FGA0221 - Inteligência Artificial

Portfólio 03

Janeiro, 2025

Universidade de Brasília

Faculdade UnB Gama

Tema do portfólio:

Problemas de Satisfação de Condições (CSPs).

Aluno: João Matheus de Oliveira Schmitz
Matrícula: 200058525
Turma: T01
Semestre: 2024.2

Janeiro, 2025

Sumário

Janeiro, 2025

1. Representação Atômica vs. Fatorada

Os algoritmos de busca, como vistos no portfólio anterior, são voltados para resolver problemas em que cada estado é **atômico**, ou seja, **não possui estrutura interna**, mas nem todo problema tem essa característica. Para resolver certos problemas de forma mais eficiente, o melhor a se fazer é representar os estados de forma **fatorada**: um **conjunto de variáveis com seus respectivos valores**, ao invés de um valor único.

Os problemas que utilizam representação fatorada tem suas **soluções em estados onde cada uma de suas variáveis tem um valor que satisfaça suas restrições**. Chamamos estes problemas de Problemas de Satisfação de Condições ou CSPs (Constraint Satisfaction Problem).

2. Definindo Problemas de Satisfação de Condições

Segundo [2](Russell & Norvig, 2009), os algoritmos de busca CSP se **aproveitam da estrutura de estados e utilizam heurísticas gerais**, ao invés de heurísticas específicas de problemas, **para resolver problemas complexos**. A ideia por trás dessa estratégia é **eliminar boa parte do espaço de busca** ao identificar combinações de variável/valor que ferem as restrições impostas.

Os Problemas de Satisfação de Condições são compostos de 3 partes:

- X, um conjunto de variáveis $\{X_1, ..., X_N\}$.
- D, um conjunto de domínios para cada variável $\{D_1, ..., D_N\}$:
- C, um conjunto de restrições que diz quais as combinações de valores possíveis $\{C_1, ..., C_N\}$:

É importante ressaltar que cada domínio D_i consiste em um conjunto de valores que a variável X_i correspondente pode ter. Além disso, cada restrição tem o formato **<escopo, rel>**, onde **escopo** é uma tupla de variáveis as quais aquela restrição se aplica e **rel** é uma relação que define os valores que essas variáveis podem assumir.

É possível representar uma relação de duas maneiras. Por exemplo, se as variáveis X e Y podem receber os valores {1, 2, 3}, mas Y é maior que X, podemos representar a relação como:

- Uma lista de todas as tuplas de valores que satisfazem a restrição, neste caso a restrição escrita é **<(X, Y), [(1, 2), (1, 3), (2, 3)]>**;
- Uma relação abstrata, como funções matemáticas, que mostram a restrição, neste caso a restrição escrita é: **<(X, Y), (X < Y)>**.

Nos CSPs, cada estado é determinado pela **atribuição** de valores a uma ou mais variáveis. Dependendo de suas características, uma atribuição pode ser:

- Consistente: não viola nenhuma restrição;
- Completa: onde todas as variáveis recebem um valor;
- Parcial: nem todas as variáveis recebem um valor.

Uma **solução** para um CSP é uma **atribuição completa e consistente**. Já uma **solução parcial** é uma **atribuição parcial e consistente**.

2.1. Problema exemplo

Para exemplificar um CSP, podemos tomar como base um **tabuleiro de xadrez**. Entre diversas configurações possíveis de peças, queremos encontrar aquela referente à **posição inicial das peças no tabuleiro antes de um jogo começar**.

Figura 01 - Posição inicial no tabuleiro de xadrez

- **As variáveis são as colunas do tabuleiro**.
  - X = {A, B, C, D, E, F, G, H};

- **Os domínios são as linhas do tabuleiro**.
  - D = {1, 2, 3, 4, 5, 6, 7, 8}, onde D_i = {Vazio, Rei Preto, Rei Branco, Rainha Preto, Rainha Branco, Bispo Preto, Bispo Branco, Cavalo Preto, Cavalo Branco, Torre Preto, Torre Branco, Peão Preto, Peão Branco}.

