UNIVERSIDADE FEDERAL DO CEARÁ - UFC
CAMPUS QUIXADÁ

Projeto 1

Curso: Ciência da Computação
Disciplina: Inteligência Artificial

Kaynan Pereira de Sousa - 540864
Victor Emanuel de Sousa Costa - 535718

Quixadá, 19 de janeiro de 2025

Conteúdo

1 Introdução

Este relatório apresenta o desenvolvimento de um projeto prático na disciplina de Inteligência Artificial, focado na implementação e análise de diferentes algoritmos de busca. O objetivo principal deste trabalho foi explorar as capacidades de diversos algoritmos para encontrar soluções ótimas em um ambiente simulado, variando diferentes parâmetros de entrada e analisando o comportamento de cada um. Foram implementados algoritmos clássicos como busca em largura, busca em profundidade, busca de custo uniforme, busca gulosa e busca A*, e cada um foi utilizado em diferentes experimentos, a fim de se entender a sua complexidade e qual deles melhor se adaptava a cada situação.

2 Implementação

2.1 Implementação

- **Linguagem de Programação:** O projeto foi desenvolvido utilizando a linguagem Python 3.10.
- **Bibliotecas e Frameworks:**
  - *heapq*: Usada para implementar filas de prioridade nos algoritmos de busca de custo uniforme, gulosa e A*.
  - *math*: Usada para cálculos matemáticos nas funções heurísticas (distância euclidiana).
  - *collections.deque*: Usada para implementar a fila na busca em largura.
  - *random*: Usada para gerar números aleatórios nos experimentos.
  - *time*: Usada para medir o tempo de execução em alguns experimentos.
  - *pandas*: Usada para criar e manipular DataFrames para armazenar e salvar os resultados dos experimentos em arquivos CSV.
- **Estrutura do Projeto:**
  - Arquivo `main.py`: Responsável por gerenciar a execução dos experimentos, definindo as funções de custo e de gerar vizinhos, e chamando os algoritmos de busca com os parâmetros necessários. Além disso, ele utiliza a biblioteca *pandas* para salvar os resultados em arquivos CSV.

- Arquivos individuais para cada algoritmo de busca:
  - `busca_em_largura.py`: O algoritmo de Busca em Largura (*BFS*) é utilizado para encontrar o menor caminho entre dois nós em um grafo, considerando a distância em termos de número de arestas ou custo. Ele explora todos os nós em um nível de profundidade antes de avançar para o próximo nível, garantindo que o caminho mais curto seja encontrado, caso exista. Na implementação fornecida, o algoritmo começa com o nó inicial e vai explorando os nós vizinhos em uma ordem de "largura", ou seja, todos os nós no mesmo nível de profundidade são visitados antes de avançar para os nós do próximo nível.

    O algoritmo utiliza uma fila (*deque*) para armazenar os nós a serem explorados, e segue a estratégia de primeiro entrar, primeiro a sair (*FIFO*). A fila é inicializada com o nó inicial, e à medida que os nós são visitados, seus vizinhos são adicionados à fila. Para evitar que um nó seja explorado mais de uma vez, a implementação utiliza um conjunto (*visitados*) para armazenar os nós já visitados. Além disso, é mantido um dicionário (*pais*) que armazena o nó anterior para cada nó visitado, o que permite reconstruir o caminho final do estado inicial até o estado objetivo. Outro dicionário, o profundidade, armazena a profundidade de cada nó, o que pode ser utilizado para calcular o custo entre os nós, caso haja algum critério de custo envolvido.

    O processo de busca é repetido enquanto houver nós na fila. Para cada nó retirado da fila, o algoritmo verifica se ele é o estado objetivo. Se for, o caminho é reconstruído utilizando o dicionário pais, e o custo do caminho é calculado com a função `custo_fun`, que leva em conta a profundidade dos nós.

    Se o estado objetivo for encontrado, o algoritmo retorna o caminho, o custo e algumas estatísticas, como o número de nós gerados e visitados. Se o objetivo não for encontrado após explorar todos os nós possíveis, o algoritmo retorna que não há caminho e que o custo é infinito.

