Relatório de Implementação do jogo Kakuro (em Scheme)

Alek Frohlich & Nicolas Goeldner, 26 de Outubro de 2019,

Executando o programa:

Detalhes para como configurar os casos de teste e como executar a aplicação se encontram no README do repositório.

Regras do jogo:

Cada tabuleiro de Kakuro consiste em uma grade em branco com dicas de soma em vários lugares. O objetivo é preencher todos os quadrados vazios usando os números de 1 a 9, para que a soma de cada bloco horizontal seja igual à pista à esquerda e a soma de cada bloco vertical seja igual à pista no topo. Além disso, nenhum número pode ser usado no mesmo bloco mais de uma vez. (tradução literal de: https://www.conceptispuzzles.com/index.aspx?uri=puzzle/kakuro/rules).

Relatório de implementação:

A implementação proposta busca solucionar um tabuleiro de Kakuro, ou em outras palavras, busca uma configuração no qual o tabuleiro não esteja inconsistente, testando possibilidades e realizando backtrack ao detectar uma configuração inconsistente.

```
;; Skip impossible board states, also propagate finished state.
(define (work pos w)
    (begin
        (write! pos w)
        (if (continue? pos w)
            (finished? pos)
            continue)))

;; Try one iteration of work.
(define (try pos w)
    (if (= w limit)
        unsolvable
        (if (solved? (work pos w))
            solved
            (try pos (+ w 1)))))

;; Solves Kakuro boards!
(define (solve pos)
    (try pos 1))
```

Figura 1. As principais funções do programa.

Na figura 1, podemos ver as principais funções que compõem a dada solução: **solve**, responsável por começar a iteração sobre a posição passada; **try**, responsável pela iteração entre os possíveis valores **w** que uma posição branca pode assumir (0-9) e pela propagação dos possíveis estados na árvore de computação do tabuleiro **[solved, unsolvable, continue]**; **work**, responsável por testar a configuração concebida pela última iteração de **try** sob o valor de **w**, decidindo se o ramo de computação deve **continue** ou se ele já falou, **unsolvable**. Caso **work** possa proceder (**continue?**) e, ao mesmo tempo, detecte um estado final (**finished?**), propaga o término de volta para a função **try**.

Estado inconsistente, neste contexto, significa um estado em que as regras do Kakuro estão feridas. As regras são:

1. Não podem haver elementos repetidos entre duas casas pretas na mesma coluna ou linha.
2. Uma linha ou uma coluna completa deve ter a soma de seus elementos igual a dica contida na casa preta que a guia.
3. Casas pretas guiam as casas brancas abaixo e/ou a sua direita. Podemos ver na figura 2 abaixo que as casas brancas de índice (1,1) e (1,2), as quais contém os elementos 9 e 8, respectivamente, têm seus valores ditados pelas casas pretas de índices (0,1), (0,2) e (1,0), no sentido que 9 e 8 tem que somar 17, 9 e 3 tem que somar 12 e 8,2,1,3 e 7 tem que somar 21.

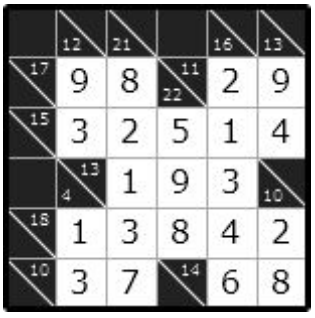

Figura 2. Um tabuleiro de Kakuro.

Testamos as regras imediatamente após a chamada de **write!** No procedimento **work** através da função **continue**. Nela, nem todas as condições são testadas todas as vezes: Como se pode ver pela figura 3, abaixo, só testamos se a linha/coluna atingiu a soma indicada pela casa guia assim que chegamos na última casa branca desta/daquela linha/coluna. Note que nas cláusulas que iniciam com **and**: Os testes de soma - **checkLine** e **checkCol** só são executados caso o programa esteja no último elemento da linha/coluna (**last_i?** e **last_j?**) ou uma quando uma casa preta é encontrada (**black?**).

```
;; Check if compute branch can continue.
(define (continue? pos w)
    (cond ((lineRepeats? (left pos) w) #f)
          ((colRepeats? (up pos) w) #f)
          ((and (or (last_j? pos) (and (not (last_j? pos)) (black?
(right pos)))) (not (checkLine pos 0))) #f)
          ((and (or (last_i? pos) (and (not (last_i? pos)) (black? (down
pos)))) (not (checkCol pos 0))) #f)
          (else #t)))
```

Figura 3. Teste de inconsistência.