- **As restrições são aquelas que forçam as peças a estarem em suas posições iniciais**:
  - <(A, H), (Torre Branco, Peão Branco, Vazio, Vazio, Vazio, Vazio, Peão Preto, Torre Preto)>;
  - <(B, G), (Cavalo Branco, Peão Branco, Vazio, Vazio, Vazio, Vazio, Peão Preto, Cavalo Preto)>;
  - <(C, F), (Bispo Branco, Peão Branco, Vazio, Vazio, Vazio, Vazio, Peão Preto, Bispo Preto)>;
  - <(D), (Rainha Branco, Peão Branco, Vazio, Vazio, Vazio, Vazio, Peão Preto, Rainha Preto)>;
  - <(E), (Rei Branco, Peão Branco, Vazio, Vazio, Vazio, Vazio, Peão Preto, Rei Preto)>;

3. Tipos de condições

Nos CSPs, as condições podem ser classificadas em alguns tipos diferentes. São elas:

- **Restrição unária**: restringe o valor de uma única variável, é o tipo mais simples (e.g. A < 10);
- **Restrição binária**: relaciona duas variáveis (e.g. A > B);
- **Restrição ternária**: relaciona três variáveis (e.g. A > B >= C);
- **Restrição global**: relaciona um número arbitrário N de variáveis, onde N não necessariamente será igual ao total de variáveis (e.g. A soma das variáveis A até H deve ser igual a 100 -> A + B + C + D + E + F + G + H = 100);
- **Restrição de preferência**: são restrições opcionais que servem como desempate entre possíveis soluções (e.g. Ao criar um horário para aulas em uma faculdade temos restrições absolutas: nenhum professor pode dar mais de uma aula ao mesmo tempo; e restrições de preferência: o professor X prefere que suas aulas sejam no horário da manhã);

Uma característica importante a ser ressaltada é que, para simplificar a implementação dos algoritmos, existem métodos para **transformar qualquer CSP N-ário em um CSP binário**. Uma das formas é introduzir **variáveis auxiliares** para transformar as restrições, outra é pela transformação **grafo dual**, onde é criada uma nova variável para cada restrição original e uma restrição binária para cada par de restrições originais que compartilham variáveis. Entretanto, **nem sempre a transformação em CSP binário é algo desejável**, pois é suscetível a erros e, quando comparadas às restrições globais, não é possível utilizar alguns algoritmos de inferência especiais.

4. Consistência

Em algoritmos que trabalham buscando um espaço de estados atômicos, o avanço da busca só ocorre através da expansão para nós sucessores, porém, essa regra não se aplica aos Problemas de Satisfação de Condições. Os CSPs tem a seguinte escolha para realizar o avanço de uma busca:

- Escolher um **novo valor para a(s) variável(is)**;
- **Reduzir o número ou intervalo de valores válidos de uma variável** em uma restrição;

A segunda opção é conhecida como **Propagação de Condições**, pois a redução dos valores válidos de uma variável pode se propagar para outra variável, depois para outra e assim por diante. A ideia por trás desta estratégia é garantir a chamada **consistência local**, ou seja, a eliminação de valores inconsistentes naquela parte do espaço de busca. Existem tipos diferentes de consistência local, eles são as consistências: de nó, de arco, de trajeto e a chamada K-consistência.

4.1. Nó

A **consistência de nó** tem como característica tornar cada variável (visualizada como um nó em um grafo) **nó-consistente**. Isso é realizado ao **eliminar do domínio da variável as opções de valores que vão contra restrições unárias** desta variável. Um grafo é dito nó-consistente quando todas as suas variáveis são consistentes.

Ao realizar este processo em todas as variáveis, é possível **eliminar todas as restrições unárias de um CSP**, possibilitando, em conjunto com a transformação de restrições n-árias em binárias, que muitos algoritmos de CSP trabalhem somente com restrições binárias, o que simplifica suas implementações.