    Enquanto o algoritmo percorre o grafo, ele expande os vizinhos de cada nó, adicionando-os à fila e marcando-os como visitados. O número de nós gerados é contado a cada vez que um novo nó é inserido na fila, enquanto o número de nós visitados é incrementado a cada vez que um nó é retirado da

fila para ser processado. Esses contadores ajudam a medir a eficiência da busca. Ao final, o algoritmo retorna não só o caminho e o custo do estado objetivo, mas também as métricas de desempenho, como o número de nós gerados e visitados, o que pode ser útil para avaliar a eficiência da busca em diferentes cenários.
Esse algoritmo tem a vantagem de ser simples de entender e implementar, além de garantir que o caminho mais curto será encontrado, caso haja um. No entanto, ele pode ser ineficiente em casos onde o grafo é muito grande ou denso, pois requer muita memória para armazenar todos os nós explorados. Apesar disso, a busca em largura é uma boa escolha quando a profundidade do grafo não é muito grande e o objetivo é garantir a solução ótima sem a necessidade de heurísticas complexas.

* `busca_em_profundidade.py`: O algoritmo de Busca em Profundidade começa com a pilha inicial contendo apenas o estado de partida. A pilha é uma estrutura fundamental para o DFS, pois a busca se baseia em explorar os caminhos de forma LIFO (Last In, First Out). Cada elemento da pilha é uma tupla composta pelo estado atual, seu pai e sua profundidade no caminho. A profundidade é importante para o cálculo do custo, caso o algoritmo precise computá-lo.
A busca inicia retirando o último estado da pilha, que é o mais recente adicionado, e verifica se ele é o estado objetivo. Se o estado atual for o objetivo, o algoritmo começa a reconstruir o caminho percorrido desde o estado inicial até o estado objetivo. Para isso, ele segue o pai de cada estado (armazenado no dicionário `pilha_busca_profundidade`) até o nó inicial, acumulando o custo utilizando a função `custo_fun`, que calcula o custo entre dois estados com base na profundidade.
Se o estado atual não for o objetivo, o algoritmo verifica se ele já foi visitado. Caso não tenha sido visitado, ele é marcado como visitado e seus vizinhos são gerados pela função `gerar_vizinhos`. Para cada vizinho não visitado, o algoritmo adiciona o vizinho à pilha junto com o estado atual como seu pai e a profundidade aumentada em 1. Ao mesmo tempo, o número de nós gerados (`nos_gerados`) é incrementado.
A pilha de estados cresce conforme o DFS avança e diminui à medida que o algoritmo retrocede para explorar novos cami-

nhos.
A profundidade de cada estado é armazenada no dicionário `profundidade_busca_profundidade`, e o estado pai de cada nó é armazenado no dicionário `pilha_busca_profundidade`. Esses dicionários são fundamentais para a reconstrução do caminho e para o cálculo do custo do caminho, se necessário. Caso o algoritmo encontre o estado objetivo, ele retorna o caminho completo desde o estado inicial até o objetivo, o custo total do caminho e as estatísticas sobre o número de nós gerados e visitados. Se o estado objetivo não for encontrado após explorar todos os caminhos possíveis, o algoritmo retorna que não existe caminho e o custo é infinito.
Uma característica importante do DFS é que ele pode ser menos eficiente em termos de memória em grafos muito profundos ou em grafos com muitos caminhos, já que pode acabar gerando um grande número de estados na pilha. No entanto, em problemas com árvores ou grafos com menor profundidade, a busca em profundidade pode ser uma solução eficiente, principalmente quando o objetivo é encontrar uma solução rapidamente sem precisar explorar o grafo inteiro.

* `custo_uniforme.py`: O algoritmo de Busca de Custo Uniforme (UCS) é utilizado para encontrar o caminho de menor custo entre um estado inicial e um estado objetivo em um grafo. A implementação apresentada utiliza uma fila de prioridade para garantir que os estados sejam explorados em ordem crescente de custo acumulado, uma característica que torna o algoritmo ótimo e completo em grafos com custos não negativos.
A função `busca_custo_uniforme` recebe como entrada o estado inicial, o estado objetivo, uma função de custo que calcula o valor da transição entre dois estados, e uma função que gera os vizinhos de um estado dado. A fila de prioridade, implementada com a biblioteca *heapq*, armazena as informações necessárias para a exploração de cada estado: o custo acumulado, o estado atual, o estado pai (de onde ele foi alcançado) e a profundidade no grafo.
Ao longo da execução, os estados já processados são armazenados em um conjunto chamado visitados, evitando que o mesmo estado seja explorado mais de uma vez. Além disso,

