UNIVERSIDADE FEDERAL DE SANTA CATARINA

CAMPUS TRINDADE

INE-DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

INE5416 - PARADIGMAS DE PROGRAMAÇÃO

Alunos:

João Victor Cabral Machado — **22204113**

Pedro Costa Casotti — **22200374**

Relatório do Trabalho 2 - Puzzle Kojun

Florianópolis

2024

1 - Introdução

O puzzle Kojun é um desafio de lógica que requer preencher um tabuleiro com números, garantindo que cada número apareça apenas uma vez por região e siga certas restrições de proximidade. O tabuleiro é composto por várias regiões não retangulares, cada uma contendo células preenchidas com números de 1 ao total de células na região. A resolução desse puzzle é complexa e fascinante, especialmente para algoritmos de resolução.

2 - Descrição da Solução

2.1 - Modelagem do Tabuleiro

Para implementar a solução em Lisp, utilizamos a estrutura de dados de matrizes e tabelas de hash. Inicialmente, cada célula do tabuleiro é preenchida com zero, indicando células vazias. Em seguida, o tabuleiro é atualizado usando um algoritmo de backtracking para encontrar os números corretos para cada posição.

```
; Implementação da estrutura de dados do puzzle
(defvar puzzle-w 8)
(defvar puzzle-h 8)
(defvar puzzle(make-array `(,puzzle-w ,puzzle-h)
:initial-contents '(; Estado inicial do puzzle)))
```

2.2 - Algoritmo do Backtracking

O algoritmo de backtracking é implementado pela função solve, que tenta preencher o tabuleiro recursivamente. A função verifica se uma célula já está preenchida. Se não estiver, tenta encontrar um número válido para essa célula e prossegue para a próxima célula. Se todos os números possíveis são esgotados e nenhuma solução válida é encontrada, a função retrocede, resetando a célula para zero e tentando um novo caminho.

```
; Função de backtracking para resolver o puzzle
(defun solve (x y) (cond
   ((>= x puzzle-w) t)
   ((>= y puzzle-h) (solve (1+ x) 0))
   (t
      (if (= 0 (getmatrix puzzle x y))
         (trynumbers x y (get-available-numbers x y))
         (solve x (1+ y))))))
```

Uma parte essencial da solução é a função get-available-numbers, que determina quais números podem ser colocados em uma célula específica sem violar as regras do Kojun. Essa função considera as restrições de números na mesma linha, coluna e região para determinar os números disponíveis.

```
; Função para obter os números disponíveis para uma célula
(defun get-available-numbers (x y)
   (let (
      (a (get-orthogonals x y))
      (b (get-region-numbers x y))
      (c (get-vertical-numbers x y))
   )
   (set-difference c (union a b))))
```

Uma otimização crucial foi a implementação eficiente da função acima, que minimiza o espaço de busca eliminando rapidamente números inviáveis antes de fazer chamadas recursivas. Isso reduz significativamente o número de caminhos explorados pelo backtracking.

3 - Entrada e Saída

A entrada do puzzle é definida estaticamente no código. O estado inicial do puzzle é codificado diretamente na função main. A saída é apresentada no console, mostrando o tabuleiro completamente resolvido ou uma mensagem indicando que não foi encontrada uma solução. É possível alterar o tabuleiro e os números, modificando diretamente o código.

Pode ser possível alterar o tabuleiro e os números, mudando diretamente no código.

A execução do código está descrita do README.md.

4 - Desafios e Soluções

Desafio 1: Eficiência do Backtracking

Problema: Backtracking pode ser extremamente lento se o espaço de busca não for adequadamente restrito, especialmente em puzzles complexos com muitas possibilidades.

Solução: Para lidar com isso, implementamos a função get-available-numbers para reduzir o número de opções inviáveis em cada passo, minimizando o número de estados inválidos explorados pelo algoritmo

Desafio 2: Complexidade de Implementação

Problema: Gerenciar o estado mutável e a complexidade das regras de validação pode tornar o código difícil de entender e manter.

Solução 2: Gerenciar o estado mutável e a complexidade das regras de validação pode tornar o código difícil de entender e manter. Para resolver isso, utilizamos estruturas de dados e funções claras, modularizando o código para torná-lo mais gerenciável e legível.

5 - Conclusões

A implementação do resolvedor para o puzzle Kojun em Lisp foi uma oportunidade para aplicar técnicas avançadas de programação funcional. As otimizações implementadas melhoraram significativamente a performance do algoritmo, tornando viável a resolução de puzzles dentro do tamanho limite estipulado.

O trabalho foi feito de maneira distribuída entre os membros do grupo, Pedro teve foco na solução inicial enquanto João teve foco na otimização da solução do problema.