FGA0221 - Inteligência Artificial

Portfólio 02

Janeiro, 2025

Universidade de Brasília

Faculdade UnB Gama

Tema do portfólio:

Resolvendo problemas por busca.

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

Janeiro, 2025

Sumário

Janeiro, 2025

1. Agente de soluções de problemas

Os agentes de soluções de problemas são caracterizados pelos seguintes pontos:

- São agentes **baseados em objetivos**, capazes de considerar os impactos que suas ações terão no futuro, realizando aquelas que o aproximam de seus objetivos a longo termo;
- O **ambiente** em que se encontra é **representado atômicamente**, ou seja, eles veem seus ambientes de forma indivisível, onde cada estado é único e não tem nada em comum com outros estados;

Para que um agente seja capaz de solucionar problemas, existem alguns passos que devem ser tomados. O primeiro entre eles é a **formulação de seus objetivos**, pois estes irão ajudar o agente a se comportar da maneira correta, descartar possíveis ações que vão contra os objetivos e, no geral, tornando o processo de tomada de decisão mais simples e eficaz.

O segundo passo é a **formulação de problemas**, onde são decididos quais ações e estados do ambiente o agente deve levar em consideração, tomando como base o objetivo definido. O agente será então responsável por realizar uma **sequência de ações** de modo a alcançar seu objetivo.

O processo para encontrar a sequência correta de ações é chamado de **busca** e é caracterizado pela examinação de ações futuras do agente, "buscando" a sequência ótima de ações que levarão ao objetivo. Os algoritmos que realizam este processo, chamados de **algoritmos de busca**, são responsáveis por retornar uma solução (sequência de ações) para cada problema que receber.

É importante dizer que o agente, enquanto realiza as ações recomendadas pelo algoritmo de busca, ignora suas percepções do ambiente, pois já sabe qual ação irá realizar sem precisar destas informações. Um sistema onde o agente realiza suas ações deste modo é conhecido como um sistema de **malha aberta**. [2](Russell & Norvig, 2009)

2. Problemas de malha aberta e de malha fechada

Sistemas de malha aberta e sistemas de malha fechada são conceitos básicos da teoria de Controle de Sistemas, a qual tem como objetivo otimizar um processo para chegar em uma solução ótima a partir de uma entrada. Estes dois tipos de sistemas são utilizados em diferentes categorias de problemas, cada um tendo vantagens e desvantagens próprias. [3](Cardoso, 2020)

2.1. Malha aberta

Sistemas de malha aberta são caracterizados por **"ignorar" suas percepções do ambiente** uma vez que tem sua sequência de ações definida. Agentes que se utilizam disso trabalham melhor em ambientes mais simples e/ou com nenhuma possibilidade de perturbações externas. Isso ocorre devido ao agente ser **incapaz de recalcular sua sequência de ações** quando está no processo de realizá-las. Para exemplificar, podemos utilizar o sistema de navegação de um carro, o qual precisa ir de uma cidade à outra. Vamos imaginar o seguinte cenário:

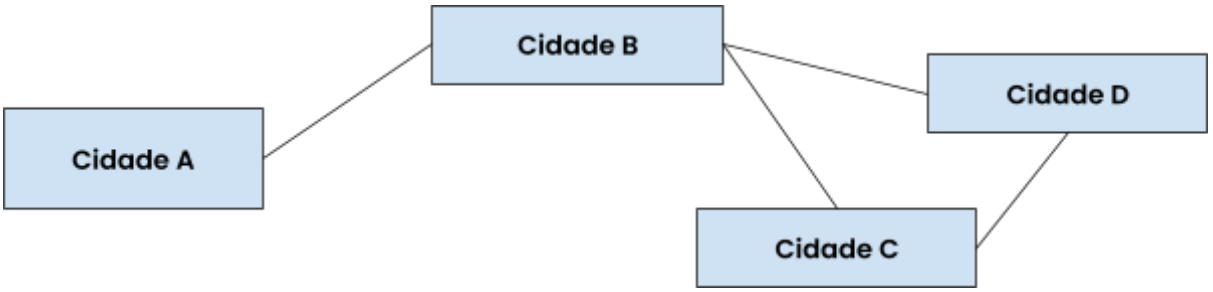

Figura 01 - Rotas entre cidades

Cidade A -> Cidade B = 30 minutos;
Cidade B -> Cidade C = 20 minutos;
Cidade B -> Cidade D = 25 minutos;
Cidade C -> Cidade D = 10 minutos;

