Relatório - Prova II - Programação Funcional

Paradigmas de Programação (INE5416)

Gabriel Baiocchi de Sant'Anna e Robson Zagre Júnior

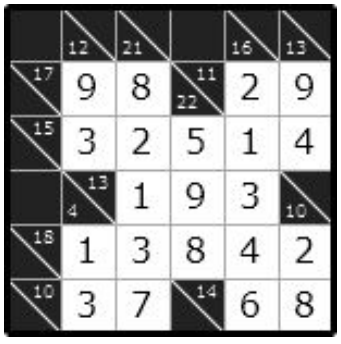
Enunciado

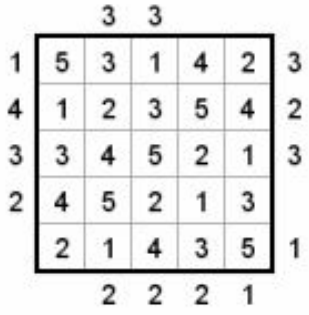
No trabalho desta prova utilizamos a linguagem **Scheme,** do paradigma de programação funcional, para resolver problemas através do método de *backtracking*. Foram implementados resolvedores para ambos os *puzzles* Wolkenkratzer e Kakuro**.**

Kakuro, também conhecido como *Cross Sums*, é um um quebra-cabeça que se assemelha à palavras cruzadas onde um certo quadro deve ter seus espaços em branco preenchidos com números de a 1 a 9. Existem no quadro algumas restrições que valem para a linha formada pelos espaços à direita da restrição e/ou na coluna formada pelos espaços abaixo da mesma. A restrição indica o valor que deve ser obtido ao somar os valores de todas as casas referentes à mesma. Não é permitido repetir números para atender uma dada restrição.

Já o jogo **Wolkenkratzer**, também conhecido como *Skyscrapers*, consiste em um tabuleiro quadrado de lado N com casas que devem ser preenchidas por números de 1 a N (algumas instâncias exigem valores de 0 a N-1) que não podem ser repetidos entre linhas ou entre colunas. Em torno do tabuleiro são colocadas algumas restrições que representam o número de prédios que poderiam ser vistos por um observador naquela posição que olhasse na direção do tabuleiro. Assim, cada número colocado em uma casa do tabuleiro representa um prédio daquela altura que encobre prédios vizinhos de menor altura em cada direção.

(Exemplos de instâncias resolvidas, respectivamente do Kakuro e do Wolkenkratzer)





Solução

Backtracking

Para resolver os *puzzles* empregamos um procedimento de alta ordem que efetua o *backtracking* sob um dado problema de maneira genérica; este utiliza *continuations* para efetuar cada passo da computação de maneira reversível, retornando a primeira solução que for encontrada ou falso quando as opções de resolução forem exauridas.

O algoritmo genérico recebe inicialmente duas continuações: a de sucesso, que é invocada passando a solução encontrada como argumento, e a de falha, que é invocada sobre nenhum argumento quando não for encontrada uma solução.

```
(lambda (solution) <...>) ;; continuacao de sucesso
(lambda () <...>)         ;; continuacao de falha
```

Além disso, o procedimento toma algumas funções que serão invocadas durante a sua execução, a mais complexa sendo a que tenta um caminho e possibilita voltar atrás nessa tentativa caso ela acabe não levando a uma solução válida.

```
(define (collapse try/else problem fail tip)
  <do-something-with-tip>
  (try/else <reduced-problem>
            (lambda ()
              <undo-that-something>
              (try/else <otherthings> fail))))
```

O primeiro passo é conferir se o espaço de possibilidades dado no problema é provavelmente válido, retornando falso se e somente se for possível concluir que a tentativa atual levou a um resultado que não é solução. Se for o caso (o procedimento concluiu que o problema é decididamente impossível com as possibilidades dadas), então a continuação de falha é invocada. Esta realizará o *backtrack* até uma tentativa anterior ou cessará o algoritmo quando todos os caminhos já tiverem sido tentados.

Em seguida, o algoritmo procura no espaço de possibilidades alguma ambiguidade, ou seja, se existem múltiplas formas de tentar resolvê-lo. Sendo assim, deve-se reduzir o problema colapsando algum conjunto de possibilidades através de uma tentativa que seguirá um ramo de computação (possivelmente arbitrário).

