Trabalho Pratico 1: Algoritmo de Busca para Menor Caminho

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

Leonardo de Oliveira Maia: 2019042139
lom2019@ufmg.br

1. Introdução:

Este trabalho tem como objetivo praticar os conceitos e algoritmos de busca em espaço de estados. Além disso, proporcionará uma forma de comparar diferentes tipos de métodos.

2. Principais Ferramentas:

As principais ferramentas utilizadas foram linguagem C++, Python, Makefile, GDB.

3. Apresentação das Estruturas e Modelagem dos Componentes da Busca

3.1 Estruturas Principais

1. **Mapa e Representação do Ambiente**

O mapa é representado como uma matriz de caracteres (`std::vector<std::vector<char>>`),

onde cada célula descreve um tipo de terreno ou obstáculo:

- . : Grama, custo base de 1.0.
- ; : Grama alta, custo de 1.5.
- +: Água, custo de 2.5.
- @: Obstáculo intransponível.
- Outros caracteres são considerados obstáculos com custo alto (6.0).

```cpp
// Define o custo de cada tipo de terreno
double get_cost(char terrain) {
    switch (terrain) {
        case '.': return 1.0; // Grama
        case ';': return 1.5; // Grama Alta
        case '+': return 2.5; // Água
        default: return 6.0; // Fogo
    }
}
```

Segue imagens de como esta sendo feito a criação da estrutura do mapa de acordo com o caminho, onde se encontra o arquivo do mapa, sera retornado as dimensões da matriz do mapa que são encontradas no topo do arquivo e a matriz do mapa de acordo com o restantes das linhas do arquivo.

```cpp
auto [size, map] = read_map(map_path);
```

```cpp
int rows, cols;
if (sscanf(size_line.c_str(), "%d %d", &rows, &cols) != 2) {
    throw std::runtime_error("Formato inválido para a primeira linha do arquivo. Esperado: '<colunas> <linhas>'");
}

// Ler o restante do mapa
std::vector<std::vector<char>> map;
std::string line;
while (std::getline(file, line)) {
    map.emplace_back(line.begin(), line.end());
}
```

2. **Estado**

- Cada estado no espaço de busca é definido por:
    - Coordenadas `(x, y)` representando a posição no mapa.
    - O custo acumulado para alcançar essa posição.
    - A profundidade (usada no **IDS**).
- Estados são armazenados em diferentes estruturas dependendo do algoritmo:
    - **BFS**: Fila (`std::queue`).
    - **IDS**: Pilha (`std::stack`).
    - **UCS, Greedy, A***: Fila de prioridade (`std::priority_queue`).

3. **Custo e Visitados**

- `cost`: Matriz bidimensional que armazena o custo acumulado para alcançar cada célula do mapa.
- `visited`: Matriz bidimensional que indica se uma célula já foi visitada.
- `parent`: Matriz bidimensional que armazena o pai de cada célula, usado para reconstruir o caminho.

```
std::vector<std::vector<double>> cost(size.first, std::vector<double>(size.second, std::numeric_limits<double>::infinity()));
std::vector<std::vector<bool>> visited(size.first, std::vector<bool>(size.second, false));
std::vector<std::vector<std::pair<int, int>>> parent(size.first, std::vector<std::pair<int, int>>(size.second, {-1, -1}));
```

3.2 Modelagem dos Componentes

1. **Função Sucessora**

    - As direções de movimento possíveis são modeladas como deslocamentos: `{(-1, 0), (1, 0), (0, -1), (0, 1)}`.
    - Para cada estado atual `(x, y)`, os sucessores são calculados verificando os deslocamentos em relação ao mapa e o custo de transição.

```
// Explorar as direções possíveis
for (const auto &[dx, dy] : directions) {
    int nx = x + dx;
    int ny = y + dy;
```

2. **Condição de Terminação**

    - A busca termina quando o estado atual corresponde à posição-alvo `(xf, yf)`.
    - Se não houver caminho para o destino, o algoritmo retorna que nenhum caminho foi encontrado.