dois dicionários auxiliares são utilizados: o primeiro, `pilha_busca_custo_uniforme`, associa cada estado ao seu estado pai, possibilitando a reconstrução do caminho ao final; o segundo, `profundidade_busca_custo_uniforme`, registra a profundidade de cada estado no grafo.
O algoritmo processa a fila de prioridade até que ela esteja vazia ou o estado objetivo seja encontrado. Em cada iteração, o estado com menor custo acumulado é removido da fila. Se esse estado corresponde ao objetivo, o algoritmo reconstrói o caminho percorrido, partindo do estado objetivo até o inicial, utilizando as informações registradas nos dicionários auxiliares. Durante esse processo, o custo total é calculado somando os custos de transição entre os estados do caminho. Por fim, o algoritmo retorna o caminho encontrado, o custo total e estatísticas como o número de nós gerados e visitados.
Se o estado atual não for o objetivo e ainda não tiver sido processado, ele é marcado como visitado, e seus vizinhos são gerados pela função fornecida. Para cada vizinho que ainda não foi visitado, o algoritmo calcula o custo acumulado para alcançá-lo e o insere na fila de prioridade. Além disso, atualiza os dicionários auxiliares para registrar o estado pai e a profundidade do vizinho.
Caso a fila de prioridade seja esvaziada sem que o estado objetivo seja encontrado, o algoritmo retorna uma estrutura indicando que o caminho não foi identificado, com o custo total sendo considerado infinito, acompanhado das estatísticas sobre nós gerados e visitados.
Essa implementação assegura que o algoritmo sempre priorize a exploração do caminho de menor custo acumulado, atendendo aos critérios de completude e optimalidade para problemas com custos de arestas não negativos.

* `busca_gulosa.py`: A busca gulosa é um algoritmo de busca informada que utiliza uma função heurística para direcionar a exploração do espaço de estados. A implementação apresentada prioriza a exploração dos estados que apresentam o menor custo heurístico estimado para alcançar o estado objetivo. Essa abordagem é eficiente em muitos casos, mas não garante a otimalidade ou completude, pois a busca pode ser enganada por heurísticas inadequadas.

A função `busca_gulosa` recebe como parâmetros o estado inicial, o estado objetivo, uma função heurística que estima o custo para alcançar o objetivo, uma função de custo que calcula o valor da transição entre dois estados, e uma função para gerar os vizinhos de um estado dado. A fila de prioridade utilizada é implementada com a biblioteca *heapq*, permitindo que os estados sejam processados em ordem crescente de custo heurístico.
A fila de prioridade armazena tuplas contendo o custo heurístico estimado, o estado atual, o estado pai (de onde foi alcançado), e a profundidade no grafo. O conjunto *visitados* mantém o registro dos estados que já foram processados, enquanto dois dicionários auxiliares são utilizados:
o dicionário `pilha_busca_gulosa`, que associa cada estado ao seu estado pai para reconstruir o caminho ao final, e o dicionário `profundidade_busca_gulosa`, que registra a profundidade de cada estado.
O algoritmo processa a fila de prioridade enquanto ela não estiver vazia. Em cada iteração, remove-se o estado com o menor custo heurístico acumulado. Caso o estado atual seja o estado objetivo, o algoritmo reconstrói o caminho percorrido utilizando o dicionário de estados pais, calcula o custo total somando os custos das transições entre os estados do caminho, e retorna essas informações juntamente com estatísticas sobre os nós gerados e visitados.
Se o estado atual não for o objetivo e ainda não tiver sido processado, ele é marcado como visitado, e seus vizinhos são gerados pela função fornecida. Para cada vizinho não visitado, o algoritmo calcula o custo heurístico estimado e o insere na fila de prioridade. Simultaneamente, registra o estado pai e a profundidade do vizinho nos dicionários auxiliares.
Caso a fila de prioridade se esgote sem encontrar o estado objetivo, a função retorna uma estrutura indicando que o caminho não foi encontrado, com o custo considerado infinito, juntamente com o número de nós gerados e visitados.
Essa implementação explora o espaço de estados de forma eficiente ao priorizar os caminhos que parecem promissores com base na heurística. Contudo, sua eficácia depende diretamente da qualidade da heurística utilizada. Heurísticas não admissíveis ou inconsistentes podem levar a caminhos subótimos ou falhas na busca.

