

Trabalho Prático I - Introdução à Inteligência Artificial

Luís Felipe Ramos Ferreira

lframes.lf@gmail.com

1 Introdução

O Trabalho Prático I da disciplina de Introdução a Inteligência Artificial teve como objetivo a implementação de 5 algoritmos de busca diferentes em um problema de menor caminho entre dois pontos em um mapa bidimensional.

2 Implementação

O projeto foi implementado utilizando C++ e Python. O código principal, que contém a implementação dos 5 algoritmos citados foi escrito em C++, versão 17, e está todo contido no arquivo `main.cpp`. Arquivos utilitários utilizados para realização de *benchmarks* comparativos entre os algoritmos, criação dos gráficos para análise dos dados foram criados utilizando Python, versão 3.12.3, e os pacotes utilizados foram gerenciados com o gerenciador de pacotes pip.

Instruções de como executar o programa principal e os arquivos utilitários, assim como a listagem de dependências, estão presentes no arquivo `README.md`, no diretório raiz do projeto.

Os testes apresentados na análise de resultados foram feitos em uma máquina com Ubuntu 24.04.01 e 16GB de memória RAM, na CPU 11th Gen Intel i5-11400F (12) @ 4.400GHz.

3 Descrição dos algoritmos

Nesta seção, é feita uma breve descrição dos algoritmos utilizados e suas principais diferenças. Para todos eles, consideramos um *branch factor* b , que a solução está no nível d e m como a profundidade máxima da árvore de busca.

- *Breadth First Search (BFS)*

A busca em largura é um algoritmo em que, a cada iteração, se expande o nó mais raso ainda não expandido na busca. Em termos informais,

primeiro se expande a raiz, depois os sucessores da raiz, depois os sucessores dos sucessores da raiz, e assim se segue. O algoritmo pode ser implementado com uma fila, pois a ideia dele segue um arquitetura de FIFO (*First In First Out*). No algoritmo podemos fazer algo chamado *Early Goal Test*, em que checamos se um nó adicionado na fila já é o nó final antes de processá-lo ao sair da fila. Faz-se isso pois com isso o algoritmo não irá precisar adicionar mais nós à fila e nem processar os que já estão lá se o nó final já estiver pronto para ser processado.

- **Completo**
- **Ótimo**, se e somente se o custo seja uma função não decrescente da profundidade do nó (Por exemplo, na busca de menor caminho em um grafo sem pesos nas arestas.).
- **Complexidade**: tempo $\mathcal{O}(b^d)$ e espaço $\mathcal{O}(b^d)$

- *Iterative Depth Search (IDS)*

A busca com profundidade iterativa é uma junção dos algoritmos BFS e DFS. Nele, aplicamos uma busca em profundidade iterativa, isto é, a cada iteração, aumentamos a profundidade permitida na busca. Nesse sentido, os benefícios da BFS e da DFS são combinados.

- **Completo**.
- **Ótimo**, considerando custo crescente como no caso da busca em largura.
- **Complexidade**: tempo $\mathcal{O}(b^d)$ e espaço

- *Uniform Cost Search (UCS)*

O algoritmo de busca de custo uniforme é muito semelhante à BFS, mas o próximo nó expandido é na verdade o nó com menor custo até o momento. Para fazer isso, é necessário manter uma ordenação dos custos dos nós, e pra isso podemos utilizar uma fila de prioridades ou um *heap* na implementação.

- **Completo**, se e somente se cada passo do algoritmo tem custo positivo (Considerando um grafo com pesos nas arestas, se houver uma aresta com peso negativo, o algoritmo não funcionaria).
- **Ótimo**, uma vez que seguimos o menor custo
- **Complexidade**: se C^* for a solução ótima, o pior caso de tempo e espaço é $\mathcal{O}(b^{1+\frac{C^*}{\epsilon}})$.