3. **Reconstrução do Caminho**

    - Após encontrar o destino, o caminho é reconstruído iterativamente a partir do nó destino usando a matriz `parent`, que armazena o nó anterior de cada posição.

```cpp
// Reconstrução do caminho do destino até a origem
std::vector<std::pair<int, int>> path;
for (int cx = xf, cy = yf; cx != -1 && cy != -1; std::tie(cx, cy) = parent[cx][cy]) {
    path.push_back({cx, cy});
}

// Imprimir o caminho do início ao fim (sem a necessidade de inverter)
for (int i = path.size() - 1; i >= 0; --i) {
    std::cout << " (" << path[i].second << "," << path[i].first << ")";
}
std::cout << std::endl;
```

4. **Controle de Limites e Obstáculos**
    - Cada movimento considera:
        - Limites do mapa.
        - Presença de obstáculos (células marcadas com @).

```cpp
// Verificar se a nova posição está dentro dos limites e não é um obstáculo
if (nx >= 0 && ny >= 0 && nx < size.first && ny < size.second && map[nx][ny] != '@') {
```

5. **Medição de Desempenho**
    - Para cada algoritmo, o número de estados expandidos e o tempo de execução são registrados em um arquivo de saída (`outfile`), permitindo comparações quantitativas entre abordagens.

```cpp
// Variaveis para Comparacao
int expanded = 0;
auto start_time = std::chrono::high_resolution_clock::now(); // Inicia o temporizador
```

4. Descrição das Principais Diferenças entre os Algoritmos:

4.1 BFS (Busca em Largura)

- **Estrutura Principal**
    - Utiliza uma fila (`std::queue`) para processar os nós em ordem de descoberta.
- **Características**:
    - Explora todos os nós a um determinado custo antes de passar para custos maiores.
    - Não leva em conta o custo do terreno de forma detalhada, mas apenas a ordem de descoberta.
    - Melhor para encontrar o caminho mais curto em grafos onde todas as arestas têm o mesmo custo.
    - Não é ideal para mapas com custos diferentes para os terrenos, mas aqui adapta o custo acumulado.
    - Completo: sim

- Ótimo: sim, desde que o custo seja uma função não decrescente da profundidade do nodo (ex. custo 1 para cada ação)
- Complexidade: Considerando branch factor b, solução no nível d temos
  - Tempo: b+b2+b3+… + bd = O(b^d)
  - Espaço: igual

4.2 IDS (Busca em Profundidade Iterativa)

- **Estrutura Principal**:
  - Utiliza uma pilha (`std::stack`) para simular a busca em profundidade e realiza múltiplas iterações aumentando a profundidade máxima permitida.
- **Características**:
  - Combina as vantagens da busca em profundidade (menor uso de memória) e da busca em largura (garante encontrar a solução mais curta).
  - Explora os caminhos em profundidade até um limite e incrementa esse limite iterativamente.
  - Pode ser ineficiente em tempo devido à repetição de busca em níveis mais baixos a cada iteração.
  - Adequado para problemas onde a profundidade da solução não é conhecida antecipadamente.
  - Completo e Ótimo (considerando custo crescente)
  - Complexidade
    - Tempo: O(b^d), onde b é o fator de ramificação (o número médio de sucessores de cada nó) e d é a profundidade da solução.
    - Espaço: O(b*d)

4.3 UCS (Busca de Custo Uniforme)

- **Estrutura Principal**:
  - Utiliza uma fila de prioridade (`std::priority_queue`) para expandir os nós com menor custo acumulado primeiro.
- **Características**:
  - Prioriza nós com menor custo total até o momento.
  - Ideal para mapas com custos variados, garantindo encontrar o caminho de menor custo total.
  - Similar ao algoritmo A*, mas sem heurística (considera apenas o custo real percorrido).
  - Eficiência em encontrar o caminho ótimo, mas pode expandir muitos nós se os custos forem uniformemente pequenos.
  - Completo: sim (se cada passo tem um custo ≥ ε)
  - Ótimo: sim (segue o menor custo)

- Complexidade: Considerando C* como o custo da solução ótima e que cada passo tem um custo de pelo menos ε, no pior caso: O(b^(1+C*/ ε)) para tempo e espaço.

