Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Ciências da Computação
INE5416 - Paradigmas de Programação
Professor: Maicon Rafael Zatelli
Alunos: Maykon Marcos Junior - 22102199 e Tiago Faustino de Siqueira - 22102193

Trabalho 1
Tabuleiro de Vergleichssudoku em Haskell

Florianópolis, 25 de Outubro de 2023

1. Introdução

O Vergleichssudoku é uma variação desafiadora do jogo clássico Sudoku. Nesse jogo, os jogadores devem preencher um tabuleiro com números de forma que sejam respeitadas as relações de comparação entre as células vizinhas. As relações de comparação incluem símbolos ">" (maior) e "<" (menor), adicionando uma camada de complexidade ao desafio. O objetivo deste trabalho é desenvolver um resolvedor de Vergleichssudoku eficiente em Haskell, aplicando conceitos de programação funcional para encontrar uma solução.

2. Estratégia utilizada pelo grupo

Nesta seção, vamos detalhar a estratégia adotada pelo grupo para resolver o problema. Isso incluirá a modelagem do tabuleiro, os algoritmos utilizados e as otimizações empregadas.

2. 1 Compreensão do Jogo

Inicialmente, dedicamos tempo para compreender as regras e a mecânica do Vergleichssudoku. Analisamos como as relações de comparação entre células impactam na resolução para podermos começar a pensar no algoritmo.

2 . 2 Desenvolvimento em Linguagem Familiar

Optamos por desenvolver um algoritmo em Python, uma linguagem com a qual a equipe estava familiarizada. Isso nos permitiu modelar e testar a lógica de resolução de forma mais ágil antes de transferir a solução para Haskell.

2. 3 Tradução para Haskell

Após validar o algoritmo em Python, realizamos a tradução do código para Haskell, mantendo uma abordagem funcional, conforme demandado pela disciplina.

3. Alguns detalhes importantes sobre o algoritmo

3. 1 Tamanho dos tabuleiros

O algoritmo final é capaz de solucionar 3 tamanhos de tabuleiro diferentes:

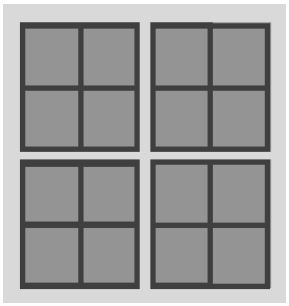

Figura 1 - Tabuleiro 4x4

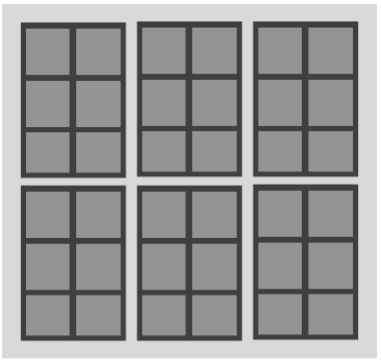

Figura 2 - Tabuleiro 6x6

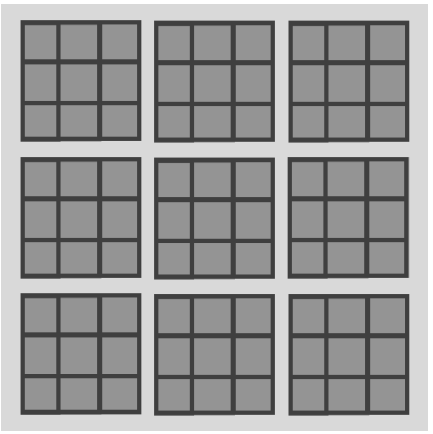

Figura 3 - Tabuleiro 9x9

No algoritmo, o tamanho é descrito com a variável “SIZE”

3. 2 Coordenadas no tabuleiro

Todas as células no algoritmo podem ser acessadas seguindo essa lógica:

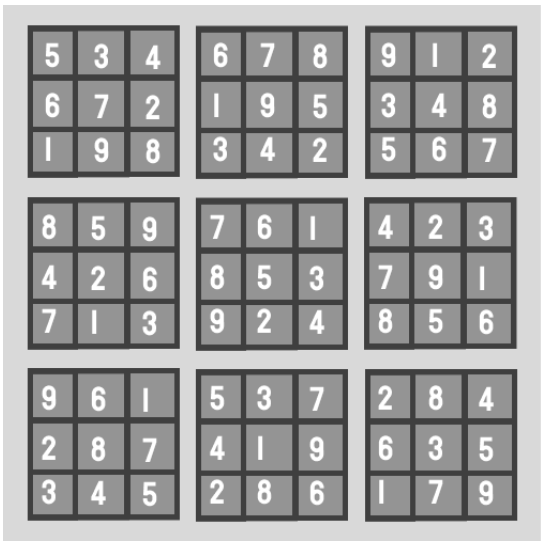

Figura 4 - Tabuleiro 9x9 preenchido

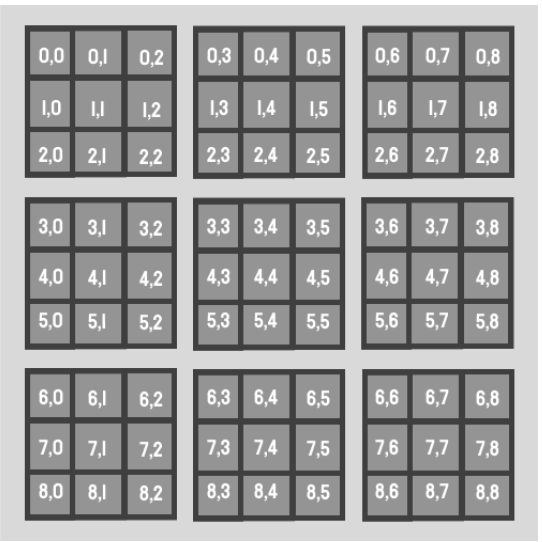

Figura 5 - Tabuleiro 9x9 mapeado

3. 3 Formatação das células

Cada célula é uma quíntupla, contendo as informações sobre seus 4 vizinhos:

" {num} {left} {up} {right} {down} "

Figura X - Formatação das células

Exemplo 1) Ilustrando visualmente a configuração da célula destacada:

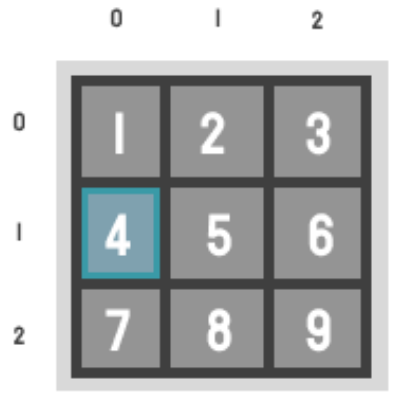

Figura 6 - Célula destacada

As relações da célula com os 4 vizinhos é dada da seguinte maneira:

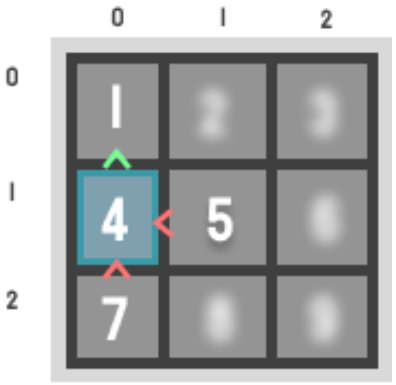

Figura 7 - Relação da célula com seus vizinhos

Logo, a formatação final da célula fica assim:

" {4} {'/'} {'>'} {'<'} {'<'} "

Figura X - Formato final da célula

Por fim, acessamos as informações daquela célula da seguinte maneira:

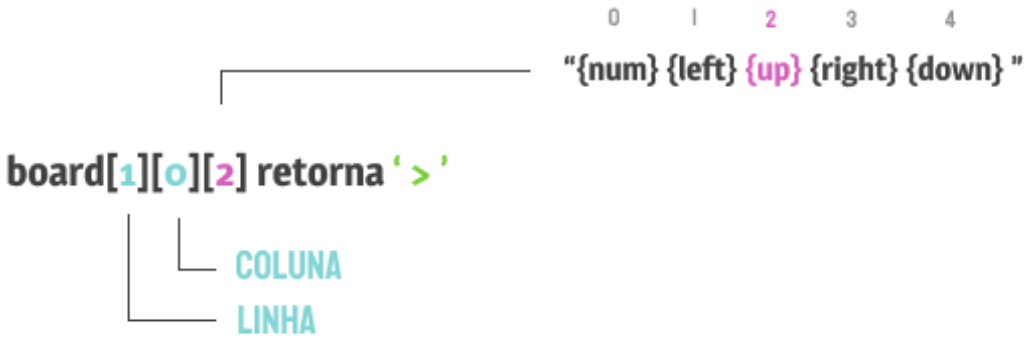

Figura X - Descrição de como fazer acesso dos dados da célula

3. 4 Box de cada tabuleiro

Os tabuleiros são subdivididos em regiões. De acordo com os 3 tamanhos de tabuleiro disponíveis, temos as dimensões das "Box" (regiões) de cada um deles:

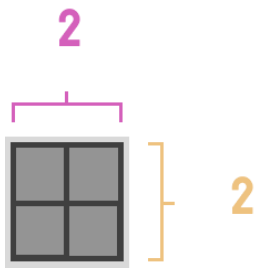

Figura 8 - Box 2x2 (Tabuleiro 4x4)

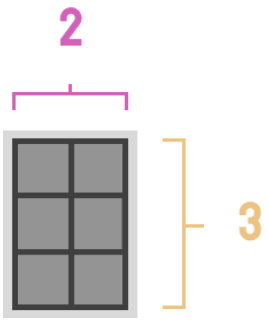

Figura 9 - Box 3x2 (Tabuleiro 6x6)

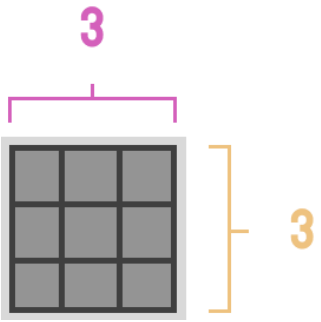

Figura 10 - Box 3x3 (Tabuleiro 9x9)

■ LIMITE DA COLUNA
■ LIMITE DA LINHA

4. Algoritmo em Python (Passo a Passo)

Aqui está um resumo do algoritmo em Python utilizado como referência para a implementação em Haskell:

4. 1 Primeiro passo: Encontrar célula vazia (*find_empty*)

Chama a função *find_empty* para encontrar a próxima célula vazia no tabuleiro (*board*). Se não encontrar nenhuma célula vazia, isso significa que o tabuleiro está resolvido e a função retorna *True*.

4. 1. 1 Função "find_empty"

Descrição: Essa função tem como objetivo encontrar a próxima célula vazia (com valor igual a 0) em um tabuleiro de Sudoku representado por uma matriz. A função aceita três parâmetros: *board* (a matriz do tabuleiro), *row* (a linha atual, padrão é 0) e *col* (a coluna atual, padrão é 0).

Exemplo de uso:

Ao chamar a função *find_empty*(board, row=0, col=0), teremos esse fluxo de execução:

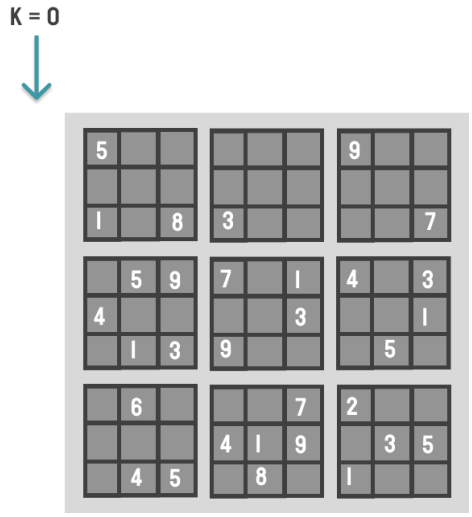

Figura 11 - começo da execução da função *find_empty()*

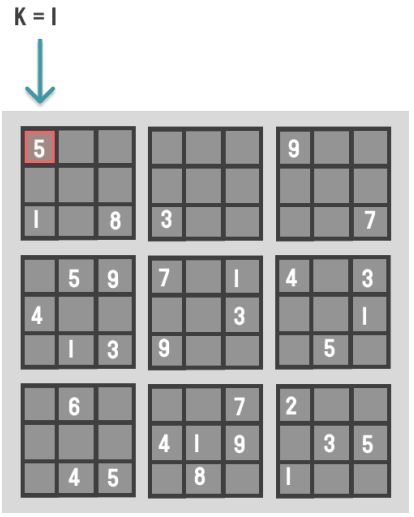

Figura 12 - primeira iteração da função *find_empty()*

Se a célula não estiver vazia, avança para a próxima célula (próxima coluna). Se a coluna atual atingir o limite da linha [3.4], reinicia a coluna e avança para a próxima linha. Em seguida, chama recursivamente a função find_empty para a próxima célula.

Verificação de célula vazia:Verifique se a célula atual está vazia (seu valor é 0). Se for, retorna a posição da célula (row, col).

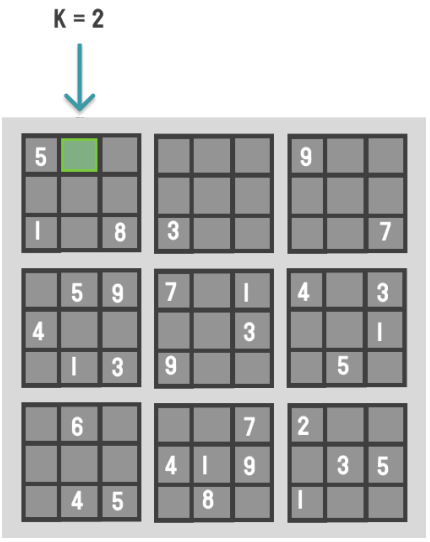

Figura 13 - função *find_empty()* encontrou célula vazia

Após achar a primeira posição vazia disponível, retorna as coordenadas [3.2] da mesma.

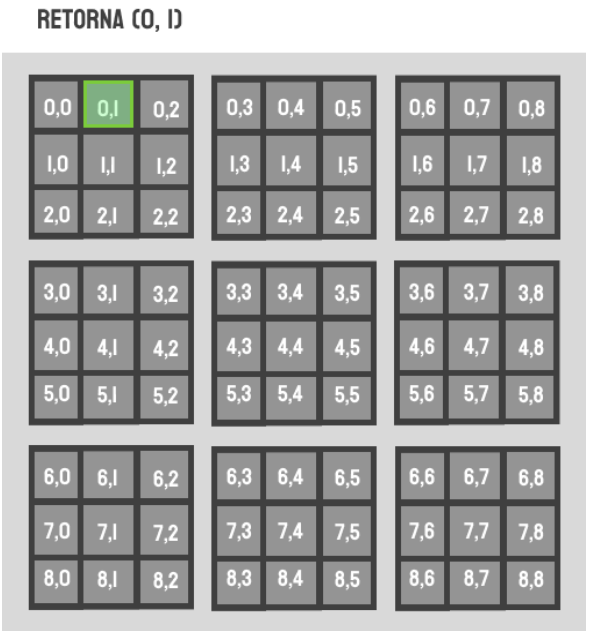

Figura 14 - função *find_empty()* retorna as coordenadas da célula

4. 2 Segundo passo: Recursão e tentativa de inserção:

Para a célula vazia encontrada (*row, col*), itera sobre os números de 1 ao tamanho do tabuleiro e tenta inserir cada número (*num*) na célula, tendo como requisito ele ser válido de acordo com as regras do Sudoku e as regras comparativas.