* `a_estrela.py`: Este código implementa o algoritmo A*, que busca o menor caminho entre um estado inicial e um estado objetivo em um espaço de estados, utilizando uma combinação de custos acumulados e heurísticas. Ele começa inicializando uma fila de prioridade, que organiza os nós a serem explorados com base na função de avaliação $f(n) = g(n) + h(n)$. Aqui, $g(n)$ é o custo acumulado para chegar até o nó atual, e $h(n)$ é a estimativa heurística de custo até o objetivo. A fila é iniciada com o estado inicial e sua heurística como prioridade.
  O programa também utiliza um conjunto de nós visitados para evitar reexplorar estados, bem como dicionários para rastrear os pais dos estados, seus custos acumulados, e suas profundidades na árvore de busca. Essas estruturas garantem que o algoritmo consiga reconstruir o caminho encontrado e calcular estatísticas, como o número de nós gerados e visitados.
  Durante a execução, o A* retira o nó com menor $f(n)$ da fila, marca-o como visitado e verifica se ele é o objetivo. Se for, o caminho é reconstruído recursivamente a partir do dicionário de pais, começando pelo estado objetivo e voltando até o estado inicial. Caso contrário, o algoritmo expande os vizinhos do nó atual, utilizando a função `gerar_vizinhos`, que retorna posições válidas (no exemplo, um grid 2D limitado entre 0 e 30 em cada eixo).
  Para cada vizinho, o custo acumulado $g(n)$ é calculado somando o custo do caminho até o estado atual com o custo da transição para o vizinho. Além disso, a heurística é calculada, podendo ser a distância Euclidiana ou Manhattan, dependendo da configuração. O algoritmo utiliza essa soma para calcular $f(n)$ e decide se deve adicionar ou atualizar o vizinho na fila de prioridade. Isso ocorre se o vizinho não foi visitado ou se um caminho de menor custo até ele foi encontrado.
  A função de custo inclui um parâmetro adicional para considerar a profundidade do nó na árvore de busca. Isso pode ser útil em cenários onde o custo de transição depende de fatores como o número de passos dados. Ao final do loop, se a fila esvaziar sem encontrar o objetivo, o algoritmo retorna que nenhum caminho foi encontrado, junto com as estatísticas acumuladas.
  Essa implementação é flexível e genérica, permitindo ajustes para diferentes tipos de grafos, funções de custo e heurísticas,

além de destacar detalhes importantes, como a contabilização de nós gerados e visitados, que é útil para analisar o desempenho do algoritmo.

- Experimentos realizados:
  - `executar_experimentos_parte1`:
    A função `executar_experimentos_parte1` tem como objetivo comparar o desempenho de três algoritmos de busca (Busca em Largura, Busca em Profundidade e Busca de Custo Uniforme) em um espaço de estados, utilizando diferentes funções de custo. O experimento é repetido múltiplas vezes, permitindo a análise da variabilidade dos resultados. Para cada repetição, os algoritmos são executados com funções de custo predefinidas, e os resultados de cada busca são salvos em um arquivo para análise posterior. A execução de cada algoritmo é acompanhada pelo tempo total de processamento, possibilitando uma comparação entre o tempo de execução e a eficiência dos algoritmos em diferentes cenários. Dessa forma, a função permite avaliar o comportamento dos algoritmos de busca em relação a diferentes funções de custo em um ambiente controlado.

  - `executar_experimentos_parte2`:
    A função `executar_experimentos_parte2` expande os experimentos realizados na função anterior, incorporando agora também a busca A* em conjunto com diferentes funções de custo e heurísticas. O objetivo principal é comparar o desempenho da Busca de Custo Uniforme e da Busca A* utilizando várias combinações de funções de custo (c1, c2, c3, c4) e heurísticas (euclidiana e manhattan) em um espaço de estados. Para cada repetição do experimento, a função executa a Busca de Custo Uniforme com diferentes funções de custo, e para cada combinação de função de custo e heurística, a Busca A* é executada. Os resultados de cada execução são salvos para análise posterior, e o tempo total de execução de todas as buscas é monitorado, permitindo a comparação entre os algoritmos em termos de eficiência e desempenho. Essa estrutura permite avaliar a eficácia da Busca A* com diferentes heurísticas em relação à Busca de Custo Uniforme.