4.2. Arco

Uma variável é considerada **arco-consistente** com outra variável quando todos os **valores de seu domínio satisfazem as restrições binárias em que as variáveis estão envolvidas**. Consequentemente, um CSP é considerado

arco-consistente se todas as suas variáveis são arco-consistentes umas com as outras.

Para melhor visualizar essa consistência, podemos utilizar o exemplo da restrição $X = Y^2$, onde X e Y são variáveis, e o domínio de ambas as variáveis começa como D={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Para torná-las arco-consistente, devemos transformar seus domínios em Dx={0, 1, 2, 3} e Dy={0, 1, 4, 9}, pois caso X assuma qualquer valor além daqueles em Dx, a variável Y não terá um valor válido correspondente e, consequentemente, podemos eliminar os valores de Y que não correspondem a um valor de X.

É possível extrapolar a consistência de arco para incluir restrições não somente binárias, mas também as n-árias. Neste caso temos a chamada **consistência de arco generalizada** ou **consistência de hiperarco**.

4.3. Trajeto

Figura 02 - Mapa da Austrália

A consistência de arco consegue resolver vários problemas, porém, em exemplos como a coloração do mapa da Austrália (Figura 02), onde cada região deve ser pintada de uma cor diferente de suas vizinhas, ela não é suficiente. Isso ocorre pois, em uma situação onde só podemos usar duas cores, mesmo que cada par de variáveis seja arco-consistente entre si, não haverá soluções para o problema. A **consistência de caminho** ou **consistência de trajeto** é responsável por **fixar restrições binárias ao utilizar restrições implícitas**, as quais são inferidas pela verificação do triplo de variáveis.

Segundo [2](Russell & Norvig, 2009), "Um conjunto de duas variáveis $\{Xi, Xj\}$ é consistente de caminho em relação a uma terceira variável Xm se, para cada atribuição $\{Xi = a, Xj = b\}$ consistente com as restrições em $\{Xi, Xj\}$, houver uma atribuição para Xm que satisfaça as restrições em $\{Xi, Xm\}$ e $\{Xm, Xj\}$. **Isso se chama consistência de caminho porque se pode cogitar isso ao olhar para um caminho de Xi para Xj com Xm ao meio**".

4.4. K

Ao utilizar a **K-consistência**, podemos definir formas mais fortes de propagação. Um CSP é k-consistente se, **para qualquer conjunto de k-1 variáveis consistentes, a k-ésima variável puder receber um valor consistente**. Neste caso, podemos dizer que a 1-consistência é a consistência de nó, 2-consistência é a consistência de arco e 3-consistência, em redes de restrição binária, é a consistência de caminho.

Uma definição importante é a de uma rede **fortemente k-consistente**. Para receber essa definição, uma rede deve ser k-consistente, (k-1)-consistente, (k-2)-consistente e assim por diante até 1-consistente.

5. Algoritmos

5.1. AC-3

O algoritmo AC-3 é o **mais popular para a realização da consistência de arco**. A estratégia do AC-3 é manter uma fila de arcos a serem considerados, a qual começa com todos os arcos do CSP. O algoritmo então é responsável por pegar um arco (Xi, Xj) e tornar Xi arco-consistente com Xj. Caso isso diminua o domínio de Xi, então é necessário reintroduzir na fila os arcos (Xk, Xi), pois ao analisar a arco-consistência deste par, é possível diminuir o domínio de Xk, mesmo que Xk já tenha sido analisado. Esse loop é repetido até que **não haja mais arcos na fila**, indicando que o CSP se tornou arco-consistente, ou caso o **domínio de uma variável se torne nulo**, o que indica que o CSP não tem uma solução consistente, fazendo o algoritmo retornar uma falha.

