Portifólio 2

Inteligência Artificial - Resolvendo problemas por busca

Autor: Wildemberg Sales da Silva Junior

Matrícula: 202017503

Data: 24/12/2024

Instituição/Universidade: Universidade de Brasília(UnB)

Disciplina: Inteligência Artificial - FGA0221

Resumo

Este artigo aborda temas relacionados a soluções de problemas por meio de buscas realizadas por agentes inteligentes. Foi explorado o que são agentes de resolução de problemas e como eles atuam na busca de soluções, analisando as características e propriedades dos ambientes em que esses modelos de agentes atuam. Também obtivemos uma visão geral sobre algoritmos de buscas, desde sistemáticos, até em ambientes complexos, vendo suas aplicações em problemas reais, e explorando novos modelos de algoritmos não discutidos. Por fim, analisamos algoritmos genéticos que é um pilar importante no conceito de busca por agentes.

Visão Geral

Este artigo traz informações relevantes para o estudo de Inteligência Artificial, pois aborda um tema muito utilizado no dia a dia que é a busca, todos os dias realizamos diversos tipos de buscas e em diferentes situações, então entender esses tipos de agentes inteligentes e os modelos de algoritmos, agrega um valor muito valioso, pois expande o pensamento e a visão de implementações em que pode ser utilizado esses agentes.

Introdução

Neste artigo, iremos abordar e discorrer sobre um tópico importante da área de IA(Inteligência Artificial) que serão os agentes de resolução de problemas por meio de busca, em que vemos como um agente pode encontrar uma sequência de ações que alcança seus objetivos quando nenhuma ação isolada é capaz de fazê-lo (RUSSEL; NORVIG, 2010).
No decorrer do artigo entenderemos o que são esses agentes de resolução de problemas, os ambientes em que eles atuam, os problemas que são capazes de resolver, tipos de algoritmos de busca para resolução de problemas em ambientes simples ou complexos, entre outros temas.
A seguir, iremos começar os nossos estudos entendendo o que são agentes de soluções de problemas, que será o tema base para o nosso artigo.

Agente de resolução de problemas

Agentes de resolução de problemas são agentes baseados em objetivos que se diferenciam de agentes mais simples pelo fator de considerarem ações futuras e o quanto seus resultados são desejáveis. Outra característica importante dos agentes de resolução de problemas é que eles se utilizam de representações atômicas, onde os estados do mundo são considerados como um todo sem estrutura interna visível para os algoritmos (RUSSEL; NORVIG, 2010).

É importante entender que agentes inteligentes maximizam suas medidas de desempenho ao tomar decisões baseadas na meta de atingir seu objetivo, com isso esses agentes simplificam seus problemas de decisão ao definir um objetivo específico, tornando a resolução de problemas mais fácil de se gerenciar.

Um ponto que devemos entender é a definição de objetivos e a formulação de problemas realizadas pelos agentes, pois são a chave para a resolução de problemas. Os objetivos são definidos como um conjunto de estados desejados no mundo, onde esses estados são os necessários para que o objetivo seja satisfeito, com isso o agente deve se verificar esses estados e entender como ele irá chegar no estado do objetivo, analisando as ações necessárias e os estados intermediários que levam ao estado objetivo. Com esses dados em análise o agente pode então realizar a formulação do problema que vai fornecer uma visão geral para o agente sobre quais ações, decisões e estados são os corretos e mais eficientes para alcançar o objetivo.

Um agente pode ou não ter informações disponíveis para tomar suas decisões, com isso, se pensarmos em um agente que tem o objetivo de ir de um **ponto A** a um **ponto B**, e possui diversas rotas disponíveis para cumprir seu objetivo, se não possuir informações sobre as rotas, o método que será adotado pelo agente é tentar caminhos aleatórios até cumprir o seu objetivo, mas, se o agente possuir informações sobre as rotas como distância, semáforos, desvios e etc, ele pode otimizar seu modo de agir, prevendo ações futuras e planejando uma sequência de ações otimizadas para chegar ao objetivo.