Além das funções auxiliares que serão exploradas na última seção, a função **continue?** faz o uso de 4 testes:

1. Elemento repete na linha? (**lineRepeats?**)
2. Elemento repete na coluna? (**colRepeats?**)
3. Soma dos elementos horizontais conformante com a dica da casa preta à esquerda? (**checkLine)**
4. Soma dos elementos verticais conformante com a dica da casa preta acima? (**checkCol)**

As funções **checkLine**, **checkCol**, **lineRepeats?** e **colRepeats?** são triviais pois simplesmente recorrem pelas respectivas linhas/colunas para as quais foram invocados os teste, verificando suas condições. Podemos ver da figura 3 que chamamos **lineRepeats?** e **colRepeats?** nos índices anteriores a posição que estamos atualmente (**left** e **up**), fazemos isto para evitar testar se a casa que estamos é igual a ela mesma.

O terceiro e quarto teste podem, à primeira vista, parecer complicados, mas, na prática, só verificam se estamos visitando a última casa branca da linha/coluna. Se estivermos, procede testando se a soma dos elementos da linha/coluna está de acordo com a casa preta que as guia. Pode-se ver que invocamos as funções checkLine e checkCol passando 0 como o terceiro argumento, sum. Fazemos isso para indicar que a soma dos elementos da linha/coluna começa em zero, sendo então acrescida aos valores pertencentes à sequência de posições visitadas pela recursão.

A seguir, a implementação dos quatro testes de **continue?**

```
;; Check if line sums up to hint.
(define (checkLine pos sum)
    (if (white? pos)
        (checkLine (left pos) (+ sum (first pos)))
        (= sum (second pos))))

;; Check if column sums up to hint.
(define (checkCol pos sum)
    (if (white? pos)
        (checkCol (up pos) (+ sum (first pos)))
        (= sum (first pos))))

;; Check if element repeats on given line.
(define (lineRepeats? pos w)
    (cond ((black? pos)  #f)
          ((= (first pos) w) #t)
          (else (lineRepeats? (left pos) w))))
```

```
;; Check if element repeats on given column.
(define (colRepeats? pos w)
    (cond ((black? pos) #f)
          ((= (first pos) w) #t)
          (else (colRepeats? (up pos) w))))
```

Figura 4. Testes de **continue?**

Tendo entendido como é feito o teste de inconsistência, podemos concluir a explicação dos procedimentos **try** e **work**. Como já mencionado antes, **work** é responsável por identificar estados de aceitação. Para isso, faz uso do procedimento **finished?** Abaixo:

```
;; Test finish conditions. Calls next position to try.
(define (finished? pos)
    (define nextP (nextWhitePos (nextPos pos)))
    (if (and (lastPos? nextP) (or (last_j? pos) (black? last_pos)))
        solved
        (if (solved? (solve nextP))
            solved
            continue)))
```

Figura 5. Condição de término da recursão.

A função testa se o programa se encontra na última casa branca, se não for o caso, avança na árvore de computação invocando **solve** sobre a próxima casa branca. É importante lembrar que o programa só invoca **finished?** caso **continue?** tenha retornado **#t**, indicando que o tabuleiro está em um estado consistente. Portanto, se **finished?** retornar **solved**, temos garantia que o tabuleiro foi resolvido.

Por fim, resta a função **nextWhitePos**, que encontra a próxima posição branca a partir desta:

```
;; Returns next white position on board.
;; If there isn't one left, returns last_pos
;; instead.
(define (nextWhitePos pos)
    (if (> (car pos) last_i)
        last_pos
        (if (white? pos)
            pos
            (nextWhitePos (nextPos pos)))))
```

Figura 6. Busca de posições brancas.

A função recorre sobre as posições do tabuleiro, evitando casas pretas. Ela chama outra auxiliar, nextPos, que retorna o próximo índice do tabuleiro seguindo a ordem de visitação row-major. Caso estejamos na última casa branca ao invocarmos esta função, ela nos retornará a última posição do tabuleiro **last_pos**.

Estruturas de dados:

Para esta implementação optamos por utilizar a estrutura **vector** da linguagem **Scheme** porque possui tempos de busca e escrita constantes (**vector-ref** e **vector-set**). Indexamos a estrutura de maneira bidimensional, mas a armazenamos linearmente de forma a só precisar de duas camadas de **vector**: uma para o tabuleiro e outra para as informações de uma dada posição.