- *Greedy*

O algoritmo de busca gulosa é um tipo de algoritmo de busca com informação. Nele, expande-se o nó do espaço de busca que está mais próximo do estado final de acordo com alguma função heurística definida previamente. O termo "guloso", conforme visto nos slides da disciplina, significa

que o método procura reduzir o custo imediato para alcançar o objetivo na expansão de cada nó, porém sem se preocupar com o custo total do caminho. Esse algoritmo também é chamado de *best first search*.

- **Completo:** não, ele pode entrar em um laço infinito se isso não for tratado
- **Ótimo:** não
- **Complexidade:** $\mathcal{O}(b^m)$ para tempo e espaço

- **A***

O algoritmo A* é um algoritmo de busca com informação que se aproveita das vantagens dos algoritmos de busca uniforme e guloso, ou seja, a escolha do próximo nó a ser expandido é feita de acordo com o custo real de chegar ao nó atual mais a **estimativa** de custo de se chegar ao estado final a partir do nó atual.

- **Completo**
- **Ótimo** se a heurística escolhida for admissível, isto é, o custo indicado pela heurística é menor ou igual ao custo real para o estado final. Também é eficientemente ótimo, ou seja, expande o menor número de nodos possível dentre os algoritmos ótimos.
- **Complexidade:** Apesar da eficiência ainda pode ser exponencial no número de nós expandidos a depender da heurística escolhida. Os nós são mantidos na memória, o que também pode causar problemas.

4 Modelagem

Para fazer a leitura dos dados, uma função denominada `parse_input_file` foi criada. Ela recebe como entrada o caminho para o arquivo de entrada e retorna uma matriz que representa o mapa. Cada entrada da matriz é um elemento de uma enumeração auxiliar criada para identificar cada tipo de terreno com mais facilidade.

Para cada algoritmo foi criada uma função que recebe como entrada uma referência para a estrutura que representa o mapa e os quatro inteiros que representam as coordenadas de começo e fim do caminho a ser percorrido pelo agente. Para padronizar a saída dos algoritmos, uma `struct` auxiliar foi criada, denotada como *SearchMethodOutput*. Ela contém as informações de saída de um algoritmo de busca, sendo elas o custo e as coordenadas dos estados no caminho encontrado pelo agente.

Conforme especificado na descrição dos algoritmos, cada um deles requer uma estrutura de dados específica para ser implementado, como no caso da busca em largura, que pode ser implementada utilizando uma fila. Desse modo, a biblioteca padrão da linguagem C++ foi utilizada, pois ela contém implementações eficientes e já prontas dessas estruturas, as quais foram utilizadas

na implementação da lógica dos algoritmos. Alguns exemplos de estruturas de dados utilizadas são pilha, fila, fila de prioridades e vetor.

A função `main` identifica a partir da entrada padrão qual o algoritmo que deve ser utilizado e invoca esse procedimento com a entrada correta após fazer a leitura do arquivo de entrada. Mais detalhes podem ser encontrados nos comentários do código no arquivo `main.cpp`.

Os arquivos `plot_path.py` e `plot_expanded_states.py` são arquivos utilitários escritos em *Python* para gerar os gráficos que mostram o caminho encontrado por cada algoritmo e os estados expandidos por eles. Por serem apenas arquivos utilitários o código utilizado foi copiado de fóruns online como *GeeksForGeeks* e *StackOverflow*, para que eles fossem criados mais rapidamente, para que assim mais foco pudesse ser dada às análises de resultados e à implementação dos algoritmos.

5 Heurísticas utilizadas

Nesta seção são apresentadas as heurísticas utilizadas nos algoritmos *Greedy* e A^* . Em particular, a heurística escolhida para ambos os algoritmos foi a distância *Manhattan* entre dois estados/coordenadas, pois é uma heurística extremamente simples de implementar e utilizar e, além disso, muito eficiente para esse problema em particular. Dados duas coordenadas (x_i, y_i) e (x_f, y_f) , a distância *Manhattan* entre elas é formalmente expressada como $|x_i - x_f| + |y_i - y_f|$.

