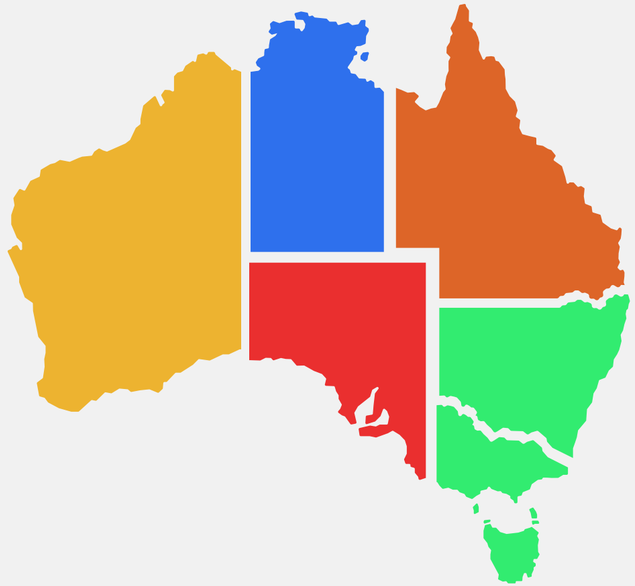
Introdução à Inteligência Artificial (9918/31)

COLORAÇÃO DE MAPAS

Problemas de Satisfação de Restrições



Gabriel Saraiva R.A.: 129145

Olga Maria R.A.: 130002

Yasser Farid R.A.: 129706

Professor Doutor Wagner Igarashi

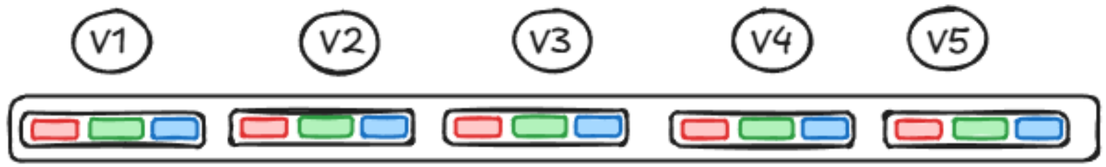
Índice

Introdução

O problema de coloração de mapas é um de Problema de Satisfação de Restrições (CSP).

Ele consiste em atribuir cores diferentes a regiões vizinhas de um mapa, de forma que nenhuma região que compartilhe uma fronteira tenha a mesma cor.

Modelagem



Variáveis:
V = {V1,V2,V3,V4,V5}

Domínio:
Di = {Vermelho, verde, azul} para todo i em V

Restrições:
Regiões vizinhas não podem ter a mesma cor

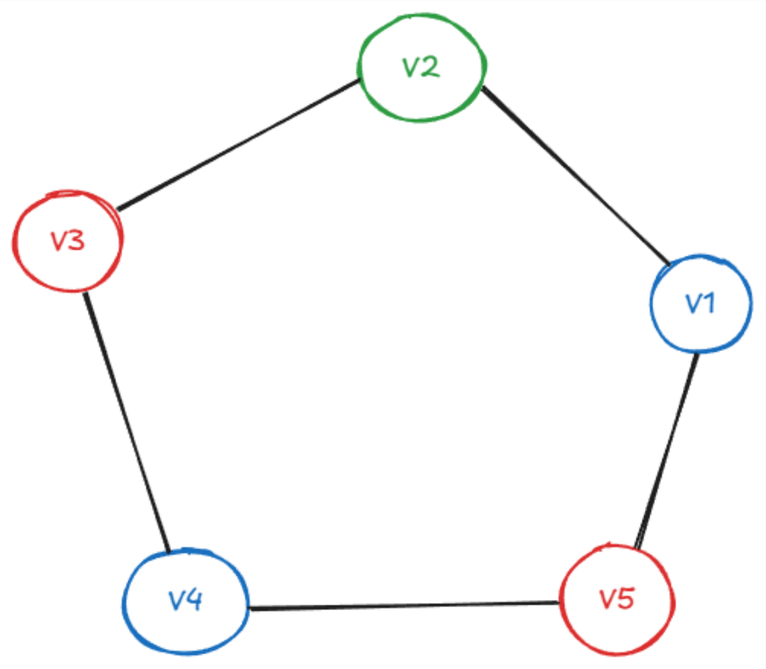
C = {V1 ≠ V2, V1 ≠ V3, V1 ≠ V5,
V2 ≠ V3, V3 ≠ V4, V5 ≠ V3}