Tendo em mente que essa tentativa pode dar errado (ou seja, esse ramo não alcança uma solução), é construída uma nova continuação de falha que deve desfazer a redução anterior e continuar o algoritmo sabendo que aquela abordagem não deu certo. Assim o espaço de possibilidades é reduzido iterativamente.

Por fim, quando o espaço de possibilidades na atual iteração sobre o problema passou pelo teste de validação e não é ambíguo, basta retorná-lo imediatamente através da continuação de sucesso.

```
(define (solve problem possible? ambiguous? collapse)
  (call/cc (lambda (return)
    ;; retry eh um processo iterativo
    (let retry ((solutions problem)
                (backtrack (lambda () (return #f))))
          ;; se o problema eh decididamente invalido, backtrack
      (cond ((not (possible? solutions)) (backtrack))
            ;; se ainda houverem ambiguidades
            ((ambiguous? solutions) =>
               ;; tenta alguma delas, podendo voltar atrás
               (lambda (ambiguity)
                 (collapse retry solutions backtrack ambiguity)))
            ;; caso contrario, encontrou a solucao
            (else (return solutions)))))))
```

Dessa forma, os procedimentos específicos de cada problema se resumem a:

A. Representar o problema como um espaço de possíveis soluções.
B. Determinar se uma solução é possivelmente válida ou decididamente inválida.
C. Determinar se um espaço de possibilidades fornecido é ambíguo ou não.
D. Reduzir o espaço de possibilidades (tentando alguma delas) e prover uma forma de reverter essa redução e continuar tentando de outras maneiras.

Exemplo: DFS

O algoritmo genérico foi testado com o exemplo canônico de procurar um elemento nas folhas de uma árvore binária: efetuamos uma busca em profundidade na sub-árvore à esquerda e se não der certo voltamos atrás, partindo em seguida para a sub-árvore à direita e assim sucessivamente até encontrar a folha de interesse.

```
(define bin-labyrinth '(((LLL LLR) (LRL LRR)) ((RLL RLR) (RRL RRR))))

(define (tree-prune try root fail _)
  (let ((left (car root)) (right (cadr root)))
    (display "Bifurcacao: ") (display root) (newline)
```

```
    (display "Tentando pela esquerda...\n")
    (try left
         (lambda ()
           (display "Oops! De volta para ") (display root) (newline)
           (display "Tentando pela direita...\n")
           (try right fail)))))

(define (dfs tree node)
  (solve tree                    ;; arvore === espaco de possibilidades
         (lambda (possibility) ;; confere se ainda eh possivel
           (or (list? possibility) (equal? possibility node)))
         list?                   ;; testa se eh ambiguo
         tree-prune))            ;; o que fazer em cada nodo raiz
```

O algoritmo se comportou como esperado e a busca pelos caminhos da árvore através de *backtracking* foi efetuada com sucesso, como ilustrado pela saída do teste:

```
(dfs bin-labyrinth 'RLR)
Bifurcacao: (((LLL LLR) (LRL LRR)) ((RLL RLR) (RRL RRR)))
Tentando pela esquerda...
Bifurcacao: ((LLL LLR) (LRL LRR))
Tentando pela esquerda...
Bifurcacao: (LLL LLR)
Tentando pela esquerda...
Oops! De volta para (LLL LLR)
Tentando pela direita...
Oops! De volta para ((LLL LLR) (LRL LRR))
Tentando pela direita...
Bifurcacao: (LRL LRR)
Tentando pela esquerda...
Oops! De volta para (LRL LRR)
Tentando pela direita...
Oops! De volta para (((LLL LLR) (LRL LRR)) ((RLL RLR) (RRL RRR)))
Tentando pela direita...
Bifurcacao: ((RLL RLR) (RRL RRR))
Tentando pela esquerda...
Bifurcacao: (RLL RLR)
Tentando pela esquerda...
Oops! De volta para (RLL RLR)
Tentando pela direita...
RLR
```

Kakuro

Para atender ao mecanismo de *backtracking* genérico implementado, cada célula que possui o valor inicial de 0 no tabuleiro, ou seja, cada célula que deve ser completada, possui uma lista de 1 à 9, demonstrando os possíveis valores para essa posição. No entanto, já buscando melhorias na eficiência da solução do puzzle, as

células do tabuleiro não são preenchidas com possibilidades de 1 à 9, mas sim por meio de uma função que gera a *range* já delimitado com base na restrição imposta.

```
(define (pa-sum a1 an n)
  (/ (* (+ a1 an) n) 2))

;; make a pair (lo . hi) with the cell possibility bounds
(define (return-options restr n)
  (let* (
 (free (- n 1))
 (higher-ascending (min 9 (- restr (pa-sum 1 free free))))
 (lower-descending (max 1 (- restr (pa-sum 9 (- 9 (- free 1)) free)))))
    (cons (max 1 lower-descending)
          (min higher-ascending 9))))
```