Chama a função *is_valid* para verificar se a inserção do número *num* na célula é válida.

4. 2. 1 Função “is_valid”

Descrição: Essa função tem como objetivo verificar se é válido inserir um determinado número (*num*) em uma determinada posição (r*ow, col*) no tabuleiro de Sudoku. A função aceita quatro parâmetros: board (a matriz do tabuleiro), num (o número a ser verificado), row (a linha da posição) e col (a coluna da posição).

Passos da função:

Primeiro passo: Determinação do tamanho do tabuleiro
Obtém o tamanho do tabuleiro, que será usado para calcular as regras de comparação com as células vizinhas.

Segundo passo: Verificação da linha
Verifica se o número num já existe na linha row percorrendo cada coluna. Se existir, retorna False.

Terceiro passo: Verificação da coluna
Verifica se o número num já existe na coluna col percorrendo cada linha. Se existir, retorna False.

Quarto passo: Verifica se o número num já existe na box.
Utiliza cálculos para determinar a caixa atual e percorre suas células. Se existir, retorna False.

Quinto passo: Verifica as regras de comparação
Verifica as regras de comparação para o número *num* em relação às células vizinhas (esquerda, acima, direita, abaixo). Se violar alguma regra de comparação (maior ou menor), retorna False.

Exemplo de execução: Exemplo trivial onde só uma célula se encontra vazia:

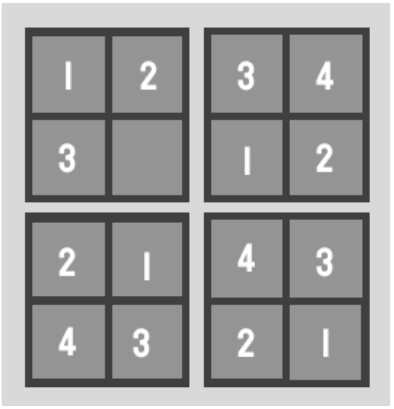

Figura 15 - exemplo ilustrando tabuleiro com apenas 1 célula vazia

Função *is_valid()* é chamada para verificar se o número **4** pode ser inserido naquela célula:

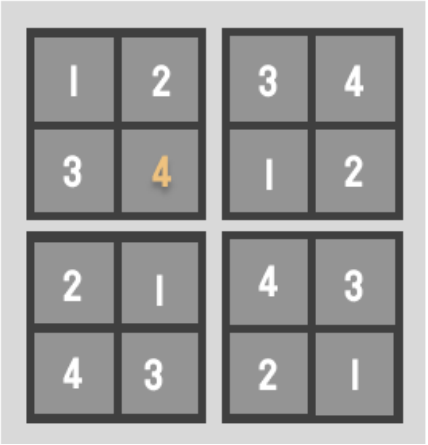

Figura 16 - Tentando inserir o número “4” na célula

Agora, varre a região para ver se o "**4**" já faz parte dela. Faz o mesmo para a coluna e linha.