Instância Fácil

Ciclo simples de 5 nós


```
{
  "variaveis": ["V1", "V2", "V3", "V4", "V5"],
  "dominios": {
    "V1": ["vermelho", "verde", "azul"],
    "V2": ["vermelho", "verde", "azul"],
    "V3": ["vermelho", "verde", "azul"],
    "V4": ["vermelho", "verde", "azul"],
    "V5": ["vermelho", "verde", "azul"]
  },
  "vizinhos": {
    "V1": ["V2", "V5"],
    "V2": ["V1", "V3"],
    "V3": ["V2", "V4"],
    "V4": ["V3", "V5"],
    "V5": ["V4", "V1"]
  },
```


Grafo de Restrições

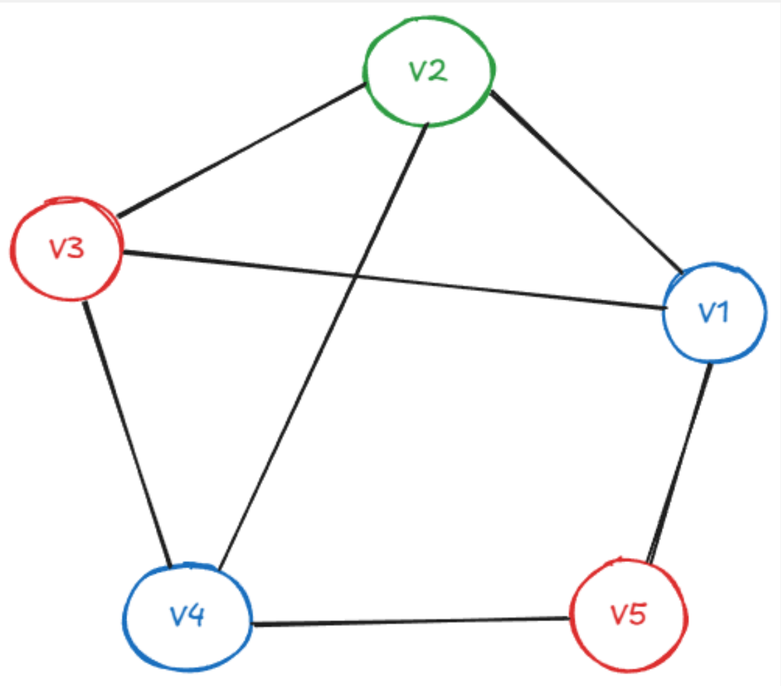


Instância Média

Diagonais adicionadas


```
{
  "variaveis": ["V1", "V2", "V3", "V4", "V5"],
  "dominios": {
    "V1": ["vermelho", "verde", "azul"],
    "V2": ["vermelho", "verde", "azul"],
    "V3": ["vermelho", "verde", "azul"],
    "V4": ["vermelho", "verde", "azul"],
    "V5": ["vermelho", "verde", "azul"]
  },
  "vizinhos": {
    "V1": ["V2", "V5", "V3"],
    "V2": ["V1", "V3", "V4"],
    "V3": ["V2", "V4", "V1"],
    "V4": ["V3", "V5", "V2"],
    "V5": ["V4", "V1"]
  },
```