Neste caso, o objetivo do agente é ir da Cidade A para a Cidade D no menor tempo possível. Ao buscar pela sequência de ações mais eficaz, iria ser encontrado o seguinte caminho:

Cidade A -> Cidade B -> Cidade D (30 + 25 = 55 minutos)

Em um problema de malha aberta, o agente irá seguir este caminho independente do que ocorrer durante sua trajetória. Se nada ocorrer e os tempos para ir de uma cidade à outra continuarem iguais, a solução será ótima, assim como o esperado. Porém, **se o ambiente sofrer uma mudança** no período em que o agente está em deslocamento, **a solução pode não ser mais a ideal**. Apesar de tal desvantagem, sistemas de malha aberta ainda são muito utilizados por serem de **fácil implementação e de baixo custo**.

2.2. Malha fechada

Sistemas de malha fechada tem sua principal vantagem e diferença com os de malha aberta devido a seguinte característica: são **capazes de realizar mudanças nas ações** que o agente irá tomar durante sua realização. Isso ocorre pois estes sistemas estão sempre **processando suas percepções do ambiente** e verificando se houve alguma mudança que irá interferir na capacidade do agente de chegar a seu objetivo.

Vamos retomar o exemplo anterior. O agente decide seguir o caminho:

Cidade A -> Cidade B -> Cidade D (30 + 25 = 55 minutos)

Entretanto, quando chega na Cidade B, ele percebe que os tempos de deslocamento entre as cidades mudou para os seguintes valores:

Cidade A -> Cidade B = 30 minutos;
Cidade B -> Cidade C = 20 minutos;
Cidade B -> Cidade D = 35 minutos;
Cidade C -> Cidade D = 10 minutos;

Isso pode ter ocorrido devido a vários fatores, como acidentes ou chuva na estrada. Caso o agente continuasse a seguir o mesmo caminho, ele iria levar um total de 65 minutos, o que não seria a melhor solução. Portanto, o sistema de malha fechada decide mudar sua sequência de ações, seguindo agora o seguinte caminho, fazendo a solução voltar a ser ótima:

Cidade A -> Cidade B -> Cidade C -> Cidade D (30 + 20 + 10 = 60 minutos)

As desvantagens deste tipo de sistema são um **custo mais elevado de processamento** e uma **maior complexidade de implementação**.

3. Algoritmos de Busca

Como dito anteriormente, os algoritmos de busca são responsáveis por encontrar uma sequência de ações que irá permitir ao agente alcançar seu objetivo. Para realizar essa busca são utilizadas estruturas de dados como **filas, pilhas e heaps**, além disso, as possíveis sequências geralmente são visualizadas na forma de outra estrutura de dados: a árvore. A **árvore de busca**, como é conhecida, tem o estado inicial do ambiente na raiz, os ramos são as ações e os nós correspondem aos outros estados do ambiente.

Um ponto a ser destacado é que, em muitos casos, existem **mais de uma sequência viável de ações** para solucionar determinado problema. Neste caso, algoritmos de busca diferentes possivelmente terão soluções diferentes uns dos outros. Os motivos para isso ocorrer podem ser:

- Nem todos os algoritmos se preocupam em alcançar uma solução ótima, **aceitando a primeira solução** que encontrarem;
- Existe mais de uma solução ótima e um algoritmo irá encontrar umas dessas soluções primeiro enquanto outro algoritmo irá se deparar primeiro com outra e no final, como **é comum manter a primeira solução encontrada**, a solução retornada será diferente;

Estes dois motivos elencam uma das diferenças entre algoritmos de busca: **ele retorna uma solução ótima, ou uma solução qualquer?** Porém, esse não é o único método de categorização desses algoritmos, pois podemos separá-los também de acordo com seus métodos de busca: **ela é realizada de forma cega ou informada?**; **qual é a ordem em que os nós são explorados?**

Ao utilizar um algoritmo de busca para resolver um problema, é importante **escolher aquele que mais se encaixa no contexto**, afinal, utilizar um algoritmo que não se preocupa com a solução ser ou não ótima em um problema onde você precisa da melhor solução possível irá, muito provavelmente, causar problemas.