Essa restrição se baseia no seguinte princípio. Preenchemos as células da restrição começando por 1 e incrementando até a (n-1) célula. Verifica-se qual é o somatório até o momento, e define como sendo o teto máximo de possibilidades, a diferença entre a restrição e tal somatório. O segundo passo é baseado no primeiro, porém possui a ideia de ordem decrescente. Preenchemos as células da restrição começando por 9 e decrementando até a (n-1) célula. Verifica-se qual é o somatório até o momento, e define como sendo a base das possibilidades, a diferença entre a restrição e tal somatório. Após realizar tais operações, é definido como sendo a base das possibilidades, o máximo valor entre 1 e base encontrada na segunda operação, e como sendo o teto, o valor mínimo entre 9 e o teto encontrado na primeira operação.

Com tais limitações no *range* de possibilidades, roda-se tal preenchimento com base nas restrições das linhas, preenchendo os valores 0 com as primeiras possibilidades. Em seguida, executa o mesmo processo para as restrições das colunas, porém agora, é feito uma intersecção entre os valores retornados pela restrição da coluna e o valor atual da célula, permitindo que apenas possua possibilidades válidas na restrição da linha e da coluna.

```
;; fill sequence with possibles values limiteds by restriction
(define (restriction-aware restr sequence)
  (let ((bounds (return-options restr (length sequence))))
    (let iter ((seq sequence))
      (if (null? seq) '()
          (let ((cell (car seq)))
            (if (pair? cell)
                (cons (intersect-bounds cell bounds) (iter (cdr seq)))
                (cons bounds (iter (cdr seq)))))))))

;; build range defined by the intersection of given bounds
(define (intersect-bounds a b)
```

```
  (range (max (car a) (car b))
         (min (cdr a) (cdr b))))
```

Para facilitar o processo de solucionar o puzzle, a lista de possibilidades de cada célula, agora já limitada pelas restrições, sofre um processo de *shuffle* pois sempre pegamos o primeiro item da lista. Com essa certa "aleatoriedade", as chances de se acertar o número mais cedo aumentam, diminuindo as tentativas inúteis no processo de *backtracking*.

```
(define (shuffle-kakuro list)
  (if (restriction? list) list (shuffle list)))

(define (set-kakuro n)
  (let ((kakuro (kakuro-ref n)))
    (restriction-fill! kakuro)
    (matrix-map shuffle-kakuro kakuro)))
```

Além disso, para agilizar o processo de *backtracking*, quando se é fixado um valor em uma célula, remove-se tal possibilidade da linha e da coluna a qual essa célula faz parte - tomando o cuidado para não sair da restrição referente àquela célula - diminuindo consideravelmente o tempo de validação.

```
;; fix a cell in the board with respect to the cell in given position
(define (fix-cell! board y x cell)

  (define (purge-kakuro-row! i j)
    (if (>= j (matrix-length board)) 'done
        (let ((others (matrix-ref board i j)))
          (if (restriction? others) 'done ;; stop on next restriction
              (begin
                (if (list? others)
                    (matrix-set! board i j (delete cell others)))
                (purge-kakuro-row! i (+ j 1)))))))

  (define (purge-kakuro-col! i j)
    (if (>= i (matrix-length board)) 'done
        (let ((others (matrix-ref board i j)))
          (if (restriction? others) 'done ;; stop on next restriction
              (begin
                (if (list? others)
                    (matrix-set! board i j (delete cell others)))
                (purge-kakuro-col! (+ i 1) j))))))

  (matrix-set! board y x cell) ;; fix that cell
  (purge-kakuro-row! y (+ x 1)) ;; horizontal purge
  (purge-kakuro-col! (+ y 1) x)) ;; vertical purge
```

