Grupo:

Gabriel da Silva Cardoso - 20100524

Julio Gonçalves Ramos - 19203165

Links:

Repositório Git com o código: https://github.com/Cardoz-0/Suguru-Solver/tree/main/Lisp

Apresentação: https://www.youtube.com/watch?v=m175trgHkfY

SUGURU SOLVER EM LISP

O problema

Suguru é um quebra cabeça onde o tabuleiro é dividido em grupos ou regiões que devem ser preenchidos com números de acordo com as regras do jogo.

Existem duas regras para como os números podem ser postos no tabuleiro, que é onde a complexidade do puzzle é criada:

1. um número não pode ser posto nas oito casas em torno de um igual.
2. as regiões devem ser completadas obedecendo a regra acima, com números de 1 a N, onde N é o tamanho da região.

Solução

"Parte da dificuldade de realizar um algoritmo para o resolver suguru é que vários caminhos para a solução são sem saída e é preciso checar por esses casos e voltar o quanto for necessário para resolver o problema.

Para isso, é utiliado a estratégia de backtracking, assim, várias soluções são testadas em sequência e caso ela não seja adequada, é descartada. Isso é feito até chegar na solução de fato."

Trecho do relatório do trabalho 1, que ainda se aplica aqui já que é o mesmo puzzle.

O Haskell e o Lisp são linguagens um tanto parecidas, então foi possível pegar os algoritmos desenvolvidos em Haskell e converte-los para esse trabalho. Houveram algumas dificuldades, que serão exploradas a seguir.

Entrada e saída

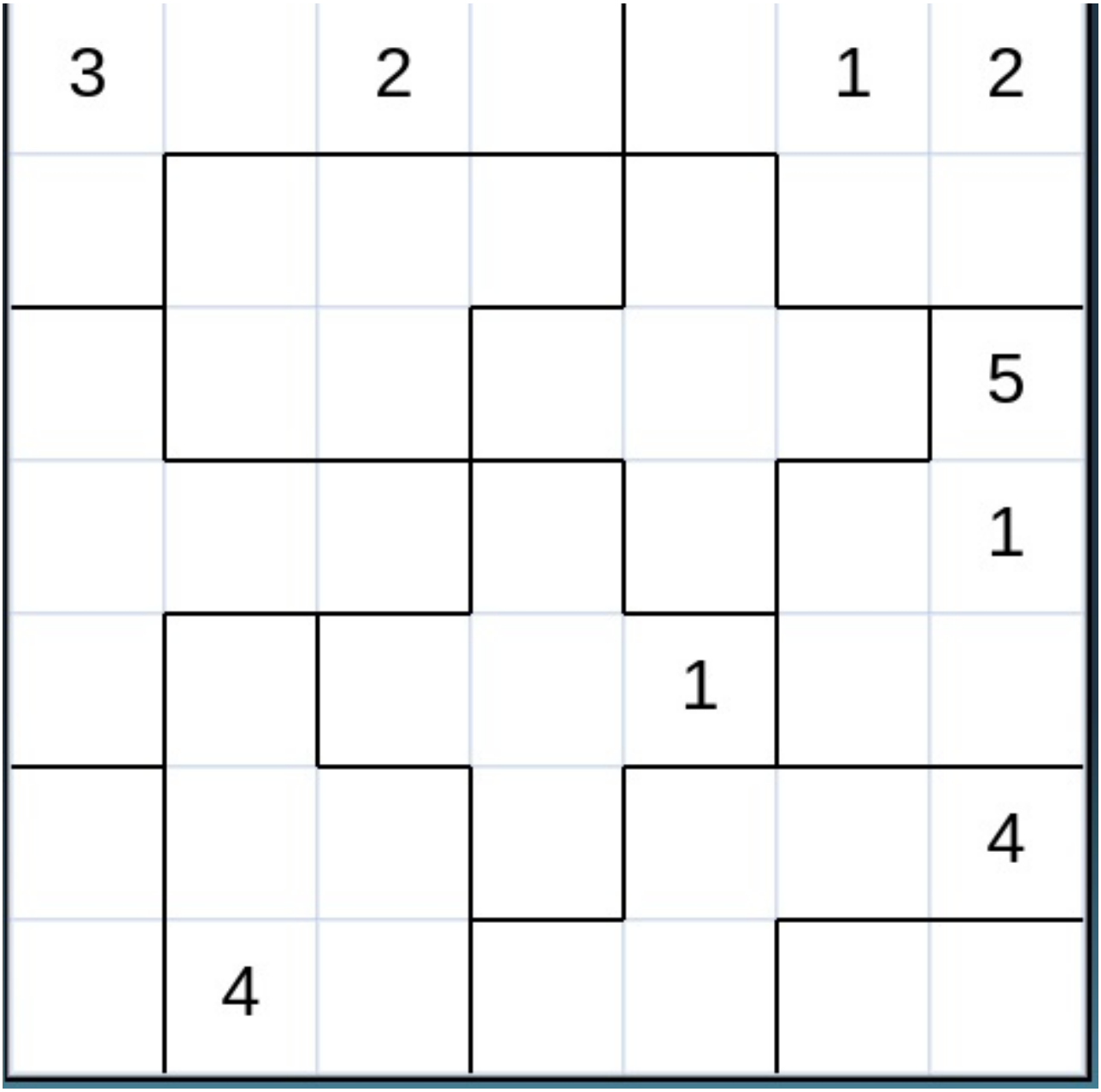
A entrada dos tabuleiros de jogo é feita diretamente no código. O armazenamento dos tabuleiros é feito manualmente pelo usuário, armazenados como matrizes (lista de listas) em variáveis a serem usadas no método chamado pelo main. Espaços vazios são representados pelo número 0.