3.1. Busca cega

Um dos métodos que a busca pode ser realizada é conhecido como Busca cega. Neste método o agente responsável não tem **nenhum conhecimento prévio** de onde a solução pode ser encontrada, portanto, ele procura por esta de

“olhos fechados”. Se o agente der sorte, a solução será encontrada rapidamente, se não, a solução pode estar, no pior dos casos, no último estado do ambiente que o agente irá olhar.

Estes algoritmos vão explorando e verificando os nós visitados, de um em um, procurando por nós objetivos. Devido às **diferentes formas de ordenar a exploração dos nós** de uma árvore de busca, diferentes algoritmos foram criados. Seguem aqui alguns **exemplos de algoritmos de busca cega**, cada um com sua ordem de exploração:

3.1.1. BFS (Breadth First Search)

Também conhecido como Busca em Largura, é baseado na estratégia de **olhar todos os nós com um mesmo grau de profundidade primeiro**, antes de ir para uma profundidade maior.

Um exemplo é procurar, em uma árvore genealógica, qual o parente mais próximo de uma pessoa que tinha determinado tipo sanguíneo. Neste caso, a busca em largura irá primeiro ver os pais daquela pessoa, depois os avós, depois os bisavós, e assim por diante, seguindo a ordem de ‘**quem está mais próximo da origem da busca primeiro**’.

3.1.2. Busca de Custo Uniforme

Tem como base o funcionamento da Busca em Largura, mas, ao invés de escolher o nó mais próximo da origem para explorar, esta busca **escolhe o nó com o menor custo de caminho para chegar até ele**.

Por exemplo, queremos sair da Cidade A e chegar na Cidade C, e temos os seguintes caminhos:

Cidade A -> Cidade B = 10 minutos;
Cidade A -> Cidade C = 30 minutos;
Cidade B -> Cidade C = 12 minutos;

Na Busca em Largura escolheríamos o caminho que leva da Cidade A à Cidade C em 30 minutos, mas na Busca de Custo Uniforme iríamos escolher o seguinte caminho: Cidade A -> Cidade B -> Cidade C (10 + 12 = 22 minutos).

3.1.3. DFS (Depth First Search)

Também conhecido como Busca em Profundidade, tem como base a estratégia de **se aprofundar o máximo possível na árvore de busca**, seguindo a borda desta até chegar no nó mais profundo, **só depois irá retornar e começar a 'alargar' a busca**.

Seguindo o exemplo da árvore genealógica, a busca em profundidade irá ver o pai da pessoa, o avô por parte de pai, o pai deste avô, e assim por diante, até chegar ao fim da árvore genealógica, vamos dizer que no tataravô. É importante destacar que esta busca **sempre irá priorizar ir para nós mais profundos**, então se ela chegar ao final, retornar um nó e ver que consegue ir para outro nó mais fundo novamente, ela irá fazer isso ao invés de retornar de forma contínua (sair do tataravô para o bisavô, e ir para a tataravó ao invés de voltar para o avô).

Em alguns problemas, principalmente em espaços de estados infinitos, esse comportamento de ir o mais fundo possível **pode gerar problemas**, como a possibilidade de ignorar uma solução bem mais próxima ou um maior custo de tempo e processamento devido a "falta de sorte". Devido a isso, existem algoritmos que derivam da DFS. Eles são chamados de **Busca em Profundidade Limitada**, onde é colocado um limite na profundidade que o algoritmo irá explorar, e **Busca de Aprofundamento Iterativo (IDS)**, onde o limite de profundidade é gradualmente expandido até encontrar o primeiro nó objetivo.

3.1.4. Busca bidirecional

Nesta busca é utilizada a estratégia de rodar **duas buscas ao mesmo tempo**: uma a partir do estado inicial e outra a partir do estado objetivo; e esperar que elas se **encontrem em um ponto intermediário** e complementem uma à outra. É comum que o método pelo qual as duas são realizadas seja a Busca em Largura, mas é possível que sejam realizadas com qualquer um dos métodos discutidos acima.

3.2. Busca informada

A busca informada tem como principal diferença, quando comparada a busca cega, a seguinte característica: ela **tem conhecimento sobre os estados**

do ambiente que vai além da definição do problema. Portanto, é possível ter um "norte" de onde a solução irá ser encontrada antes mesmo de começar o processo de busca, permitindo encontrar soluções de forma mais eficiente. Tais informações adicionais são estimativas e vêm na forma de uma **Função Heurística**.

