Trabalho Prático I - Introdução à Inteligência Artificial

Luís Felipe Ramos Ferreira

lframos.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 contêm 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 ds dados foram criados utilizando Python, versão 3.12.3, e os pacotes utilizados foram manejados 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 combinado.

  - `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 contêm 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 contêm implementações eficientes e já prontas dessas estruturas, as quais foram utilizadas na implementação da lógica dos algoritmos.

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*.

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 distãncia *Manhtann* entre elas. Logo, a heurística é sempre menor ou igual ao custo real.Consistente?

- Distância *Manhattan*
- NO mais barato atualmente.

6 Resultados

6.1 Número de estados expandidos

6.2 Tempo de execução

7 Discussão dos resultados

blabla