Como exemplo podemos considerar a situação onde A, B, C e D são variáveis, seus domínios são D={1, 2, 3} e as restrições são:

$$(A > C)\,,\ (A < B)\,,\ (C = D)$$

É importante destacar que precisamos colocar as restrições tanto da forma como estão, quanto de forma oposta, na fila. Portanto a fila de arcos será:

$$(A > C) \text{ -> } (C < A) \text{ -> } (A < B) \text{ -> } (B > A) \text{ -> } (C = D) \text{ -> } (D = C)$$

O algoritmo então irá seguir os seguintes passos:

-	A > C	C < A	**A < B**	B > A	C = D	D = C	**C < A**	D = C
Da	{2, 3}	{2, 3}	{2}	{2}	{2}	{2}	{2}	{2}
Db	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{3}	{3}	{3}	{3}	{3}
Dc	{1, 2, 3}	{1, 2}	{1, 2}	{1, 2}	{1, 2}	{1, 2}	{1}	{1}
Dd	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2}	{1, 2}	{1}

Tabela 01 - Visualização dos passos do algoritmo AC-3

Os passos em negrito foram responsáveis por adicionar arcos novamente: $(A < B)$ causou a reentrada de $(C < A)$, que causou a de $(D = C)$. **No final, o algoritmo conseguiu tornar o CSP arco-consistente.**

5.2. Busca com backtracking

Ao realizar buscas por soluções em CSPs que não podem ser resolvidos por inferências, é possível observar que **os ramos de busca crescem em quantidade muito rapidamente**. De acordo com [2](Russell & Norvig, 2009), "Geramos uma árvore com $n!\,d^n$ folhas, embora existam apenas d^n atribuições completas possíveis!".

Essa quantidade explosiva de folhas ocorre devido a ignorarmos uma propriedade muito importante dos CSPs: a **comutatividade**. A comutatividade diz que a ordem dos fatores não altera o resultado, portanto só precisamos considerar **uma variável por nó da árvore de busca**.

Os algoritmos de busca com backtracking em CSPs funcionam com a seguinte estratégia: utilizam uma **busca em profundidade** que **atribui valor a uma variável de cada vez** e **retorna para o nó anterior quando uma variável não tem mais valores válidos** para atribuição. [2](Russell & Norvig, 2009)

Para melhorar o desempenho destes algoritmos, existem alguns métodos diferentes. Eles são derivados das seguintes questões:

- Qual variável deve ser atribuída em seguida e qual a ordem em que seus valores devem ser experimentados?
- Que inferências devem ser realizadas a cada etapa de busca?
- Quando a busca chega a uma atribuição que viola uma restrição, a busca pode evitar que essa atribuição se repita?

5.2.1. Ordenação de variáveis e valores

Derivada da primeira questão, a ordenação de variáveis e valores pode ocorrer de algumas formas diferentes. São elas:

- **Ordenação de variáveis:**
  - Estática: escolhe de acordo com a ordem em que as variáveis são declaradas no CSP;
  - Aleatória: escolhe de modo aleatório;
  - Valores mínimos restantes: também chamada de "primeira falha", escolhe primeiro a variável com o menor número de valores válidos para atribuição em determinado momento;

- Heurística de grau: escolhe a variável envolvida no maior número de restrições com outras variáveis não atribuídas, tentando reduzir o número de ramos gerados futuramente;

- **Ordenação de valores:**
  - Valores menos restritivos: escolhe o valor que elimina o menor número possível de opções para as variáveis vizinhas;

Portanto, podemos dizer que, geralmente, escolhemos primeiro as **variáveis com mais chance de falhar primeiro** e os **valores com mais chance de falhar por último**. Deste modo, temos a possibilidade de reduzir o tamanho da árvore de busca gerada e também diminuir o tempo necessário para encontrar uma solução.

5.2.2. Intercalando busca e inferências

Como visto anteriormente, o AC-3 é responsável por diminuir os domínios das variáveis antes da realização de uma busca. Entretanto, as **inferências** realizadas podem ser ainda mais úteis, pois também podemos **realizá-las durante a busca**.

Essa estratégia tem como base realizar inferências buscando diminuir os domínios das variáveis a cada vez que um valor é atribuído a uma variável qualquer, ou seja, **a cada passo da busca**. Uma das formas mais comuns de realizarmos esse processo é através do algoritmo de **Forward Checking** ou **Verificação à frente**.

