Problemas de Satisfação de Restrições (PSR)

Formulação e Resolução com Backtracking e Propagação de Restrições

Márcio Nicolau

2025-10-16

Table of contents

List of Figures

Introdução

Até agora, exploramos como os agentes inteligentes podem encontrar caminhos em espaços de estados usando estratégias de busca cega e informada. No entanto, muitos problemas em Inteligência Artificial não se encaixam naturalmente no paradigma de encontrar uma "sequência de ações" para um estado objetivo. Em vez disso, eles envolvem a busca por um **estado** (uma configuração de variáveis) que satisfaça um conjunto de **restrições** específicas. Estes são os **Problemas de Satisfação de Restrições (PSRs)**.

Nesta aula, definiremos o que são PSRs, como formalizá-los e, mais importante, como utilizar técnicas poderosas como a busca com **backtracking** e a **propagação de restrições** para resolvê-los de forma eficiente. PSRs são onipresentes em áreas como planejamento, agendamento, visão computacional e projeto de circuitos.

Objetivo de Aprendizagem

Ao final desta aula, você será capaz de:

- Definir e identificar os componentes de um Problema de Satisfação de Restrições (PSR).
- Formular problemas do mundo real como PSRs.
- Compreender e aplicar o algoritmo de busca com Backtracking para resolver PSRs.
- Entender a importância e aplicar técnicas de aprimoramento do Backtracking, como ordenação de variáveis, ordenação de valores e propagação de restrições (Forward Checking).
- Implementar soluções para PSRs simples em Python utilizando essas técnicas.

O que são Problemas de Satisfação de Restrições (PSR)?

Um **Problema de Satisfação de Restrições (PSR)** é um modelo para problemas nos quais o objetivo é encontrar uma atribuição de valores para um conjunto de variáveis, de forma que todas as restrições sobre essas variáveis sejam satisfeitas (Russell; Norvig, 2004, p. 136).

Componentes de um PSR

Formalmente, um PSR é definido por três componentes principais:

1. **Variáveis (V):** Um conjunto de variáveis $V = \{V_1, V_2, \dots, V_n\}$.
2. **Domínios (D):** Para cada variável V_i, há um domínio D_i, que é um conjunto de valores possíveis que V_i pode assumir.
3. **Restrições (C):** Um conjunto de restrições $C = \{C_1, C_2, \dots, C_m\}$. Cada restrição C_j especifica uma combinação permitida de valores para um subconjunto das variáveis.

Uma **solução** para um PSR é uma atribuição consistente (sem violação de restrições) de um valor de seu domínio para cada variável.

Exemplos Clássicos de PSRs

- **Coloração de Mapas:** Atribuir cores a regiões de um mapa de forma que regiões adjacentes tenham cores diferentes.
- **Problema das N-Rainhas:** Posicionar N rainhas em um tabuleiro N x N de xadrez de forma que nenhuma rainha ataque outra.
- **Sudoku:** Preencher uma grade 9x9 com dígitos de 1 a 9, de forma que cada linha, coluna e subgrade 3x3 contenha todos os dígitos sem repetição.
- **Agendamento:** Agendar tarefas em horários e recursos limitados.

Grafo de Restrições

Um PSR pode ser visualizado como um **grafo de restrições**, onde:

- Os nós do grafo são as **variáveis**.
- As arestas conectam variáveis que compartilham uma **restrição**.

Formalizando um PSR: Exemplo de Coloração de Mapas

Vamos usar o problema de colorir o mapa da Austrália para ilustrar a formalização de um PSR.

Problema: Colorir cada estado do mapa da Austrália com uma de três cores (vermelho, verde, azul) de forma que estados adjacentes não tenham a mesma cor.

Componentes do PSR:

- **Variáveis (V):**
  - $\{WA, NT, SA, Q, NSW, V, T\}$ (os estados australianos, onde T é Tasmânia, mas não é adjacente a nenhum outro e pode ser ignorado por enquanto para simplificar o grafo conectado).
  - Vamos usar apenas as variáveis do grafo acima: $\{WA, NT, SA, Q, NSW, V\}$.
- **Domínios (D):** Para cada variável V_i, o domínio é $D_i = \{\text{vermelho, verde, azul}\}$.
- **Restrições (C):** Para cada par de estados adjacentes, eles devem ter cores diferentes.
  - $C(WA, NT) : WA \neq NT$
  - $C(WA, SA) : WA \neq SA$
  - $C(NT, SA) : NT \neq SA$
  - $C(NT, Q) : NT \neq Q$
  - $C(SA, Q) : SA \neq Q$
  - $C(SA, NSW) : SA \neq NSW$
  - $C(SA, V) : SA \neq V$
  - $C(Q, NSW) : Q \neq NSW$
  - $C(NSW, V) : NSW \neq V$

Técnicas de Resolução de PSRs

A resolução de PSRs pode ser vista como um problema de busca. Um estado na busca é uma atribuição parcial de valores às variáveis, e uma ação é atribuir um valor a uma variável não atribuída.