Ela é uma heurística admissível pois o custo indicado por ela é menor ou igual ao custo real para chegar ao estado final. Isso pois o melhor caso possível, isto é, o caminho de menor custo possível entre duas coordenadas, nesse problema, seria justamente a distância *Manhattan* entre elas. Logo, a heurística é sempre menor ou igual ao custo real.

6 Resultados

Para o cálculo dos resultados, uma série de testes foi realizada. Em particular, o arquivo `benchmark.py` é um código em *Python* utilizado para realizar essa bateria de testes. Como o mapa *cidade* possui dimensão 256 por 256, os testes realizados seguiram a seguinte lógica para avaliarmos como os algoritmos se comportam conforme o estado final se distancia do estado inicial.

- O estado inicial é sempre o estado com coordenada (1, 1)
- O estado final começa a partir do estado com coordenada (2, 2) e iterativamente aumenta com um salto de 4, indo para (6, 6), (10, 10), etc.

Os resultados para cada caso de estado final foram coletados e serão apresentados nessa seção. A escolha do pulo de tamanho 4 foi feita para que os *benchmarks* não demorassem muito para serem executados mas ainda assim

uma boa quantidade de informação pudesse ser coletada. Caso o estado final seja uma parede (isto é, um local que não se pode passar) apenas o ignoramos. De modo geral, isso acontece pouco e não impacta na análise.

6.1 Custo dos caminhos encontrados

O primeiro resultado que se mostra importante na comparação de algoritmos de busca é o custo dos caminhos encontrados. Os gráficos abaixo mostram a evolução dos custos dos caminhos para ambos os mapas disponibilizados, conforme o estado final se distancia do inicial.

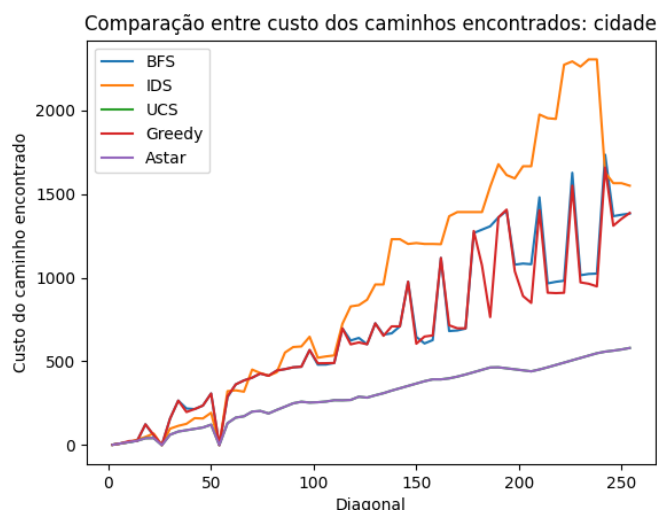
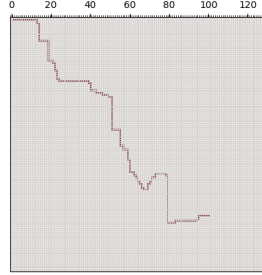


Figure 1: Comparaç o entre custos dos caminhos - cidade

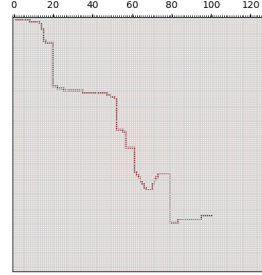
Pode-se notar algumas coisas interessantes. Primeiramente, notamos que o custo da UCS e do A* s o iguais. Isso acontece pois ambos algoritmos s o  timos, ou seja, sempre encontram o menor caminho poss vel.   importante notar tamb m como os algoritmos guloso e a BFS apresentam custos similares, enquanto a IDS  , de longe, o algoritmo que encontra o pior custo.

