**Universidade Federal de Santa Catarina**
**Relatório Paradigmas T3**
**Arthur Philippi Bianco (17203358)**
**Ezequiel Pilcsuk da Rosa (17203126)**

## 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.

## Itens necessários

Programação de restrições é dividida em duas etapas, a modelagem do problema e a busca por solução. A parte relacionada a modelagem é feita definindo um domínio em que a solução se encontra, assim, encontra um conjunto de relações entre os componentes da solução, sem nomeá-los, sem perda de generalidade.

Exemplo de restrição em que todas as linhas da matriz não podem ter elementos repetidos:

```
maplist(all_distinct,Matrix)
```

A busca, então, se trata da nomeação de variáveis que satisfazem as relações decorrentes das restrições impostas.

Exemplo de busca, onde são escolhidos elementos que satisfaçam restrições impostas:

**maplist(label,Matrix)**

A utilização de programação de restrição trouxe diversas vantagens, no quesito modelagem, diminuiu significativamente o número de linhas no código, ao utilizar funções dispostas na biblioteca, tornando o código muito

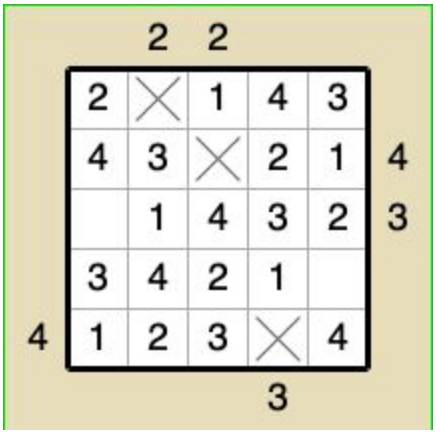
mais legível e próximo de uma descrição de solução em linguagem natural. No quesito performance observa-se grande ganho, visto que ao utilizar de representações dos problemas em grafos, a biblioteca dispõe de algoritmos muito mais eficientes, sem necessidade de exaurir todas as possíveis combinações. A maior desvantagem encontrada no uso de programação de restrições foi a perda da facilidade da depuração do código, visto que em qualquer chamada que utilizasse algum elemento da biblioteca, centenas de sub chamadas eram geradas.

O paradigma lógico trabalha com predicados e asserções, em que funções são executadas utilizando a base de crenças para restringir suas tentativas e chegar no resultado mais rapidamente. Por outro lado, nossa solução no paradigma de programação funcional, procura tentar todas as possibilidades retirando os casos em que não são válidos. Em geral foram parecidas as implementações de solucionadores de wolkenkratzer, tanto em prolog quanto em haskell, no sentido em que tentam-se todas as possibilidades. A maior diferença está na necessidade ou não da implementação do backtracking que não foi necessária em Prolog, mas foi em Haskell.

## Exemplo de entrada do usuário:

- Exemplo de declaração de puzzle:
  (Para solucionar esse puzzle maxValue = 4 e puzzleSize = 5)

```
problem(218,Mx,Lr,Ud) :-
      Mx = [[_,_,_,_,_],
            [_,_,_,_,_],
            [_,_,_,_,_],
            [_,_,_,_,_],
            [_,_,_,_,_]],
      Lr = [[0,0],
            [0,4],
            [0,3],
            [0,0],
            [4,0]],
      Ud = [[0,0],
            [2,0],
            [2,0],
            [0,3],
            [0,0]].
```




A descrição em Prolog a esquerda se refere ao puzzle a direita em que:

**Lr** são as colunas esquerda e direita que informam a altura vista.
**Ud** são as linhas cima e baixo que informam a altura vista.

Para solucionar o puzzle, realiza-se uma consulta assim:

```
problem(218, M,X,Y),
wolkenkratzer(M,X,Y),
concurrent_maplist(label, M),
concurrent_maplist(portray_clause, M).
```

## Dificuldades encontradas:

Uma das principais dificuldades encontradas na implementação do solucionador foi a impossibilidade de se monitorar a execução do programa com trace, uma vez que a biblioteca **clpfd** realiza uma quantidade exorbitante de chamadas de outras funções para qualquer utilidade da biltioteca, até mesmo um simples 2 #= X -1.

Uma das facilidades que encontramos foi, a desnecessidade de implementar funções bastante complexas que realizem o backtrack propriamente dito ou outras. Esses tipos de função Prolog tem pronto ou se tornam triviais com outras funções ja prontas do Prolog.

Nossa solução, inicialmente, ficou bastante parecida com a solução original em haskell no sentido em que tentávamos todas as soluções, utilizando pouco a biblioteca ***clpfd.*** Para podermos utilizar a biblioteca melhor, modificamos a função para que: dada uma quantidade de prédios, ela, utilizando as restrições da biblioteca, encontra uma lista com todas possibilidades de linhas que satisfazem a altura necessária. Desta maneira o tempo de soluçaão de um puzzle 5x5 aumentou de minutos para poucos segundos.