Trabalho 2 - Resolvedor em *Common Lisp* para o jogo Suguru

INE5416 - Paradigmas de Programação

Matheus Dhanyel C. Roque e Pedro H. Aquino Silva

Abril 2021

1 Análise do problema

O trabalho consiste em desenvolver um programa em linguagem que suporta o paradigma funcional que encontre uma solução para um *puzzle* do tipo Suguru. A linguagem escolhida foi `Common Lisp`.

Este jogo é um quebra-cabeças com números, que tem como base um tabuleiro $n \times n$, dividido em regiões — ou *tectons*. Dentro de cada *tecton*, as posições livres devem ser preenchidas com cada número entre 1 e a quantidade de posições do *tecton*, sendo que dois números iguais não podem ser vizinhos ortogonais ou diagonais. Algumas posições já são preenchidas no tabuleiro de entrada.

2 Descrição da solução

2.1 Visão geral

De maneira geral, o funcionamento da solução pode ser separada em 3 tópicos:

- **Estruturas e entrada de dados:** Primeiramente é definido como os dados serão armazenados e como se estrutura a entrada de dados para que eles possam ser devidamente manipulados.
- **Checagem de valores válidos:** Em seguida, é necessário definir funções para o acesso dessas estruturas e para fornecer quais números podem ser alocados em cada posição.
- **Solução por tentativa e erro:** Agora que conseguimos obter os movimentos válidos para cada posição, podemos executar um algoritmo de tentativa e erro, até a completude da matriz de tabuleiro.

2.2 Estruturas e entrada de dados

Primeiramente foi decidido como seria feita a modelagem do *puzzle* Suguru em nosso programa. Após pesquisar e analisar as regras do puzzle, decidimos por uma abordagem que representa o puzzle com uma estrutura chamada *game*, que possui dois elementos: uma matriz com os valores de cada espaço, tomando 0 como valor das posições que ainda devem ser preenchidas; e uma lista de todos os *tectons* do puzzle, sendo que cada região é uma lista de posições.

```
(defstruct game
    ; position matrix for the board, use with make-array
    board
    ; list of tectons, each a list of positions
    tectons
    )
```

Dessa forma, um `game` pode ser inicializado no programa criando apenas algumas listas, como segue no exemplo de Suguru 6×6:

```
(setq example1  ; https://www.janko.at/Raetsel/Suguru/001.a.htm
    (make-game
        :board (make-array '(6 6)
            :initial-contents '((4 0 0 0 0 0)
                                (0 0 0 0 0 0)
                                (0 0 4 0 0 1)
                                (0 0 0 2 0 0)
                                (5 0 0 3 5 0)
                                (0 0 0 0 0 0)))
        :tectons (make-array '(8)
            :initial-contents '(((0 0) (0 1) (1 0) (2 0))
                                ((0 2) (1 1) (1 2) (1 3) (2 2))
                                ((0 3) (0 4) (0 5) (1 4) (2 4))
                                ((3 0) (4 0) (5 0) (3 1) (2 1))
                                ((3 2) (3 3) (3 4) (2 3) (4 3))
                                ((1 5) (2 5) (3 5) (4 5) (5 5))
                                ((4 1) (4 2))
                                ((5 1) (5 2) (5 3) (5 4) (4 4))))))
```

O grupo decidiu que a instância de Suguru a ser solucionada deveria ser definida no arquivo `main` do trabalho, de modo que a entrada de dados é *hard-coded* no programa Lisp.

As estruturas de dados utilizadas são análogas às utilizadas no resolvedor de Makaro desenvolvido em Haskell para o Trabalho 1. Naquele momento, utilizamos um *array* 2-D para representar o tabuleiro, e listas de tuplas para representar cada *tecton*. Como não existem tuplas em Lisp, utilizamos listas nesta solução.

2.3 Checagem de movimentos válidos

Quanto à checagem de valores válidos, existem três critérios que devem ser considerados. As jogadas válidas são limitadas pelo tamanho da região em que a posição se encontra, pelos valores já existentes na mesma região e pelos vizinhos em todas as direções. Para que o código fosse mais compreensível e menos inclinado a erros difíceis de rastrear, esses três elementos foram computados separadamente e em seguida utilizados na função `validMoves`:

```
(defun validMoves (puzzle i j)
    (let ((moves (initialValues (length (findTectonPosition puzzle i j))))
          (invalidMoves (union (invalidMovesNeighbours puzzle i j) (
    invalidMovesTecton puzzle i j))))
          (remove-if (lambda (val) (member val invalidMoves)) moves)))
```

Desta forma, a função `validMoves` retorna uma lista de todos os valores possíveis naquela posição. Primeiramente `initialValues` cria o vetor `moves` com os valores de 1 a N, sendo N o tamanho da região em que a posição se encontra. A função `invalidMovesNeighbours` retorna um vetor com todos os vizinhos preenchidos e `invalidMovesTecton` um vetor com todos os valores já presentes no tecton em questão, sendo que os retornos de ambas são unidos em uma única lista `invalidMoves`. Por fim, são removidos quaisquer valores na lista `moves` que estejam presentes em `invalidMoves`.

