                             conflicts += 1;
24                         }
25                     }
26                 }
27             }
28         }
29     }
30     return conflicts;
31 };

```

## Anexo 3 - Função de mutação

```

1  void SudokuBoard::mutate(SudokuBoard &individual, double
   mutation_rate) const {
2      auto size = rank * rank;
3      for (int i = 0; i < size; ++i) {
4          for (int j = 0; j < size; ++j) {
5              auto index = i * size + j;
6              if (nodes[index] == 0 && (rand() % 100) < mutation_rate *
   100) {
7                  individual.nodes[index] = rand() % size + 1;
8              }
9          }
10     }
11 };

```

#### Anexo 4 - Função de cruzamento

```
1 std::pair<SudokuBoard, SudokuBoard>
2 SudokuBoard::crossover(const SudokuBoard &first, const SudokuBoard
   &second) {
3     auto size = first.rank * first.rank;
4     SudokuBoard first_child;
5     SudokuBoard second_child;
6     first_child.nodes = second_child.nodes = std::vector<cell_type>(
       size * size);
7     first_child.rank = second_child.rank = first.rank;
8     int crossoverPoint = rand() % (size - 1) + 1;
9
10    for (int i = 0; i < size; ++i) {
11        for (int j = 0; j < size; ++j) {
12            auto index = i * size + j;
13            if (j < crossoverPoint) {
14                first_child.nodes[index] = first.nodes[index];
15                second_child.nodes[index] = second.nodes[index];
16            } else {
17                first_child.nodes[index] = second.nodes[index];
18                second_child.nodes[index] = first.nodes[index];
19            }
20        }
21    }
22
23    return std::make_pair(first_child, second_child);
24 };
```