Alguns dos exemplos de algoritmos de busca informada são:

3.2.1. Greedy Best First Search

A base deste algoritmo é a estratégia de, **a cada passo**, **chegar o mais perto possível do objetivo**, tendo como critério as estimativas presentes na Função Heurística. A vantagem dessa estratégia é que o **custo necessário para encontrar uma solução é mínimo**. Sua desvantagem é que o algoritmo tem uma "visão de túnel", ou seja, **sempre olha para o melhor caminho** que está na sua frente **sem considerar outras opções** que, a longo prazo, poderiam ser melhores.

Para exemplificar, podemos utilizar o seguinte contexto: O agente precisa sair da Cidade A e chegar na Cidade D, percorrendo o menor caminho possível; A Função Heurística terá os valores da distância em linha reta entre as cidades.

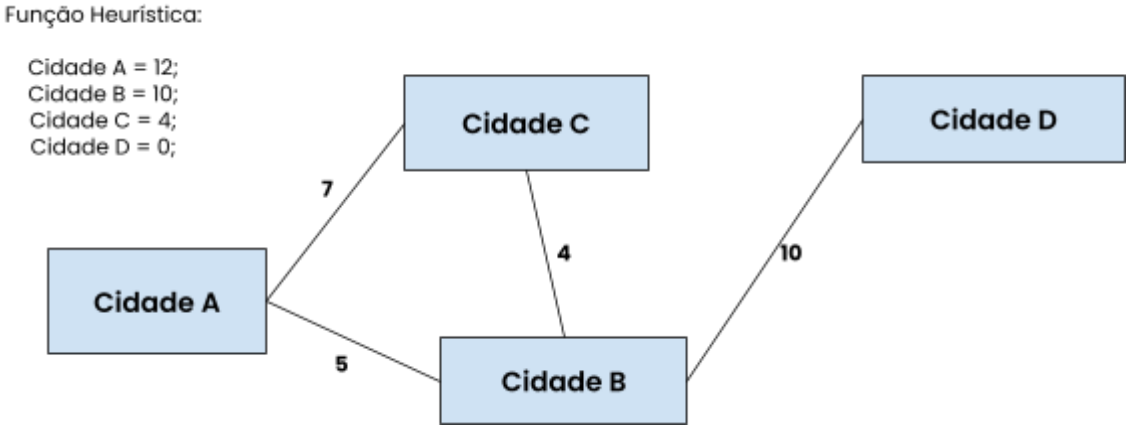

Figura 02 - Árvore de busca e função heurística

Neste caso, o algoritmo iria seguir o caminho: Cidade A -> Cidade C -> Cidade B -> Cidade D. Entretanto, ao calcular a distância percorrida, vemos que este caminho tem custo igual a 21, enquanto o caminho Cidade A -> Cidade B -> Cidade D tem um custo igual a 15. Portanto, neste caso, o algoritmo **encontrou uma solução não-ótima devido à "ganância" de chegar logo ao objetivo**.

3.2.2. Busca A*

É o algoritmo mais comum quando se fala de busca informada. Sua estratégia de busca se baseia, assim como o Greedy Best First Search, em utilizar a função heurística e escolher o nó com menor custo estimado até o objetivo. Porém, ele introduz outro valor além da heurística para definir qual o nó com menor custo em determinado momento: **o custo do caminho para chegar até aquele nó**. Com isso, a equação que define o custo de um nó é igual a:

$$F(n) = g(n) + h(n)$$

- g(n) é o custo do caminho até o nó;
- h(n) é o valor da heurística;

Com essa mudança, o algoritmo A* se prova capaz de fornecer soluções melhores ou iguais às do algoritmo Greedy Best First Search, com a desvantagem de um **custo mais elevado de busca**. É importante comentar que, de acordo com [2](Russell & Norvig, 2009), o algoritmo só funciona de maneira ótima caso a função heurística for:

- **Admissível:** "... nunca superestima o custo de atingir o objetivo";
- Ou **consistente** (busca em grafos): "... para cada nó *n* e para todo sucessor *n'* de *n* gerado por uma ação *a*, o custo estimado de alcançar o objetivo de *n* não for maior do que o custo do passo de chegar a *n'* mais o custo estimado de alcançar o objetivo de *n'*."