Para que seja possível o agente poder analisar ações futuras, o ambiente em que o agente está inserido deve possuir as seguintes propriedades:

- Observável: Onde o agente sempre conhece o estado atual;
- Discreto: Onde existe um número finito de ações disponíveis em cada estado em que o agente se encontra;
- Conhecido: O agente sabe o resultado das ações que ele tomar;
- Determinístico: Cada ação que o agente tomar leva a um único resultado;

Com essas características de ambientes bem definidas, é possível afirmar que a solução para um problema é uma sequência fixa de ações (RUSSEL; NORVIG, 2010). Vale ressaltar que essa premissa é relacionada a ambientes previsíveis, em ambiente menos previsíveis, pode haver uma complexidade maior para chegar ao seu objetivo. Um agente que realiza seu planjamento de forma a ignorar o restante do ambiente, deve estar completamente certo do que está acontecendo, isso se chama de um sistema de **malha aberta**, pois ignorar a percepção quebra o laço entre o agente e o ambiente.

Problemas de malha aberta e de malha fechada

Neste tópicos iremos abordar problemas de malha aberta e fechada, a fim de entender as diferenças das ações de um agente nesses modelos.

Antes de analisarmos os problemas, é interessante entendermos a diferença entre **malha aberta** e **malha fechada**.

- **Malha Aberta**: Quando o agente executa o plano previamente definido com base nas informações do ambiente, e executa o plano cegamente ignorando as novas percepções do ambiente, ou seja, ele não se adapta a novas alterações do ambiente;
- **Malha Fechada**: Quando o agente executa o planjedo, mas continua a observar o ambiente e se adaptar a ele, fazendo os ajustes em suas ações futuras.

Com esses dois modelos definidos, podemos agora obervar um exemplo de problema com dois modelos, onde no **exemplo 1.1** iremos analisar a resolução do problema com malha aberta, e no **exemplo 1.2** a resolução do problema com malha fechada. Para basear os exemplos, o contexto será o seguinte: > Um agente pertencente a uma esteira de linha de montagem, deve realizar a montagem de um equipamento, para o agente é fornecido um manual de intruções sobre a montagem de cada peça, e qual peça depende de outra.

Exemplo 1.1 - Malha Aberta

Se o agente executar suas ações em um modelo de malha aberta, ele irá analisar o documento fornecido com as intruções de montagem, as dependências de peças, e qual a melhor ordem de montagem, com isso, ele irá traçar um plano para executar em ordem um conjunto de ações e chegar ao seu objetivo.

Se não houver problemáticas na montagem do equipamento, tudo ocorrerá muito bem, e o agente irá conseguir terminar a montagem da maneira que planejou. Mas, se alguma peça do equipamento quebrar durante o processo e não servir para utilização, o agente não conseguirá terminar sua montagem, pois ele não tem a capacidade de mudar seu planejamento devido a esse incidente.

A seguir podemos ver a aplicação do algoritmo do agente:

```
ALGORITMO Montagem_Malha_Aberta
    ENTRADA: instrucoes, pecas_disponiveis
    PARA cada etapa EM instrucoes:
        peca <- etapa.peca
        dependencias <- etapa.dependencias

        SE dependencias NÃO ESTÃO em pecas_disponiveis:
            EXIBIR "Erro: Dependências para a peça {peca} não estão disponíveis."
            ENCERRAR

        SE peca NÃO ESTÁ em pecas_disponiveis:
            EXIBIR "Erro: A peça {peca} está quebrada. Não é possível continuar."
            ENCERRAR

        REMOVER peca DE pecas_disponiveis
        EXIBIR "Montando a peça {peca}."

    EXIBIR "Montagem concluída com sucesso."
FIM_ALGORITMO
```

Código 1: Algoritmo do agente executando em malha aberta. Fonte: OPENAI. Assistente Virtual ChatGPT, 2024

Exemplo 1.2 - Malha Fechada

Se o agente executar suas ações baseado no modelo de malha fechada, ele irá analisar o documento fornecido com as instruções de montagem, mas depencências das peças, e qual a melhor ordem de montagem para chegar ao objetivo da forma mais otimizada.