4.4 Greedy (Busca Gulosa)

- **Estrutura Principal**:
  - Utiliza uma fila de prioridade (`std::priority_queue`) para expandir os nós com menor heurística em relação ao destino.
- **Características**:
  - Prioriza estados com menor custo estimado até o objetivo.
  - Usa uma fila de prioridade ordenada pela **heurística**.
  - Ignora os custos já acumulados, o que pode levar a caminhos subótimos.
  - Não é garantido que encontre o caminho mais curto.
  - Simples de implementar e rápido para grafos pequenos.
  - Funciona bem em ambientes onde a heurística é uma boa aproximação.
  - Busca Gulosa
  - Ótimo: não
  - Completo: não (pode entrar em loop)
  - Complexidade de tempo e espaço O(b^m), onde b é o fator de expansão e m profundidade máxima da árvore de busca

4.4 Algoritmo A*

- **Estrutura Principal**:
  - Utiliza uma fila de prioridade (`std::priority_queue`) para expandir o nó com menor custo acumulado do caminho combinado com a estimativa heurística até o objetivo. É mais robusto que o Greedy e garante encontrar o caminho ótimo, desde que a heurística seja **admissível** (não superestima o custo).
- **Características**:
  - Usa uma função `f(n) = g(n) + h(n)`:
  - `g(n)` é o custo acumulado do início até o nó `n`.
  - `h(n)` é a estimativa heurística do custo de `n` ao destino.
  - Utiliza uma fila de prioridade, ordenada por `f(n)`.
  - Completo e ótimo (com heurística admissível).
  - Complexidade: Pode consumir mais memória do que outros algoritmos, dependendo do tamanho do grafo e da heurística.
    - Tempo: No pior caso O(b^d), onde b é o fator de ramificação (o número médio de sucessores de cada nó) e d é a profundidade da solução.
    - Espaço: O(b^d), devido à necessidade de armazenar todos os nós.

5. Especificação das Heurísticas Utilizadas

As heurísticas são utilizadas no contexto de dois algoritmos de busca para encontrar o caminho mais eficiente em um mapa: *Greedy* e *A**. A heurística empregada nos algoritmos é a **distância Euclidiana** entre o ponto atual e o ponto de destino. A fórmula utilizada é:

