UNIVERSIDADE FEDERAL DE SANTA CATARINA
INE - DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

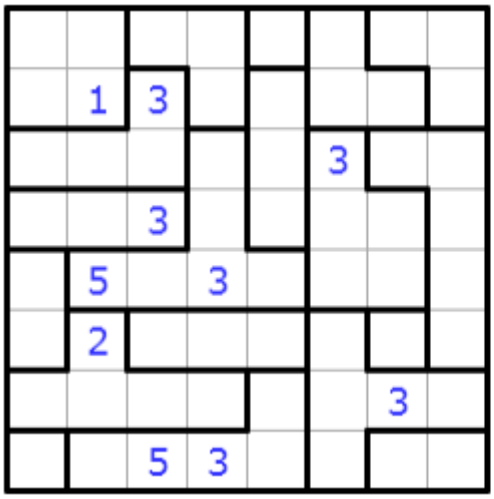
RELATÓRIO REFERENTE AO SEGUNDO TRABALHO DA DISCIPLINA DE PARADIGMAS DE PROGRAMAÇÃO (INE 5416)

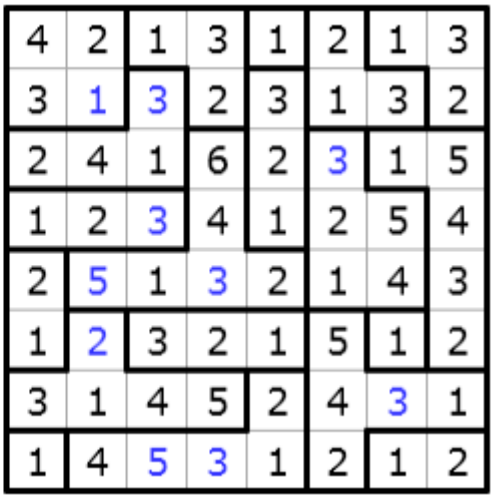
Erik Orsolin de Paula
Rafael Correa Bitencourt
Pedro Augusto da Fontoura

FLORIANÓPOLIS
2024

1. DESCRIÇÃO DO PROBLEMA (KOJUN)

O jogo consiste em preencher todas as regiões de um tabuleiro, que pode variar de 6x6 até 17x17, com números de 1 a n, sem repetição, onde n é o tamanho da região a ser preenchida. Números adjacentes não podem ser iguais e, se estiverem na mesma região, o número diretamente acima deve ser maior do que o abaixo dele.





4	2	1	3	1	2	1	3
3	1	3	2	3	1	3	2
2	4	1	6	2	3	1	5
1	2	3	4	1	2	5	4
2	5	1	3	2	1	4	3
1	2	3	2	1	5	1	2
3	1	4	5	2	4	3	1
1	4	5	3	1	2	1	2

2. SOLUÇÃO ADOTADA (BACKTRACKING)

A estratégia adotada para resolver o problema foi a utilização da técnica de backtracking, uma abordagem de busca por tentativa e erro. O backtracking explora todas as possibilidades de preenchimento do tabuleiro até encontrar uma solução que satisfaça todas as restrições impostas pelo problema. Esta técnica é escolhida por sua simplicidade e clareza na implementação, embora possa não ser a mais eficiente em termos de desempenho.

O tabuleiro e as regiões foram modelados como listas bidimensionais em LISP. Cada célula do tabuleiro contém um número, inicialmente 0 para células vazias, e cada célula da matriz de regiões contém o identificador da região correspondente.

Dentre as funções do algoritmo, as de maior importância são descritas abaixo:

2.1 POSIÇÃO VÁLIDA

Esta função verifica se um número k pode ser colocado em uma posição específica (linha, coluna) do tabuleiro. A verificação inclui:

Se o número k já existe na região.
Se o número k já existe nos vizinhos ortogonais (células adjacentes).
Se as condições de superioridade/inferioridade vertical dentro da mesma região são satisfeitas.

```
(defun eh-posicao-valida (tabuleiro regioes linha coluna k)
  (let* ((regiao (nth coluna (nth linha regioes)))  ;; Encontra a região da célula atual.
         (valores-regiao (loop for i from 0 below (length tabuleiro)  ;; Coleta valores da mesma região para checagem.
                               append (loop for j from 0 below (length tabuleiro)
                                            when (= (nth j (nth i regioes)) regiao)
                                            collect (nth j (nth i tabuleiro)))))
         (vizinhos (list (cons (1- linha) coluna) (cons (1+ linha) coluna) (cons linha (1- coluna)) (cons linha (1+ coluna))))
         (valores-vizinhos (loop for (i . j) in vizinhos  ;; Coleta valores dos vizinhos para checagem.
                                 when (and (>= i 0) (< i (length tabuleiro))
                                           (>= j 0) (< j (length tabuleiro)))
                                 collect (nth j (nth i tabuleiro)))))
    (and (not (member k valores-regiao))  ;; Verifica se k não está na região.
         (<= k (length valores-regiao))  ;; Verifica se k é menor ou igual ao número de células na região.
         (not (member k valores-vizinhos))  ;; Verifica se k não está entre os vizinhos.
         (or (= linha 0)  ;; Condições adicionais para as bordas e regiões acima e abaixo.
             (/= (nth coluna (nth (1- linha) regioes)) regiao)
             (>= (nth coluna (nth (1- linha) tabuleiro)) k))
         (or (= linha (1- (length tabuleiro)))
             (/= (nth coluna (nth (1+ linha) regioes)) regiao)
             (<= (nth coluna (nth (1+ linha) tabuleiro)) k)))))
```

- “região” obtém a região da célula atual.
- “valores-regiao” coleta todos os valores presentes na mesma região.
- “vizinhos” determina as células adjacentes.
- “valores-vizinhos” coleta os valores dos vizinhos para garantir que k não seja repetido.
- A função permite a colocação do número k, se todas as condições forem atendidas.

2.2 RESOLVE

Esta função tenta resolver o tabuleiro a partir de uma posição inicial (linha, coluna) usando a técnica de backtracking. Ela avança célula por célula e retrocede quando encontra um impasse.

```
(defun resolve (tabuleiro regioes linha coluna)
  (cond ((= linha (length tabuleiro)) tabuleiro)  ;; Caso base: se toda a linha foi processada, retorna o tabuleiro.
        ((= coluna (length (car tabuleiro))) (resolve tabuleiro regioes (1+ linha) 0))  ;; Se alcançou o fim de uma
        ((/= (nth coluna (nth linha tabuleiro)) 0) (resolve tabuleiro regioes linha (1+ coluna)))  ;; Se a célula at
        (t (tenta-colocacoes tabuleiro regioes linha coluna (loop for i from 1 to (length tabuleiro) collect i)))))
```

- Verifica se chegou ao fim da linha atual e avança para a próxima linha.
- Verifica se a célula atual já está preenchida e avança para a próxima coluna.
- Se a célula está vazia, tenta preencher com diferentes valores chamando a função “tenta-colocacoes”.

2.3 TENTA COLOCAÇÕES

Esta função tenta diferentes valores em uma célula específica do tabuleiro, chamando resolve recursivamente para cada tentativa válida.

```
(defun tenta-colocacoes (tabuleiro regioes linha coluna ks)
  (if (null ks)
      nil  ;; Se não há mais valores para tentar, retorna nil.
      (let ((k (car ks)))  ;; Pega o primeiro valor da lista para tentar.
        (if (eh-posicao-valida tabuleiro regioes linha coluna k)
            (let ((solucao (resolve (substitui-2d tabuleiro linha coluna k) regioes linha (1+ coluna))))
              (if solucao
                  solucao  ;; Se encontrou solução, retorna.
                  (tenta-colocacoes tabuleiro regioes linha coluna (cdr ks))))  ;; Caso contrário, tenta
            (tenta-colocacoes tabuleiro regioes linha coluna (cdr ks))))))  ;; Se o valor não é válido,
```

- Itera sobre os possíveis valores k para a célula atual.
- Verifica se o valor k é válido usando eh-posicao-valida.
- Se válido, substitui o valor na célula e chama resolve recursivamente.
- Se não encontrar solução, tenta o próximo valor k.