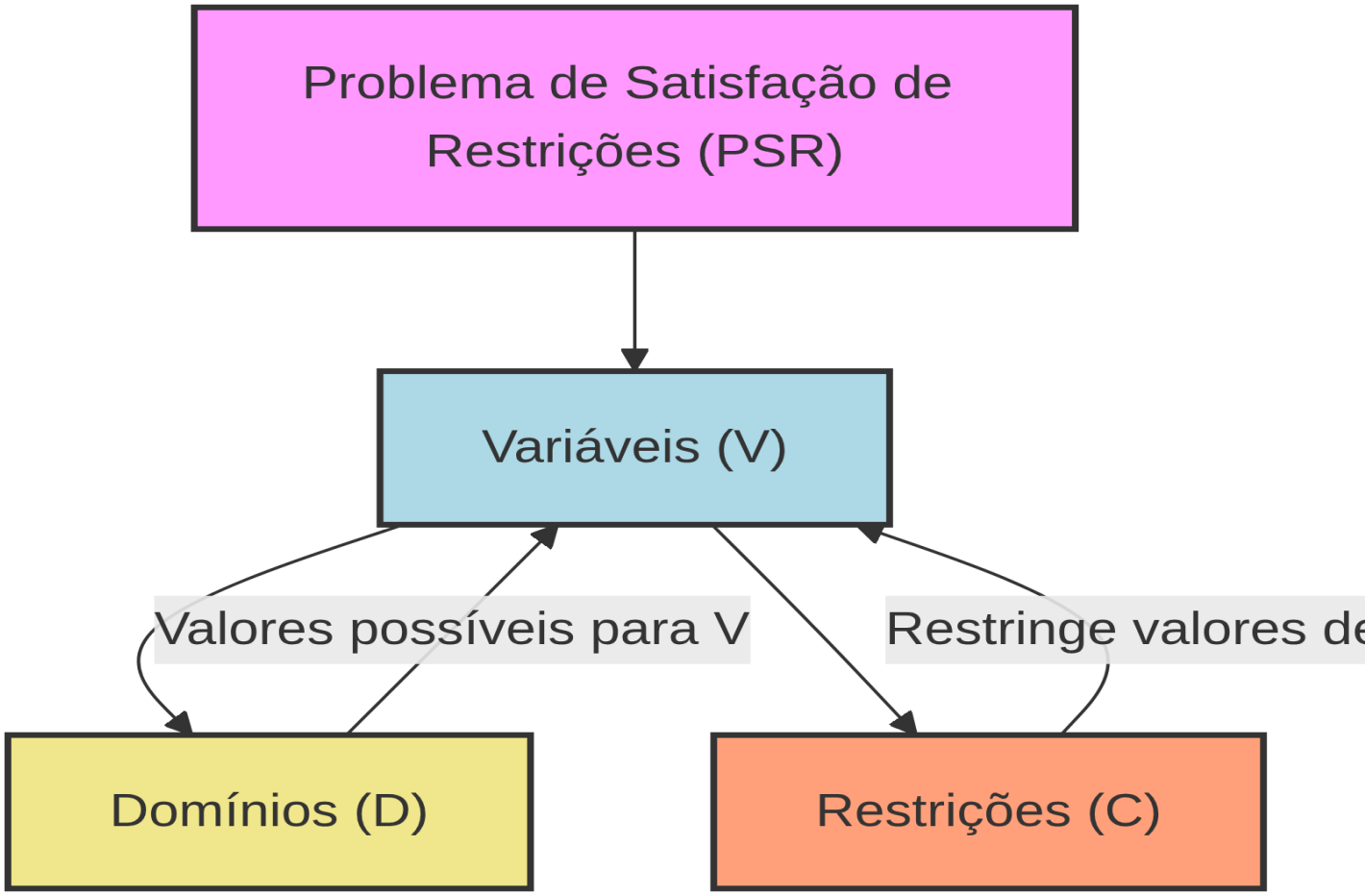


Figure 1: Componentes de um Problema de Satisfação de Restrições (PSR)