  - `executar_experimentos_parte3`:

A função `executar_experimentos_parte3` continua a extensão dos experimentos anteriores, agora incorporando a busca gulosa, além da busca A* e da busca de custo uniforme. O principal objetivo dessa função é avaliar o desempenho da Busca Gulosa e da Busca A* para diferentes combinações de funções de custo e heurísticas (euclidiana e manhattan). Para cada repetição, a função executa a Busca Gulosa utilizando diferentes heurísticas, e, em seguida, para cada função de custo, o custo total do caminho é recalculado. Após isso, a Busca A* é executada para cada combinação de função de custo e heurística. Os resultados são salvos para cada tipo de busca, e o tempo total de execução é monitorado para comparar o desempenho dos algoritmos. Esta abordagem permite uma análise mais completa dos algoritmos de busca, incluindo a Busca Gulosa como uma alternativa à Busca A* e à Busca de Custo Uniforme.

* `executar_experimentos_parte4:`
  A função `executar_experimentos_parte4` expande os experimentos anteriores, agora incorporando uma versão modificada das buscas em largura e profundidade, utilizando vizinhos gerados de forma aleatória. O experimento avalia o impacto dessa aleatoriedade no desempenho dos algoritmos de busca, comparando-os para diferentes funções de custo. O estado inicial e o objetivo são fixados, e para cada repetição, a busca em largura e a busca em profundidade são executadas com a geração de vizinhos aleatórios, utilizando cada uma das funções de custo disponíveis. Os resultados são salvos, e o tempo total de execução é monitorado para cada tipo de busca. Esse experimento permite uma análise sobre o comportamento dos algoritmos de busca quando aplicados a um ambiente mais imprevisível, onde a geração dos vizinhos não segue um padrão determinístico.

* `executar_experimentos_parte5:`
  A função `executar_experimentos_parte5` realiza uma série de experimentos onde o objetivo é encontrar o melhor caminho entre um estado inicial e um estado objetivo, passando por farmácias localizadas aleatoriamente ou definidas manualmente. Para cada repetição do experimento, são avaliados

diferentes custos e heurísticas, realizando buscas A* tanto entre o estado inicial e cada farmácia quanto entre a farmácia e o estado objetivo. O melhor caminho é aquele que apresenta o menor custo total, considerando a soma dos custos de transição entre os pontos. Além disso, a função monitora o número de nós gerados e visitados durante a execução, possibilitando uma análise detalhada do desempenho do algoritmo de busca. Os resultados de cada execução são armazenados para posterior análise. O código permite flexibilidade, permitindo a definição manual das farmácias ou a geração aleatória das mesmas.

- **Como Rodar os Algoritmos:**

  1. **Ambiente:** Certifique-se de ter o Python 3.10 instalado no seu sistema.
  2. **Instalação:** É necessário instalar a biblioteca pandas, utilize o seguinte comando:

     ```
     pip install pandas
     ```

  3. **Execução:**
     - Navegue até o diretório do projeto.
     - Execute o script `main.py` utilizando o seguinte comando:

       ```
       python main.py
       ```

     - O script gerará arquivos CSV com os resultados de cada busca executada. Com isso, basta executar todas as células do arquivo `tabelas.ipynb`
  4. **Parâmetros de Teste:**
     - Estados iniciais e objetivos são gerados aleatoriamente.
     - Funções de custo (C1, C2, C3, C4) são definidas no arquivo principal.
     - Heurísticas (Manhattan e Euclidiana) são definidas no arquivo `a_estrela.py`.
     - A quantidade de repetições de cada experimento são configuráveis nas funções `executar_experimentos_parteX`.