$$h(x, y) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```cpp
double euclidean_distance(int x1, int y1, int x2, int y2) {
    return std::sqrt(std::pow(x2 - x1, 2) + std::pow(y2 - y1, 2));
}
```

Onde:

- (x1,y1) representa a posição atual do agente.
- (x2,y2) representa a posição de destino no mapa.

Esta heurística calcula a "distância reta" entre dois pontos no espaço bidimensional.

5.1 Especificação das Heurísticas Utilizadas

Uma heurística é considerada admissível se ela nunca superestima o custo real para alcançar o objetivo, ou seja, se o valor estimado da heurística é sempre menor ou igual ao custo real de alcançar o destino.

No caso da distância Euclidiana, ela é **admissível** porque:

- A distância Euclidiana é uma forma de medir a menor distância possível entre dois pontos em um espaço contínuo e sem obstáculos.
- Em um ambiente onde o agente pode se mover livremente (sem obstáculos que precisem ser evitados), a distância Euclidiana nunca superestima o custo real para alcançar o destino, pois é, de fato, o custo mínimo necessário para viajar de um ponto a outro em linha reta.

No entanto, se houver obstáculos no mapa, como representados por células com o valor `'@'`, a heurística ainda é admissível, mas **não é consistente**, pois a distância Euclidiana pode não refletir com precisão a quantidade de trabalho necessário para percorrer o mapa contornando obstáculos.

Assim, as heurísticas utilizadas no código, ou seja, a distância Euclidiana, são **admissíveis** porque nunca superestimam o custo real para alcançar o objetivo, mas podem não ser **consistentes** em ambientes com obstáculos, o que pode afetar a eficiência de alguns algoritmos (como o Greedy). No entanto, para o algoritmo *A** a heurística é suficientemente boa para garantir a optimalidade do caminho, desde que o mapa não contenha obstáculos.

