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 1 - Puzzle Kojun

Florianópolis

2024

1 - Introdução

O puzzle Kojun é um desafio de lógica que exige preencher um tabuleiro com números de forma que cada número apareça uma vez por região e obedeça a certas restrições de proximidade. O tabuleiro é composto por várias regiões não retangulares, cada uma com células preenchidas com números que começam com 1 e vão até o número total de células na região. O principal desafio é garantir que as soluções respeitem todas as regras, o que torna o puzzle particularmente complicado e fascinante para um algoritmo de resolução.

2 - Descrição da Solução

2.1 - Modelagem do Tabuleiro

Para implementá-lo, foi utilizada a estrutura de dados do Haskell "*IOArray*", que permite mudanças dentro de um contexto de interconexão de dados. Ao começar, cada célula do tabuleiro é preenchida com zero, indicando células vazias. Em seguida, o tabuleiro é atualizado usando o algoritmo para encontrar números corretos para cada posição.

```
type Puzzle = IOArray Int Int
newPuzzleMatrix (w, h) = newListArray (1, w * h)
```

A escolha de *IOArray* foi crucial devido à natureza do algoritmo de backtracking, que frequentemente atualiza o estado do tabuleiro ao tentar diferentes números nas células.

2.2 - Algoritmo do Backtracking

O backtracking é implementado pela função solve, que tenta preencher o tabuleiro recursivamente. A função verifica se uma célula já está preenchida; se não, ela 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.

```
solve puzzle (x, y)
   | y > puzzleH = return True
   | x > puzzleW = solve puzzle (1, y + 1)
   | otherwise = do
       value <- getPuzzle puzzle (x, y)
       case value of
           0  ->  getAvailableNumbers  puzzle  (x,  y)>>=
tryNumbers puzzle (x, y)
           _ -> solve puzzle (x + 1, y)
```

A função *getAvailableNumbers* é essencial para determinar quais números podem ser colocados em uma célula específica sem violar as regras do Kojun. Ela combina as restrições de números na mesma linha, coluna e região para determinar os números disponíveis.

```
getAvailableNumbers p (x, y) = do
   a <- getOrthogonalNumbers p (x, y)
   b <- getRegionNumbers p (x, y)
   c <- getVerticalNumbers p (x, y)
   return (c \\ (a `union` b))
```

Uma otimização crucial foi a implementação eficiente de *getAvailableNumbers*, 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 é definida estaticamente no código, onde 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.

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: Implementei a função *getAvailableNumbers* para reduzir drasticamente o número de opções inviáveis em cada passo, utilizando

informações das restrições ortogonais e de região para cada célula. Isso ajudou a acelerar o processo de encontrar uma solução ao reduzir o número de estados inválidos que o algoritmo precisa explorar.

Desafio: 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: A utilização de estruturas de dados e funções claras, cada uma responsável por uma parte específica da lógica do puzzle (como *getRegionNumbers*, *getOrthogonalNumbers*), ajudou a modularizar o código, tornando-o mais gerenciável e legível.

5 - Conclusões

A implementação de um resolvedor para o puzzle Kojun em Haskell apresentou uma excelente oportunidade para aplicar técnicas avançadas de programação funcional, especialmente o uso de arrays mutáveis em um contexto de IO para realizar operações complexas de backtracking. As otimizações empregadas 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.