Ao tomar como base o exemplo da Figura 02, podemos ver como o algoritmo funciona. Primeiro ele calcularia os custos da Cidade B: $5 + 10 = 15$; e da Cidade C: $7 + 4 = 11$; e iria para a segunda, onde calcularia um custo de $11 + 10 = 21$ para a Cidade B. Ao invés de seguir este caminho, o algoritmo percebe que consegue chegar na Cidade B com um custo menor, através da Cidade A. Portanto, ele retorna e segue pelo caminho Cidade A -> Cidade B, onde percebe que consegue chegar à Cidade D com o custo de 15 e, em seguida, retorna a solução: Cidade A -> Cidade B -> Cidade D.

4. Funções Heurísticas

Como dito na seção anterior, os algoritmos de busca informada utilizam a chamada Função Heurística para ter um "norte" de onde a solução para um problema estaria. Essa função é composta por valores estimados até a solução, partindo de cada um dos estados do problema. Entretanto, **não existe uma função heurística única para cada problema**.

Existem contextos onde uma função heurística específica é muito mais eficiente do que as outras e também contextos onde todas as heurísticas têm uma eficiência parecida, entretanto, nem sempre é fácil "imaginar" uma heurística viável para um problema. De acordo com [2](Russell & Norvig, 2009), existem algumas maneiras do agente gerar tais heurísticas de forma automática. Elas são:

- Problemas relaxados:

  As heurísticas que o agente gera são **os valores das soluções ótimas de problemas relaxados** do contexto original. Os problemas relaxados são definidos como **problemas com menos restrições nas ações** para resolvê-los do que o original.

- Subproblemas:

  As heurísticas também podem ser **derivadas do custo da solução ótima de subproblemas** do problema original. Neste caso, surge a ideia de **bancos de dados de padrões**, que é armazenar os custos de solução de todas as instâncias possíveis de um subproblema. Cada banco de dados origina uma heurística admissível.

- Aprender com a experiência:

  Neste caso, as heurísticas seriam geradas a partir do custo das soluções ótimas de cada instância do problema. O agente seria responsável por, a cada vez que resolver um determinado problema, **alinhar alguma característica de cada estado com o custo para chegar a solução** naquela instância e usar isso para **prever o custo de solução na próxima vez** que for resolver este problema. Conforme o agente vai aprendendo, as previsões vão ficando mais precisas e as heurísticas ficando melhores, possibilitando resolver o problema de forma cada vez mais rápida, até alcançar uma otimização.

5. Busca em ambientes complexos

Os algoritmos discutidos anteriormente utilizam diferentes estratégias para descobrir uma sequência de ações para uma solução, seja ela ótima ou não, ou seja, eles se preocupam com o caminho até o estado objetivo. Porém, nem todo problema tem essa necessidade, **se preocupando somente com o estado final**.

Além disso, estes algoritmos são direcionados para funcionar em ambientes observáveis, determinísticos e conhecidos. Consequentemente, em **ambientes parcialmente observáveis e/ou não-determinísticos** estes algoritmos irão apresentar um comportamento falho. Com o objetivo de resolver tais problemas, foram criados algoritmos próprios para estes contextos. Os algoritmos que resolvem problemas onde somente o estado final importa são conhecidos como: **algoritmos de busca local**.

A estratégia utilizada por esses algoritmos é **se preocupar somente com o estado atual e seus vizinhos**, tomando decisões locais para encontrar uma solução para o problema. Esse comportamento gera duas vantagens e uma desvantagem quando comparado aos algoritmos de busca vistos nas seções anteriores. Elas são:

Vantagens:

- Usam pouquíssima memória;
- Conseguem encontrar boas soluções em espaços de ambientes grandes ou infinitos (contínuos);

Desvantagem:

- Não exploram o ambiente de busca de maneira sistemática, podendo nunca explorar um local onde uma solução se encontra;

Os algoritmos de busca local também são muito utilizados na resolução de **problemas de otimização**, onde precisam encontrar o melhor estado do ambiente de acordo com uma **função objetivo**. Para visualizar estes tipos de problema, é comum a utilização da **topologia do espaço de estados**, exemplificada pela figura a seguir.

