Trabalho I - Suguru Solver em Haskell

João Vitor Maia Neves Cordeiro (19100532)
Bernardo Schmidt Farias (19100519)

23 de março de 2021

1 Introdução

Dentre os dois problemas disponíveis o grupo optou por resolver o jogo Suguru, também conhecido como Tectonic Puzzle. O jogo consiste em um puzzle semelhante ao Sudoku, porém com regras ligeiramente diferentes. Dado um tabuleiro de NxN casas, com áreas internas definidas, o jogador precisa organizar números da seguinte forma:

- Uma área de tamanho M deve ser preenchida com números de 1 até M, que apareçam exatamente uma vez dentro da área.
- Não devem existir números iguais em casas ortogonais ou diagonais.
- Todo o tabuleiro deve ser preenchido.

Um exemplo de tabuleiro 8x8 e sua versão resolvida podem ser vistos abaixo:

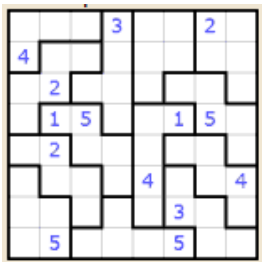

Figura 1: Tabuleiro 8x8 de Suguru

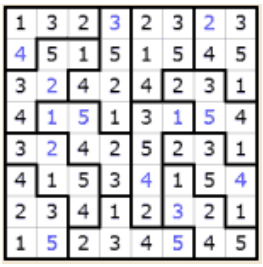

Figura 2: Tabuleiro anterior resolvido

2 Metodologia

Todo o trabalho foi desenvolvido em dupla, utilizando como ferramentas o Discord (para comunicação) e a extensão VSCode Live Share que permite iniciar sessões remotas colaborativas de edição de código.

Para resolver o problema passamos por várias etapas, a primeira sendo a compreensão do puzzle e algumas "partidas"manuais dele, assim poderiamos entender melhor o funcionamento e pensar em estratégias para resolver. Depois tentamos uma primeira solução em uma linguagem de programação que o grupo possuia mais costume, a linguagem escolhida foi Python e tivemos sucesso em resolver dessa forma, o código fonte se encontra no mesmo repositório do projeto. Abaixo uma *screenshot* do script Python resolvendo o exemplo dado acima em poucos milissegundos, ele utiliza *backtracking* e já possui algumas otimizações.

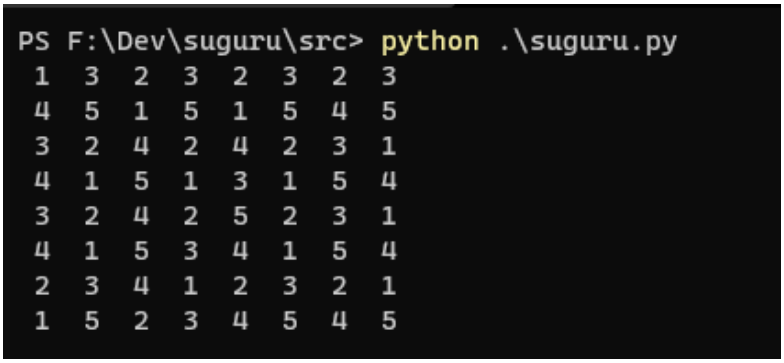

Figura 3: Tabuleiro anterior resolvido

Por último, com uma boa ideia de como seria o algoritmo para resolver o

puzzle, partimos para a solução em Haskell que será detalhada a seguir.

3 Solução em Haskell

3.1 Modelagem dos tipos

A primeira barreira encontrada, tanto na solução em Python quanto em Haskell, foi a modelagem do tabuleiro, pois diferentemente do tabuleiro de sudoku que possui áreas fixas, o suguru pode assumir várias formas. Logo, além de receber como input o valor inicial de cada casa do tabuleiro também precisariamos de alguma forma receber o formato das áreas.

Nas duas soluções, optamos por receber junto do valor de cada casa um identificador dizendo a qual área do tabuleiro essa casa pertence, os identificadores iriam de 0 até o número de áreas totais do tabuleiro, e a ordem escolhida como convenção foi de numerar a partir do *top-left*.