Cada vez que uma variável receber um valor, o Forward Checking irá estabelecer a arco-consistência desta variável. Ao utilizar esse método, não precisamos utilizar o AC-3 como forma de pré-processamento, já que a arco-consistência vai sendo calculada dinamicamente.

Entretanto, o Forward Checking pode deixar passar uma inconsistência. Isso ocorre devido a sua característica de realizar a arco-consistência na variável atual, mas não realizá-la para as outras variáveis. Portanto, foi criado o algoritmo de **Manutenção da Consistência de Arco** ou **MAC** (Maintaining Arc Consistency) que funciona como uma variação do AC-3 capaz de detectar tal inconsistência, o que o torna superior ao Forward Checking.

5.2.3. Backtracking inteligente

A busca com backtracking comum utiliza o chamado **backtracking cronológico**, onde o algoritmo retorna para a tomada de decisão mais recente. Uma abordagem mais inteligente é **retornar a atribuição de valor a uma variável que causou o backtracking**, ou seja, que **tornou inválido um valor da variável atual**.

Para realizar esse método, precisamos acompanhar o conjunto de atribuições que causaram o conflito e **retornar para a atribuição mais recente deste conjunto**. Este conjunto é chamado de **conjunto de conflito**. Além disso, podemos encontrar e separar um conjunto mínimo de variáveis daquelas presentes no conjunto de conflito, processo chamado de **aprendizado de restrição**. Ao separar essas variáveis e seus valores, podemos **armazená-las com o intuito de impedir que sejam utilizadas novamente**. Esse processo também pode ser utilizado em algoritmos de Forward Checking, não somente os de backtracking, e se tornou essencial para algoritmos modernos de CSPs.

5.3. Busca local

Os algoritmos de busca local são muito eficazes na resolução de CSPs. Sua característica é **atribuir valores para todas as variáveis**, gerando um estado inicial que, na maioria das vezes, está violando várias restrições. O algoritmo então irá **modificar o valor de uma variável a cada passo**, buscando **corrigir as violações que está realizando** no momento.

Entretanto, existe um ponto de dúvida nesse processo: **qual critério deve ser utilizado na atribuição de um novo valor para as variáveis**? A resposta é a seguinte: a **heurística de conflitos mínimos**. Ela se caracteriza por escolher um novo valor com o intuito de diminuir ao máximo os conflitos atuais daquela variável.

Além da heurística de conflitos mínimos, também podemos utilizar a técnica de **ponderação de restrição**, a qual atribui um peso (inicialmente 1) para cada restrição. A cada passo, o algoritmo escolherá o par de variável/valor que irá minimizar o valor da soma dos pesos das restrições e ajustará os pesos de acordo, incrementando o valor das restrições violadas no estado atual.

6. Estrutura de problemas

Para facilitar a resolução de um CSP, podemos realizar um processo comum para a resolução de diversos problemas complexos: **dividi-lo em subproblemas**. Ao utilizarmos o exemplo de coloração do mapa da Austrália (Figura 02), podemos ver facilmente que a Tasmânia está desconectada do resto do mapa, portanto uma solução para a Tasmânia sempre poderá se conectar sem problemas com uma solução com o restante do mapa.

Um bom método para verificar a independência entre variáveis é a busca por **componentes conectados** no grafo de restrição, os quais cada um se tornará um subproblema diferente.

Entretanto, subproblemas completamente independentes como os da Tasmânia são raros, mas felizmente podemos utilizar outras estruturas de grafos para encontrar uma resolução de forma fácil. Um exemplo é transformar o grafo de restrições em uma árvore onde duas variáveis estão ligadas somente por um único caminho. Neste caso podemos utilizar uma **consistência orientada de arco**, que de acordo com [2](Russell & Norvig, 2009), é "... **uma ordenação de variáveis** $X1, X2, ..., Xn$ **se e somente se cada** Xi **for arco-consistente com cada** Xj **para** $j > i$.".