3 Experimentação

3.1 Descrição da solução

Este experimento teve como foco a comparação de diferentes algoritmos de busca para encontrar um caminho entre dois pontos em um ambiente simulado. A implementação envolveu a criação de funções para cada algoritmo de busca (largura, profundidade e custo uniforme), e a exploração de como diferentes funções de custo e diferentes entradas afetam o seu comportamento. A solução buscou analisar o desempenho e a aplicabilidade de cada algoritmo em cenários com variações nos custos das ações e nas coordenadas de origem e destino.

3.1.1 Parte 0: Experimentação Livre

O objetivo deste experimento era explorar o comportamento dos algoritmos de busca individualmente, variando as funções de custo e, quando aplicável, as heurísticas. Foi possível observar como cada algoritmo reage a diferentes configurações e parâmetros.

3.1.2 Parte 1: Largura vs. Profundidade vs. Custo Uniforme

- **Objetivo:** Comparar os algoritmos de busca em largura, profundidade e custo uniforme, variando as funções de custo e observando como cada algoritmo se comporta em diferentes situações.
- **Observações:**
  - **Busca em Largura:** A busca em largura garante encontrar o caminho mais curto (em número de passos) em um grafo não ponderado, mas pode ser mais custosa em termos de memória, pois precisa manter todos os nós da "fronteira"na fila.
  - **Busca em Profundidade:** A busca em profundidade explora um caminho até o fim antes de retornar para explorar outros, ela pode não encontrar o melhor caminho, e a solução pode depender da ordem como os vizinhos são gerados.
  - **Busca de Custo Uniforme:** A busca de custo uniforme sempre encontra o caminho de menor custo, mas pode ser mais lenta e necessita de mais memória do que a busca gulosa, por exemplo.
  - **Influência da Função de Custo:** A escolha da função de custo não afeta a ordem de execução da busca em largura e profundidade, mas afeta diretamente o custo do caminho encontrado.

- **Coordendas:** Variações nas coordenadas de origem e destino afetam a quantidade de nós explorados e, consequentemente, o tempo de execução dos algoritmos.

3.1.3 Parte 2: Custo Uniforme vs. A*

- **Objetivo:** Comparar os algoritmos de busca de custo uniforme e A*, utilizando diferentes funções de custo e heurísticas, a fim de entender qual algoritmo é mais adequado para cada cenário.
- **Observações:**
  - **Busca de Custo Uniforme:** A busca de custo uniforme encontrou sempre o caminho mais curto, mas pode ser mais lenta dependendo da configuração dos estados. Por ser uma busca não informada, ela não faz uso de heurísticas para guiar a exploração dos nós, resultando em maior tempo de execução em alguns casos.
  - **Busca A*:** A busca A* encontrou sempre o melhor caminho e foi mais eficiente quando utilizando as heurísticas. Este algoritmo combina a função de custo e a heurística, permitindo que ele explore menos nós em comparação com a busca de custo uniforme.
  - **Influência da Função de Custo:** A função de custo altera diretamente o custo do caminho e o tempo de execução, influenciando a seleção dos nós tanto na busca de custo uniforme quanto na busca A*.
  - **Influência da Heurística:** A escolha da heurística impacta na performance da busca A*. A heurística de Manhattan, por exemplo, apresentou melhores resultados em cenários onde as distâncias entre os nós seguem padrões ortogonais, enquanto a heurística Euclidiana se mostrou mais eficaz em espaços abertos e sem restrições de movimento.

3.1.4 Parte 3: Busca Gulosa vs. A*

- **Objetivo:** Comparar os algoritmos de busca gulosa e A*, utilizando diferentes funções de custo e heurísticas.
- **Observações:**
  - **Busca Gulosa:** A busca gulosa não garante que encontra o melhor caminho, mas foi mais rápida que a busca A* em termos

de tempo de execução. Isso ocorre porque ela se baseia exclusivamente na heurística para guiar a busca, ignorando os custos acumulados.

- **Busca A*:** A busca A* sempre encontrou o melhor caminho, e sua performance é diretamente afetada pela heurística utilizada. Ela demonstrou maior robustez ao lidar com cenários complexos e custo variável entre os nós.
- **Funções de Custo e Heurísticas:** A combinação de funções de custo e heurísticas influenciou significativamente os resultados, evidenciando que a escolha adequada destes parâmetros depende do cenário específico.