5.2 Prova para a distância euclidiana

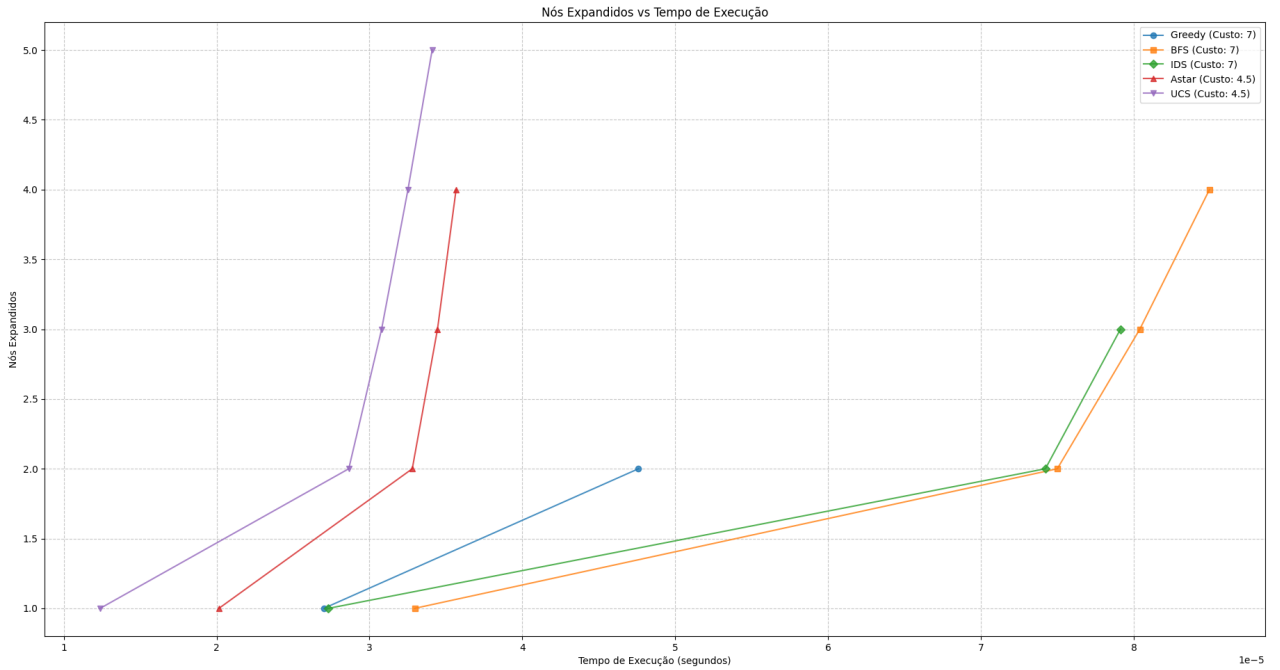
O custo real h*(n) entre n e o destino é pelo menos igual à distância euclidiana, considerando que qualquer movimento válido na matriz segue uma sequência de passos cuja soma das distâncias sempre será maior ou igual à linha reta entre os dois pontos.

- **Se o caminho for direto** (sem obstáculos), h(n)=h*(n), pois a distância euclidiana coincide com o custo real.
- **Se houver obstáculos**, o custo real h*(n) será maior que h(n), pois qualquer desvio para evitar obstáculos adicionará ao custo do caminho.

Portanto, h(n)≤h*(n), para qualquer n. Isso demonstra que a heurística não superestima o custo real, sendo, portanto, **admissível**.

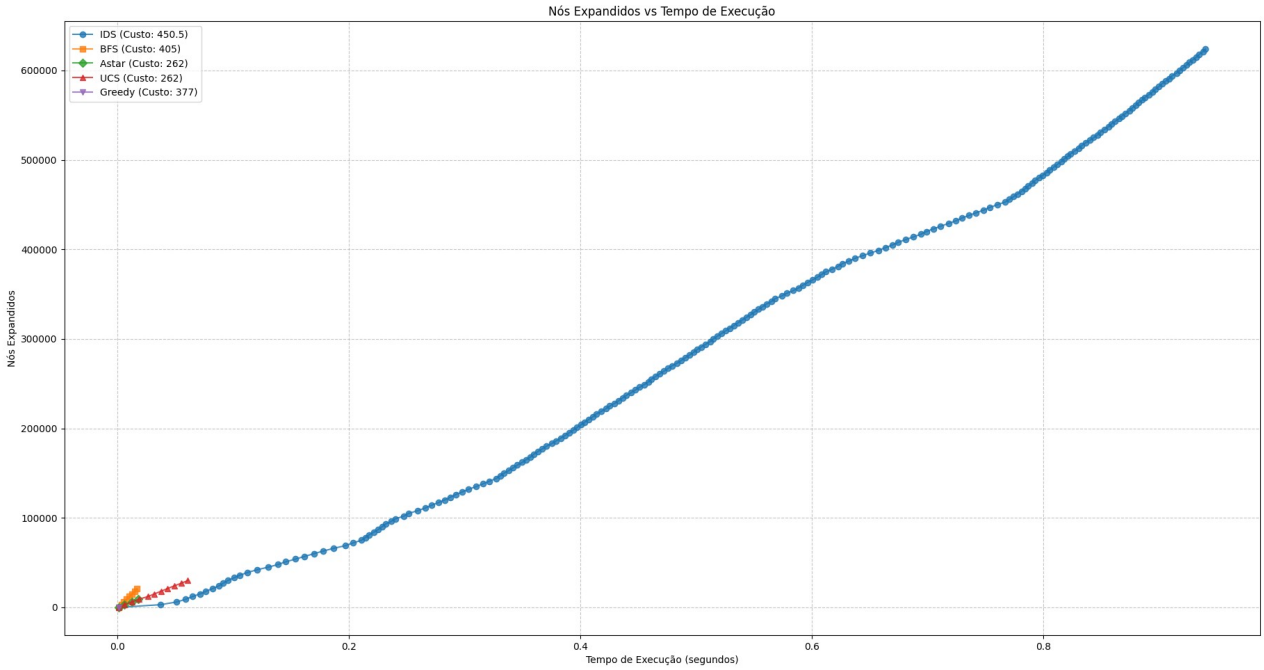
6. Análise quantitativa

6.1 Gráfico para mapa_teste.map posição inicial (1,1) e final (3,1)

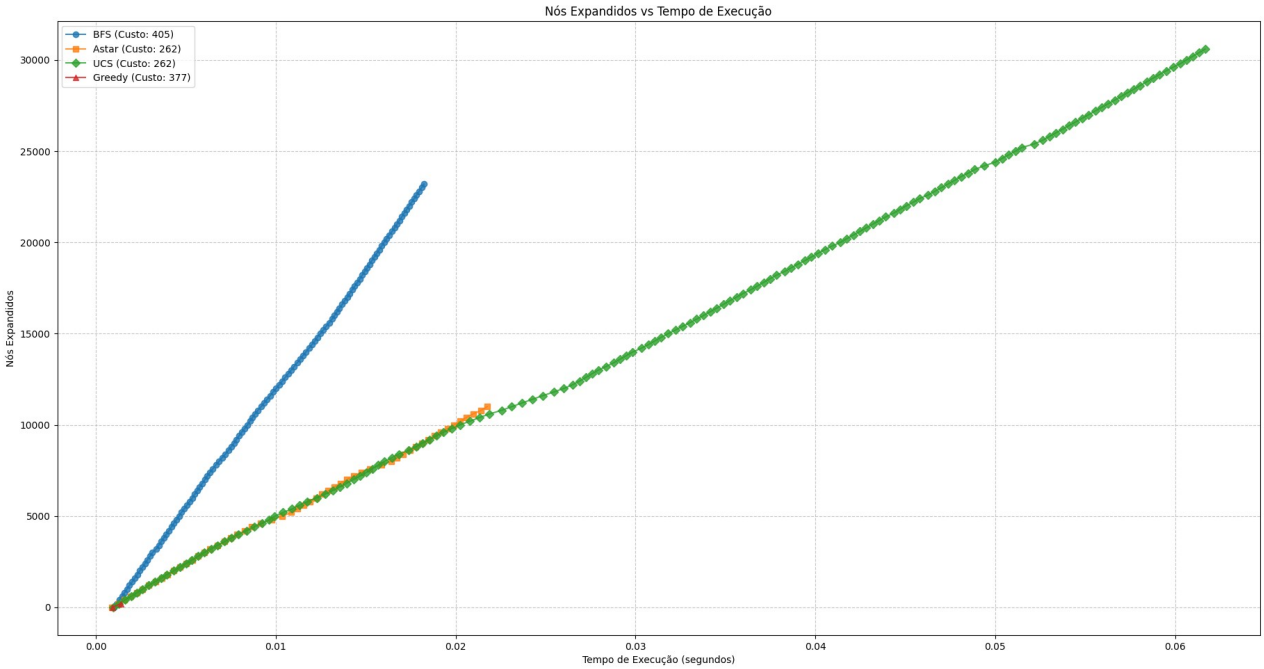


Como esperado, o mapa apresenta custos variados para se deslocar de um ponto a outro, o que tornou os algoritmos BFS/IDS/Greedy não ótimos, encontrando um caminho mais custoso de 7. Já os algoritmos A* e UCS encontraram o caminho ótimo de custo 4,5.
Notamos que o UCS precisou expandir mais nós entre todos (5), enquanto o A* expandiu menos (4) em relação ao UCS devido ao uso da heurística euclidiana, que influenciou a expansão dos nós na direção do objetivo final, levando em consideração a otimalidade em relação ao custo de