Se não houver problemáticas o agente conseguirá chegar ao seu objetivo. Mas supondo que ocorra o mesmo problema do exemplo 1.1, onde durante a montagem uma das peças acabou se quebrando, o agente irá realizar uma nova análise das formas de contornar esse problema, seja solicitando uma nova peça para a esteira de produção, ou ajustando seu plano para montar o equipamento sem e peça, ou deixando a montagem da peça quebrada para o final. Todas essas formas podem ser utilizadas pelo agente para continuar as ações para alcançar o objetivo, pois ele analisou os estados em que se encontrava e ajustou suas ações para isso.

A seguir temos a aplicação do algoritmo do agente:

```
ALGORITMO Montagem_Malha_Fechada
    ENTRADA: instrucoes, pecas_disponiveis
    PARA cada etapa EM instrucoes:
        peca <- etapa.peca
        dependencias <- etapa.dependencias

        ENQUANTO dependencias NÃO ESTÃO em pecas_disponiveis:
            EXIBIR "Aguardando dependências para a peça {peca}."
            ADICIONAR dependencias PARA pecas_disponiveis

        ENQUANTO peca NÃO ESTÁ em pecas_disponiveis:
            EXIBIR "A peça {peca} está quebrada. Solicitando nova peça."
            ADICIONAR peca PARA pecas_disponiveis

        REMOVER peca DE pecas_disponiveis
        EXIBIR "Montando a peça {peca}."

    EXIBIR "Montagem concluída com sucesso."
FIM_ALGORITMO
```

Código 2: Algoritmo do agente executando em malha fechada. Fonte: OPENAI. Assistente Virtual ChatGPT, 2024

Algoritmos de busca

Como foi visto anteriormente, agentes inteligentes fazem um planejamento para chegar ao seu objetivo, este planejamento é feito baseado no entendimento do ambiente e através da previsão se ações e estados futuros em que o agente pode estar, com isso, se nos aprofundarmos no planejamento das ações do agente, será possível entender que é composto por uma série de buscas que analisam várias sequências de ações possíveis.

Essas séries de buscas realizadas pelo agente que analisam diversas sequências de ações para identificar a melhor que chegará ao objetivo, acaba gerando uma árvore de busca, onde tal árvore possui na raiz o estado inicial do agente, em suas ramificações existem as possíveis ações, e nos nós os estados possíveis que o agente pode estar. Os agentes percorrem essas árvores de busca diversas vezes para analisar e compreender qual a melhor "rota" possível para chegar ao objetivo.

No decorrer desta seção iremos abordar dois importante tópicos de busca, a **busca cega** e a **busca informada**, que trazem aspectos e modelos diferentes de atuação de um agente na resolução de um problema.

Busca Cega

A busca cega ou busca sem informação, é um modelo de busca realizado pelo agente, onde ele não tem nenhuma informação adicional sobre os estados do

ambiente, além das informações que recebeu na definição do problema, ou seja, ele não tem uma "visão" privilegiada sobre os estados mais promissores a serem explorados, portanto esses agentes funcionam apenas com a capacidade de gerar sucessores e distinguir se chegaram ao objetivo ou não.

