Aluno: Gilson Trombetta Magro
Matrícula: 18202192
Disciplina: INE5416 - Paradigmas de Programação

RELATÓRIO: TRABALHO II

Novembro de 2020

1 ANÁLISE DO PROBLEMA

O problema escolhido para o desenvolvimento deste trabalho foi o Straights (ou Str8s). Este puzzle é um jogo similar ao Sudoku, e é composto de um tabuleiro quadrado (usualmente de tamanho 6 x 6 ou 9 x 9) que possui casas brancas e negras. O tabuleiro deve ser preenchido de acordo com as seguintes regras:

- Todas as casas brancas devem ser preenchidas com números de 1 a N (considerando um tabuleiro NxN), e nenhuma linha/coluna pode conter números repetidos;
- O tabuleiro pode ser separado em "regiões": uma região é composta apenas por casas brancas conectadas vertical ou horizontalmente e é delimitada por casas negras e/ou pelas bordas do tabuleiro;
- Toda região (vertical ou horizontal) deve conter um conjunto de números consecutivos, porém não necessariamente em ordem;
- Um número em uma casa negra serve para remover a possibilidade de uso daquele número em sua respectiva linha/coluna;
- Casas negras não fazem parte de nenhuma região.

2 SOLUÇÃO PROPOSTA

2.1 ORGANIZAÇÃO

No trabalho 1 (implementado em Haskell), eu decidi dividir o código em quatro arquivos diferentes, cada um com sua funcionalidade particular. Neste trabalho, porém, a linguagem utilizada foi o LISP. A implementação com esta linguagem rendeu um número

muito menor de linhas de código; portanto, achei mais apropriado manter o código todo em um único arquivo.

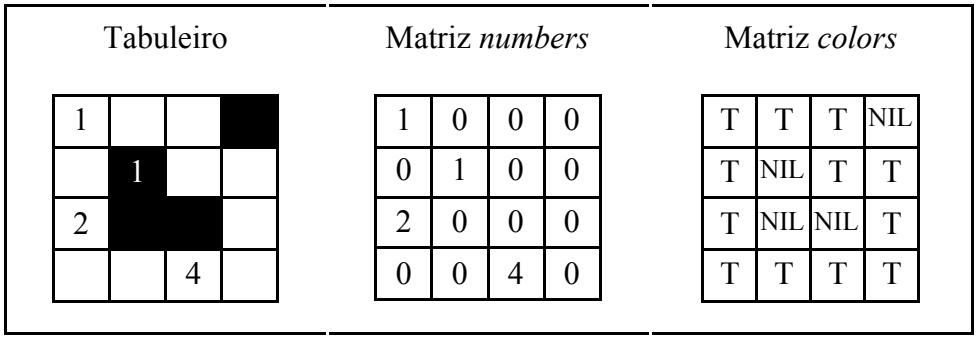
O código é composto de quatro funções: *parse-input*, *solve*, *possible* e *parse-output*. A funcionalidade de cada uma delas é listada a seguir:

- *parse-input*: lê a entrada, monta o tabuleiro e chama o resolvedor;
- *solve:* resolve o tabuleiro de entrada;
- *possible:* função auxiliar que recebe o tabuleiro, uma posição (x, y) e um valor *value* e decide se *value* pode ou não ser colocado na posição indicada;
- *parse-output*: recebe um tabuleiro e imprime-o na saída.

2.2 REPRESENTAÇÃO

O tabuleiro é representado por duas matrizes de tamanho NxN: *numbers* e *colors*. A primeira é uma matriz de inteiros, e representa os valores de cada casa do tabuleiro; casas vazias são preenchidas com o número zero. A segunda é uma matriz de booleanos, e representa a cor das casas do tabuleiro, onde T representa uma casa branca, e NIL representa uma casa negra.

