Trabalho Prático 1
Introdução a Inteligencia Artificial

João Vitor Soares Santos
2023002138

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Belo Horizonte - MG - Brasil

joaosoares@dcc.ufmg.br

1 Introdução

Este trabalho explora conceitos fundamentais de busca em espaço de estados, com foco na aplicação prática de algoritmos de Path-Finding. O objetivo principal é implementar e comparar cinco métodos de busca: Busca em Largura (BFS)[1], Aprofundamento Iterativo (IDS)[2], Busca de Custo Uniforme (UCS)[3], Greedy[4] e A*[5]. Esses algoritmos serão utilizados para resolver o problema de encontrar o menor caminho entre dois pontos em um mapa discretizado, levando em consideração custos associados a diferentes tipos de terreno.

2 Modelagem

A modelagem foi definida em três tópicos principais: a definição de estados, que representam as configurações possíveis do problema; a função sucessora, responsável por gerar novos estados a partir do atual; e as estruturas de dados utilizadas nos algorítimos.

2.1 Estado

Neste trabalho, um **estado** é definido como uma estrutura que contém duas informações essenciais:

1. **Coordenadas x e y**: Representam a posição no mapa discretizado.
2. **Custo**: Indica o custo de percorrer a posição no mapa discretizado.

Essa definição de estado é fundamental para que os algoritmos de busca possam navegar eficientemente pela árvore de busca e calcular os custos dos caminhos para construir a solução final.

2.2 Função sucessora

Neste trabalho, os movimentos permitidos para transitar de um estado para o outro são: esquerda, direita, baixo e cima. Dado que o mapa é representado por uma matriz bidimensional, a função sucessora primeiro verifica se é possível fazer cada movimento sem ultrapassar os limites da matriz. Em seguida, verifica-se se o movimento não leva o agente a uma coordenada representando um terreno do tipo parede, que é considerado intransitável. Por fim, a função `sucessorFunction`,

que foi desenvolvida nesse trabalho, retorna um vetor contendo apenas os movimentos válidos — esquerda, direita, cima e baixo — que permanecem dentro dos limites do mapa e não levam a uma coordenada do tipo parede.

2.3 Estruturas de dados

Diferentes estruturas de dados foram utilizadas para cada algoritmo, incluindo filas [6] e filas de prioridade[7], escolhidas de acordo com as necessidades específicas de cada caso.

2.3.1 Fila (Queue)

Para este trabalho, foi utilizada a estrutura de dados *fila* (`std::queue`) da biblioteca padrão do *C++*. Essa estrutura segue o princípio FIFO (First In, First Out), onde os elementos são inseridos no final da fila e removidos do início.

Principais métodos

- `void push(const T& value)`: Insere um elemento no final da fila ($O(1)$).
- `void pop()`: Remove o elemento no início da fila ($O(1)$).
- `T& front()`: Retorna uma referência ao primeiro elemento ($O(1)$).
- `bool empty()`: Verifica se a fila está vazia ($O(1)$).

A utilização da `std::queue` facilitou o desenvolvimento, fornecendo uma solução pronta e confiável para os algoritmos que necessitam de acesso sequencial aos estados.

2.3.2 Fila de prioridade (Indexed Priority Queue)

Para este trabalho, a estrutura de dados *heap indexado* foi implementada. Sua principal diferença comparada com a sua versão padrão é o fato de que é possível verificar se determinado elemento está no *heap*, retornando um ponteiro para o elemento, em tempo constante $O(1)$. Graças a isso, a operação de atualizar determinado item do *heap* através de sua chave é facilitada. Para isso ser possível, um *array* auxiliar que guarda qual elemento está em determinada posição do *heap* é mantido.

Principais métodos

- `void insert(Key key, Value value)`: $O(\log(\text{heapsize}))$
- `Tuple<Key, Value>* contains(Key key)`: $O(1)$
- `void update(Key key, Value value)`: $O(\log(\text{heapsize}))$
- `Tuple<Key, Value> remove()`: $O(\log(\text{heapsize}))$

Com o objetivo de manter o *heap* ordenado, as funções auxiliares `heapifyDown`, que ordena o *heap* de baixo para cima, e `heapifyUp`, que ordena o *heap* de cima para baixo, são usadas.

3 Algorítimos

Os algoritmos de Path-Finding diferem em suas estratégias de busca, no uso de heurísticas e nas estruturas de dados empregadas:

- **BFS (Busca em Largura)**: Explora todos os estados de um mesmo nível antes de avançar para o próximo. Garante a solução ótima para grafos não ponderados. Utiliza uma fila para explorar os estados.
- **IDS (Busca Iterativa em Profundidade)**: Combina a busca em profundidade com a busca em largura, repetindo a DFS[8] com profundidades limitadas crescentes. Garante a solução ótima em grafos não ponderados, mas é menos eficiente em tempo. Utiliza uma pilha[9] ou uma estratégia recursiva para explorar os estados (O último foi usado neste trabalho).
- **UCS (Dijkstra)**: Encontra o caminho mais curto em grafos ponderados usando uma abordagem de busca por custo acumulado. Não utiliza heurísticas. Utiliza uma fila de prioridades.
- **A***: Estende o Dijkstra ao usar uma heurística admissível para priorizar os caminhos mais promissores. É eficiente e garante a solução ótima, desde que a heurística seja consistente. Utiliza uma fila de prioridades.
- **Greedy**: Baseia-se exclusivamente na heurística para escolher o próximo estado. Não garante solução ótima, mas pode ser mais rápido em determinados cenários. Utiliza uma fila de prioridades.