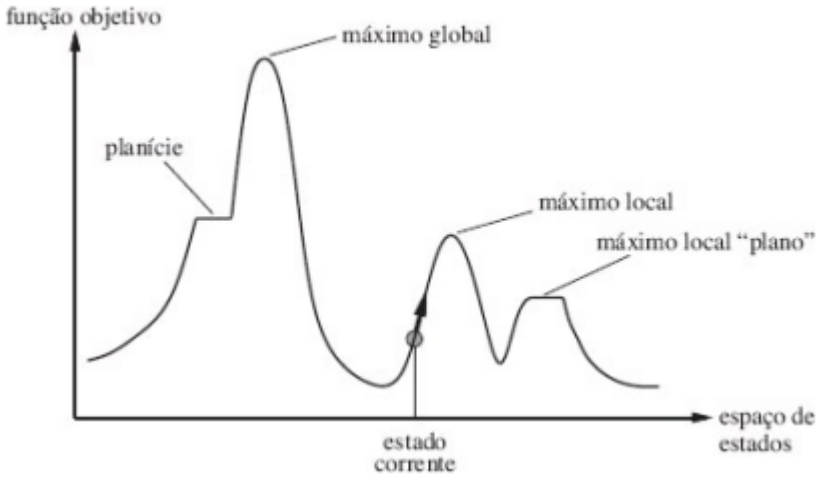

Figura 03 - Topologia de um espaço de estados

Neste caso, o objetivo é encontrar o ponto máximo da função objetivo, portanto é caracterizado como um algoritmo de **Hill Climbing (Subida de Encosta)**. Caso o objetivo fosse encontrar o ponto mínimo da função, então seria um algoritmo de **Gradient Descent (Descida de Gradiente)**.

No exemplo da Figura 03, o algoritmo irá olhar para o valor do estado atual e compará-lo com os estados adjacentes. **Se um estado adjacente tiver valor maior que o atual, então ele irá se tornar o novo estado atual**, se isso não ocorrer significa que o estado atual é um máximo, pois não importa para qual lado ele andar, o valor será menor que o dele.

Devido ao algoritmo não realizar a busca de forma sistemática, é bem possível que o máximo encontrado não seja o chamado **máximo global** (solução ótima), assim como no exemplo da figura, em que a busca irá acabar quando o estado atual chegar ao **máximo local** mais próximo (solução).

Além disso, o algoritmo base pode retornar um "máximo falso", pois existe o conceito de **platôs**: um intervalo no espaço de estados onde todos os estados têm o mesmo valor. Os platôs podem ser subdivididos em duas categorias:

- **Máximo local plano**: onde não existe a possibilidade do valor continuar subindo;
- **Planície**: onde ainda é possível progredir;

Caso o platô for uma planície e o algoritmo só transformar um estado adjacente em atual se o valor for **maior**, então ele irá parar antes de verificar se a subida continua do outro lado do platô, retornando um "máximo falso". Portanto, é comum que algoritmos verifiquem se o valor de um estado adjacente é **maior**

ou igual ao do estado atual, impedindo que planícies atrapalhem na busca de uma solução. Consequentemente, para evitar que o algoritmo fique indo de um lado ao outro do de um máximo local plano, é importante impor um limite no número de "movimentos horizontais" que podem ser realizados, evitando assim um loop infinito.

Devido a cada problema ter uma topologia de estados distinta, foram criadas várias versões do Hill Climbing. São elas:

- Permitir **movimento horizontais**, como comentado acima;
- **Stochastic Hill Climbing**: Escolhe de forma aleatória o caminho de subida no caso de ambos os estados adjacentes serem maiores que o atual, ao invés de escolher aquele com o maior valor como ocorre normalmente;
- **First Choice Hill Climbing**: Escolhe de forma aleatória qual estado adjacente será verificado primeiro, mudando o estado atual quando o primeiro estado adjacente se provar maior que ele. É muito usado quando cada estado atual pode ter um número muito grande de estados adjacentes.
- **Random Restart Hill Climbing**: Executa diversas buscas seguidas umas das outras a partir de pontos iniciais aleatórios e, no final, verifica qual é a melhor solução entre aquelas retornadas.
- **Simulated Annealing**: Para evitar que a busca se prenda frequentemente em máximos locais, este algoritmo permite que um menor valor se torne o estado atual. Neste caso, o estado adjacente será escolhido de forma aleatória e se o valor for melhor, será escolhido de imediato, se não ele terá uma chance menor do que 100% de ser escolhido com base em quão ruim é o valor deste estado quando comparado ao atual.
- **Local Beam Search**: Ao invés de acompanhar somente um estado atual, este algoritmo utiliza K estados atuais, gerados de forma aleatória. Embora pareça com o Random Restart, tem como diferenças os fatos de estar realizando buscas em todos os estados de forma simultânea e fazendo com que os estados "foquem" nas áreas que estão retornando maior valor. Para evitar que os K estados foquem muito em uma mesma área, existe uma variação chamada de **Stochastic Local Beam Search** que escolhe estados adjacentes de forma aleatória.