Grafo de Restrições



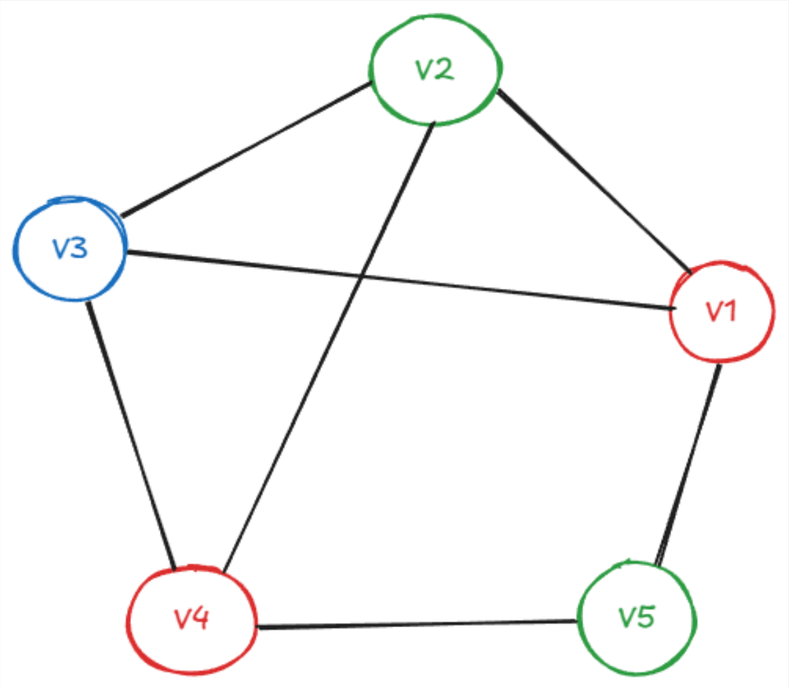
Adição das Diagonais {V1, V3} e {V2, V4}

Instância Difícil

Restrição dos domínios


```
{
 "variaveis": ["V1", "V2", "V3", "V4", "V5"],
 "dominios": {
  "V1": ["vermelho", "verde", "azul"],
  "V2": ["vermelho", "verde"],
  "V3": ["vermelho", "verde", "azul"],
  "V4": ["vermelho", "verde"],
  "V5": ["vermelho", "verde", "azul"]
 },
 "vizinhos": {
  "V1": ["V2", "V5", "V3"],
  "V2": ["V1", "V3", "V4"],
  "V3": ["V2", "V4", "V1"],
  "V4": ["V3", "V5", "V2"],
  "V5": ["V4", "V1"]
 },
```