Em Haskell, criamos o tipo `Point`, como uma tupla contendo `Value` e `Sector`, e esses tipos Value e Sector são apenas *aliases* para `Int`. Dessa forma, o nosso tabuleiro, nomeado como o tipo `Board` é uma matriz de `Point`. Abaixo está o código com nessas declarações de tipo exatas.

```
1  -- O identificador de setor utilizado
2  type Sector = Int
3
4  -- Um valor de 1 a N, onde N e o tamanho do setor no qual o
       ponto esta
5  type Value = Int
6
7  -- Um par de coordenadas X, Y
8  type Location = (Int, Int)
9
10 -- Um ponto no tabuleiro, no formato (Value, Sector)
11 type Point = (Value, Sector)
12
13 -- Um tabuleiro de Suguru
14 type Board = [[Point]]
```

Em ambos os códigos, a entrada é dada diretamente no código por uma matriz de tuplas (Haskell) ou uma matriz de `Point` (Python). A saída do programa é mostrada no terminal após a execução.

3.2 Funções auxiliares

Tendo os tipos já modelados, escrevos algumas funções auxiliares para esses tipos, por exemplo funções como `getValue` e `getSector` que recebem um ponto e retornam respectivamente seu `Value` e `Sector`. Existem outras funções como essa que servem apenas para buscar e acessar valores dentro das nossos tipos customizados, todas elas funcionam de forma parecida e trivial, portanto vamos demonstrar abaixo apenas as duas citadas acima, as outras podem ser vistas no código fonte.

```
1 -- Retorna o setor de um ponto
2 getSector :: Point -> Sector
3 getSector (_, sector) = sector
4
5 -- Retorna o valor de um ponto
6 getValue :: Point -> Value
7 getValue (value, _) = value
```

Com essas funções auxiliares já podemos começar a implementar a primeira parte da nossa lógica. Dividimos a lógica basicamente em duas partes: a primeira parte checa se é possível adicionar um valor `V` em um ponto de coordenadas `(X, Y)` dentro da matriz do tabuleiro, enquanto a segunda parte utiliza *backtracking* para verificar cada posição e seus pontos possíveis, gerando recursivamente o resultado final (ou retornando a matriz inicial caso não exista uma solução possível).

3.3 Checando viabilidade do valor

Para checarmos se um valor por ser adicionado a um determinando ponto sem quebrar as regras precisamos realizar duas verificações iniciais: verificar se no setor interno existe um outro ponto já com o mesmo valor e verificar se existem pontos vizinhos com o valor.

Para a primeira verificação, utilizamos duas funções, sendo que uma filtra a matriz do tabuleiro em busca de todos os pontos do setor `N`, enquanto a outra filtra a lista de pontos do setor `N` em busca de valores `X`, retornando `True` caso o valor exista em pelo menos um ponto.

```
1 -- Retorna todos os pontos de um setor indicado pelo indice n
2 -- Primeiro acessando a linha, depois acessando a coluna.
3 getSectorPoints :: Board -> Sector -> [Point]
4 getSectorPoints board n = [(board !! row) !! col | row <- [0
    .. size], col <- [0 .. size], getSector ((board !! row) !!
     col) == n]
```

```
5
6  -- Checa a existencia de um ponto com o valor [value] no [
       sector]
7  sectorHasValue :: Board -> Sector -> Value -> Bool
8  sectorHasValue board sector value =
9      not ( null ( [point | point <- getSectorPoints board
       sector, getValue point == value] ) )
```