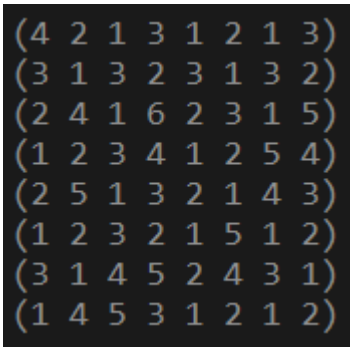
3. ENTRADA E SAÍDA

O usuário pode informar a entrada modificando as variáveis tabuleiro e regiões diretamente no código, conforme o exemplo abaixo:

```
(defun principal ()
  (let ((tabuleiro '((0 0 0 0 0 0 0 0)
                     (0 1 3 0 0 0 0 0)
                     (0 0 0 0 0 3 0 0)
                     (0 0 3 0 0 0 0 0)
                     (0 5 0 3 0 0 0 0)
                     (0 2 0 0 0 0 0 0)
                     (0 0 0 0 0 0 3 0)
                     (0 0 5 3 0 0 0 0)))

        (regioes '((1 1 2 2 3 4 5 5)
                   (1 1 6 2 7 4 4 5)
                   (6 6 6 8 7 9 10 10)
                   (11 11 11 8 7 9 9 10)
                   (12 8 8 8 8 9 9 10)
                   (12 13 14 14 14 15 16 10)
                   (13 13 13 13 17 15 15 15)
                   (18 17 17 17 17 15 19 19))))
```

O resultado é apresentado na forma de uma impressão do tabuleiro resolvido no console, ou uma mensagem informando que nenhuma solução foi encontrada.



4. VANTAGENS E DESVANTAGENS DA UTILIZAÇÃO DO LISP

4.1 VANTAGENS DO LISP:

- LISP é muito flexível e expressivo, permitindo manipulação direta de listas e estruturas de dados.
- A sintaxe homogênea de LISP facilita a criação de metaprogramas.
- A capacidade de tratar código como dados (e vice-versa) permite a implementação de algoritmos complexos de forma mais natural.

4.2 DESVANTAGENS DO LISP:

- LISP pode ser menos eficiente em termos de desempenho em comparação com Haskell.
- A falta de um sistema de tipos forte como o de Haskell pode levar a mais erros em tempo de execução.
- A sintaxe de LISP, baseada em parênteses, pode ser menos legível para aqueles que não estão familiarizados com a linguagem.

5. MUDANÇAS RELATIVAS AO HASKELL

- Em Haskell, usamos funções puras e recursão para manipular o estado do tabuleiro, enquanto em LISP usamos mutabilidade.
- A verificação de condições e a manipulação de listas são mais explícitas e verbosas em LISP.
- Haskell permite a definição de tipos algébricos e o uso de pattern matching, o que pode tornar o código mais conciso e expressivo.

6. ORGANIZAÇÃO DO GRUPO

Nosso grupo foi composto por três membros, cada um responsável por diferentes aspectos do projeto:

Coordenação e Integração (Rafael):

- Coordenou o projeto e integrou o código final.
- Organizou reuniões semanais e entregas regulares

Desenvolvimento e Algoritmo (Erik):

- Implementou o algoritmo de backtracking.
- Garantiu a correção da lógica e eficiência do código.

Testes e Documentação (Pedro):

- Desenvolveu casos de teste e validou o funcionamento do algoritmo.
- Elaborou a documentação e preparou a apresentação.

7. DIFICULDADES

Inicialmente, tivemos dificuldade em entender e implementar corretamente a técnica de backtracking. A complexidade do algoritmo e a necessidade de garantir que todas as condições do puzzle fossem atendidas simultaneamente demoraram para serem integralmente compreendidas.

Implementar a verificação de todas as regras do puzzle Kojun, especialmente a condição de que os números em células verticalmente adjacentes na mesma região devem obedecer a uma ordem específica, também foi um grande desafio. Mas procurando implementações em outras linguagens para o mesmo problema e traduzindo-as para LISP resolvemos logo.

Coordenar as diferentes partes do código desenvolvidas por membros diferentes do grupo e integrá-las em um programa coeso foi uma tarefa complexa. Garantir que todas as funções funcionassem bem juntas exigiu muita comunicação e colaboração.