Dentro deste modelo de busca cega, temos alguns submodelos que possuem características e funcionamentos diferentes, a seguir vamos entender um pouco mais sobre eles:

- **Busca em Largura:** Este modelo de busca expande todos os nós da árvore de busca, independente se durante o processo já encontrou o estado objetivo, ele segue um padrão de abrir todos os nós de um nível, antes de expandir os nós de outro nível. Isso garante que o agente sempre encontre o melhor caminho para se chegar ao objetivo
- **Busca de Custo Uniforme:** Este modelo é parecido com o de busca em largura, mas com uma pequena diferença, ele não expande todos os nós de um nível, ele expande os nós com o menos custo de caminho, este modelo armazena através de uma fila de prioridade os custos e prioriza os de menor valor.
- **Busca em Profundidade:** Este modelo realiza um tipo de busca diferente dos citados anteriormente, ele realiza uma busca através de uma das ramificações até chegar no objetivo ou ao final da ramificação, caso ele encontre o final, ele volta um nível, e abre as ramificações. De forma mais simples, podemos pensar que ele vai sempre seguindo uma direção da árvore, por exemplo, em todo nó, ele sempre continua seguindo à esquerda, até que chegue ao final ou ao objetivo, caso chegue ao final, ele volta um nível, e começa a pegar a direita, e assim por diante até percorrer toda a árvore, ou até encontrar o objetivo. Uma diferença deste modelo para os anteriores, é que ele não garante o melhor caminho até o objetivo.
- **Busca de Bidirecional:** Este modelo aplica uma lógica de encontro de buscas, onde a sua busca é iniciada em dois lugares, uma na raiz da árvore e uma no objetivo, e essas buscas devem se encontrar em um caminho intermediário.
- **Busca de Profundidade Limitada:** Este modelo é o mesmo do de busca em profundidade, a diferença entre eles é que este modelo tem um limite de nível, onde caso ele chegue a esse limite, ele volta um nível e contiua sua busca. De forma simples, é como se fosse definido onde é o "final" da árvore.

Agora que entendemos alguns dos modelos existentes de algoritmos de busca cega, podemos agora pensar em um exemplo onde a aplicação deste modelo pode ser eficiente.

Exemplo: Imagine que temos um labirinto para percorrer, e neste labirinto há somente uma saída, para isso temos um agente que irá percorrer o labirinto em busca da saída. Podemos pensar de três modos com este problema, o primeiro onde o agente deve encontrar o melhor caminho para chegar a saída, e o outro onde o agente deve encontrar a saída o mais rápido possível, nestes dois exemplos,

podemos citar na mesma ordem três algoritmos para chegar a esses objetivos, o primeiro seria o de busca em largura para o primeiro objetivo de melhor caminho, onde o agente iria percorrer todas as rotas possíveis do labirinto para descobrir qual a melhor rota, e o segundo seria a de busca em profundidade e a busca bidirecional, onde o agente assim que encontrasse a saída concluiria seu objetivo.

A seguir no *código 3* podemos ver a implementação do algoritmo de busca bidirecional, aplicado ao exemplo desenvolvido anteriormente.

```
from collections import deque

def bidirectional_search(labyrinth, start, goal):
    def get_neighbors(position):
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        neighbors = []
        for dx, dy in directions:
            nx, ny = position[0] + dx, position[1] + dy
            if 0 <= nx < len(labyrinth) and 0 <= ny < len(labyrinth[0]) and labyrinth[nx][ny
                neighbors.append((nx, ny))
        return neighbors

    # Frontiers and visited sets for both searches
    frontier_start = deque([start])
    frontier_goal = deque([goal])
    visited_start = {start: None}  # Keeps track of visited nodes and predecessors
    visited_goal = {goal: None}

    while frontier_start and frontier_goal:
        # Expand the frontier from the start side
        current_start = frontier_start.popleft()
        for neighbor in get_neighbors(current_start):
            if neighbor not in visited_start:
                visited_start[neighbor] = current_start
                frontier_start.append(neighbor)

                if neighbor in visited_goal:  # Intersection found
                    return reconstruct_path(neighbor, visited_start, visited_goal)

        # Expand the frontier from the goal side
        current_goal = frontier_goal.popleft()
        for neighbor in get_neighbors(current_goal):
            if neighbor not in visited_goal:
                visited_goal[neighbor] = current_goal
                frontier_goal.append(neighbor)

                if neighbor in visited_start:  # Intersection found
```

```
                    return reconstruct_path(neighbor, visited_start, visited_goal)

    return None  # No path found

def reconstruct_path(meeting_point, visited_start, visited_goal):
    # Reconstruct path from start to meeting point
    path_start = []
    current = meeting_point
    while current is not None:
        path_start.append(current)
        current = visited_start[current]

    # Reconstruct path from goal to meeting point
    path_goal = []
    current = visited_goal[meeting_point]
    while current is not None:
        path_goal.append(current)
        current = visited_goal[current]

    return path_start[::-1] + path_goal  # Combine both parts

# Example labyrinth (0 = free space, 1 = wall)
labyrinth = [
    [0, 0, 1, 0, 0],
    [1, 0, 1, 0, 1],
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0]
]

start = (0, 0)  # Starting point
goal = (4, 4)   # Goal point

path = bidirectional_search(labyrinth, start, goal)
if path:
    print("Path found:", path)
else:
    print("No path found.")
```