O solver do kakuro retorna falso caso haja alguma célula nula, chamando novamente o *backtracking*. Caso haja ambiguidade ou caso as restrições de todas as colunas e linhas sejam validadas, o solver retorna true, continuando o *backtracking* colapsando a próxima opção, ou achando a solução do puzzle.

```
;; verify if kakuro is valid
(define (kakuro-solver k)
  ; no blank spaces
  (and (not (matrix-find-pos
              (lambda (i j) (null? (matrix-ref k i j))) k))
       ; and both directions check
       (kakuro-solver-row k 0)
       (kakuro-solver-col k 0)))

;; verify if kakuro a line of kakuro is solved
(define (kakuro-solver-seq seq get-restr)
  (if (null? seq) #t
      (let ((sum (get-restr (caar seq)))
            (cells (cdar seq)))
        (if (or (= sum 0)
                (any list? cells)
                (= sum (apply + cells)))
            (kakuro-solver-seq (cdr seq) get-restr)
            #f))))
```

Wolkenkratzer

Para este jogo, representamos o tabuleiro como uma matriz e cada sequência de restrições (superior, esquerda, inferior e direita) como uma lista de pares contendo o valor da restrição propriamente dito e o índice da linha ou coluna associada. Assim, o espaço de possibilidades é dado individualmente para cada casa do tabuleiro, podendo ser uma lista (os possíveis valores a serem preenchidos ali) ou um único valor (caso tenha sido fixado durante a resolução).

```
;; try some possibility, providing a way to backtrack
(define (consider-construction try board fail position)
  (let* ((i (car position)) (j (cadr position))
         (possibilities (matrix-ref board i j))
         (checkpoint (matrix-copy board))) ; save previous board
    (block-prune! board i j (car possibilities))
    (try board
         (lambda ()
           ; restore previous board and reduce possibility space
           (matrix-set! checkpoint i j (cdr possibilities))
           (try checkpoint fail)))))
```

```
;; find a position with an ambiguity, false if there isn't any
(define (find-amb-terrain brd)
  (matrix-find-pos (lambda (i j) (list? (matrix-ref brd i j))) brd))
```

Dessa forma, uma casa encontra-se em um estado ambíguo se for uma lista e colapsar essa ambiguidade se resume a escolher o primeiro elemento dessa lista de possibilidades e fixá-lo no tabuleiro. Tendo fixado um valor em uma posição específica podemos remover aquele valor dos espaços de possibilidade das casas na mesma linha e coluna. Ao voltar atrás no *backtracking*, basta restaurar a matriz do tabuleiro ao seu estado original e manter naquela lista de possibilidades todos os outros exceto o primeiro. Uma heurística aplicada foi a de embaralhar essas listas de possibilidades no início do algoritmo.

```
;; prune the board with respect to the cell in given position
(define (block-prune! board y x cell)
  ;; fix that cell
  (matrix-set! board y x cell)

  ;; remove it from the rest of the row
  (matrix-for-each-pos-in-row
    (lambda (i j)
      (let ((others (matrix-ref board i j)))
        (if (list? others)
            (matrix-set! board i j
              (filter (lambda (value) (not (= value cell))) others)))))
    board y)

  ;; as well as from the rest of the column
  (matrix-for-each-pos-in-col
    (lambda (i j)
      (let ((others (matrix-ref board i j)))
        (if (list? others)
            (matrix-set! board i j
              (filter (lambda (value) (not (= value cell))) others)))))
    board x))
```