Para resolver CSPs dessa maneira, a ordem das variáveis deve estar de acordo com o conceito de **classificação topológica**: cada variável aparece após sua variável pai, conforme mostra a Figura 03.

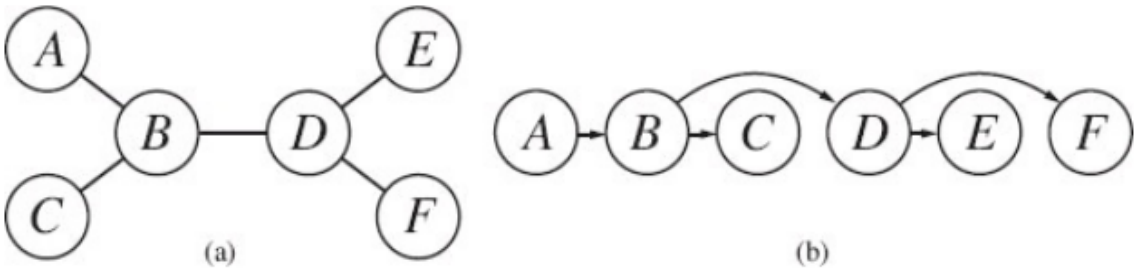

Figura 03 - (a) Grafo de restrições; (b) Árvore reordenada em classificação topológica

Ao orientar a árvore de acordo com essa classificação, podemos então torná-la arco-consistente e, consequentemente, atribuir valores para as variáveis de acordo com sua ordem que, no caso da Figura 03, é de A até F.

Entretanto, nem todo grafo de restrições pode ser transformado em uma árvore tão facilmente. Portanto, foram desenvolvidas duas formas de realizar tal feito, as quais serão comentadas a seguir.

6.1. Remover nós do grafo

No caso da Figura 02, podemos **retirar um nó** (Austrália Meridional) do grafo **ao fixar seu valor** e **diminuir os domínios de seus vizinhos**. É importante comentar que o número de nós removidos é variável de acordo com o grafo e recebe o nome de **conjunto de corte de ciclo (CCC)**. Além disso, caso o algoritmo não encontre uma solução de imediato, podemos configurá-lo para tentar novamente, só que desta vez atribuindo valores diferentes às variáveis presentes no CCC, até acabar com as possíveis variações nos valores destas ou encontrar uma solução.

6.2. Condensar os nós do grafo

Se baseia em **decompor o grafo de restrições em um conjunto de subproblemas conectados em forma de árvore**. Neste caso, cada subproblema é resolvido independentemente e suas soluções são combinadas posteriormente. Segundo [2](Russell & Norvig, 2009), uma decomposição em árvore deve satisfazer os seguintes requisitos:

- Toda variável no problema original aparece em **pelo menos** um dos subproblemas.
- Se duas variáveis estiverem conectadas por uma restrição no problema original, elas deverão aparecer juntas (e juntamente com a restrição) em **pelo menos** um dos subproblemas.
- Se uma variável aparecer em dois subproblemas na árvore, ela deverá aparecer em **todo** subproblema ao longo do caminho que conecta esses dois subproblemas.

Se algum dos subproblemas provar não ter solução enquanto os resolvemos, então pode-se concluir que não há solução geral. Se todos apresentarem solução, então podemos tentar construir uma solução geral de acordo com a estratégia da seção 6.