Código 3: Implementação do algoritmo de busca bidirecional aplicada ao exemplo anterior, Fonte: OPENAI. Assistente Virtual ChatGPT, 2024

Busca Informada

A busca informada de acordo com RUSSEL e NORVIG, 2010, é a que utiliza conhecimento de um problema específico além da definição do problema em si —

pode encontrar soluções de forma mais eficiente do que uma estratégia de busca sem informação.

Esse modelo de busca se dá quando o agente tem informações a mais sobre o ambiente e os estados que ele pode possuir ou possui, os modelos pertencente a busca informada se utilizam de funções heurísticas, funções para medição do quão "longe" está um nó da solução, para avaliar os caminhos com o menos custo baseado nas informações do problema. De forma simplificada podemos pensar na ideia de ir de uma cidade a outra com um mapa, onde o agente teria as informações sobre os caminhos que ele pode percorrer como distância, desvios, problemas e etc, e baseado nessas informações ele poderia calcular a rota com o menor custo para ele.

A seguir iremos explorar alguns modelos de busca informada para entendermos melhor e nos aprofundarmos no assunto:

- **Busca Gulosa de Melhor Escolha:** Esse modelo se utiliza da função heurística para escolher o próximo nó a ser expandido na árvore de busca, se baseando nos nós futuros que parecem estar mais próximo do objetivo, não contabilizando o caminho já percorrido;
- **Busca A*:** Realiza a mesma função da busca gulosa, mas com um diferencial que é a contabilização do caminho percorrido até o nó atual, fazendo com que a função heurística possua esse parâmetro para o cálculo do próximo nó a ser expandido;
- **Busca Iterativa de Aprofundamento com A* (IDA*):** Esse modelo é uma variação do A*, onde é aplicado a ideia de limite de profundidade, onde o modelo realiza a busca até esse limite imposto, e se não encontrar o objetivo, ele modifica o limite para o menor valor da função heurística já encontrado que excedeu o limite anterior, e repete sua busca;
- **Busca Recursiva de Melhor Escolha (RBFS):** Baseado no modelo de A*, se utiliza do mesmo princípio, mas trabalha de modo recursivo em sua abordagem, fazendo com que toda vez que chegue em um limite de uma ramificação, ele retorna e recalcula o limite.
- **Busca Heurística Bidirecional:** Funciona de maneira similar a busca bidirecional de busca cega, onde são realizadas duas buscas simultaneamente, uma partindo da raiz e outra partindo do objetivo, e ambas as buscas utilizam heurísticas específicas, para que ambas as buscas se encontrem no meio do "caminho".