6. Algoritmos genéticos

Os algoritmos genéticos são uma variação da Stochastic Local Beam Search na qual **os estados sucessores são uma combinação de dois ou mais estados pais**, os quais são escolhidos entre todos os estados do feixe de busca devido a serem os mais bem avaliados, ou seja, aqueles que estão mais próximos da solução.

Segundo [2](Russell & Norvig, 2009), o algoritmo inicia com um conjunto de K estados gerados de forma aleatória, o qual denominamos de **população**. Cada estado, ou **indivíduo**, é então representado através de uma cadeia de símbolos de um alfabeto finito, como letras ou algarismos.

Uma função chamada de **função de avaliação** ou **função de adaptação** é utilizada para avaliar a aptidão (seguindo a terminologia da teoria de evolução de Darwin) dos indivíduos de uma população. Os indivíduos mais aptos são então selecionados para realizar o processo de **reprodução** ou **recombinação**. Durante esse processo pode ocorrer uma **mutação**, onde um dos caracteres da cadeia de símbolos que forma um indivíduo é alterado de forma aleatória. Os indivíduos gerados através da reprodução serão então usados para preencher uma nova população e o processo se repete até que uma solução seja encontrada.

O algoritmo utiliza os seguintes parâmetros em sua execução:

- **Tamanho da população**: Quantidade de indivíduos;
- **DNA**: A representação de cada indivíduo;
- **Número de parentes**: Determina quantos indivíduos serão usados no processo de reprodução;
- **Função de avaliação**: Método de avaliação usado para escolher os indivíduos pais;
- **Procedimento de recombinação**: Método utilizado para recombinar os parentes (50% de cada, 25% de um e 75% de outro, ...);
- **Taxa de mutação**: Determina a probabilidade de ocorrer uma mutação durante a reprodução;
- **Nova geração**: Quantos indivíduos serão mantidos a cada geração?

Para que o algoritmo funcione da melhor maneira possível, é extremamente importante que estes parâmetros sejam ajustados para cada contexto em que o algoritmo é utilizado.

7. Exemplos de código

7.1. Algoritmo de busca cega

Um dos algoritmos brevemente discutidos na seção 3.1.3 foi o Depth Limited Search, uma variação do algoritmo Depth First Search.

```
def depthLimitedSearch(graph, start, goal, limit):
  def dlsRecursive(node, depth):
    if depth > limit:
      return False  # Excedeu o limite de profundidade
    if node == goal:
      return True  # Objetivo encontrado
    for neighbor in graph.get(node, []):
      if dlsRecursive(neighbor, depth + 1):
        return True
    return False

  return dlsRecursive(start, 0)


# Código rodando:
graph = {
  'A': ['B', 'C', 'D'],
  'B': ['E'],
  'C': ['G'],
  'D': ['H'],
  'E': [],
  'F': [],
  'G': [],
  'H': ['F']
}

startNode = 'A'
goalNode = 'F'
depthLimit = 2

result = depthLimitedSearch(graph, startNode, goalNode, depthLimit)

if result:
  print(f"Nó objetivo '{goalNode}' encontrado dentro do limite de
```

```
produndidade '{depthLimit}'.")
else:
  print(f"Nó objetivo '{goalNode}' NÃO foi encontrado dentro do limite
de produndidade '{depthLimit}'.")
```

7.2. Algoritmo de busca informada

A busca binária é considerada um dos algoritmos mais eficientes de busca em um vetor, pois é otimizada para fazer buscas em vetores com uma determinada característica: eles precisam estar ordenados. Ao usar o conhecimento desta característica, a busca binária consegue procurar por uma solução de forma mais eficiente.

```
def binarySearch(array, target):
  left = 0
  right = len(array) - 1

  while left <= right:
    mid = left + (right - left) // 2
    if array[mid] == target:
      return mid  # Retorna o valor encontrado
    elif array[mid] < target:
      left = mid + 1  # Procura na metade da direita
    else:
      right = mid - 1  # Procura na metade da esquerda

  return -1  # Caso não tenha encontrado


# Código rodando:
sortedArray = [1, 3, 5, 7, 9, 11, 13]
target = 13
result = binarySearch(sortedArray, target)

if result != -1:
  print(f"O número {target} foi encontrado na posição {result}.")
else:
  print(f"O número {target} não foi encontrado no vetor.")
```

7.3. Algoritmo de busca em ambiente complexo

O algoritmo a seguir realiza uma busca pelo máximo global de uma topologia de espaço de estados ao levar em conta a simples função: $sin(X) + Y$. Devido ao parâmetro atual onde X e Y variam de 0 à 10, podemos afirmar que o máximo global será um estado onde $sin(X) = 1$ e $Y = 10$, para um valor total de 11. Saber disso facilita a confirmação da eficácia do algoritmo conforme aumentamos seu número de iterações máximas (atualmente é 1000).