4 Heurísticas

Os algorítimos de busca com informação requerem a utilização de heurísticas admissíveis para que a solução final seja ao máximo otimizada. Uma heurística é considerada admissível quando nunca supera o custo real para alcançar o objetivo, ou seja, é uma heurística otimista. Isso garante que o algorítimo não descarte caminhos que possivelmente podem levar a solução ótima, mantendo a eficiência da busca.

A heurística escolhida para auxiliar os algorítimos *A** e *Greedy* foi a distancia de Manhattan[10]. Considerando duas coordenadas, a distancia de Manhattan é calculada ao somar o modulo da diferença das coordenadas no eixo-x com o modulo da diferença das coordenadas no eixo-y.

Visto que o operador só pode se movimentar para cima, baixo, esquerda e direita a distancia de Manhattan é exatamente o número de movimentos mínimos que um operador precisa para se deslocar de um ponto ao outro, considerando que todo caminho tem custo um e que não existem impedimentos, sendo assim, é uma heurística admissível. Considerando outras heurísticas, como a distancia Euclideana[11], nossa heurística é firme, visto que a soma dos dois lados de um triangulo é maior que sua diagonal.

5 Análise quantitativa

A quantidade de nós expandidos tem uma relação direta no tempo de execução dos algorítimos de busca em espaços de estados, seja pela quantidade ou pela forma como o próximo nó a ser expandido é escolhido.

A Figura 1 apresenta uma comparação entre os algoritmos, mostrando o número de estados expandidos em função da distância dos caminhos explorados.

O *IDS* se destaca como o algorítimo que mais expande estados, isso se deve ao fato dele executar repetidamente o *DFS* com profundidade limitada crescente. Também é possível observar que o *BFS* e o *UCS* expandem aproximadamente a mesma quantidade de nós. Isso ocorre porque ambos os algoritmos exploram os estados de forma semelhante: o *BFS* explora em camadas crescentes,

enquanto o *UCS* o faz a partir de distâncias crescentes. Por último, vale ressaltar que uma boa heurística pode reduzir significativamente o número de nós expandidos. Isso é evidente nos algoritmos *A* e *Greedy*, que utilizam a distancia de Manhattan para uma busca mais eficiente. Como o *Greedy* se baseia apenas no valor da heuristica, que tende a decrescer, ele expande menos nós que o *A**.

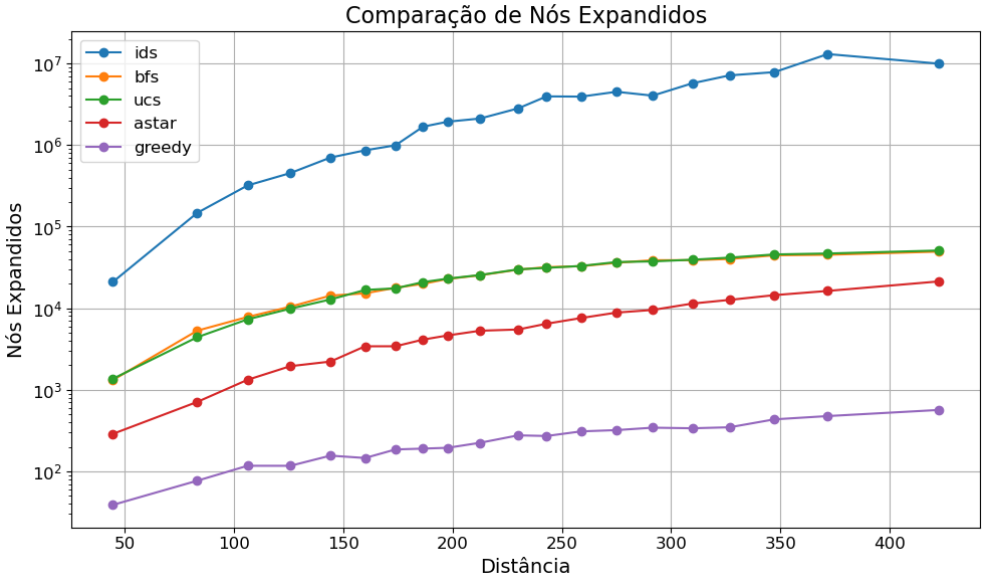

Figura 1: Relação entre a distância de um caminho e a quantidade de estados expandidos

A Figura 2 apresenta uma comparação entre os algoritmos, mostrando o tempo de execução em função da distância dos caminhos explorados.