7. Exemplo de código

Como exemplo de código, irei utilizar um algoritmo de resolução de CSP através de **Backtracking** com variáveis ordenadas a partir do método **Valores mínimos restantes** e valores ordenados a partir do método **Valores menos restritivos**. Os valores do CSP são de acordo com o exemplo dado na seção 5.1.

```
# Backtracking CSP com variáveis e valores ordenados

def backtracking_search(csp):
  def is_consistent(assignment, var, value):
    # Checa se atribuir um valor para uma variável é consistente
    for other_var in assignment:
      if (var, other_var) in csp['constraints']:
        constraint = csp['constraints'][(var, other_var)]
        if not constraint(value, assignment[other_var]):
          return False
      if (other_var, var) in csp['constraints']:
        constraint = csp['constraints'][(other_var, var)]
        if not constraint(assignment[other_var], value):
          return False

    return True

  def select_unassigned_variable(assignment):
    # Seleciona uma variável sem valor através da ordem MRV
    unassigned = [var for var in csp['variables'] if var not in
assignment]
    return min(unassigned, key=lambda var: len(csp['domains'][var]))

  def order_domain_values(var, assignment):
    # Ordena o valor do domínio através da ordem LCV.
    def count_constraints(value):
      count = 0
      for other_var in csp['variables']:
        if other_var != var and other_var not in assignment:
          if (var, other_var) in csp['constraints']:
            constraint = csp['constraints'][(var, other_var)]
            count += sum(1 for other_value in csp['domains'][other_var]
if not constraint(value, other_value))
          if (other_var, var) in csp['constraints']:
            constraint = csp['constraints'][(other_var, var)]
```

```
            count += sum(1 for other_value in csp['domains'][other_var]
if not constraint(other_value, value))
      return count

    return sorted(csp['domains'][var], key=count_constraints)

  def backtrack(assignment):
    # Backtracking recursivo.
    if len(assignment) == len(csp['variables']):
      return assignment

    var = select_unassigned_variable(assignment)
    for value in order_domain_values(var, assignment):
      if is_consistent(assignment, var, value):
        assignment[var] = value
        result = backtrack(assignment)
        if result is not None:
          return result
        del assignment[var]

    return None

  return backtrack({})

# Código rodando
if __name__ == "__main__":
  # Definição do CSP
  csp = {
    'variables': {'A', 'B', 'C', 'D'},
    'domains': {
      'A': {1, 2, 3},
      'B': {1, 2, 3},
      'C': {1, 2, 3},
      'D': {1, 2, 3}
    },
    'constraints': {
      ('A', 'B'): lambda a, b: a < b,
      ('B', 'A'): lambda b, a: b > a,
      ('A', 'C'): lambda a, c: a > c,
      ('C', 'A'): lambda c, a: c < a,
      ('C', 'D'): lambda c, d: c == d,
      ('D', 'C'): lambda d, c: d == c
```

```
    }
  }

  solution = backtracking_search(csp)
  if solution:
    print("Solução encontrada:", solution)
  else:
    print("Não existe solução")
```

8. Impressões sobre o conteúdo

Os Problemas de Satisfação de Condições aparentam ser muito simples, afinal são compostos de somente 3 partes distintas: as variáveis, seus domínios e as restrições. Porém, ao estudar as diferentes consistências e os algoritmos responsáveis por sua resolução, fica claro que, mesmo parecendo simples, os CSPs são capazes de representar contextos e resolver questões complexas, como o agendamento de voos de uma companhia aérea.

O mais incrível é a quantidade de otimizações que torna possível a resolução destes problemas em custos de tempo, computacional e de memória razoáveis. Desde o modo como restrições n-árias podem ser transformadas em binárias até as diferentes formas de otimizar um algoritmo de Backtracking: seja através do uso de heurísticas que apontam a melhor ordem de escolha das variávei, do uso de Forward Checking ou tornando o processo de backtracking em si mais inteligente; o esforço teórico por trás da relativa simplicidade atual dos algoritmos (comparado com o que poderia ser) demonstra ser de suma importância para a viabilidade da resolução dos CSPs.

Por fim, a representação fatorada dos estados do ambiente mostrou um aumento de potencial gigantesco na diversidade dos problemas que os agentes de inteligência artificial conseguem resolver. Afinal, os ambientes da vida real estão longe de serem atômicos.

9. Referências

[1] SOARES, Fabiano Araujo. Slides da aula 10 à 11. Apresentação do PowerPoint.

[2] Russell, S. & Norvig, P. **Artificial Intelligence - A Modern Approach**. 3ª ed. Pearson Education, Inc. 2009.