Relatório II Lisp

Aluno: Jéssica Regina dos Santos
Matrícula: 22100626
Disciplina: INE5416 - Paradigmas de Programação
Professor: Maicon Rafael Zatelli
Data: 05/12/2024

---

1. Introdução

Este relatório apresenta um programa em Common Lisp para resolver Sudokus de tamanho n x n com a adição de restrições personalizadas que definem relações entre elementos da grade.

O programa gera todas as possíveis soluções e verifica quais atendem aos critérios fornecidos.

Vergleichssudoku (Logikrätsel, Zahlenrätsel)

---

2. Estrutura do Código

2.1 Função permutacao-valida

Esta função verifica se uma permutação atende a um conjunto de restrições. Cada restrição é uma tripla que define uma relação entre dois índices e o operador da comparação.

```
(every (lambda (restricao)
         (let* ((i (first restricao)) ;; Índice
                (j (second restricao)) ;; Índice
                (op (third restricao)) ;; Operad
                (val-i (nth i permutacao)) ;; Va
                (val-j (nth j permutacao))) ;; V
           ;; Avalia a relação entre val-i e val
           (cond
             ((char= op #\>) (> val-i val-j)) ;;
             ((char= op #\<) (< val-i val-j)) ;;
             (t t)))) ;; Se não houver operador
       restricoes)) ;; Verifica todas as restriç
```

2.2 Função gerar-permutacoes

Gera todas as permutações possíveis de uma lista, usada para explorar combinações válidas para cada linha da grade.

```
(defun gerar-permutacoes (lista)


  ;; Função que gera  gera todas as permutações possíveis de uma lista.
  ;; Exemplo, para (1 2 3) as permutações seriam [(1 2 3) (1 3 2) (2 1 3) ...]

  (if (null lista) ;; Caso base - a lista vazia retorna uma lista contendo outra lista vazia
      '(())
      ;; Caso recursivo - para cada elemento da lista combina-o com as permutações do restante
      (mapcan (lambda (elemento)
                ;; Para cada permutação do restante da lista, adiciona o elemento atual no início
                (mapcar (lambda (permutacao)
                          (cons elemento permutacao))
                        ;; Remove o elemento atual e gera as permutações do restante
                        (gerar-permutacoes (remove elemento lista :count 1))))
              lista)))
```

2.3 Função dividir-em-blocos

Divide uma lista linear em blocos de tamanho n, transformando uma grade linear em uma matriz n x n.

```
(defun dividir-em-blocos (n lista)


  ;; Função que divide uma lista em blocos de tamanho n
  ;; Para n = 3 e lista = (1 2 3 4 5 6), deve retornar [(1

  (if (null lista) ;; Caso base - a lista vazia retorna um
      '()
      ;; Pega os primeiros n elementos como o primeiro blo
      (cons (subseq lista 0 n)
            (dividir-em-blocos n (subseq lista n)))))


(defun verificar-grade (grade restricoes)
```

2.4 Função verificar-grade

Verifica se a grade inteira atende às restrições fornecidas. Essa função transforma índices lineares em índices da matriz antes de avaliar as relações.

```lisp
(defun verificar-grade (grade restricoes)


  ;; Verifica se uma grade (a matriz) atende as restrições
  ;; As restrições são aplicadas entre os índices absolutos da grade

  (let ((n (length grade))) ;; n é o número de linhas (ou colunas) da grade
    (every (lambda (restricao)
             (let* ((i (first restricao)) ;; Índice linear do primeiro elemento
                    (j (second restricao)) ;; Índice linear do segundo elemento
                    (op (third restricao)) ;; Operador da relação (#\> ou #\<)
                    ;; Converte os índices lineares para (linha, coluna) e obtém os valores correspondentes
                    (val-i (nth (mod i n) (nth (floor i n) grade)))
                    (val-j (nth (mod j n) (nth (floor j n) grade))))
               ;; Avalia a relação entre val-i e val-j
               (cond
                 ((char= op #\>) (> val-i val-j))
                 ((char= op #\<) (< val-i val-j))
                 (t t)))) ;; Se a restrição for inválida assume valor verdadeiro
           restricoes)))
```