deslocamento. O Greedy foi o que menos expandiu (2), pois dá preferência em seguir na direção do objetivo, mesmo que não haja obstáculos no caminho. Já o algoritmo BFS foi equivalente ao A* em relação à quantidade de nós expandidos, mas o IDS precisou de menos expansões (4) devido ao fato de que sua exploração estava inicialmente já direcionada para o caminho do objetivo.
Não podemos afirmar muito em relação ao tempo, pois todos os algoritmos expandiram poucos nós, e o computador utilizado pode ter influenciado no tempo desde o início até o fim.

6.2 Gráfico para cidade.map posição inicial (121,16) e final (253,5)



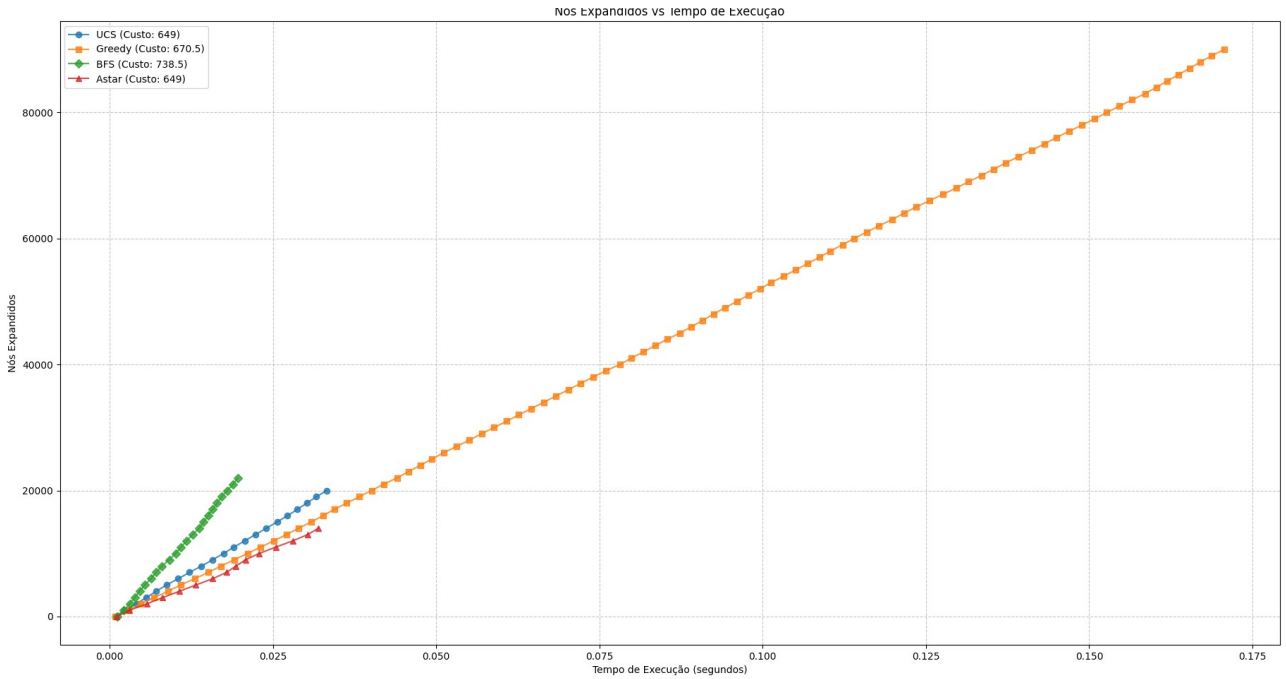
Agora que temos um mapa maior, podemos observar que o IDS terá que expandir muitos mais nós do que os outros, além de demandar mais tempo. Além disso, dependendo da direção que ele der preferência com sua lógica de Busca em Profundidade (DFS), ele pode apresentar um custo maior em comparação aos outros (450,5). Vamos retirar o IDS do gráfico para tornar os outros algoritmos mais visíveis.



Como esperado, o A* começa a se destacar por conseguir atingir o valor ótimo realizando menos buscas ao redor, ao contrário do UCS, que atingiu o mesmo valor, mas precisou expandir mais de 20.000 nós que o A* para encontrar o ótimo, demandando mais tempo. O BFS expande menos nós

que o UCS, mas, em contrapartida, aumenta o custo do trajeto encontrado em mais de 100, tornando-se ineficiente. No entanto, ainda se mantém um pouco mais rápido em comparação ao A*, porém teve que expandir mais nós. O Greedy, talvez por sorte, encontrou um trajeto muito rápido até o objetivo, mas com um custo mais de 100 maior que o ótimo.