Grafo de Restrições

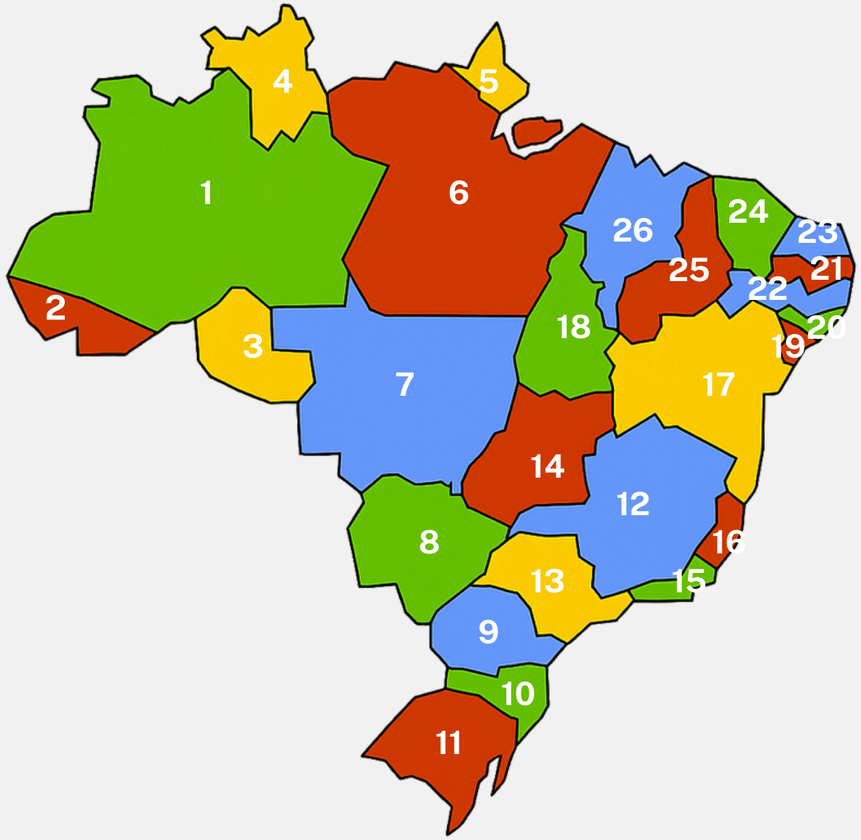


Domínios de V2 e V4 restringidos para {vermelho, verde}

Instância Complexa

Restrição dos domínios

```
MAPA_BRASIL = {
    1: [2, 3],                  # ACRE
    2: [1, 3],                  # RONDÔNIA
    3: [1, 2, 4, 6, 7],         # AMAZONAS
    4: [3, 5, 6],               # RORAIMA
    5: [4, 6],                  # AMAPÁ
    6: [3, 4, 5, 7],            # PARÁ
    7: [3, 6, 8, 9, 12, 13, 14], # MATO GROSSO
    8: [7, 9, 12],              # MATO GROSSO DO SUL
    9: [7, 8, 10, 11, 12],      # PARANÁ
    10: [9, 11],                # SANTA CATARINA
    11: [9, 10],                # RIO GRANDE DO SUL
    12: [7, 8, 9, 13, 14],      # GOIÁS
    13: [7, 12, 14, 15],        # DF
    14: [7, 12, 13, 15, 16, 18], # MINAS GERAIS
    15: [13, 14, 16],           # ESPÍRITO SANTO
    16: [14, 15, 17, 18],       # RIO DE JANEIRO
    17: [16, 18, 19, 20],       # BAHIA
    18: [14, 16, 17, 19],       # TOCANTINS
    19: [17, 18, 20, 21],       # SERGIPE
    20: [17, 19, 21, 22],       # ALAGOAS
    21: [19, 20, 22, 23],       # PERNAMBUCO
    22: [20, 21, 23],           # PARAÍBA
    23: [21, 22, 24],           # RIO GRANDE DO NORTE
    24: [23, 25, 26],           # CEARÁ
    25: [24, 26],               # PIAUÍ
    26: [24, 25],               # MARANHÃO
}
```



Restrições de dominios:

Amazonas = {vermelho, verde, azul },
Paraná = {azul},
São Paulo = {amarelo},
Rio de Janeiro = {azul, verde, amarelo},
Bahia = {amarelo},

Ferramentas utilizadas nas Implementações

Python	MiniZinc	Java/Prolog
• Biblioteca *python-constraint* • Algoritmo Principal: *Backtracking (BacktrackingSolver)* • Heuristicas: Grau (Degree) e Mínimo de Valores Restantes (MRV)	• Solver Gecode 6.3.0 • Consistência de Arco • Backtracking • Heuristicas first_fail (MRV)	• Choco Solver • Backtracking • Seleção de variável: Dom/Wdeg • Heuristica: LCV (Least Constraining Value)

Python

A Biblioteca python-constraint

O que é?

- Uma biblioteca Python de código aberto projetada para resolver Problemas de Satisfação de Restrições (PSR)

Principal Vantagem:

- Permite uma modelagem **declarativa** do problema. O desenvolvedor se concentra apenas em *definir* as variáveis, seus domínios e as restrições, enquanto a biblioteca cuida da complexa implementação do algoritmo de busca.