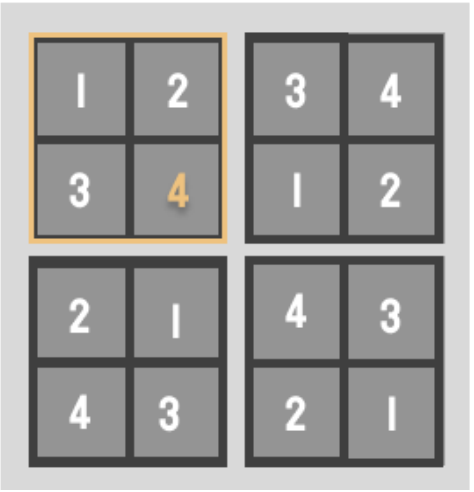

Figura 17 - Verifica região, linha e coluna associadas

Caso não faça parte, compara com os vizinhos

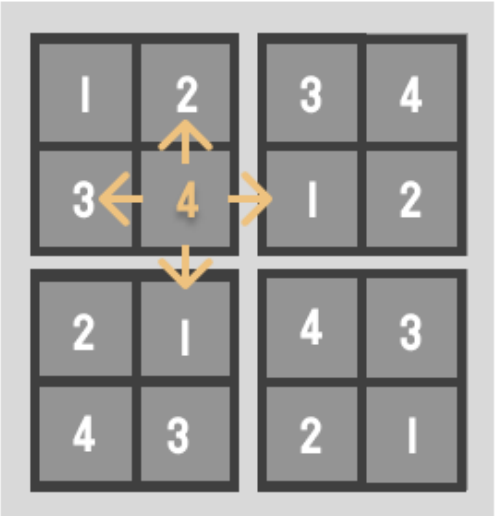

Figura 18 - Comparação com os vizinhos

Se o número num passar por todas as verificações, retorna *True*, indicando que é válido inserir esse número na posição especificada.

4. 3 Terceiro passo: Atualização da célula

Se a inserção do número *num* for válida, atualiza a célula (*board[row][col][0] = num*) e chama recursivamente a função *solve_comparative* para continuar a resolver o restante do tabuleiro.

Se a recursão for bem-sucedida e resolver o tabuleiro, retorna *True* indicando que o tabuleiro foi resolvido com sucesso.

4. 4 Quarto passo: Backtracking

Se a recursão não for bem-sucedida ou não resolver o tabuleiro, reverte a última tentativa de inserção (b*oard[row][col][0] = 0*) e continua o loop para tentar o próximo número.

4. 5 Cenário em que não há solução

Se todos os números foram testados para a célula atual e nenhum levou a uma solução, retorna False, indicando que essa configuração do tabuleiro não leva a uma solução válida.

5. Conversão do código para Haskell

Após a validação do algoritmo em Python, o próximo passo crucial foi a tradução do código para Haskell, uma linguagem de programação funcional que enfatiza a imutabilidade e a expressão clara da lógica de programação. Essa tradução exigiu um cuidadoso ajuste da abordagem para se adequar à natureza funcional de Haskell.

6. Entrada e saída

Aqui vamos esclarecer como os usuários podem fornecer a entrada ao programa e como o resultado é apresentado.

Para executar o programa Haskell a partir do terminal use o seguinte comando:

```
ghc solver.hs -o saida
./saida
```

O programa lerá o conteúdo do arquivo "tabuleiro.txt", resolverá o Vergleichssudoku e apresentará o resultado.

7. Organização do grupo

O grupo realizou uma série de reuniões, tanto online quanto presenciais, para discutir a estratégia de resolução do problema.

A montagem do código em Python foi feita de forma colaborativa usando a ferramenta Live Share do VS Code, que permite a edição simultânea do código por várias pessoas em tempo real.

O histórico de commits e o código-fonte do projeto estão disponíveis no repositório compartilhado no GitHub: https://github.com/dexeme/vergleichssudoku-puzzle-solver.

8. Dificuldades encontradas

Nesta seção, abordaremos as principais dificuldades encontradas durante a implementação do resolvedor do puzzle e as soluções que o grupo adotou para superar esses desafios.

Uma das principais dificuldades foi a tentativa inicial de implementar uma solução alternativa que não se baseasse em backtracking, que é uma abordagem mais direta. Contudo, após investir um tempo considerável nessa abordagem, percebemos que a complexidade crescente da lógica funcional estava tornando o problema mais desafiador do que o necessário. Decidimos, então, optar por uma solução mais direta e funcional, que se alinhasse melhor com os conceitos da linguagem Haskell.

A experiência de pensar em algoritmos de forma funcional representou um desafio considerável, mas também se revelou uma oportunidade valiosa de aprendizado. Mesmo com as dificuldades encontradas, todos os membros da equipe apreciaram a experiência e aprimoraram suas habilidades de programação funcional.