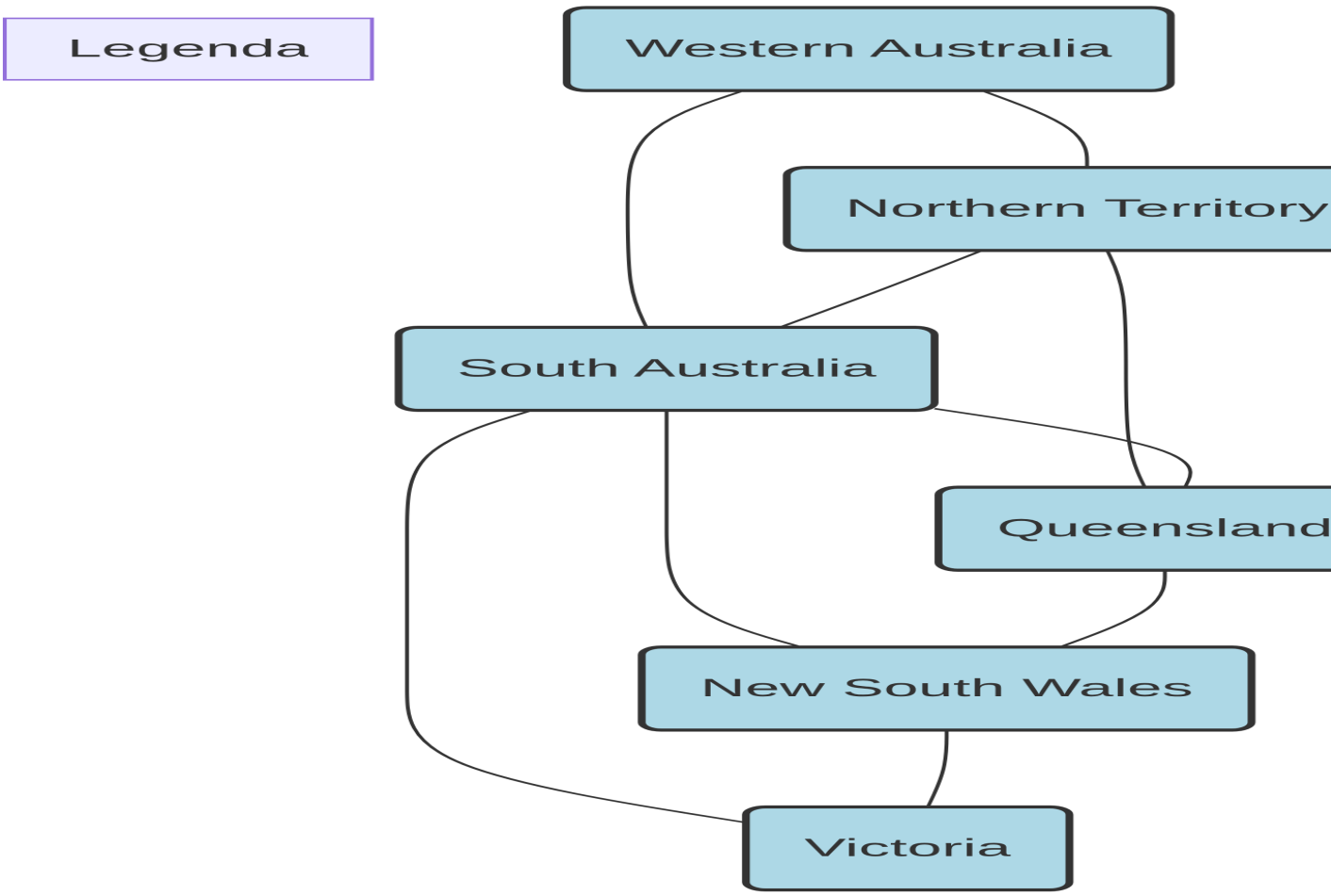


Figure 2: Mapa da Austrália com Grafo de Restrições para Coloração

Busca com Backtracking (Backtracking Search)

A Busca com Backtracking é o algoritmo básico para resolver PSRs. É uma forma de Busca em Profundidade (DFS) que, em cada nó da árvore de busca, tenta atribuir um valor a uma variável. Se a atribuição viola alguma restrição, ela retrocede (backtracks) e tenta outro valor.