Componentes Principais:

- *Problem( )*: A classe principal onde o problema é modelado.
- *addVariable( )/addVariables( )*: Métodos para definir as variáveis e seus domínios de valores possíveis.
- *addConstraint( )*; Método para definir as regras (restrições) que a solução final deve oberdecer.
- *getSolution( )/ getSolutions( )*: Métodos que invocam um “solver” para encontrar uma ou todas as soluções.

O BacktrakingSolver

O Motor da Busca:

- O BacktrackingSolver é um dos solvers disponibilizados pela biblioteca. Conforme a especificação do trabalho, ele implementa o algoritmo de **Busca com Retrocesso (Backtracking)**

Como Funciona?

- Ele constrói a solução de forma incremental, atribuindo um valor a uma variável por vez.
- A cada nova atribuição, ele verifica se a solução parcial ainda é consistente com todas as restrições.
- Se uma atribuição leva a um conflito (violação de restrição), o solver retrocede (faz o backtrack): ele desfaz a atribuição inválida e tenta um novo valor para a mesma variável.
- Se todos os valores de uma variável falham, ele retrocede ainda mais, para a variável anterior, e tenta um novo caminho a partir dela.

Heurísticas MRV e Degree

Por que usar Heurísticas?

- A ordem em que o backtracking escolhe as variáveis afeta drasticamente a performance. Uma ordem "ruim" pode levar à exploração de partes inúteis da árvore de busca.
- Heurísticas são regras inteligentes que guiam o solver a tomar melhores decisões, podando a árvore de busca e encontrando a solução (ou a falha) muito mais rápido.

A Combinação do BacktrakingSolver:

- Analisando o código-fonte da biblioteca, vemos que o BacktrackingSolver utiliza uma combinação poderosa de duas heurísticas clássicas:
  - Heurística de Grau (Degree Heuristic):
  - Heurística MRV (Minimum Remaining Values):

Heurísticas MRV e Degree

Heurística de Grau (Degree Heuristic):

- **Princípio:** Prioriza a variável que participa do maior número de restrições com outras variáveis ainda não atribuídas.
- **Objetivo:** Tenta resolver primeiro as partes mais "influentes" do problema, o que ajuda a restringir as opções das variáveis vizinhas mais cedo.

Heurística MRV (Minimum Remaining Values):

- **Princípio:** Prioriza a variável com o menor número de valores válidos restantes em seu domínio.

- **Objetivo:** É uma heurística de "falha-primeiro" (fail-first). Ao lidar com a variável mais restrita, o algoritmo descobre rapidamente se um caminho levará a um beco sem saída, permitindo um backtrack mais cedo e mais eficiente.

MiniZinc

MiniZinc

O MiniZinc é uma linguagem de modelagem para descrever problemas de otimização e satisfação de restrições que se conecta a um "solver" que resolve o problema.

Solver utilizado foi o Gecode 6.3.0
O Gecode utiliza uma abordagem que combina dois elementos:

- Propagação de Restrições: A função do propagador é reduzir o domínio das outras variáveis, eliminando valores que se tornaram impossíveis com a nova atribuição. O Gecode usa versões avançadas de algoritmos de consistência.
- Busca por Retrocesso (Backtracking): A busca por retrocesso é o algoritmo que explora a árvore de busca do problema. O Gecode aprimora a busca com a heurística para decidir qual caminho seguir.
  - Heuristica MRV (Minimum Remaining Values)

```
% Número de regiões
n = 5;
% Número de relações
m = 7;

regiao1 = [1, 1, 1, 2, 2, 3, 4];
regiao2 = [2, 3, 5, 3, 4, 4, 5];

% Cores que a região não deve assumir
%                 1    2       3      4      5
restricoes = [{}, {Azul}, {}, {Azul}, {}];
```

```
% Algoritmo para o problema de coloração de mapas

% Dominio das variáveis
enum Cores = {Vermelho, Verde, Azul};

% Número de variáveis
int: n;
int: m;

set of int: V = 1..n;

array[1..m] of int: regiao1;
array[1..m] of int: regiao2;

% conjunto de cores restritas para cada variável
array[V] of set of Cores: restricoes;

% variáveis de decisão
array[V] of var Cores: cor;

% Restrição: vizinhos não podem ter mesma cor
constraint
  forall(k in 1..m) (
    cor[regiao1[k]] != cor[regiao2[k]]
  );

% restrições extras de domínio
constraint
  forall(i in V) (
    not (cor[i] in restricoes[i])
  );

solve :: int_search(cor, first_fail, indomain_min, complete) satisfy;

output [ "V" ++ show(i) ++ ": " ++ show(cor[i]) ++ "\n" | i in V ];
```

