Universidade Federal de Santa Catarina
Relatório Paradigmas T1
Arthur Philippi Bianco (17203358)
Ezequiel Pilcsuk da Rosa (17203126)

1. Descrição do problema:

O jogo escolhido foi o Wolkenkratzer, que consiste em um *puzzle* onde:

1. As células podem conter números de zero à um máximo valor dado.
2. Cada linha e coluna contém no máximo uma instância dos números de zero ao valor dado.
3. As bordas das colunas e linhas podem conter um número referente à quantidade de prédios visíveis da posição. Um prédio é visível somente se não há prédios mais altos na frente dele. O objetivo é preencher todas as posições do Wolkenkratzer obedecendo às bordas e à regra 2.

O problema consiste em um algoritmo que dado um jogo vazio, retorne sua solução.

2. Solução do problema

Para melhor representação do problema foi adotado uma estrutura de dados chamada Linha, que contém um valor inteiro, uma lista de inteiros e outro valor inteiro, representando o número de prédios a ser visto pela esquerda, o tamanho dos prédios em ordem e o número de prédios a serem vistos pela direita, respectivamente. Outra estrutura usada para facilitar o entendimento foi a estrutura Game, que contém uma lista de Linha que representam o jogo. O algoritmo usa de dois valores que devem ser alterados de acordo com o jogo que está sendo solucionado, que são puzzleSize que é referente ao número de linhas do puzzle, e maxValue, que representa o maior valor que deve aparecer na solução do puzzle, caso puzzleSize == maxValue, significa que a solução do puzzle não possui zeros. Para representar também as colunas, a estrutura Game possui duas vezes a quantidade de linhas do jogo, assim, as linhas a partir da metade representam as colunas do jogo. As definições são:

data Linha = L Int [Int] Int *(representada no código como: (L x linha y)*
e
data Game = G [Linha] *(representado no código como: (G game)*

Algumas das funções de verificação de corretude foram utilizadas simples listas para representar as linhas. Alguns exemplos das implementações das principais funções de checagem são:

- Função que dada uma lista conta quantos prédios são vistos a partir da esquerda.

```
nPredios :: [Int] -> Int
nPredios [] = 0
nPredios a = nPrediosOP a 0
        where   nPrediosOP [ ] _ = 0
             nPrediosOP (a:b) atualMaior | (a > atualMaior) = 1 + (nPrediosOP b a)
                                         | otherwise = nPrediosOP b atualMaior
```

- Função que verifica se existe repetição de elemento em uma linha.

```
repeatsElement :: Linha -> Bool
repeatsElement (L x [] y) = False
repeatsElement (L x (a:b) y) | (elem a b) = True
                             | otherwise = repeatsElement (L x b y)
```

- Assim, para a checagem da corretude de uma linha a função é simples, primeiro é aplicada nPredios a lista, e comparado ao valor da esquerda, depois é aplicada nPredios na lista reversa e comparado com o valor a direita, então é verificado a não repetição de elemento na linha, caso todas as checagens retorne Verdadeiro, a linha é uma linha válida.
-

```
verificaLinhaValida :: Linha -> Bool
verificaLinhaValida (L x a y) = (verificaDir x a)  && (verificaDir y (reverse $ a))
```

Um jogo é correto se todas suas linhas são corretas, inclusive as que representam as colunas.

Para a solução do jogo foi adotada uma estratégia de exaustão pro backtracking, foi desenvolvida uma função que dada uma Linha não preenchida, retorna todas as possíveis configurações de soluções dadas as exigências de prédios da linha. Assim, o backtracking pode ser feito sobre cada linha, e não sobre cada elemento, assim, conforme cresce o número de exigências de prédios vistos no puzzle, mais rápida é a solução do mesmo.