Cada posição do tabuleiro possui três campos: A sua cor (**white/black**) e seus campos **first** e **second**. **first,** em casas brancas, contém o número atualmente lá presente. Já em casas pretas, **first** representa a dica da coluna. O campo **second** só é utilizado para casas pretas e representa a dica da linha.

Abaixo segue a implementação da interface com a estrutura:

```
;; BEGIN VECTOR INTERFACE

(define (write! pos val)
    (vector-set! (vector-ref board (index pos)) 1 val))

(define (black? pos)
    (= (vector-ref (vector-ref board (index pos)) 0) black))

(define (white? pos)
    (= (vector-ref (vector-ref board (index pos)) 0) white))

(define (first pos)
    (vector-ref (vector-ref board (index pos)) 1))

(define (second pos)
    (vector-ref (vector-ref board (index pos)) 2))

;; END VECTOR INTERFACE
```

Figura 7. Interface **vector**.

Ponto de entrada:

Aproveitando que **Scheme** é uma linguagem de script, não declaramos nenhuma função **main**, nem algo do tipo. Simplesmente invocamos a função **solve** sobre a primeira casa branca do tabuleiro:

```
;; ENTRY POINT
(define nex (nextWhitePos(cons 0 0)))

(newline)

;; Solve the board
```

```
(if (solved? (solve nex))
    (display-board)
    (display "Unsolvable!"))
```

E depois imprimimos o estado final de **board** com a função **display-board**:

```
;; Print solved board
(define (display-board)
    (define (print-pos pos)
        (if (black? pos)
            (display "*")
            (display (first pos))))
    (define (iter pos)
            (if (= (car pos) height)
                (newline)
                (begin
                    (print-pos pos)
                    (if (last_j? pos)
                        (begin
                            (newline) (iter (nextPos pos)))
                        (iter (nextPos pos))))))
    (iter (cons 0 0)))
```

Por fim, nos restam as funções auxiliares.

Funções auxiliares:

As funções auxiliares abstraem problemas simples como encontrar as posições ao redor de uma dada posição (**down, right, up, left)**; Testar se dada posição é o último índice da linha/coluna (**last_i** e **last_j**) ou a última do tabuleiro (**lastPos?**); Por fim, também encontram a próxima posição em relação a posição atual (**nextPos**).

```
;; BEGIN AUXILIARES

;; Has the board been solved?
(define (solved? status)
    (= status solved))

;; Position up of pos.
(define (down pos)
    (cons (+ (car pos) 1) (cdr pos)))

;; Position right of pos.
(define (right pos)
```

```
    (cons (car pos) (+ (cdr pos) 1)))

;; Position down of pos.
(define (up pos)
    (cons (- (car pos) 1) (cdr pos)))

;; Position left of pos.
(define (left pos)
    (cons (car pos) (- (cdr pos) 1)))

;; Convert 2d position to linear
(define (index pos)
    (+ (* (car pos) width) (cdr pos)))

;; Last position on column?
(define (last_i? pos)
    (= (car pos) last_i))

;; Last position on line?
(define (last_j? pos)
    (= (cdr pos) last_j))

;; Next position on board.
(define (nextPos pos)
    (if (last_j? pos)
        (cons (+ (car pos) 1) 0)
        (cons (car pos) (+ (cdr pos) 1))))

;; Last position on board?
(define (lastPos? pos)
    (and (= (car pos) last_i) (= (cdr pos) last_j)))

;; END AUXILIARES
```

Figura 8. Funções auxiliares.

Comparação entre linguagens: Haskell vs Scheme

Embora **Haskell** possua um sistema de tipagem bem mais forte e interessante que o loop de avaliação e aplicação de **Scheme** (**eval-apply loop**), a segunda possui a vantagem de ser mais fácil de se expressar e, portanto, mais rápida/fácil de se desenvolver código para. Em questões de técnicas utilizadas, nenhuma das duas linguagens, pelo que sabemos, oferece uma maneira significantemente melhor do que a outra para aplicar backtracking. Na verdade, em relação a primeira linguagem utilizada, **C++**, as linguagens funcionais se saíram bem mal (em nosso ver), pois complicaram a solução sem deixá-la mais eficiente em termos de espaço e tempo. Para este tipo de problema preferimos linguagens imperativas.