Ambas as funções que buscam por valores inválidos apenas chamam funções auxiliares, sendo a mais complexa a `squareElementList`. Essa função retorna um vetor com todos os valores que existem em um quadrado 3x3 ao redor da posição, ignorando valores de posições inválidas, valores vazios ou valores repetidos. Basicamente, ela funciona passando dois iteradores como argumentos em suas chamadas recursivas, utilizando eles como posição checada e como critério de parada para a função.

```
(defun squareElementList (puzzle i j iti itj elemList)
    (let ((val (getValue puzzle (+ i iti) (+ j itj))))
        (cond
            ((= 3 iti) elemList)  ; se i = 3, foram as 3 linhas
            ((= 3 itj) (squareElementList puzzle i j (1+ iti) 0 elemList))
   ; se j = 3, passa pra proxima linha
            ((or
                (null val)  ; se o valor em i ou j esta fora da matriz
                (= val 0)  ; se nao foi preenchido
                (member val elemList))  ; se o valor ja esta na lista
                    (squareElementList puzzle i j iti (1+ itj) elemList))  ;
    nao colocamos em elemList
            (t (squareElementList puzzle i j
                iti (1+ itj)
                (cons val elemList))))))  ; caso contrario, coloca na lista
```

2.4 Solução por tentativa e erro

A solução implementada segue a mesma estratégia utilizada no Trabalho 1. Primeiramente, com a ajuda das funções auxiliares criadas, gera-se uma lista de possíveis valores em dada posição em branco, isto é, uma lista de movimentos que obecedem as regras do jogo. A partir desta lista, itera-se sobre as opções disponíveis buscando encontrar a solução válida. Diferentemente do Trabalho 1, em que utilizamos a função `concatMap` e comprehensions como modo de implementar a ideia de não-determinismo, aqui a busca sobre as soluções é feita iterativamente, desenvolvendo um nodo de computação ao máximo até encontrarmos uma solução ou um beco sem saída. Para coordenar as tentativas e erros, temos a função `suguru-solver`:

```
(defun suguru-solver (puzzle)
    (let ((next (nextOpenPosition puzzle)))
        (cond
            ((null next) puzzle)  ; resolvido
            ((null (validMoves puzzle (elt next 0) (elt next 1))) nil)  ;
   deadtrail
            (t (loop for try in (validMoves puzzle (elt next 0) (elt next 1)
   )
                do (let ((result
                    (suguru-solver (update puzzle (elt next 0) (elt next 1)
   try))))
                    (if (null result)
                        (progn
                            (update puzzle (elt next 0) (elt next 1) 0)
                            nil)
                        (return-from suguru-solver result))))))))
```

Aqui, temos três situações possíveis. O nodo atual avaliado contém uma solução válida, isto é, não temos outra posição vazia a preencher; o nodo encontrou um caminho sem saída e a solução parcial avaliada não pode ser desenvolvida, ou seja, a lista de movimentos possíveis retorna vazia; ou o nodo ainda pode ser desenvolvido em mais um passo de computação, e neste caso, fazemos a chamada da função `suguru-solver` em cada filho gerado iterativamente no loop por `update`:

```
; this considers puzzle a mutable variable
(defun update (puzzle i j value)
    (progn
        (setf (aref (game-board puzzle) i j) value)
        puzzle))
```

Na implementação de `update` utilizada, a alteração feita na matriz de inteiros do tabuleiro é destrutiva, por isso, caso o resultado seja retornado nulo, isto é, um *deadtrail*, precisamos restaurar a matriz para a configuração anterior. Isto é feito chamando novamente a função update e inserindo um zero na posição previamente alterada. Caso contrário, e a função sugurusolver em algum filho retornar um valor não-*nil*, então utilizamos a *macro* `return-from` para sair do loop e retornar o puzzle resolvido.

Nestes pontos, esse trabalho difere da implementação em Haskell, em que fazíamos uso da *lazy evaluation* da linguagem, e computávamos os filhos utilizando compreensões de listas, nos aproveitando da imutabilidade de variáveis que Haskell tem por padrão. Aqui, utilizamos alguns recursos que aproximam Common Lisp do paradigma imperativo, o que gerou efeitos colaterais que precisaram ser remediados (como a mutabilidade da matriz de inteiros). Seria possível utilizar um estilo puramente funcional, mas o grupo decidiu que seria mais fácil utilizar as possibilidades da linguagem e propor uma solução em estilo misto.

Ressaltamos que a estratégia de solução adotada permite resolver *puzzles* até bastante grandes, sendo que um teste foi realizado com um tabuleiro 10x10, no problema 117 do *site* Janko. Isto ocorre porque, ao calcular separadamente todos os movimentos válidos, evitamos a criação de novas versões do tabuleiro que serão colocadas em pilha, e mantemos somente as versões realmente essenciais para a tentativa e erro.

3 Detalhes sobre a execução do trabalho

3.1 Divisão de tarefas

A carga de implementação foi dividida de forma que ambos pudessem trabalhar independentemente, um responsável pelas funções relativas à checagem de que números são válidos em cada espaço vazio e o outro pelas funções relativas à resolução do puzzle através de *backtracking*.

Para facilitar o trabalho assíncrono entre as duas partes foi criado um repositório no Github para colocar os arquivos referentes ao trabalho. Houve comunicação de ambas as partes ao longo da implementação, de modo que erros que surgiram foram identificados e solucionados rapidamente. Desta forma, consideramos que a divisão de tarefas e cargas de trabalho foi satisfatória para ambos os membros.

3.2 Dificuldades

No geral, não encontramos grandes dificuldades na modelagem do problema, uma vez que adotamos basicamente a mesma estratégia de resolução utilizada no Trabalho 1. Contudo, a pequena quantidade de aulas voltadas a Common Lisp e a variedade de recursos sobre Lisp como um todo na internet dificultaram um pouco a implementação da solução.

À medida que os membros tiveram mais prática na linguagem, as dificuldades foram se reduzindo, e os recursos utilizados tornaram-se mais compreensíveis.