- Como existe duplicação de dados uma vez, o algoritmo de backtracking só é aplicado sobre a primeira metade da lista de Linha do jogo, que representam as linhas, e as Linhas restantes que representam as colunas são preenchidas com as informações das linhas usando a função. A

função que preenche a Linha preenchida que representa a Coluna n é definida como:

```
fillColumn :: Int -> Game -> Linha
fillColumn n (G a) =
(L (getX (getLinhaG (n+puzzleSize) (G a))) (operator n [getValues (getLinhaG i (G a)) | i <- [1..puzzleSize]])  (getY (getLinhaG (n+puzzleSize) (G a))))
    where operator n [] = []
          operator n (a:b) = [(a!!(n-1))]++(operator n b)
```

- A função que resolve o jogo é definida como:

```
solveGame :: Game -> Int -> IO (Maybe Game)
solveGame thisGame n
  | (n == (puzzleSize+1)) = do
    if (verificaCorretude (fillAllColumns thisGame))
        then printGame thisGame >> return (Just thisGame)
    else return Nothing
  | otherwise = do v <- return (getLinhaG n thisGame)
            case v of
                (L x [] y) -> gameBackTrack thisGame n (getOptions v)
                _  -> solveGame thisGame (n+1)
```

Para solucionar um jogo ela é chamada sobre o Game e 1, referente a linha 1, a partir de onde o algoritmo tenta solucionar. Primeiro o algoritmo faz uma checagem se já terminou, que seria quando chegou na linha (puzzleSize+1), se terminou verifica se o jogo é solução, se for solução mostra na tela. Caso não seja solução não mostra nada. Caso o número da linha seja menor ou igual a puzzleSize ele verifica se a linha correspondente está preenchida, se estiver, vai para a próxima linha, se não estiver, calcula todas as possíveis soluções da linha chamando o backTrack passando o jogo, a linha atual e o resultado de getOptions da linha, que filtra entre todas as possíveis configurações de uma linha sem restrições, deixando somente as que solucionam as que satisfazem as restrições de prédios vistos da linha.

- A definições de getOptions e de allPossibleSolutions (função usada em getOptions que retorna todas as possíveis soluções de uma linha sem restrições de prédios) são:

```
allPossibleOptions :: [[Int]]
allPossibleOptions =
    | (puzzleSize == maxValue) = permutations [1..maxValue]
```

```
    | otherwise = excludeEquals (allPermutations (includeZeros (powerSet [1..maxValue])))

getOptions :: Linha -> [[Int]]
getOptions (L x [] y) = [a | a <- allPossibleOptions, verificaValido (L x a y)]
getOptions (L _ a _) = []
```

- O algoritmo responsável pelo backTracking, que tenta uma solução de uma linha e tenta solucionar o jogo a partir dela, depois desfaz sua alteração e tenta chama a si mesmo com o restante das opções de solução da linha, é implementado da seguinte maneira:

```
gameBackTrack :: Game -> Int -> [[Int]] -> IO (Maybe Game)
gameBackTrack thisGame n [] = return Nothing
gameBackTrack thisGame n (v:vs) = do
  solveGame (writeLineG thisGame (L lineX v lineY) n) n
  gameBackTrack (writeLineG thisGame (L lineX [] lineY) n) n vs
    where lineX = getLineX (getLinhaG n thisGame)
          lineY = getLineY (getLinhaG n thisGame)
```

2.1 Paralelização da solução

Descobrimos também que a complexidade de um algoritmo de backtracking pode ser proibitiva para puzzles de tamanhos maiores do que 5, especialmente se contém zeros na solução, o que aumenta significativamente a quantidade de possíveis soluções por linha, assim, tentamos implementar a paralelização das tentativas do solucionador, e também implementamos uma versão que consegue solucionar o jogo bem mais rapidamente utilizando dos outros núcleos do processador. Utilizamos a biblioteca Control.Concurrent e fizemos algumas alterações nas funções solveGame e backTrack, para possibilitar uma implementação mais simples, visto que estávamos aprendendo a utilizar paralelismo em Haskell.