Também é necessário uma matriz para especificar as regiões do tabuleiro, também inserida pelo usuário. Cada região é representada por um grupo de elementos com o mesmo valor adjacentes um ao outro. Ao contrário do tabuleiro, 0 é utilizado como índice.

As matrizes de tabuleiro e região são agrupadas em uma estrutura "puzzle" por conveniência.

```
(defstruct puzzle
    ; Números iguais representam uma mesma região
    region
    ; Entrada do programa contendo os números presentes no tabuleiro
    ; Espaços vazios são representados por zeros
    tabuleiro
)

(setq puzzle_7
    (make-puzzle
        :region (list '(0 0 0 0 1 1 1)
                      '(0 2 2 2 3 1 1)
                      '(4 2 2 3 3 3 5)
                      '(4 4 4 6 3 5 5)
                      '(4 8 6 6 6 5 5)
                      '(7 8 8 6 9 9 9)
                      '(7 8 8 9 9 10 10))
        :tabuleiro (list '(3 0 2 0 0 1 2)
                         '(0 0 0 0 0 0 0)
                         '(0 0 0 0 0 0 5)
                         '(0 0 0 0 0 0 1)
                         '(0 0 0 0 1 0 0)
                         '(0 0 0 0 0 0 4)
                         '(0 4 0 0 0 0 0))
    )
)
```



Para compilar basta rodar :

```
clisp -c suguru.lisp
```

Para executar:

```
clisp -c suguru.lisp
```

Após ser executado o programa printa a matriz do tabuleiro resolvido

```
(3 4 2 1 5 1 2)
(5 1 5 3 4 3 4)
(2 4 2 1 5 2 5)
(3 1 5 4 3 4 1)
(4 2 3 2 1 2 3)
(1 5 1 5 3 5 4)
(2 4 3 2 1 2 1)
```

Implementação

Foram utilizados os mesmo algoritmos escritos para o Haskell, com algumas funções adicionais para as peculiaridades de Lisp. Algumas operações de lista e matriz puderam ser simplificadas por causa de funções pré-existentes do Lisp, mas na maior parte foi utilizada a mesma lógica recursiva do trabalho anterior.

Abaixo a explicação utilizada no relatório passado:

"É gerado um vetor de tamanhos, que contém um número com o tamanho de cada região.

O algoritmo principal primeiro checa se o número esta dentro dos limites da matriz e se ele se encaixa como solução.

A partir daí, existem vários casos de teste dependendo da condição do número testado e do tabuleiro, por exemplo:

- Checar nova linha (backtrackingTestNextColumn)
- Checar nova coluna (backtrackingTestNextLine)
- Testar próximo número (backtrackingTestNewNumber)
- Achar próxima posição para tentar um número (backtrackingFindNextPos)
- Validar uma posição proposta (backtrackingValidate)
- Entre outros. "

Separação de tarefas

As tarefas do trabalho foram separadas entre os integrantes do grupo: Um dos integrantes ficou a cargo de passar o código em haskell para Lisp e comentar o código novamente, o outro montou o relatório e realizou as tarefas relacionadas a edição de vídeo e pesquisa sobre arrays em Lisp.

Dificuldades

Nesse trabalho houveram menos dificuldades únicas, mas algumas já encontradas retornaram.

O Lisp possui um tipo específico para matrizes, o array que pode ser configurado para ter duas dimensões, mas tivemos bugs ao tentar utilizar ele e resolvemos focar na implementação com lista de listas.

A sintáxe do Lisp também foi um problema. Ela é de certo modo ainda mais obtusa do que o Haskell, e a grande quantidade de parênteses torna todo o código muito menos legível.

Porém, os algoritmos utilizados no trabalho anterior foram reutilizados aqui, diminuindo a carga de trabalho consideravelmente. Passar tudo para Lisp ainda foi trabalhoso, mas não precisamos ter que gastar tempo para bolar a solução.