Funcionamento

1. **Atribuição Incremento:** O algoritmo atribui um valor a uma variável por vez.
2. **Verificação de Restrições:** Após cada atribuição, verifica se alguma restrição é violada.
3. **Backtracking:** Se uma atribuição viola uma restrição, o algoritmo "desfaz" a atribuição e tenta outro valor para a mesma variável. Se não houver mais valores para tentar, ele retrocede para a variável anterior e muda sua atribuição.

Pseudocódigo Básico (Russell; Norvig, 2004, p. 139)

```
function BACKTRACKING-SEARCH(csp) returns solution or failure
    return BACKTRACK( {}, csp )

function BACKTRACK(assignment, csp) returns solution or failure
    if assignment is complete then return assignment
    var <- SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment and csp's constraints then
            add {var = value} to assignment
            result <- BACKTRACK(assignment, csp)
            if result is not failure then return result
            remove {var = value} from assignment
    return failure
```

- `SELECT-UNASSIGNED-VARIABLE`: Escolhe a próxima variável a ser atribuída.
- `ORDER-DOMAIN-VALUES`: Decide a ordem em que os valores serão testados para a variável selecionada.

Limitações do Backtracking Simples

O backtracking simples pode ser muito ineficiente. Ele pode perder muito tempo explorando caminhos de busca que são obviamente inválidos, o que é conhecido como **"thrashing"** (perder tempo repetindo os mesmos erros, pois não "lembra" de onde veio o problema).

Melhorando o Backtracking

Podemos melhorar significativamente a eficiência do backtracking adicionando heurísticas e técnicas de propagação de restrições.

1. Ordem de Seleção de Variáveis (Variable Ordering)

Qual variável devemos atribuir primeiro?

- **MRV (Minimum Remaining Values - Variável Mais Restrita):** Escolha a variável com o menor número de valores legais restantes em seu domínio. Isso tende a identificar falhas mais cedo.
- **Heurística do Grau (Degree Heuristic):** Se houver um empate no MRV, escolha a variável que está envolvida no maior número de restrições com variáveis ainda não atribuídas. Isso tenta "desprender" as variáveis mais conectadas primeiro.

2. Ordem de Seleção de Valores (Value Ordering)

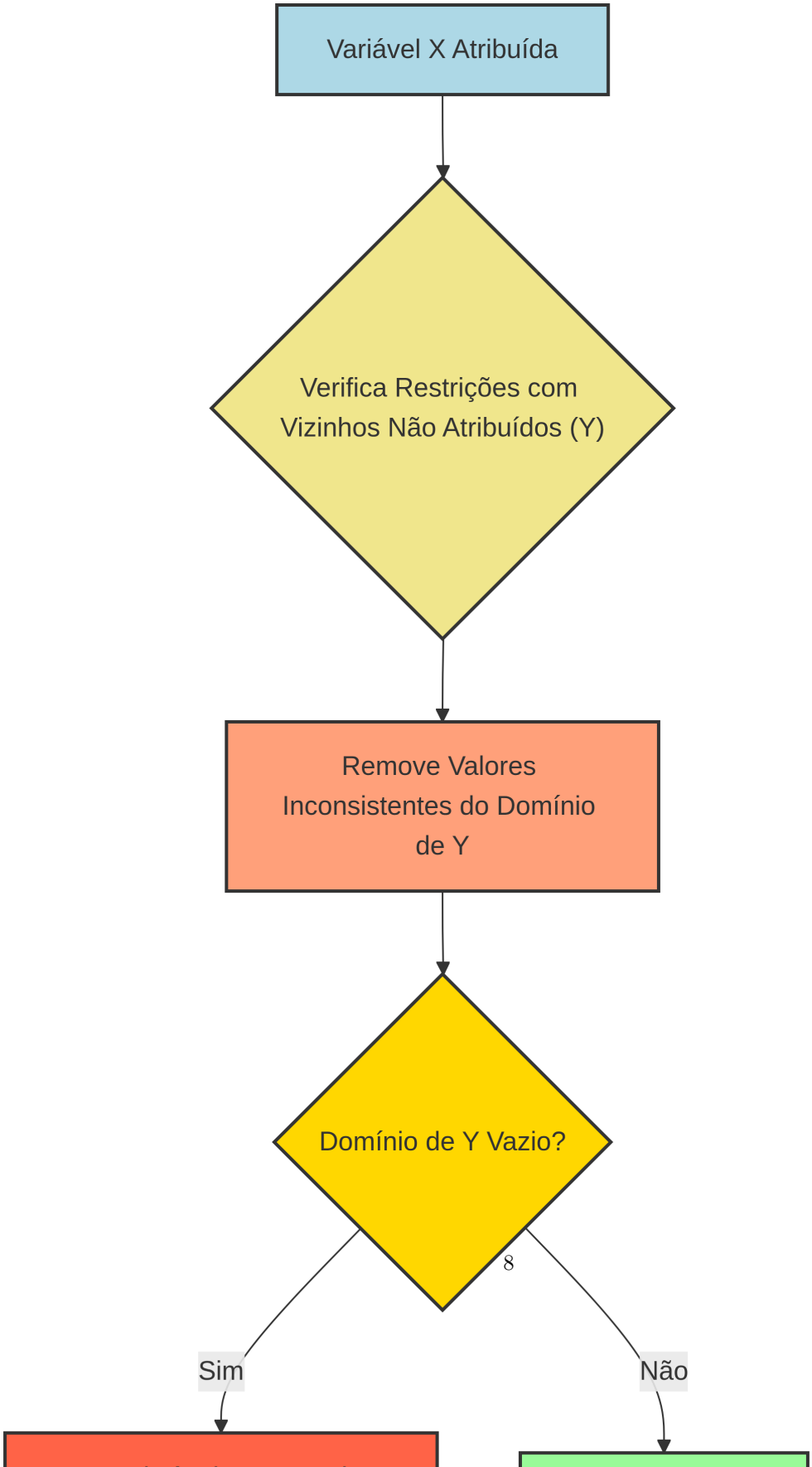
Uma vez que uma variável foi selecionada, qual valor devemos tentar primeiro?