- A implementação do solucionador do jogo utilizando paralelismo ficou bem semelhante a versão sequencial, na função solveGame só foi necessário alterar seus retorno, para isso não é possível não printar nada ao encontrar uma solução incorreta, assim decidimos mostrar um . para cada solução incorreta encontrada:

solveGame :: Game -> Int -> IO ()

```
solveGame thisGame n
  | (n == (puzzleSize+1)) = do
    if (verificaCorretude (fillAllColumns thisGame))
        then printGame thisGame
    else putChar('.')
  | otherwise = do v <- return (getLinhaG n thisGame)
                   case v of
                       (L x [] y) -> gameBackTrack thisGame n (getOptions v)
                       _  -> solveGame thisGame (n+1)
```

- Na função de backTrack que foi efetivamente implementada a paralelização, chamando uma nova thread para tentar solucionar o jogo a partir da opção tentada. Foi implementada uma sincronização básica onde a thread pai espera o retorno dos filhos para finalizar, através do retorno de uma MVar ao final da execução da thread, caso contrário threads filhas podem ser encerradas sem terminar, comprometendo a checagem de todas as possíveis configurações.

```
gameBackTrack :: Game -> Int -> [[Int]] -> IO ()
gameBackTrack thisGame n [] = putChar('.')
gameBackTrack thisGame n (v:vs) = do
  children <- myForkIO(do
    solveGame (writeLineG thisGame (L lineX v lineY) n) n
    gameBackTrack (writeLineG thisGame (L lineX [] lineY) n) n vs)
  takeMVar children
    where lineX = getX (getLinhaG n thisGame)
          lineY = getY (getLinhaG n thisGame)
```

Como utilizamos bibliotecas que possivelmente não estarão disponíveis no teste do código, decidimos manter a solução sequencial como principal, por ser mais limpa, sem os retornos dos pontos para as soluções erradas.

Para entrada de novos puzzles o usuário deve criar uma variável do tipo game, com a lista de Linha correspondentes e chamar a função solveGame passando o jogo e o número 1. Lembrar também de alterar as variáveis maxValue e puzzleSize com os valores corretos.

3. Exemplo de entrada do usuário:

- Exemplo de declaração do puzzle referente a:
  (Para solucionar esse puzzle maxValue = puzzleSize = 5)

" 0 3 3 0 0 "
"1[5,3,1,4,2]3"
"4[1,2,3,5,4]2"
"3[3,4,5,2,1]3"
"2[4,5,2,1,3]0"
"0[2,1,4,3,5]1"
" 0 2 2 2 1 "

myCorrectGame :: Game
myCorrectGame = (G [(L 1 [5,3,1,4,2] 3), (L 4 [1,2,3,5,4] 2), (L 3 [3,4,5,2,1] 3), (L 2 [4,5,2,1,3] 0), (L 0 [2,1,4,3,5] 1), (L 0 [5,1,3,4,2] 0), (L 3 [3,2,4,5,1] 2) , (L 3 [1,3,5,2,4] 2), (L 0 [4,5,2,1,3] 2), (L 0 [2,4,1,3,5] 1)])

myEmptyGame :: Game
myEmptyGame = (G [(L 1 [] 3), (L 4 [] 2), (L 3 [] 3), (L 2 [] 0), (L 0 [] 1), (L 0 [] 0), (L 3 [] 2) , (L 3 [] 2), (L 0 [] 2), (L 0 [] 1)])

4. Dificuldades encontradas:

Uma das principais dificuldades encontradas na implementação do solucionador foi a adaptação dos retornos dos Monads possibilitando o retorno de todas as soluções possíveis, pois inicialmente pensamos que o algoritmo de backTracking poderia retornar um tipo diferente do solveGame, mas após pesquisar melhor, foi compreendida a incompatibilidade dos retornos.

A adaptação de um algoritmo de backTracking, que estávamos acostumados em fazer em linguagens imperativas, de forma recursiva se mostrou um empecilho, mas, no fim, o uso dos Monads, que possibilita criação de código sequencial, facilitou a implementação de forma similar a implementações em outras linguagens.

Outra dificuldade que foi subestimada ao começar a implementação do algoritmo foi a escassez de informação sobre Haskell quando comparado a linguagens mais utilizadas, que dificulta a pesquisa e complica a solução de problemas mesmo que simples.