O *IDS*, embora use uma estrutura de dados simples (pilha/recursão), tem o maior tempo de execução entre os algorítimos por repetidamente executar o *DFS*. Também vale mencionar, mesmo que a quantidade de nós expandidos entre *BFS* e o *UCS* seja próxima, o *UCS* utiliza uma fila de prioridades, essa complexidade adicional o faz ter um tempo de execução maior que o *BFS*. Por fim, pelo fato do *A** e o *Greedy* expandirem menos nós devido a distância de Manhattan, mesmo que também utilizem filas de prioridade, seus tempos de execução são pequenos comparados aos outros algorítimos.

6 Resultados e Considerações Finais

Os resultados indicam que o *IDS* deve ser evitado devido ao seu alto tempo de execução e ao grande número de nós expandidos. Esse comportamento ocorre porque o algorítimo repete a execução do *DFS*, o que resulta em uma exploração ineficiente do espaço de estados. No caso de grafos não ponderados, o *BFS* é preferível ao *UCS*, pois, apesar de ambos explorarem uma quantidade similar de estados, o *BFS* apresenta um tempo de execução menor, já que não envolve a sobrecarga de uma fila de prioridades como o *UCS*. Além disso, o uso de heurísticas é uma boa estratégia já que ajuda a reduzir o número de nós expandidos e o tempo de execução. Por fim, quando o foco é a minimizar tempo de execução e o número de nós expandidos, o *Greedy* é uma excelente opção, pois, embora não seja ótimo como o *UCS* e o *A**, ele expande menos nós e tem um desempenho superior, principalmente quando a heurística é admissível.

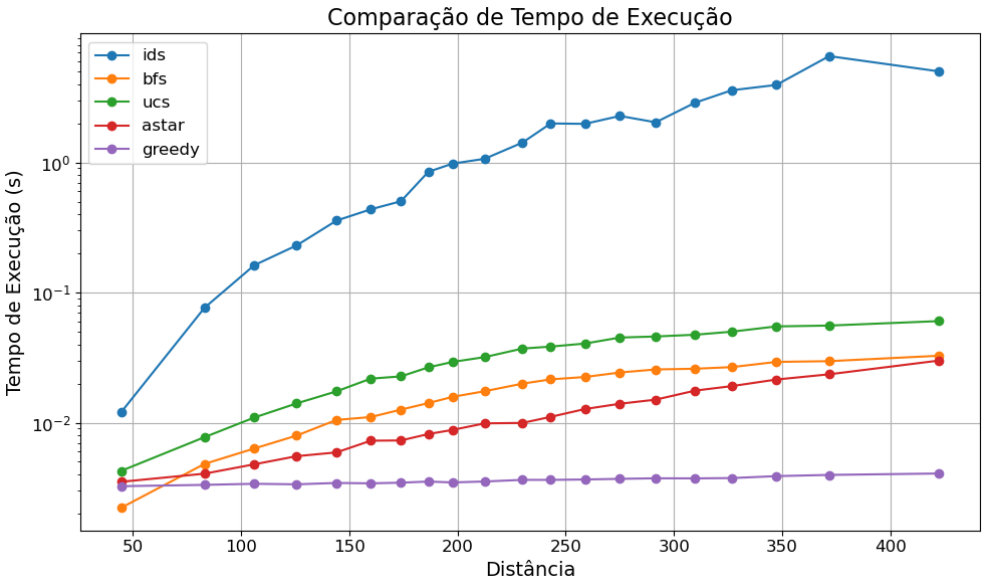

Figura 2: Relação entre a distância de um caminho e o tempo de execução

Por meio da resolução desse trabalho, foi possível praticar conceitos de algoritmos de busca em espaços de estados com e sem informação. Além disso, ficou evidente a relação entre quantidade de nós expandidos e tempo de execução e também a importância de escolher abordagens que equilibram eficiência e precisão, dependendo do contexto do problema. Esse processo proporcionou não apenas um aprendizado técnico, mas também uma visão mais crítica sobre como analisar resultados e justificar escolhas em soluções computacionais.

Referências

[1] Wikipedia. *Busca em largura*. URL: https://pt.wikipedia.org/wiki/Busca_em_largura.

[2] Wikipedia contributors. *Iterative deepening depth-first search*. URL: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search.

[3] Wikipedia contributors. *Dijkstra's algorithm*. URL: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[4] Wikipedia contributors. *Best-first search*. URL: https://en.wikipedia.org/wiki/Best-first_search.

[5] Wikipedia contributors. *A* search algorithm*. URL: https://en.wikipedia.org/wiki/A*_search_algorithm.

[6] Wikipedia. *Queue*. URL: https://en.wikipedia.org/wiki/Queue_(abstract_data_type).

[7] Wikipedia contributors. *Heap (data structure)*. URL: https://en.wikipedia.org/wiki/Heap_(data_structure).

[8] Wikipedia. *Busca em profundidade*. URL: https://pt.wikipedia.org/wiki/Busca_em_profundidade.

[9] Wikipedia. *Stack*. URL: https://en.wikipedia.org/wiki/Stack_(abstract_data_type).

[10] Wikipedia contributors. *Taxicab geometry*. URL: https://en.wikipedia.org/wiki/Taxicab_geometry.

[11] Wikipedia contributors. *Euclidean distance*. URL: https://en.wikipedia.org/wiki/Euclidean_distance.