**Universidade Federal de Santa Catarina**
Departamento de Informática e Estatística (INE)
Curso: Ciência da Computação
Disciplina: Paradigmas de Programação (INE5416)
Professor: Maicon Rafael Zatelli
**Estudante:** Guilherme Adenilson de Jesus (22100620)

# Relatório T2 - Scheme

- **Problema**

O presente relatório é referente ao desenvolvimento de um programa em Gambit Scheme para um solucionador do quebra-cabeça escolhido, Vergleichssudoku (Sudoku Greater Than). Esse *puzzle* é uma variante do Sudoku, em que cada posição do tabuleiro possui comparadores para as posições adjacentes a ela dentro do seu quadrado.

- **Solução**

O tabuleiro é separado em três matrizes. Uma para armazenar os valores de cada posição, uma para os comparadores horizontais e outra para os verticais. O tabuleiro utiliza o tipo *vector* pela maior facilidade de alteração dos valores em posições, enquanto os comparadores mantêm com o tipo *list*. Os comparadores são obtidos por um arquivo .txt, detalhado melhor posteriormente.

As três matrizes são enviadas para função *solve*. Que executará cada alternativa possível. Ele inicia na primeira posição (0,0) para executar recursivamente a função solve. No momento que encontrar a solução (quando ela achar um valor possível na última posição do tabuleiro), será impresso o resultado e fechará o programa.

```
;;Função que realiza o backtracking
(define (solve board row col comp_h comp_v)
  (if (and (= row (- 9 1)) (= col 9)) ;;Verifica se concluiu o backtracking
      ;;Se concluiu, imprime a solução e fecha o programa
      (begin
        (print-board board)
        (exit))
      ;;Se não, continua para a próxima posição a ser testada
      (begin
        ;;Atualiza o valor de row se chegar no fim do vetor
        (if (= col 9)
            (begin
              (set! row (+ row 1))
              (set! col 0)))
        ;;Verifica se a posição já tem um valor colocado
        (if (> (vector-ref (vector-ref board row) col) 0)
            (solve board row (+ col 1) comp_h comp_v)
            (begin
              (do ((num 1 (+ num 1))) ((> num 9) #f)
                (if (isPossible board row col num comp_h comp_v) ;;Verifica se a opção de valor é possível
                    (begin
                      (vector-set! (vector-ref board row) col num)
                      (if (solve board row (+ col 1) comp_h comp_v) ;;Verifica se o backtracking foi sucedido
                          #t
                          (vector-set! (vector-ref board row) col 0)
                      )
```

Figura 1 - Função *solve*
Fonte: autoria própria

A seguir, há a função *isPossible*. Ela avaliará se a alternativa solicitada (num) será possível baseada na atual organização do tabuleiro. Analisando o trecho de código abaixo (Figura 2), são feitas até 6 verificações: se a posição está fora do tabuleiro, se a posição já está preenchida, se num já é um elemento da linha, coluna e quadrado. Essas análises são iguais às realizadas no Sudoku, o que altera é a última, que utiliza os comparadores referentes à posição. Destaca-se que cada célula de um quadrado tem diferentes quantidades de comparadores, o que torna necessário a avaliação da posição em relação ao seu quadrado. Para realizar essa verificação final, é utilizada a função *comparing* para cada comparador.

```
;;Verifica se o número sugerido é possível
(define (isPossible board row col num comp_h comp_v)
    (cond
        ;;Verificações padrões do sudoku
        ((> (getCell board row col) 0) #f)
        ((isElem (getRow board row) num) #f)
        ((isElem (getColumn board col) num) #f)
        ((isElem (getSquare board row col) num) #f)

        ;;Verificação dos comparadores, referente a posição relativa ao quadrado que [row][col] está
        ((and ( = (modulo row 3) 0) ( = (modulo col 3) 0)) (and (comparing  num                          (getComp comp_h row col)      (getCell board row (+ col 1)))
                                                                (comparing  num                          (getComp comp_v row col)      (getCell board (+ row 1) col)))
        )

        ((and ( = (modulo row 3) 0) ( = (modulo col 3) 1)) (and (comparing (getCell board row (- col 1)) (getComp comp_h row (- col 1)) num )
                                                                (comparing  num                          (getComp comp_v row col)      (getCell board row (+ col 1)))
                                                                (comparing  num                          (getComp comp_h row col)      (getCell board (+ row 1) col)))
        )
        ((and ( = (modulo row 3) 0) ( = (modulo col 3) 2)) (and (comparing (getCell board row (- col 1)) (getComp comp_h row (- col 1)) num )
                                                                (comparing  num                          (getComp comp_h row col)      (getCell board (+ row 1) col)))
        )
        ((and ( = (modulo row 3) 1) ( = (modulo col 3) 0)) (and (comparing  num                          (getComp comp_h row col)      (getCell board row (+ col 1)))
                                                                (comparing  num                          (getComp comp_v row col)      (getCell board (+ row 1) col))
                                                                (comparing (getCell board (- row 1) col) (getComp comp_v (- row 1) col) num))
```