Java

A Biblioteca Choco Solver

O que é?

- Uma biblioteca Java de código aberto projetada para modelar e resolver Problemas de Satisfação de Restrições (PSR) e Problemas de Otimização.
- É amplamente usada em pesquisa e indústria para lidar com problemas combinatórios complexos, como coloração de mapas, escalonamento, roteamento e alocação de recursos.

Principal Vantagem:

- Permite uma modelagem declarativa e modular dos problemas.
- O desenvolvedor se concentra em definir variáveis, domínios e restrições, enquanto o Choco cuida internamente das estratégias de busca, propagação e otimização.
- Oferece suporte a heurísticas sofisticadas (como Dom/Wdeg, LCV, Impact-Based Search, entre outras), permitindo resolver instâncias grandes com eficiência.

A Biblioteca Choco Solver

Componentes Principais:

- *Model( ): Classe principal onde o problema é construído. Define variáveis e restrições*

.

- *IntVar / BoolVar / RealVar: Representam variáveis inteiras, booleanas e reais, respectivamente, com seus domínios.*

- *arithm( ) / allDifferent( ) / sum( ): Métodos usados para criar restrições entre variáveis.*

- *getSolver( ): Retorna o objeto responsável pela busca e controle da resolução.*

- *solve( ) / findSolution( ) / getSolutions( ): Métodos que executam o processo de busca e retornam uma ou todas as soluções possíveis.*

Heurísticas Dom/Wdeg e LCV

Heurística Dom/Wdeg (Domain over Weighted Degree):

- Princípio: Seleciona a variável com o menor quociente entre o tamanho do domínio e o grau ponderado de restrições que ela participa. Quanto mais restrita e mais “conflitante” for a variável, maior sua prioridade.

- Objetivo: Combina as ideias da heurística MRV e da heurística de Grau, focando em variáveis críticas que causam mais falhas durante a busca. Isso ajuda o solver a identificar cedo os pontos de maior dificuldade no problema, acelerando a convergência.

Heurística LCV (Least Constraining Value):

- Princípio: Escolhe para cada variável o valor que restringe menos os domínios das variáveis vizinhas.

- Objetivo: Minimizar o impacto futuro de cada decisão. Ao escolher valores menos restritivos, o algoritmo mantém o espaço de busca mais amplo, reduzindo o número de retrocessos (backtracks) e melhorando a eficiência global.

Resultados

Fácil

	Python	MiniZinc	Java
Resultados	• V1: Azul • V2: Verde • V3: Azul • V4: Verde • V5: Vermelho	• V1: Vermelho • V2: Verde • V3: Vermelho • V4: Verde • V5: Azul	• V1: Vermelho • V2: Verde • V3: Azul • V4: Vermelho • V5: Verde
Tempo	0.000 milissegundos	150 msec	13 msec