6.2 Caminho encontrado

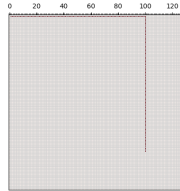
O caminho encontrado   uma forma interessante de se analisar como cada algoritmo se comporta. Para a an lise que ser  feita aqui, optou-se por imprimir o caminho encontrado por cada algoritmo entre o estado inicial (1, 1) e o estado final (256, 256). Em particular, mostramos os caminhos encontrados por cada algoritmo no mapa *cidade*.



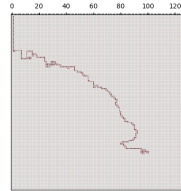
(a) UCS



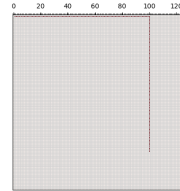
(b) A*



(c) Greedy



(d) IDS



(e) BFS

Primeiramente, nota-se que o caminho encontrado do UCS e do A* são iguais, o que está correto, pois o A* é basicamente idêntico ao UCS, mas com o uso de uma heurística para guiar o agente mais rapidamente em direção ao estado final, diminuindo o número de estados expandidos. O algoritmo guloso e a BFS também encontraram o mesmo caminho, mas isso foi uma coincidência dessa instância de análise. A IDS, por sua vez, encontrou um caminho diferente dos outros, naturalmente apresentando um caminho que seria encontrado por uma busca em profundidade.

6.3 Número de estados expandidos

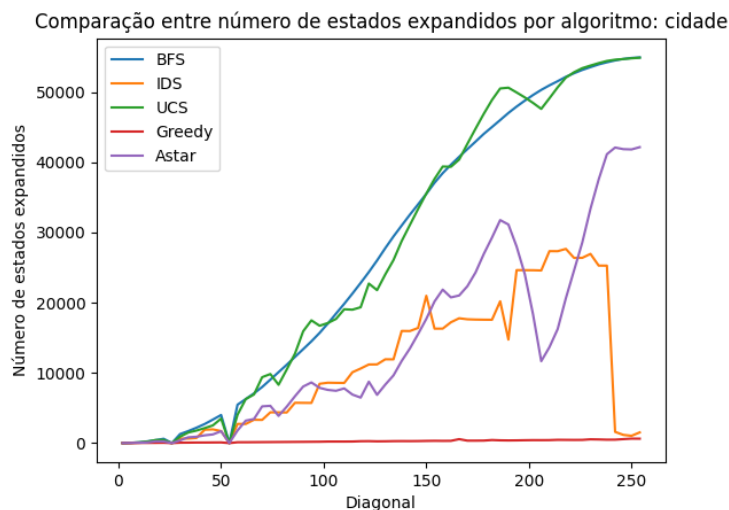


Figure 2: Comparação de número de estados expandidos - cidade

Em relação ao número de estados expandidos, podemos ver claramente a diferença entre os algoritmos. A BFS e a UCS expandem uma grande quantidade de estados. Isso vem naturalmente da natureza exploratória de ambos os algoritmos. É curioso enxergar como a curva do algoritmo BFS é suave. Isso ocorre devido ao fato do algoritmo não se guiar de maneira alguma em direção ao estado final, e apenas explora em largura o espaço de busca. O algoritmo A*, por sua vez, se aproveita do uso de heurísticas para otimizar a busca feita pelo UCS, e por isso o número de estados expandidos para ele é menor que a do UCS. No entanto, mesmo assim, muitos estados ainda são expandidos, quando comparamos com o algoritmo guloso. O guloso não encontra a resposta ótima, mas como o agente segue um caminho gulosamente que diminui a distância *Manhattan* até o estado final, pouquíssimos estados são expandidos, o que é uma grande vantagem do algoritmo. A IDS, por fim, também expande poucos estados, devido à natureza do uso de uma busca em profundidade.