Agora que vimos alguns exemplos de modelos de busca informada, é interessante explorarmos um exemplo para aprofundar o entendimento:

Exemplo: Suponha que temos um jogo de tabuleiro, e esse jogo seja xadrez, jogos de tabuleiro como xadrez costumam ter a necessidade de bastante planejamento para as jogadas, avaliando as possíveis situações e resultados que as ações podem causar. Com essa ideia em mente, seria possível aplicarmos um algoritmo de busca informada para jogar xadrez, basta entendermos que em um jogo de xadrez o agente iria possuir diversas informações sobre o ambiente em que está

atuando, poderia planejar jogadas baseadas nas jogadas do seu oponente, nas posições das peças, entre outros diversos dados que iria possuir. Deste modo a aplicação de um algoritmo como o RBFS seria uma boa escolha para este caso, a seguir no *código 4* podemos ver uma versão simplificada da ideia do tabuleiro de xadrez.

```
import chess

def heuristic(board):
    """Avalia o tabuleiro com base no material de cada lado."""
    piece_values = {
        chess.PAWN: 1,
        chess.KNIGHT: 3,
        chess.BISHOP: 3,
        chess.ROOK: 5,
        chess.QUEEN: 9,
        chess.KING: 1000
    }
    score = 0
    for piece_type in piece_values:
        score += len(board.pieces(piece_type, chess.WHITE)) * piece_values[piece_type]
        score -= len(board.pieces(piece_type, chess.BLACK)) * piece_values[piece_type]
    return score

def rbfs(board, depth, alpha, beta, is_white):
    """
    Recursive Best-First Search (RBFS) para o xadrez.
    - board: o tabuleiro atual.
    - depth: a profundidade máxima de busca.
    - alpha, beta: os limites alfa-beta.
    - is_white: indica se é o turno do jogador branco.
    """
    if board.is_checkmate():
        return float('inf') if is_white else float('-inf'), None
    if board.is_stalemate() or board.is_insufficient_material() or depth == 0:
        return heuristic(board), None

    best_score = float('-inf') if is_white else float('inf')
    best_move = None

    for move in board.legal_moves:
        board.push(move)
        score, _ = rbfs(board, depth - 1, alpha, beta, not is_white)
        board.pop()

        if is_white:
```

```
            if score > best_score:
                best_score, best_move = score, move
            alpha = max(alpha, score)
        else:
            if score < best_score:
                best_score, best_move = score, move
            beta = min(beta, score)

        if beta <= alpha:
            break

    return best_score, best_move

# Configuração inicial do tabuleiro
board = chess.Board()

# Simulação de uma jogada usando RBFS
depth = 3  # Profundidade de busca limitada
is_white_turn = True
score, move = rbfs(board, depth, float('-inf'), float('inf'), is_white_turn)

print("Melhor movimento sugerido:", move)
if move:
    board.push(move)
    print(board)
```

Código 4: Implementação do algoritmo de busca RBFS aplicada ao exemplo anterior, Fonte: OPENAI. Assistente Virtual ChatGPT, 2024

Busca em ambientes complexos

Até o presente momento, vimos diversos algoritmos de buscas sistemáticos que se preocupavam em alcançar o seu objetivo descobrindo um caminho para chegar nele, mas agora, iremos ver um outro modelo de busca, que atuam em ambientes complexos, onde a preocupação não será o caminho para chegar, mas sim se os estados atuais estão de acordo com o objetivo.

Um dos modelos de busca que atua dentro de ambientes complexos, é o de busca local, esse algoritmo não se preocupa com estados anteriores, ou seja, não grava a rota percorrida, economizando memória, ele se foca somente no estado atual, analisando e se deslocando para os estados vizinhos em preocupação com os estados anteriores. Uma das grandes vantagens da busca local, além de economizar memória e atuarem muito bem em grandes ou infinitos ambientes, eles também ajudam a resolver problemas de otimização, nos quais o objetivo é encontrar o melhor estado de acordo com a função objetivo (RUSSEL; NORVIG, 2010).