Figura 1 - Representação do tabuleiro



2.3 SOLUÇÃO

A solução para o problema baseia-se na técnica de *backtracking*, conforme sugerido no enunciado. A função *solve* é a que de fato soluciona o tabuleiro. Seu código é apresentado abaixo:

Função *solve*

```
1  (defun solve (numbers colors)
2    (if
3      (block solve-block
4        (dotimes (y SIZE)
5          (dotimes (x SIZE)
6            (if (and (= (aref numbers y x) 0) (aref colors y x))
7              (progn
8                (dotimes (value SIZE)
9                  (if (possible numbers colors y x (+ value 1))
10                   (progn
11                     (setf (aref numbers y x) (+ value 1))
12                     (solve numbers colors)
13                     (setf (aref numbers y x) 0))))
14               (return-from solve-block NIL)))))
15       T
16     )
17     (parse-output numbers colors)
18   )
19 )
```

O funcionamento da função *solve* pode ser resumido aos seguintes passos, apresentados abaixo:

1. Itera-se sobre todas as posições (y, x) do tabuleiro, até que se encontre uma posição branca e vazia.
2. Então, para cada número inteiro *value* $\in$ [1, N], sendo N o tamanho do lado do tabuleiro, o algoritmo testa se é possível colocar *value* na posição (y, x) sem quebrar nenhuma regra do *puzzle*. Note que essa verificação acerca de um valor poder ou não ocupar uma casa é feita pela função *possible*.
3. Se for possível, o valor *value* é escrito na posição (y, x) e a função *solve* é chamada novamente, agora com o tabuleiro atualizado. Quando *solve* retorna, escreve-se zero na posição (y, x) e parte-se para o próximo valor. Isso caracteriza o *backtracking*.
4. Se nenhuma posição vazia for encontrada, então o tabuleiro foi totalmente preenchido e encontrou-se uma solução válida para o problema. Portanto, a configuração atual do tabuleiro é impressa na tela através da função *parse-output*.

Ao término da execução, a função *solve* terá imprimido todas as soluções possíveis para o tabuleiro de entrada. Na eventualidade de que a configuração de entrada não possua solução, nenhum tabuleiro será impresso na saída.

A estratégia utilizada para implementar a função *possible* é bastante similar à utilizada pela função *getAvailable* no trabalho 1, com a diferença de que no primeiro trabalho calculava-se todos os valores possíveis para uma posição (y, x). Neste trabalho, porém, optou-se pela seguinte abordagem: dado um valor *value* e uma posição (y, x), deseja-se saber se *value* pode ser escrito na posição (y, x) sem quebrar nenhuma regra do jogo.

Nesse sentido, identificar se um valor já aparece na linha y ou na coluna x é trivial: basta percorrer a linha/coluna e comparar os valores. Ademais, deve-se identificar os limites da região vertical (de casas brancas conectadas) à qual a casa (y, x) pertence. Então, calcula-se o tamanho da região encontrada. Em seguida, o algoritmo percorre a região de modo a aferir se o valor *value* extrapola, em algum momento, a condição de consecutividade. Naturalmente, repete-se o processo para a região horizontal.

Apresenta-se abaixo a seção da função *possible* que implementa o processo descrito acima, relativo à região vertical de (y, x):

Trecho da função *possible*

```
…  (defun possible (numbers colors y x value)
…    (block outer-block
…      ...
1      (setf north y)
2      (setf south y)
3      (loop while (and (> north 0) (aref colors (- north 1) x))
4            do (setf north (- north 1)))
5      (loop while (and (< south (- SIZE 1)) (aref colors (+ south 1) x))
6            do (setf south (+ south 1)))
7
8      (setf space (+ (- south north) 1))
9      (loop for row from north to south
10           do (if (and (/= (aref numbers row x) 0) (>= (abs (- (aref numbers row x) value)) space))
11           (return-from outer-block NIL)))
…       ...
```