- **LCV (Least Constraining Value - Valor Menos Restritivo):** Escolha o valor que descarta o menor número de valores nos domínios das variáveis adjacentes (que ainda não foram atribuídas). Isso maximiza as chances de encontrar uma solução sem precisar de backtracking.

3. Propagação de Restrições (Constraint Propagation)

A propagação de restrições remove valores dos domínios das variáveis que são inconsistentes com as atribuições atuais ou com outras restrições. Isso ajuda a detectar falhas mais cedo e a reduzir o espaço de busca.

- **Forward Checking (FC):** Após atribuir um valor a uma variável `X`, o Forward Checking verifica cada variável `Y` não atribuída que é adjacente a `X` (ou seja, tem uma restrição com `X`). Para cada `Y`, ele remove do seu domínio quaisquer valores que sejam inconsistentes com a atribuição de `X`. Se o domínio de `Y` se tornar vazio, a atribuição de `X` é inconsistente, e o algoritmo retrocede.


Variável X Atribuída
Verifica Restrições com Vizinhos Não Atribuídos (Y)
Remove Valores Inconsistentes do Domínio de Y
Domínio de Y Vazio?
Sim
Não

- **Arc Consistency (AC-3):** Uma forma mais poderosa de propagação de restrições. Ela garante que cada arco `(Xi, Xj)` em um grafo de restrições seja consistente. Um arco `(Xi, Xj)` é consistente se, para cada valor `x` no domínio de `Xi`, existe *pelo menos um* valor `y` no domínio de `Xj` tal que a atribuição `(Xi=x, Xj=y)` satisfaz a restrição entre `Xi` e `Xj`. O AC-3 itera sobre os arcos, removendo valores inconsistentes até que nenhuma remoção possa ser feita (ou um domínio se torne vazio).

Exemplo: Problema das N-Rainhas com Backtracking e Forward Checking

O problema das N-Rainhas é um PSR clássico: posicionar N rainhas em um tabuleiro N x N de modo que nenhuma rainha ataque outra (nenhuma na mesma linha, coluna ou diagonal).

- **Variáveis:** $Q_1, \ldots, Q_N$, onde Q_i é a linha da rainha na coluna i.
- **Domínios:** Para cada Q_i, $D_i = \{1, \ldots, N\}$ (as linhas possíveis).
- **Restrições:** Para qualquer par de rainhas (Q_i, Q_j) (com $i \neq j$):
  - $Q_i \neq Q_j$ (não na mesma linha)
  - $|Q_i - Q_j| \neq |i - j|$ (não na mesma diagonal)

Vamos implementar uma solução para o problema das N-Rainhas usando backtracking e Forward Checking.