Neste ambito de busca local, podemos observar alguns dos diversos modelos de buscas existes, a seguir iremos entender um pouco mais os algoritmos escolhidos:

- **Busca de Subida de Encosta:** Um dos modelos de busca local mais simples, onde durante o seu processo de busca do objetivo, ele tenta melhorar constantemente e progressivamente a solução atual, ele começa em um estado aleatório ou definido, e durante sua "jornada" sempre tenta escolher o melhor vizinho baseado no valor da função objetivo, onde dependendo do problema pode ser o maior valor ou o menor valor;
- **Têmpera Simulada:** É um modelo completo mas ineficiente, sua lógica é baseada em movimentos "aleatórios" onde ele fica em um processo de subida e descida nos valores da função objetivo, se o movimento realizado melhorar a situação do agente, esse movimento sempre será aceito, se não, o modelo irá aceitar o modelo após uma análise na probabilidade de melhora;
- **Busca em Feixe Local:** Este modelo é baseado na busca em largura, mas com um foco diferente, ele busca explorar um número de estados maior em cada nível, e aos invés de explorar todos os nós de um nível, ele seleciona apenas os melhores nós, e com isso limita a quantidade de estados.

Para entendermos melhor como esses algoritmos podem funcionar, vamos trazer um exemplo e a aplicação de um desses modelos para a solução do problema:

Exemplo: Imagine que um agente é responsável por fazer a organização de um armazém, ele tem diversos produtos e precisa organizá-los de uma forma otimizada para facilitar o processo de coleta e transporte desse produto, a aplicação de um algoritmo de Subida de Enconsta pode ajudar, onde durante suas análises ele tentará achar a melhor e mais eficiente solução.

A seguir no *código 5*, podemos ver a implementação deste exemplo em python.

```python
import random

# Função que calcula a distância total percorrida para pegar todos os produtos (simulação s
def calcular_distancia(arranjo):
    # Aqui a distância é simplesmente a soma das distâncias entre produtos consecutivos
    distancia = 0
    for i in range(len(arranjo) - 1):
        distancia += abs(arranjo[i] - arranjo[i+1])  # Distância entre posições consecutivas
    return distancia

# Função para gerar um estado vizinho (movendo dois produtos de lugar)
def gerar_vizinho(arranjo):
    novo_arranjo = arranjo[:]
    i, j = random.sample(range(len(arranjo)), 2)  # Seleciona dois índices aleatórios
    novo_arranjo[i], novo_arranjo[j] = novo_arranjo[j], novo_arranjo[i]  # Troca os produtos
    return novo_arranjo
```

```python
# Algoritmo de Subida de Encosta (Hill Climbing)
def subida_de_encosta(arranjo_inicial):
    estado_atual = arranjo_inicial
    custo_atual = calcular_distancia(estado_atual)

    while True:
        # Gerar um vizinho e calcular seu custo
        vizinho = gerar_vizinho(estado_atual)
        custo_vizinho = calcular_distancia(vizinho)

        # Se o vizinho for melhor (menor custo), mova para ele
        if custo_vizinho < custo_atual:
            estado_atual = vizinho
            custo_atual = custo_vizinho
        else:
            # Se não encontrar melhorias, termina o algoritmo
            break

    return estado_atual, custo_atual

# Exemplo de uso
# Inicializando o arranjo de produtos no armazém (ex: posições em um eixo linear)
arranjo_inicial = [4, 2, 8, 1, 6, 5, 7, 3]

# Rodando o algoritmo de subida de encosta
melhor_arranjo, custo_final = subida_de_encosta(arranjo_inicial)

# Exibindo o melhor arranjo e o custo final
print(f"Melhor arranjo encontrado: {melhor_arranjo}")
print(f"Custo final (distância total): {custo_final}")
```

Código 5: Implementação do algoritmo de busca de Subida de Enconsta aplicada ao exemplo anterior, Fonte: OPENAI. Assistente Virtual ChatGPT, 2024

Algoritmos genéticos

Os algoritmos genéticos são uma variação da busca em feixe estocástica na qual os estados sucessores são gerados pela combinação de dois estados pais, em vez de serem gerados pela modificação de um único estado (RUSSEL; NORVIG, 2010).