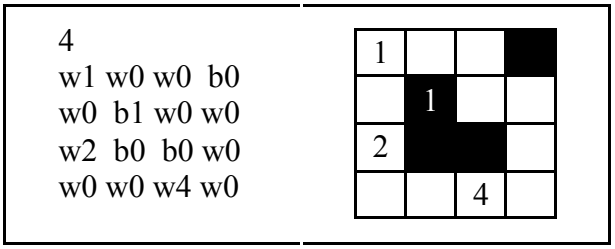
3 ENTRADA E SAÍDA

3.1 ENTRADA

A entrada para o programa se dá através de um arquivo de texto ou do próprio terminal. O formato de entrada consiste de um inteiro N >= 1 na primeira linha, que corresponde ao tamanho do lado do tabuleiro NxN. As N linhas subsequentes deverão conter,

cada uma, N números (∈ [0, N]) separados por espaços. Cada um desses números deve ser precedido de um “w”, caso corresponda a uma casa branca, ou “b”, caso corresponda a uma casa negra. O número zero corresponde a uma casa vazia. A figura 2 ilustra a entrada de um tabuleiro 4x4. Exemplos adicionais são fornecidos junto com o código-fonte, na pasta “examples”.

Figura 2 - Exemplo de entrada do programa



3.2 SAÍDA

Como saída, o programa exibirá o tabuleiro inicial e uma lista com todas as soluções possíveis, derivadas do tabuleiro inicial. Se nenhuma solução for impressa na tela, então o tabuleiro inicial não tem solução. A figura 3 ilustra a saída do programa, considerando a entrada apresentada na figura 2. Note que, ao imprimir os tabuleiros, casas negras são representadas por colchetes em volta do número. Novamente utiliza-se o número zero para representar casas vazias.

Figura 3 - Exemplo de saída do programa

```
gilsontm@gilsontm-S550CA:~/Documents/ufsc/ine5416/trabalho2$ clisp main.lisp < examples/005
::: Lendo inteiro (>=1) que representa o tamanho do tabuleiro...
::: Tamanho selecionado: 4x4
::: Lendo o tabuleiro...
::: Tabuleiro de entrada:

 1  0  0 [0]
 0 [1] 0  0
 2 [0][0] 0
 0  0  4  0
::: Solucoes validas:

 1  3  2 [0]
 4 [1] 3  2
 2 [0][0] 3
 3  2  4  1
::: Concluido!
```

4 DIFERENÇAS ENTRE HASKELL E LISP

Considerando o primeiro trabalho, implementado em Haskell, e agora este, implementado em LISP, é possível destacar algumas das principais diferenças entre o uso das duas linguagens.

Apesar de ambas utilizarem o paradigma funcional, a sensação que tive é de que o Haskell tenta ser muito mais complicado que o necessário. Um exemplo disso está nas classes numéricas do Haskell, que são extremamente complicadas e me fizeram perder muito tempo nos exercícios e no primeiro trabalho. Ademais, o conceito de monads também não me pareceu intuitivo, nem amigável a programadores iniciantes na linguagem.

Em contrapartida, o LISP apresenta a facilidade de poder atribuir valores à variáveis, criar laços e executar instruções sequenciais de maneira relativamente simples. Há também a enorme conveniência de poder realizar operações de I/O a qualquer momento dentro de uma função. Contudo, o LISP peca em possuir poucas funções nativas para lidar com strings.

No geral, pode-se afirmar que a sintaxe de ambas as linguagens foge do padrão das linguagens mais comuns, como C/C++, Java e Python; mas esse quesito causa estranheza apenas no início.

Por fim, posso concluir que a minha experiência utilizando o LISP rendeu um processo de desenvolvimento mais suave, se comparado à utilização do Haskell.

5 REFERÊNCIAS

COMPUTERPHILE. **Python Sudoku Solver - Computerphile**. 2020. (10m52s). Disponível em: <https://youtu.be/G_UYXzGuqvM>. Acesso em: 07 out. 2020.

HASKELL WIKI. **Sudoku:** Simple solver. Disponível em: <https://wiki.haskell.org/Sudoku#Simple_solver>. Acesso em: 07 out. 2020.