Relatório de implementação de resolvedor de puzzle Kojun

Gabriel de Vargas Coelho, 20200400

Florianópolis, 2022

Análise do Problema

O jogo escolhido para esse trabalho foi o kojun. A escolha se deu simplesmente por gosto pessoal, uma vez que não gosto de jogos no estilo sudoku, o kojun foi, dentre as opções, o mais divertido.

O kojun se trata de um jogo simples, em que o objetivo é conseguir completar um tabuleiro com números, respeitando 3 regras:

1. Casas adjacentes ortogonalmente devem possuir números distintos;
2. O número máximo para uma casa é determinado pelo número de casas do grupo, e dentro de um grupo não pode haver repetição de números;
3. Casas de um mesmo grupo e de mesma coluna, devem apresentar numeração decrescente no sentido de cima para baixo.

Assim, o desafio para a implementação de uma solução ao kojun se dá pelo fato de haverem muitas "respostas" corretas para um tabuleiro. Uma vez traçada a estratégia para o preenchimento das casas do tabuleiro, o segundo desafio é a verificação da corretude da solução. Na próxima seção são explicitados ambas as estratégias.

Linguagem Escolhida

Para a solução do problema aqui proposto foi usado a linguagem Lisp. A escolha se deu por 2 motivos:

1. Foi a linguagem apresentada em sala de aula;
2. Dentre as linguagens funcionais apresentadas talvez seja a que tenha o menor número de funções já prontas, desse modo, há um desafio a mais em implementar funções que o Haskell, por exemplo, já implementa.

Solução Proposta

A primeira coisa a ser destacada aqui é a modelagem do tabuleiro. Optou-se por utilizar a estrutura de tupla para modelar uma casa do tabuleiro, da seguinte forma: ((Int), Int), em que o primeiro valor é uma lista de todos os valores possíveis para aquela casa, e o segundo valor é um indicador do grupo ao qual aquela casa pertence. Uma vez definida a modelagem de uma casa do tabuleiro, o tabuleiro em si é somente uma matriz, em que cada posição é do tipo ((Int), Int). Para sinalizar uma casa vazia é usado uma lista com o valor 0.

```
(((0) 0) ((0) 1) ((4) 1) ((0) 1) ((2) 3) ((0) 4))
(((0) 0) ((0) 2) ((3) 1) ((0) 3) ((0) 3) ((0) 3))
(((1) 0) ((4) 0) ((0) 5) ((4) 3) ((0) 10) ((0) 10))
(((0) 6) ((5) 7) ((0) 5) ((0) 9) ((0) 9) ((2) 10))
(((0) 6) ((0) 7) ((0) 7) ((0) 8) ((3) 8) ((0) 10))
(((6) 7) ((2) 7) ((0) 7) ((2) 8) ((0) 8) ((5) 8))
```

Imagem 1: Tabuleiro 6x6 usando a modelagem proposta

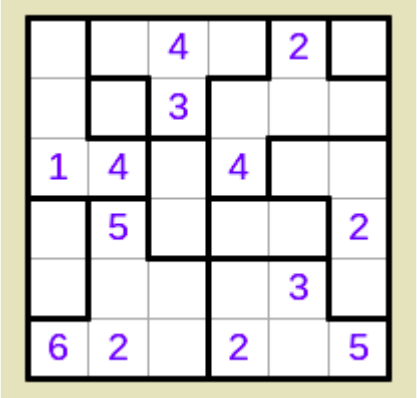
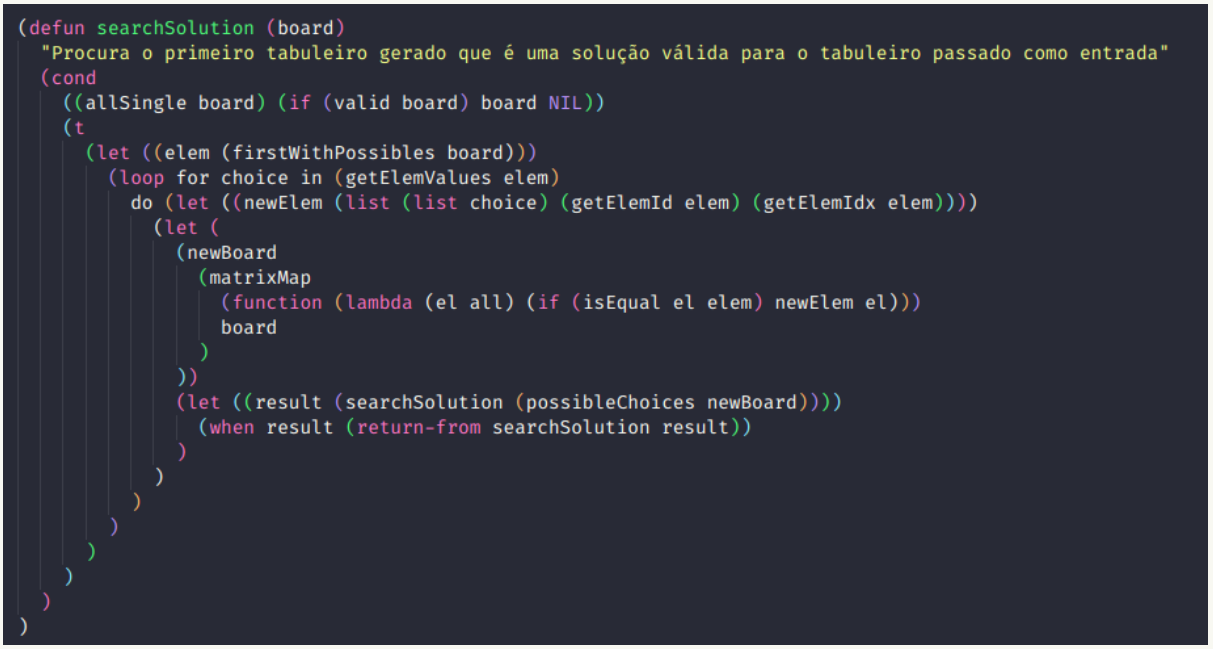