Esses algoritmos genéticos funcionam da seguinte forma, eles possuem um conjunto de estados, onde nesses estados eles avaliam os mais potenciais a resolver a solução, e fazem combinações para gerarem novos estados baseados nos anteriores (como se fosse a reprodução sexuada, por isso algortimo genético). O processo de avaliação é feito baseado em uma função de avaliação que analisa

cada estado, fornece um valor para os estados, e em seguida identifica o que aparenta ser a melhor escolha para realizar a combinação.

A seguir, iremos ver um exemplo para podermos absorver a ideia de algoritmos genéticos de forma mais tranquila.

Exemplo: Baseado no problema da mochila, pertencente ao "Os 21 problemas NP-completos, que foram introduzidos por Richard Karp em 1972 como forma de demonstrar a aplicabilidade do conceito de NP-completude" (WIKIPEDIA, 2024). Temos uma mochila com peso limitado, e uma série de itens que possuem peso e valor, com isso, temos de selecionar de maneira inteligente, os item de maior valor, sem exceder a capacidade de peso da mochila. O algoritmo irá analisar as combinações possíveis e retornar um resultado para o problema.

Podemos ver a aplicação deste exemplo no *código 6*.

```
def knapsack(values, weights, capacity):
    """
    Resolve o problema da mochila utilizando programação dinâmica.

    :param values: Lista dos valores dos itens.
    :param weights: Lista dos pesos dos itens.
    :param capacity: Capacidade máxima da mochila.
    :return: Valor máximo que pode ser colocado na mochila e os itens escolhidos.
    """
    n = len(values)
    # Matriz para armazenar os valores máximos para cada capacidade e número de itens.
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    # Preenchendo a matriz dp
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                # Inclui o item i-1 ou não
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i - 1]])
            else:
                dp[i][w] = dp[i - 1][w]

    # Determinando os itens escolhidos
    chosen_items = []
    w = capacity
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:  # Se o valor mudou, significa que o item foi incluído
            chosen_items.append(i - 1)
            w -= weights[i - 1]

    return dp[n][capacity], chosen_items
```

```
# Exemplo de uso
values = [60, 100, 120]  # Valores dos itens
weights = [10, 20, 30]   # Pesos dos itens
capacity = 50            # Capacidade máxima da mochila

max_value, items = knapsack(values, weights, capacity)

print("Valor máximo:", max_value)
print("Itens escolhidos:", items)
```

Código 6: Implementação do algoritmo de busca genética aplicada ao exemplo anterior, Fonte: OPENAI. Assistente Virtual ChatGPT, 2024

Conclusão

Baseado em tudo que estudamos neste artigo, foi possível entender diversos tópicos importantes para o entendimento sobre a solução de problemas por buscas, conseguimos obter um entendimento sobre como agentes atuam através de diferentes modelos de busca, desde buscas sistemáticas, até buscas em ambientes complexos, com exemplos claros de sua utilização e aplicação em linguagem python. Por fim aprofundamos nosso assunto de busca em um conceito importante que é a busca por algoritmo genético que tem suas aplicações em diversos setores importantes.

Referências

[1] RUSSELL, Stuart; NORVIG, Peter. *Inteligência Artificial: Uma Abordagem Moderna* – 3ª edição.

[2] OPENAI. Assistente Virtual ChatGPT. Respostas geradas com base em inteligência artificial. Disponível em: https://openai.com. Acesso em: 24 dez. 2024.

[3] WIKIPEDIA. 21 problemas NP-completos de Karp. Disponível em: https://pt.wikipedia.org/wiki/21_problemas_NP-completos_de_Karp. Acesso em: 29 dez. 2024.