6.3 Gráfico para floresta.map posição inicial (244,21) e final (104,2)



Retiramos o IDS para deixar os outros algoritmos visíveis. Neste gráfico, devido à presença de obstáculos que impedem a posição inicial de alcançar a posição final de forma mais retilínea, percebemos que o Greedy começa a realizar expansões extras, já que há nós próximos com distância euclidiana menor em comparação a outros nós que são necessários para chegar ao objetivo. Ele expande desnecessariamente até conseguir contornar o obstáculo e alcançar o trajeto final, demandando muito mais tempo em comparação aos outros.

7. Conclusão

Analisando os algoritmos de busca em relação à otimalidade, complexidade de tempo e espaço, com os gráficos obtidos de diversos mapas, percebemos que eles estão de acordo com suas restrições para serem úteis na escolha para resolver um problema. O algoritmo A* foi o que demonstrou mais eficiência para resolver problemas visando otimalidade com tempo adequado de execução. O UCS consegue atingir a otimalidade, mas quase sempre explorando mais nós desnecessários em comparação ao A*. O algoritmo IDS demonstrou demandar muito tempo e piorar sua solução, dependendo da sorte do caminho tomado na escolha do DFS e do tamanho do mapa que estamos explorando. O algoritmo Greedy demonstrou que, se estivermos usando uma matriz com poucos obstáculos, ele consegue encontrar um caminho mais rápido que todos, sendo ótimo ou pouco menos na maior parte dos casos. O BFS demonstrou ser útil para encontrar caminhos próximos quando o custo não importa, mas se torna ruim quando o custo começa a ser relevante.