Por fim, verificar se um tabuleiro ainda tem potencial para se tornar solução é feito em algumas etapas: se houver alguma posição com uma lista vazia significa que todas as possibilidades antes ali foram tentadas e deram errado, ou seja o tabuleiro atual é decididamente inválido; se a soma dos elementos não atende à alguma restrição (supondo que todos já tenham sido fixados) então essa tentativa deu errado; no último caso é possível concluir que ou existem ambiguidades não resolvidas ou todas foram fixadas e deram certo, retornando verdadeiro.

```
;; checks if skyscrapers seen comply to a restriction
(define (skyscrapers-check? r i get from step to)
  (let iter ((j from) (count 0) (tallest 0))
    (if (= j to)
        (= count r)
        (let ((curr (get i j)))
          (if (list? curr) 'skip
              (iter (step j 1)
                    (if (> curr tallest) (+ count 1) count)
                    (max tallest curr)))))))

;; puzzle solver
(define (wolkenkratzer n upper left bottom right lo hi)
  ;; check if a board stands candidate to solve the problem
  (define (may-allow? board)
    ;; no blank spaces
    (if (matrix-find-pos (lambda (i j)
                           (null? (matrix-ref board i j)))
                         board)
        #f
        ;; respects all of the puzzle's constraints
        (and (every (lambda (cnstr)
                      (let ((restr (car cnstr)) (row (cdr cnstr)))
                        (skyscrapers-check?
                          restr row
                          (lambda (i j) (matrix-ref board i j))
                          0 + n)))
                    left)
             (every (lambda (cnstr)
                      (let ((restr (car cnstr)) (row (cdr cnstr)))
                        (skyscrapers-check?
                          restr row
                          (lambda (i j) (matrix-ref board i j))
                          (- n 1) - -1)))
                    right)
             (every (lambda (cnstr)
                      (let ((restr (car cnstr)) (col (cdr cnstr)))
                        (skyscrapers-check?
                          restr col
                          (lambda (i j) (matrix-ref board j i))
                          (- n 1) - -1)))
                    bottom)
             (every (lambda (cnstr)
                      (let ((restr (car cnstr)) (col (cdr cnstr)))
                        (skyscrapers-check?
                          restr col
                          (lambda (i j) (matrix-ref board j i))
                          0 + n)))
                    upper))))
  ;; actually solving it
  (solve (matrix-map shuffle (make-matrix n n (range lo hi)))
         may-allow? find-amb-terrain consider-construction))
```

Utilização

Kakuro

O usuário poderá informar a entrada no próprio arquivo kakuro.scm, na secção "BOARDS". Há uma função denominada kakuro-ref a qual recebe um número n e retorna um tabuleiro do kakuro, sendo suas céulas em branco representadas por 0, e suas restrições de somas sendo representadas por *,(r col row)* onde *col* é a soma que tal coluna deve possuir, assumindo 0 caso não informado, e *row* é a soma que tal linha deve possuir, também assumindo 0 caso não informado.

O usuário poderá realizar o teste da solução do kakuro chamando a função solve-kakuro? a qual recebe um número n que é o número do tabuleiro o qual deseja-se buscar uma solução. A função retorna no display: "Impossible", caso não consiga encontrar uma resposta válida; ou o tabuleiro preenchido com as opções que solucionam o mesmo.

Wolkenkratzer

Aqui, pela homogeneidade dos tabuleiros do Wolkenkratzer, incluímos entrada de usuário das restrições e da altura máxima dos prédios. Quando essa altura for igual ao tamanho N, significa que são permitidos valores de 1 a N; se for N-1 significa que essa instância do *puzzle* exige casas vazias, ou seja, de 0 a N-1. Por exemplo, as restrições no tabuleiro dado na figura no início deste relatório poderia ser descrito como:

```
(0 3 3 0 0)
(1 4 3 2 0)
(0 2 2 2 1)
(3 2 3 0 1)
     5
```

Também possível descrever as entradas nesse formato em um arquivo de texto e jogá-lo (via *pipes*) à entrada do programa. Por exemplo, supondo que o exemplo anterior estivesse no arquivo *res/wolkenkratzer-5x5-janko.txt* poderíamos testá-lo com:

```
$ guile wolkenkratzer.scm < res/wolkenkratzer-5x5-janko.txt
```

Conclusão

Na linguagem Haskell foi observada uma maior facilidade na implementação de diferentes passos bases das funções, especialmente quando há vários casos diferentes mas com respostas simples e básicas. Já em Scheme, a implementação foi agilizada devido a não preocupação com tipos de variáveis e algumas facilidades que a linguagem proporciona, notavelmente os prints na tela em um momento qualquer do processamento (muito úteis para debugar o programa) e possibilidade da implementação de código sequencial.

Uma das maiores desvantagens de Scheme foi a necessidade de tornar homogênea as estruturas das matrizes (que diferem entre implementações) e ter que implementar alguns procedimentos úteis para a sua manipulação. Entretanto, esse mesmo contratempo foi tido também com Haskell. Por outro lado, a implementação do *backtracking* na linguagem Scheme foi mais fácil e eficiente e tomou-se proveito disso para torná-la genérica e reutilizável, possibilitando a implementação de ambos os *puzzles* graças à tipagem dinâmica. Além disso o suporte à continuações de primeira ordem possibilitou cortar processamento desnecessário em algumas instâncias. Dessa forma, concluímos que a linguagem Scheme nessa comparação mostrou ser mais adequada para a implementação e resolução dos problemas propostos.