Além disso, o algoritmo utiliza o método de Simulated Annealing para permitir que valores menores sejam sucessores e escolhe os possíveis sucessores de forma aleatória.

```
import numpy as np
import random

def ambientSearch(objectiveFunction, stateSpaceBounds,
maxIterations=1000, initialTemperature=100, coolingRate=0.99):
  def randomState():
    return np.array([random.uniform(low, high) for low, high in
stateSpaceBounds])

  def perturbState(state):
    perturbation = np.array([random.uniform(-0.1, 0.1) for _ in state])
    newState = state + perturbation
    return np.clip(newState, [low for low, _ in stateSpaceBounds], [high
for _, high in stateSpaceBounds])

  currentState = randomState()
  currentValue = objectiveFunction(currentState)
  bestState, bestValue = currentState, currentValue
  temperature = initialTemperature

  for _ in range(maxIterations):
    newState = perturbState(currentState)
    newValue = objectiveFunction(newState)

    if newValue > currentValue or random.random() < np.exp((newValue -
currentValue) / temperature):
      currentState, currentValue = newState, newValue

    if currentValue > bestValue:
```

```
      bestState, bestValue = currentState, currentValue

    temperature *= coolingRate
    if temperature < 1e-6:
      break

  return bestState, bestValue

def objectiveFunction(state):
  x, y = state
  return np.sin(x) + y

# Código rodando
stateSpaceBounds = [(0, 10), (0, 10)]
bestState, bestValue = ambientSearch(objectiveFunction,
stateSpaceBounds)
print(f"Melhor estado: {bestState}, Melhor valor: {bestValue:.4f}")
```

8. Impressões sobre o conteúdo

Ao iniciar este tópico, me perguntei como exatamente os algoritmos de busca se conectam com a inteligência artificial. Porém, logo percebi que minha visão estava muito focada em somente “buscar um item em uma lista”, o que não faz juz a gama de possibilidades de utilização destes algoritmos.

A busca por uma sequência de ações que irá levar um agente ao seu objetivo se conecta com vários temas discutidos no portfólio anterior. A primeira conexão que percebi é que o processo de busca compõe parte da Função Agente, pois dita a lógica que irá guiar o agente no processo de alcançar seus objetivos. Além disso, a depender das propriedades do ambiente de tarefas e da representação de seus estados, o tipo de busca utilizada será diferente.

Ao explorar este conteúdo e conectá-lo com os passados, consegui então perceber a utilidade e importância dos algoritmos de busca para a inteligência artificial, o que me permitiu ter uma visão mais ampla de como essa tecnologia funciona em situações reais.

9. Referências

[1] SOARES, Fabiano Araujo. Slides da aula 07 à 09. Apresentação do PowerPoint.

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

[3] CARDOSO, Matheus. Conceitos Básicos de Controle: Malha Aberta X Malha Fechada. **IEEE RAS**, 2020. Disponível em: https://edu.ieee.org/br-ufcgras/conceitos-basicos-de-controle-malha-aberta-x-malha-fechada/. Acesso em: 31 de dez. de 2024.

[4] THAILAPPAN, Dhanya. An Introduction to Problem-Solving using Search Algorithms for Beginners. **Analytics Vidhya**, 2024. Disponível em: https://www.analyticsvidhya.com/blog/2021/10/an-introduction-to-problem-solving-using-search-algorithms-for-beginners/. Acesso em: 01 de jan. de 2025.

[5] Compreendendo os algoritmos de busca de IA. **Elastic**. Disponível em: https://www.elastic.co/pt/blog/understanding-ai-search-algorithms. Acesso em 02 de jan. de 2025.

[6] CARVALHO, André Ponce de Leon F. de. Algoritmos Genéticos. **ICMC USP**, 2009. Disponível em: https://sites.icmc.usp.br/andre/research/genetic/. Acesso em: 03 de jan. de 2025.