Figura 2 - Função *possible*
Fonte: autoria própria

Como visto na Figura 3 abaixo, a função *comparing* recebe dois inteiros e uma string (comparador). Para acelerar as análises, é feito algumas avaliações das alternativas extremas (1 e 9). Se o valor 1 estiver ao lado de um comparador '>', é impossível que haja outro número menor que ele, então já é considerado False de imediato, mesma coisa para valor 9 com '<'. Essa verificação premeditada é importante devido ao valor 0 (posição vazia), que qualquer valor comparado a ele já se tornaria True, podendo considerar uma alternativa que nunca daria uma solução correta. Caso passe por todas essas avaliações, é feita a comparação em si entre a e b.

```
;;Compara se um número é maior/menor que outro
(define (comparing a comp b)
    (cond
        ((and (= a 1) (string=? comp ">")) #f) ;;1 nunca será maior que qualquer outro número
        ((and (= a 9) (string=? comp "<")) #f) ;;9 nunca será menor que qualquer outro número
        ((and (= b 1) (string=? comp "<")) #f) ;;1 nunca será maior que qualquer outro número
        ((and (= b 9) (string=? comp ">")) #f) ;;9 nunca será menor que qualquer outro número
        ((or  (= a 0) (= b 0)) #t) ;;Se alguns deles for zero, então pode inserir sempre
        ((string=? comp ">") (> a b))
        (else (< a b))
    )
)
```

Figura 3 - Função *comparing*
Fonte: autoria própria

- **Entrada pelo usuário**

Como exposto na Figura 4 abaixo, o programa pedirá ao usuário para escrever no terminal o arquivo de texto referente à configuração do quebra-cabeça. Caso não seja possível abrir, o programa finaliza com um aviso de erro.

```
(define (main)
    (display "Digite o arquivo do puzzle a ser resolvido: ")
    (let ((x (read-line)))
        (define compH (my-map string-split char-whitespace? (slice (readlines x) 0 9))) ;;Comparadores Horizontais
        (define compV (my-map string-split char-whitespace? (slice (readlines x) 10 19))) ;;Comparadores Verticais
```

Figura 4 - Recebimento de entrada
Fonte: autoria própria

O arquivo de configuração é similar ao apresentado na Figura 5 a seguir. Começa pelos comparadores horizontais, linha a linha. Quando não houver um comparador, é utilizado o caracter “|”, indicando troca de quadrado ou fim da linha do quebra-cabeça. Para facilitar a leitura e separação, antes da descrição dos comparadores verticais é colocado uma linha com um único caracter “.”.

```
1   < < | > > | < < |
2   < > | < > | < > |
3   > > | > > | > < |
4   > < | < < | < > |
5   > < | > < | > < |
6   > < | > < | < > |
7   > > | < > | > < |
8   < > | < > | > < |
9   < > | > > | < < |
10  .
11  < < > > > > > > >
12  > > > > > > < < <
13  | | | | | | | | |
14  < > < < > < > > <
15  < < < > > > > < >
16  | | | | | | | | |
17  > > > > < < < < >
18  > < > < > > > < <
19  | | | | | | | | |
```

Figura 5 - Configuração do quebra-cabeça
Fonte: autoria própria

O resultado do jogo é imprimido no terminal (Figura 6), mostrando os valores que estarão presentes em cada posição do quebra-cabeça. Caso não haja uma solução possível, será printado “Solução não encontrada”.

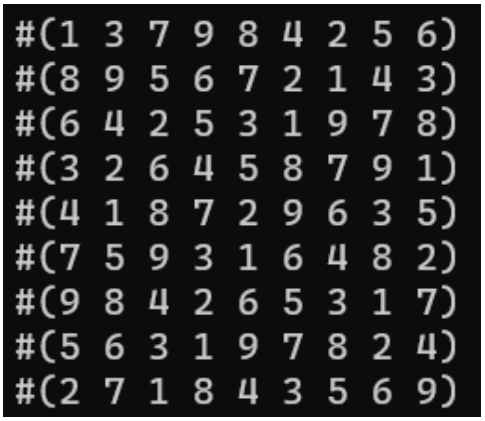

Figura 6 - Resultado do quebra-cabeça
Fonte: autoria própria

- **Dificuldades encontradas**

Uma das dificuldades foi adaptar o programa feito no trabalho anterior para essa nova linguagem, principalmente a função *solve* que possuía uma grande dependência do *lazy evaluation* de métodos que não tem no Scheme. Para isso, ela foi refeita como visto no tópico Solução.

Outro problema encontrado foi a leitura do arquivo. Para isso, foi necessário criar as funções *readlines* (leitura do arquivo), *string-split* (para criar as matrizes de comparadores), *slice* (para separar os comparadores) e *my-map* (para auxiliar o *string-split*).

Por fim, a maior dificuldade foi a otimização. Visto que o programa foi testado utilizando *gsi* (interpretador), já é notável presenciar uma perda de desempenho. Contudo, para diminuir o tempo para aproximadamente ⅓, foi adaptado o código para que a variável utilize o tipo *vector*, invés do *list*. Essa solução foi necessária devido a uma grande quantidade de alterações realizadas no tabuleiro, que com o tipo anterior obrigava percorrer a matriz até encontrar a posição a ser alterada e depois criar um novo tabuleiro com o valor substituído.