N-Rainhas com Backtracking e Forward Checking em Python

```python
def solve_n_queens(n):
    # Inicializa domínios para cada rainha (coluna)
    # domains[coluna] = set de linhas possíveis para aquela rainha
    domains = {col: set(range(n)) for col in range(n)}

    # current_assignment = {coluna: linha}
    current_assignment = {}

    # backtrack(col) tenta posicionar a rainha na coluna 'col'
    solutions = []

    def is_consistent(col, row, assignment):
        # Verifica se a nova rainha (col, row) é consistente com as rainhas já posicionadas
        for assigned_col, assigned_row in assignment.items():
            if assigned_row == row:  # Mesma linha
                return False
            if abs(assigned_col - col) == abs(assigned_row - row): # Mesma diagonal
                return False
        return True

    def forward_check(col, row, current_domains):
        # Cria uma cópia dos domínios para não modificar o original no caso de backtracking
        new_domains = {c: d.copy() for c, d in current_domains.items()}

        # O valor 'row' é atribuído à 'col'. Removemos 'row' do domínio de outras colunas se for inco
        for next_col in range(col + 1, n):
            # Remove a linha 'row' (mesma linha)
            if row in new_domains[next_col]:
                new_domains[next_col].discard(row)
```

A implementação demonstra como o `forward_check` atua para podar o espaço de busca. Quando um valor `row` é atribuído a uma rainha na `col`, o `forward_check` percorre as colunas futuras e remove as linhas que seriam atacadas. Se um domínio de alguma coluna futura se tornar vazio, sabemos que a atribuição atual é inviável, e o backtracking ocorre mais cedo, sem explorar caminhos que levariam a um beco sem saída.

Considerações Finais

Problemas de Satisfação de Restrições são uma forma poderosa e flexível de modelar uma vasta gama de desafios em IA. Enquanto o backtracking puro é uma estratégia de força bruta, a combinação com heurísticas de ordenação (MRV, Grau, LCV) e, crucialmente, com técnicas de propagação de restrições como o Forward Checking e Arc Consistency, transforma-o em um algoritmo prático e eficiente para resolver muitos PSRs complexos. A chave é a detecção precoce de inconsistências para evitar a exploração de ramos inúteis na árvore de busca.

Verificação de Aprendizagem

Responda às seguintes questões e implemente as tarefas propostas para solidificar seu entendimento.

1. **Fundamentos de PSR:**
    a) Defina um Problema de Satisfação de Restrições (PSR) e liste seus três componentes principais.
    b) Dê um exemplo de um problema do mundo real (diferente dos citados na aula) que pode ser formulado como um PSR. Identifique as variáveis, seus domínios e as restrições para seu exemplo.
2. **Backtracking:**
    a) Explique o funcionamento básico do algoritmo de busca com Backtracking para resolver PSRs. Qual é a principal desvantagem do backtracking simples?
    b) Por que as heurísticas de ordenação de variáveis (MRV e Grau) e de valores (LCV) são importantes para melhorar o desempenho do backtracking?
3. **Propagação de Restrições:**
    a) Descreva o mecanismo do **Forward Checking**. Como ele ajuda a reduzir o espaço de busca em comparação com o backtracking simples?
    b) Qual é a diferença conceitual entre Forward Checking e Arc Consistency (AC-3)? Quando você usaria um em vez do outro, considerando a complexidade?
4. **Implementação e Análise:** Considere o seguinte problema de criptoaritmética: `TWO + TWO = FOUR`. Cada letra deve ser um dígito único de 0 a 9. `T` e `F` não podem ser 0.
    a) **Formule este problema como um PSR:**
        - Liste as variáveis.
        - Defina os domínios para cada variável.
        - Liste as restrições (iguais, diferentes, e a restrição aritmética).
    b) **Simulação de Backtracking com Forward Checking (Manual):** Imagine que o algoritmo de backtracking está resolvendo este PSR. Se a primeira atribuição é `T = 2`:

- Quais variáveis seriam afetadas pelo Forward Checking (ou seja, quais domínios seriam podados)?
- Quais valores seriam removidos dos domínios dessas variáveis devido à restrição de dígitos únicos e à restrição $T \neq 0$?
- Existe alguma inconsistência detectada imediatamente por esta primeira atribuição? (Ou seja, algum domínio fica vazio?)

Referências Bibliográficas

RUSSELL, Stuart J.; NORVIG, Peter. **Inteligência Artificial: Um Enfoque Moderno**. 2. ed. Rio de Janeiro: Prentice Hall, 2004.