Imagem 2: Tabuleiro modelado pela imagem 1. Fonte: [1]

A primeira coisa feita pelo algoritmo de resolução é definir um índice para cada casa do tabuleiro, isso facilita processos de identificar cada casa do tabuleiro, e é usado no momento de gerar novos tabuleiros. Após isso é definido valores possíveis para cada casa em branco. Por exemplo, seguindo a Imagem 2, a primeira casa do tabuleiro (casa superior esquerda), tem como opções os valores 2 e 3, assim em nossa modelagem seria o seguinte: ((2 3) 0). Com isso é feita a geração de um tabuleiro possível a partir da escolha dos valores disponíveis na primeira casa em branco disponível. Em nosso exemplo, seria gerado um tabuleiro com a casa em questão tendo o valor ((2) 0).



```
(defun searchSolution (board)
  "Procura o primeiro tabuleiro gerado que é uma solução válida para o tabuleiro passado como entrada"
  (cond
    ((allSingle board) (if (valid board) board NIL))
    (t
      (let ((elem (firstWithPossibles board)))
        (loop for choice in (getElemValues elem)
          do (let ((newElem (list (list choice) (getElemId elem) (getElemIdx elem))))
            (let (
              (newBoard
                (matrixMap
                  (function (lambda (el all) (if (isEqual el elem) newElem el)))
                  board
                )
              ))
              (let ((result (searchSolution (possibleChoices newBoard))))
                (when result (return-from searchSolution result))
              )
            )
          )
        )
      )
    )
  )
)
```

Imagem 3: Função que busca o tabuleiro que satisfaz as condições do jogo

Após a escolha feita, o tabuleiro gerado vai passar por um processo de redução de escolhas. Nesse processo, ao identificar uma casa do tabuleiro que possui múltiplas escolhas, é obtido os valores já configurados para o grupo ao qual essa casa pertence, e também os valores já configurados nas casas vizinhas ortogonalmente, e é feita a subtração desses valores no leque de escolhas da casa. Exemplificando o processo: considere que existem as seguintes casas no tabuleiro ((1 3 4) 1) e ((4) 1), após passar pela redução teremos o seguinte: ((1 3) 1) e ((4) 1).

O tabuleiro então segue para a avaliação, se o tabuleiro tiver todas as casas configuradas em um só valor, e cumprir as regras do jogo, explicitadas na seção anterior, então esse tabuleiro é a solução do problema. Caso contrário, esse tabuleiro volta ao passo em que são gerados novos tabuleiros. Caso a árvore gerada para uma escolha possível não apresente resultado válido, é passado todo o procedimento sob a ótica de outra escolha, no nosso exemplo voltaríamos a primeira casa e seria definido o valor ((3) 0) para a casa, em caso de não haver solução é retornado ao usuário um array vazio.

No caso de sucesso, o usuário visualiza em seu terminal de comandos o tabuleiro, modelado com nossa proposta, com os valores da solução.

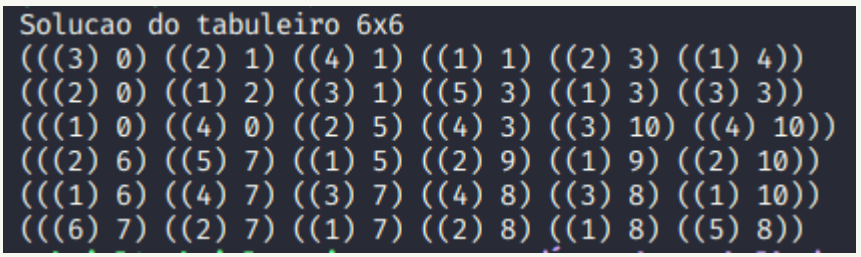


Imagem 4: Solução possível do tabuleiro das Imagens 1 e 2

Dificuldades

A grande dificuldade encontrada no presente trabalho foi conseguir implementar uma versão do algoritmo que fosse capaz de resolver tabuleiros maiores que 6x6, tanto que tal desafio não conseguiu ser transposto.

Em comparação com o trabalho desenvolvido com Haskell, a estratégia de solução do tabuleiro é a mesma, tendo até uma otimização a mais, que é considerar as casas vizinhas já configuradas para montar a lista de opções possíveis de uma casa. Porém o resultado alcançado não foi o mesmo, visto que a versão em Haskell consegue solucionar tabuleiros 10x10.

Para essa discrepância nos resultados consegue-se apontar 3 hipóteses:

1. A estratégia adotada é mais eficiente para a linguagem Haskell;
2. A implementação de funções que o Haskell já disponibiliza, não foi feita de forma igualmente eficiente;
3. O mais provável talvez seja a junção dos dois pontos acima.

Referências

[1] JANKO.AT. **Kojun**. Disponível em: <https://www.janko.at/Raetsel/Kojun/index.htm>. Acesso em: 08 out. 2022.