Para a segunda, utilizamos uma checagem baseada em offsets, então para um ponto na localização `(X, Y)` nós checamos os 8 pontos vizinhos baseado nas coordenadas deles, utilizando uma comparação `OR` para garantir que caso algum valor igual seja encontrado a função retorne `True`.

```
1  -- Checa os valores adjacentes de determinada localizacao
2  -- Retorna True caso ache um ponto igual ao valor fornecido,
       False caso o contrario
3  checkAdjacents :: Board -> Location -> Value -> Bool
4  checkAdjacents board location v =
5    checkAdjacent board (getX location,     getY location - 1 )
       v ||
6    checkAdjacent board (getX location + 1, getY location - 1 )
       v ||
7    checkAdjacent board (getX location + 1, getY location     )
       v ||
8    checkAdjacent board (getX location + 1, getY location + 1 )
       v ||
9    checkAdjacent board (getX location,     getY location + 1 )
       v ||
10   checkAdjacent board (getX location - 1, getY location + 1 )
       v ||
11   checkAdjacent board (getX location - 1, getY location     )
       v ||
12   checkAdjacent board (getX location - 1, getY location - 1 )
       v
13
14 -- Checa um valor adjacente
15 checkAdjacent :: Board -> Location -> Value -> Bool
16 checkAdjacent board target v
17   | (getX target < 0) || (getY target < 0) = False
18   | (getX target >= size) || (getY target >= size) = False
19   | v == getValueFromLocation board target = True
20   | otherwise = False
```

3.4 Backtracking

Com essas duas verificações somos capazes de saber é possível adicionar um valor a uma casa do tabuleiro sem quebrar as regras do tabuleiro já montado, mas ainda precisamos de uma maneira de iterar pelos campos e adicionar esses valores. Essa iteração não pode ser um simples loop pois um valor que inicialmente é válido pode ser mostrar um valor errado numa futura iteração, fazendo com que ele deva ser mudado. Para isso, utilizamos a técnica de *backtracking* que recursivamente avança na solução, testando novos valores e desfazendo seus passos anteriores caso chegue a um "dead end".

Para isso, decidimos como convenção sempre preencher os valores na ordem *top-left*, procurando o próximo campo vazio do tabuleiro, verificando um valor válido para ele e avançando. Caso após um número de iterações perceba-se que não existem mais valores possíveis a serem preenchidos a função retorna para um estado anterior do tabuleiro e tenta novamente incrementando o valor de entrada. Após testar todos os valores possíveis e não encontrar solução sabemos que o tabuleiro é insolúvel.

Quando implementamos essa solução em Python, utilizando programação imperativa, tivemos sucesso com um código relativamente simples, nossa função `solve()` possuia menos de 20 linhas e pode ser vista abaixo

```
1  def solve(inputBoard: Board):
2      row, column = nextEmpty(board)
3
4      if row is None:
5          return True
6
7      sector_points = list(filter(lambda p: p.sector == board.
       points[row * board.size + column].sector, board.points))
8
9      for i in range(1, len(sector_points) + 1):
10         if check(board, row, column, i):
11             board.points[row * board.size + column].value = i
12
13             if(solve(board)):
14                 return True
15
16         board.points[row * board.size + column].value = 0
```

Entretanto, ao tentar passar a solução para Haskell, obviamente com uma abordagem ligeiramente diferente por se tratar de uma linguagem funcional, não conseguimos uma solução correta. O maior problema encontrado durante

as tentativas sempre era "desfazer"os passos realizados ao se chegar em um "dead end". Quando estavamos no Python, podiamos simplesmente alterar os pontos da variável que guardava o tabuleiro para retornar a um estado anterior.

Em Haskell, por trabalharmos com a matriz de pontos em uma constante que era copiada a cada iteração, não poderiamos utilizar essa solução. A solução final que conseguimos em Haskell itera por boa parte da matriz, conseguindo um resultado muito próximo do correto, mas ainda assim errado. Sabemos que o erro está provavelmente na função que `solve` que implementa o *backtracking*, entretanto mesmo após bastante tempo desprendido ainda não conseguimos consertar.

4 Considerações finais

Mesmo sabendo que o projeto ainda apresenta problemas quanto a implementação, acreditamos que a lógica utilizada esteja correta, e possivelmente com um pouco mais de familiariade com a linguagem Haskell conseguiriamos chegar numa solução satisfatória. Infelizmente o tempo de entrega (mesmo com a extensão dada pelo professor) está a poucas horas no futuro, por isso decidimos por finalizar o relatório com a solução incompleta em Haskell, explicando os problemas no caminho e demonstrando também a solução em Python.

O código, assim como os arquivos fonte desse mesmo relatório se encontram em um repositório Git, na url `https://github.com/Leviosar/suguru`.