6.4 Tempo de execução

Em relação ao tempo de execução, os resultados encontrados indicam exatamente o que se esperava dos algoritmos. A IDS é, de longe, a que mais demora. Isso ocorre devido à natureza do algoritmo, que repete iterativamente buscas em profundidade conforme aumenta a profundidade máxima utilizada.

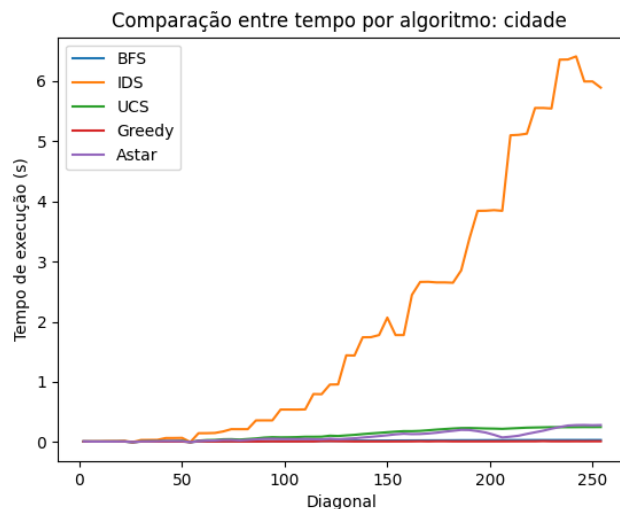


Figure 3: Comparação de tempo - cidade

Para tornar a comparação entre os algoritmos mais fácil, temos também um gráfico sem o tempo de execução da IDS.

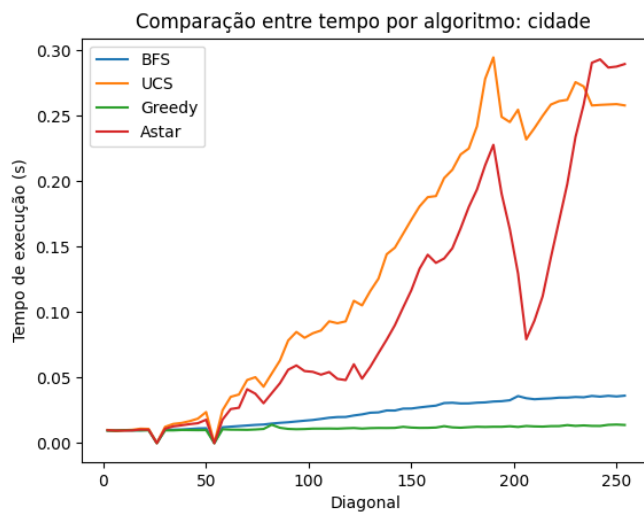


Figure 4: Comparação de tempo - cidade (sem IDS)

Os algoritmos UCS e A* possuem tempos de execução semelhantes, embora o A* seja mais rápido, devido ao uso de uma heurística para guiar o agente mais

rapidamente em direção ao estado final. A BFS é um pouco mais lenta que a busca gulosa, uma vez que não se baseia em heurística nenhuma para se guiar e explora os estados em largura. A busca gulosa é a mais rápida, naturalmente, pois ela explora poucos estados e se guia pela heurística da distância *Manhattan* em direção ao estado final, abrindo mão do custo total final.

7 Conclusão

A implementação dos algoritmos assim como os resultados encontrados ajudaram a compreender melhor como cada um dos algoritmos de busca funcionam, assim como seus pontos fracos e fortes. De maneira geral, o algoritmo A^* aparenta ser o melhor dentre todos eles para resolver o problema em questão, pois retorna o resultado ótimo, executa rapidamente, e heurísticamente é capaz de diminuir o número de estados expandidos. Em contrapartida, o algoritmo IDS parece de longe o pior, sendo disparadamente o que mais demora pra executar.