2.5 Função resolver-sudoku

Resolve o Sudoku gerando combinações para cada linha e testando todas as possibilidades até encontrar uma solução válida.

```lisp
(let* ((numeros (loop for i from 1 to n collect i)) ;; Lista de números possíveis para
       (linhas (dividir-em-blocos n valores-iniciais)) ;; Converte os valores iniciais
       ;; Gera permutações para cada linha da grade
       ;; Linhas com valores fixos permanecem como estão
       (permutacoes-todas-linhas (mapcar (lambda (linha)
                                           (if (some (lambda (x) (/= x 0)) linha) ;; S
                                               (list linha) ;; apenas usa a linha como
                                               (gerar-permutacoes numeros))) ;; Caso c
                                         linhas)))
  ;; Função recursiva para tentar resolver o Sudoku
  (labels ((tentar-resolver (permutacoes-restantes solucao)
             (if (null permutacoes-restantes) ;; Caso base: todas as permutações foram
                 ;; Verifica se a solução completa atende às restrições
                 (if (verificar-grade (dividir-em-blocos n solucao) restricoes)
                     (list solucao) ;; Se válido, retorna a solução encontrada
                     nil) ;; Caso contrário, não há solução para este caminho
                 ;; Para cada permutação da linha atual tenta resolver o restante recu
                 (mapcan (lambda (permutacao)
                           (tentar-resolver (rest permutacoes-restantes)
                                            (append solucao permutacao))) ;; Adiciona
                         (first permutacoes-restantes)))))) ;; Processa a primeira linh
    ;; Chama a função recursiva com todas as permutações possíveis e uma solução inici
    (let ((resultado (tentar-resolver permutacoes-todas-linhas '())))
      ;; Retorna o resultado como uma matriz n x n (ou nula se não houver solução)
      (if resultado
          (dividir-em-blocos n (first resultado))
          nil)))))
```

2.6 Impressão da Solução

A solução é exibida em formato de matriz n x n.

```lisp
(defun imprimir-sudoku (grade)

  ;; Função para mostrar a solução do Sudoku na tela
  ;;Imprime a grade do Sudoku linha por linha

  (dolist (linha grade)
    (format t "~a~%" linha)))
```

2.7 Função principal

Integra todas as funções e resolve o Sudoku com os parâmetros definidos pelo usuário.

```lisp
(defun principal ()

  ;; Função principal que começa e resolve o sudoku com as restrições


  (let* ((n 4) ;; Tamanho do Sudoku 4x4
         (valores-iniciais (make-list (* n n) :initial-element 0)) ;; Inicializa a gr
         (restricoes '((0 4 #\>) ;; Aqui estão as restrições entre os índices
                       (0 1 #\<)
                       (1 5 #\<)
                       (2 6 #\<)
                       (3 2 #\>)
                       (3 7 #\>)
                       (4 5 #\<)
                       (6 7 #\>)
                       (8 9 #\>)
                       (8 12 #\>)
                       (9 13 #\<)
                       (10 14 #\<)
                       (10 11 #\<)
                       (11 15 #\>)
                       (12 13 #\>)
                       (14 15 #\>))))

    ;; Soluciona o sudoku com os valores e restrições fornecidos.
    (let ((solucao (resolver-sudoku n valores-iniciais restricoes)))

      ;; Mostra a solução ou informa ao jogador que não há solução possível
      (if solucao
          (imprimir-sudoku solucao)
          (format t "Não há solução possível")))))
```

3. Funcionamento do Programa

1. Entrada:
   - n: tamanho da grade n x n.
   - valores-iniciais: lista linear com os valores fixos na grade.
   - restricoes: lista de triplas representando relações entre índices.

2. Processamento:
    - Divide a lista inicial em blocos.
    - Gera permutações possíveis para cada linha.
    - Testa combinações completas com base nas restrições.
3. Saída:
    - Solução como matriz n x n ou mensagem indicando que não há solução.

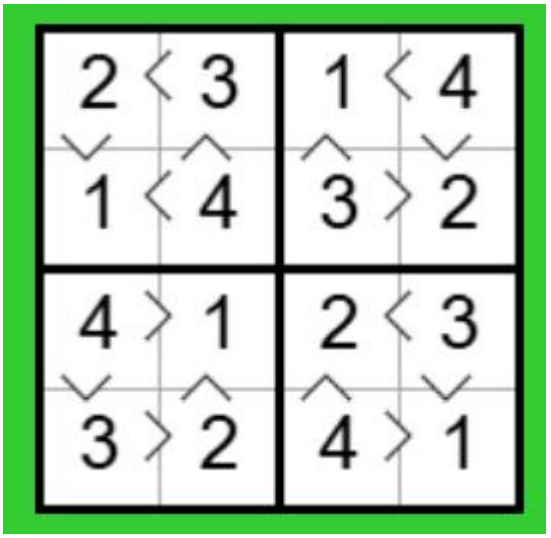
4. Testes e Exemplos

Configuração de Teste:

- Grade 4 x 4
- Restrições: Definidas entre índices, por exemplo (0 4 #\>) (valor na posição 0 deve ser maior que o valor na posição 4).

Output:

(2 3 1 4)
(1 4 3 2)
(4 1 2 3)
(3 2 4 1)



5. Conclusão

O programa resolve Sudokus. Ele utiliza uma abordagem que combina geração de permutações e validações incrementais. A flexibilidade no tamanho da grade e na definição de restrições torna-o

aplicável para variados cenários, com limitações de desempenho para tamanhos maiores devido à explosão combinatória.