3.1.5 Parte 4: Largura vs. Profundidade com Randomização da Vizinhança

- **Objetivo:** Analisar o impacto da randomização da ordem dos vizinhos nos algoritmos de busca em largura e profundidade, e como isso afeta os resultados.
- **Observações:**
  - **Busca em Largura:** A randomização não afeta o custo final da busca, pois este algoritmo garante encontrar o menor caminho em termos de passos para o objetivo. No entanto, a ordem dos vizinhos alterou os caminhos explorados antes de encontrar a solução.
  - **Busca em Profundidade:** A randomização afetou tanto o custo quanto o número de nós gerados e visitados. Soluções diferentes foram encontradas dependendo da ordem dos vizinhos gerados, o que evidencia a falta de determinismo nesse método.
  - **Variações:** A randomização na ordem dos vizinhos gerou resultados variados, permitindo uma avaliação mais ampla das possibilidades de caminhos explorados e custos gerados para diferentes combinações de estados iniciais e objetivos.

3.1.6 Parte 5: Caminho Mínimo Com Uma Parada A Mais

- **Objetivo:** Comparar o comportamento do algoritmo A* na busca de caminhos que passam por um ponto intermediário (farmácia).
- **Observações:**

  - **Busca A*:** O algoritmo A* sempre encontrou a solução mais eficiente, passando por uma das farmácias com menor custo total. Sua combinação de função de custo e heurística permitiu identificar rapidamente o melhor caminho entre todas as possibilidades.
  - **Funções de Custo e Heurísticas:** As diferentes funções de custo (*f1*, *f2*, *f3*, *f4*) e heurísticas (*Euclidiana* e *Manhattan*) resultaram em caminhos variados, com custos diferentes dependendo da localização das farmácias e o peso atribuido aos custos em cada função.
  - **Complexidade:** Este experimento apresentou maior complexidade devido à necessidade de considerar multiplas paradas intermediárias e a relação entre as diferentes funções utilizadas, impactando diretamente no tempo de execução e custo final.

- **Arquivos:** Para cada experimento, existe um arquivo de texto que contém todos os resultados obtidos.

4 Considerações finais

4.1 Considerações Finais

- A implementação dos algoritmos de busca foi concluída com êxito, permitindo a realização de comparações detalhadas entre as diferentes abordagens nos experimentos propostos.

- A busca A* destacou-se como o algoritmo mais eficiente na maioria dos cenários analisados, devido à sua capacidade de encontrar sempre o caminho ótimo enquanto mantém uma boa performance graças ao uso combinado da função de custo e heurística.

- A influência da função de custo e da heurística foi evidente ao longo dos experimentos. Observou-se que a escolha cuidadosa dessas variáveis é essencial para melhorar o desempenho dos algoritmos e adequá-los às especificidades de cada problema.

- A busca em profundidade apresentou bom desempenho em termos de rapidez, mas não garantiu a solução de menor custo, sendo altamente dependente da ordem em que os vizinhos foram gerados, especialmente em cenários com randomização.

- A randomização na ordem dos vizinhos teve impacto significativo na busca em profundidade, alterando os caminhos encontrados e os custos

associados. No caso da busca em largura, apesar de não afetar o custo final, influenciou a ordem de exploração dos nós.

- A busca gulosa demonstrou ser uma alternativa eficiente em termos de tempo de execução, mas não garantiu a solução ótima devido à sua dependência exclusiva da heurística, o que a torna menos robusta em situações de custos variados.

- Os experimentos com a busca A* em cenários mais complexos, como encontrar o caminho com uma parada intermediária, evidenciaram a escalabilidade do algoritmo, ainda que a complexidade do problema tenha aumentado consideravelmente o tempo de execução.

Este relatório apresenta uma visão abrangente das implementações, dos experimentos realizados e das conclusões obtidas. O código foi estruturado de forma modular, com arquivos separados para cada algoritmo e experimento, facilitando a leitura, manutenção e futuras extensões. As funções implementadas fornecem resultados claros, incluindo o custo do caminho, a sequência de estados percorridos e a quantidade de nós visitados e gerados. A estrutura desenvolvida atende a todos os requisitos do trabalho, permitindo a execução e avaliação dos algoritmos de forma prática e eficiente.