Matheus Henrique Schaly – 18200436

11-05-2020

Straights – Scheme

1. Como usar o programa

O tabuleiro utiliza apenas uma matriz. Cada elemento da matriz é uma lista contendo dois elementos. O primeiro elemento é o valor do campo, o segundo elemento é a cor do campo, onde 1 simboliza um campo preto e 0 um campo branco.

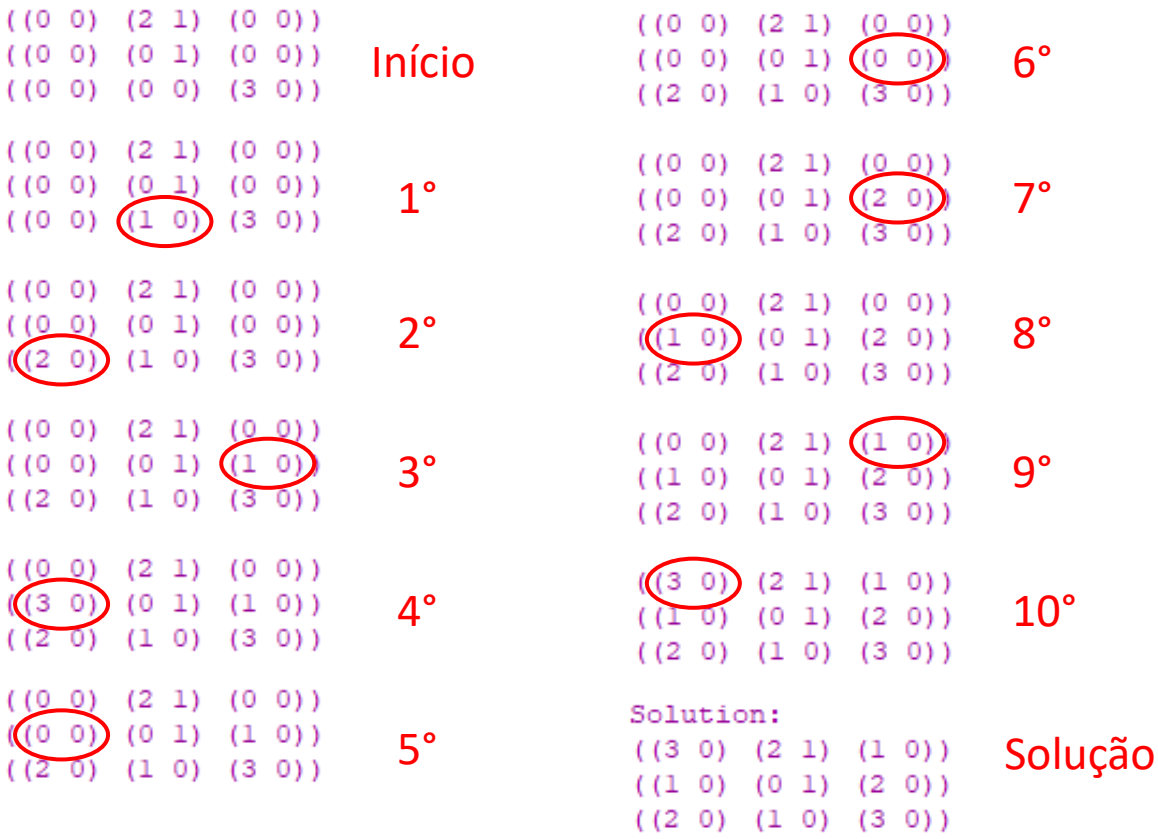
Há seis exemplos de tabuleiros já criados no código fonte. Para fornecer um novo input ao programa o usuário precisa criar ou alterar uma das definições diretamente no código fonte.

Para executar o programa, altere a linha "(define t t6)" para "(define t *x*)" onde *x* é o nome do tabuleiro já criado ou que o usuário criou.

O output é fornecido por um *display* no código. Sendo assim, o output pode ser visualizado a partir do software que chamou o programa após a string "*Solution:*" como pode ser na seção 2 abaixo. Observe que os números 0 simbolizam ausência de valor pois o campo é de cor preta e não possuía valor prévio.

2. Solução

O problema foi resolvido através do uso de recursividade e backtracking, como pode ser visto na imagem abaixo. Note que na 1° iteração do problema, há apenas uma possibilidade de valor, valor 1, que pode ser inserido no campo linha 1, coluna 2. Portanto tal valor é inserido (não há outra escolha). Ao chegar na iteração 4°, o algoritmo nota que não há valor possível para ser inserido no campo linha 0, coluna 2. Portanto, o algoritmo retorna para o último campo que ele realizou uma escolha, desfazendo todas as suas escolhas durante seu retorno. Ao retornar ao último campo que ele possuía mais de uma opção para o campo, ele abandona a primeira escolha para tal campo (pois tão escolha não deu certo) e seleciona a segunda escolha. Tal campo que possuía mais de uma possibilidade é o campo linha 1 coluna 2, como pode ser visto na 7° iteração. Sendo assim, ele substitui a opção de valor *1* pela opção de valor *2*, e retorna ao funcionamento habitual. O algoritmo continua realizando recursividade e backtracking até encontrar uma solução.