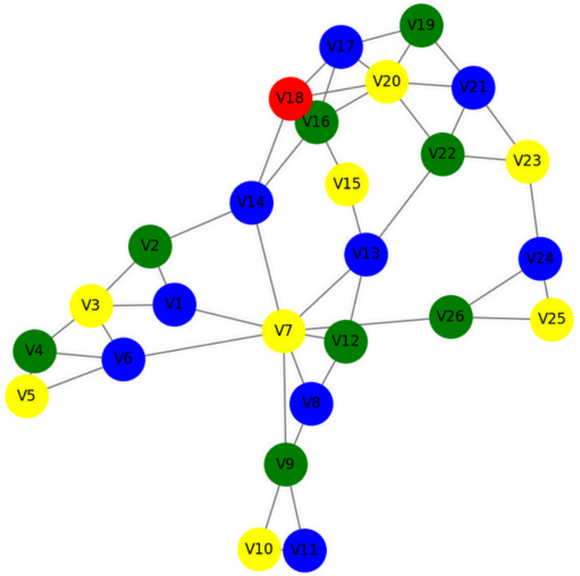
Média

	Python	MiniZinc	Java
Resultados	• V1: Azul • V2: Verde • V3: Vermelho • V4: Azul • V5: Verde	• V1: Vermelho • V2: Verde • V3: Azul • V4: Vermelho • V5: Verde	• V1: Vermelho • V2: Verde • V3: Azul • V4: Vermelho • V5: Verde
Tempo	0.000 milissegundos	141 msec	3 msec

Difícil

	Python	MiniZinc	Java
Resultados	• V1: Vermelho • V2: Verde • V3: Azul • V4: Vermelho • V5: Azul	• V1: Vermelho • V2: Verde • V3: Azul • V4: Vermelho • V5: Verde	• V1: Vermelho • V2: Verde • V3: Azul • V4: Vermelho • V5: Verde
Tempo	0.0000 milissegundos	246 msec	1 msec

Complexa

	Python	MiniZinc	Java
Resultados			V1: Azul V2: Verde V3: Amarelo V4: Azul V5: Amarelo V6: Vermelho V7: Verde V8: Amarelo V9: Azul V10: Amarelo V11: Verde V12: Azul V13: Amarelo V14: Vermelho V15: Azul V16: Verde V17: Vermelho V18: Azul V19: Verde V20: Amarelo V21: Vermelho V22: Azul V23: Amarelo V24: Verde V25: Amarelo V26: Azul
Tempo	0.000 milissegundos	199 msec	42 msec

Conclusão

A resolução do problema de coloração de mapas foi abordada por meio de três linguagens: Python (constraint), MiniZinc (Gecode), Java (Choco Solver).

Todas as implementações se basearam no algoritmo de Busca com Retrocesso (Backtracking). A diferença foi ditada pelas heurísticas empregadas para guiar a busca: Python usou a Heurística de Grau e o Mínimo de Valores Restantes (MRV), MiniZinc utilizou MRV (com first_fail), e Java empregou a combinação de Dom/Wdeg e LCV (Least Constraining Value).

Os resultados de tempo nas instâncias Fácil, Média, Difícil e Complexa, o Python apresentou tempos de resolução extremamente baixos (0,000 milissegundos), sugerindo uma eficiência excepcional da biblioteca para o modelo testado. O Java (Choco Solver) também demonstrou alta eficiência, resolvendo instâncias complexas em tempos muito rápidos (1 a 42 ms), mostrando que suas heurísticas (Dom/Wdeg e LCV) são eficazes na poda da árvore de busca.

Referências

- https://docs.minizinc.dev/en/latest/index.html
- https://www.gecode.dev
- https://www.gecode.dev/doc-latest/MPG.pdf
- https://www.youtube.com/watch?v=hdMZxZkhV_s
- python-constraint. PyPI. Disponível em: <https://pypi.org/project/python-constraint/#description>. Acesso em: 4 out. 2025.
- PYTHON-CONSTRAINT Community. python-constraint: A Python library for solving constraint satisfaction problems. Versão <ins>1.4.0</ins>. 2020. Disponível em: python-constraint/python-constraint: Constraint Solving Problem resolver for Python. GitHub. Disponível em: <https://github.com/python-constraint/python-constraint>. Acesso em: 4 out. 2025.
- https://choco-solver.org/docs
- https://javadoc.io/doc/org.choco-solver/choco-solver/5.0.0-beta.1/index.html

Agradecidos

Alguma dúvida?