Para encontrar as possibilidades de cada campo, checa-se o seguinte:

1. Todos os valores das linhas devem ser distintos. Todos os valores das colunas devem ser distintos.
2. Todos os valores das tiras de campos brancos devem seguir uma sequência válida tanto nas linhas quanto nas colunas, que foi realizado da seguinte forma tanto para as linhas quanto para as colunas:
    a. Encontra-se a tira branca e seus valores.
    b. Encontra-se o valor máximo da tira (max).
    c. Encontra-se o valor mínimo da tira (min).
    d. Cria-se uma lista que varia de *max – length (tira branca) + 1* até *min + length (tira branca) - 1)*.
3. Fazendo-se a intersecção dessas 4 listas, obtém-se os valores possíveis de serem preenchidos em um determinado campo.

```
; Gets all the possibilities that can be inserted into position row index (ri) and row column (rc)
(define (get-poss ri ci)
  (let* [(r (get-row t ri))
         (row-poss (get-list-poss r))1
         (c (get-col t ci))
         (col-poss (get-list-poss c))1
         (intersection-1 (intersection row-poss col-poss))
         (row-strip-poss (get-strip-poss r ci))2
         (intersection-2 (intersection row-strip-poss intersection-1))
         (col-strip-poss (get-strip-poss c ri))2
         (final_intersection (intersection col-strip-poss intersection-2))]3
  final_intersection))
```

```
; Finds a white strip
(define (get-strip rc i)2a
  (let* [(prev (get-prev rc (- i 1)))
         (next (get-next rc (+ i 1)))
         (strip-length (+ (+ (length prev) (length next)) 1))
         (union-list (union prev next))]
    (cond
      [(= 0 (length (remove-zeros union-list)))
       (cons all-poss (length all-poss))]
      [else
       (let [(union-list-no-zero (remove-zeros union-list))]
         (cons union-list-no-zero strip-length))])))

; Uses the formula max - N + 1 to get the lower bound of possibilities
(define (get-min-seq max N)2d
  (cond
    [(< (+ (- max N) 1) 1)
     1]
    [else
     (+ (- max N) 1)]))

; Uses the formula min + N - 1 to get the upper bound of possibilities
(define (get-max-seq min N)2d
  (cond
    [(> (- (+ min N) 1) (length all-poss))
     (length all-poss)]
    [else
     (- (+ min N) 1)]))

; Gets the possibilities of a white strip
(define (get-sequence strip)
  (let* [(max (get-max-min (car strip) >)) 2b
         (min (get-max-min (car strip) <)) 2c
         (min-seq (get-min-seq max (cdr strip)))2d
         (max-seq (get-max-seq min (cdr strip)))2d
         (poss (range min-seq max-seq))]2d
    poss))
```

3. Dificuldades

As dificuldades e soluções foram:

1. Como representar o tabuleiro: A solução adotada foi utilizar uma única matriz e usar listas de dois elementos para representar cada campo. As operações *car* e *cdr* facilitam lidar com listas, especialmente com listas de dois elementos.
2. Como encontrar tiras em branco: A solução adotada foi, dado um campo, usar uma função para encontrar os elementos antes desse campo até a linha (ou coluna) acabar ou até encontrar um campo preto; e outra função para encontrar os elementos após esse campo até a linha (ou coluna) acabar ou até encontrar um campo preto. E, em seguida, unir os elementos encontrados antes e depois do campo.
3. Como identificar uma sequência valida: A solução adotada foi a fórmula descrita no algoritmo da seção 2, especificamente na parte 2d do algoritmo descrito.
4. Como implementar a recursividade: A solução adotada foi utilizar três diferentes loops. Um para percorrer as linhas, outro para percorrer os elementos da linha e o último para percorrer as possibilidades de valores de cada campo.

4. Comparação com Haskell

Similaridades e diferenças:

- Scheme e Haskell são bem similares, em Scheme usa-se *lets* ou *lambdas*, em Haskell usa-se *where*. Em Scheme usa-se *cond* and *ifs*, em Haskell usa-se guardas. As duas linguages usam muita recursividade.
- A IDE que utilizei para Scheme (DrRacket) avisava-me de erros enquanto eu digitava o código. Já no Haskell, eu sabia dos erros apenas quando tentava carregar o programa.
- Haskell é mais fácil de ler e entender. Em Haskell é opcional o uso de *prefix*, podendo-se usar *infix*. Em Scheme, é obrigatório o uso de *prefix*. Além disso, tudo em Scheme usa parênteses, as vezes há tantos parênteses que se torna difícil de entender onde ele começou e onde ele deve acabar.
- Haskell parece oferecer mais built-ins para listas.