Relatório do Projeto de Algoritmos de Busca

Lucas de Oliveira Ferreira
Bruno Matheus Foschianni Ricardo
Arthur Queiroz Moura
Miguel Prates Ferreira de Lima Cantanhede

Novembro 2024

Disciplina: SCC0230 - Inteligência Artificial
Objetivo: Implementação e análise de algoritmos de busca aplicados em grafos, com foco em um problema de busca.

1 Introdução

O nosso projeto visa implementar diferentes algoritmos de busca para análise e experimentação em grafos. São utilizados algoritmos de busca, como Busca em Profundidade (DFS), Busca em Largura (BFS), A* (A Estrela), Dijkstra, Best-First Search e Hill Climbing. Além disso, há a utilização de um grafo de "mundo pequeno", que apresenta alta conectividade e caminhos curtos entre os nós, oferecendo uma rede realista para os testes.

O GITHUB do projeto se encontra em https://github.com/mpferreira003/GraphView, pedimos que leia o README.md para entender melhor sobre como executar as funcionalidades do nosso código.

2 Estrutura e Implementação dos Algoritmos de Busca

Os algoritmos são implementados com base na classe Navigator, que gerencia a navegação entre nós e a visualização do grafo. Abaixo, detalhamos os algoritmos implementados, sendo eles **DFS**, **BFS**, **A***, **Dijkstra**, **Best-First Search** e **Hill Climbing**.

Alguns destes algoritmos utilizam as heurísticas Euclidiana, Manhattan e Chebyshev para estimar distâncias nos grafos.

2.1 Busca em Profundidade (DFS)

Descrição: A Busca em Profundidade (DFS) explora o máximo possível ao longo de um caminho antes de voltar. Esse algoritmo é adequado para explorar

completamente um ramo antes de passar para o próximo.

Implementação: A DFS é implementada de forma iterativa, utilizando uma função auxiliar _dfs e uma pilha (stack) para simular o comportamento da recursão. No método run, o nó inicial e o objetivo são definidos. A _dfs então começa com o nó inicial e, a cada iteração, o algoritmo remove o último nó da pilha, marcando-o como visitado. Os vizinhos são empilhados em ordem contrária para garantir que a busca siga uma ordem de profundidade, explorando cada caminho o mais fundo possível antes de retroceder.

Cenários de Uso: A DFS é eficiente em grafos com estrutura de árvore ou para busca exaustiva.

Limitações: Em grafos muito grandes, pode ser ineficiente e consumir muita memória.

2.2 Busca em Largura (BFS)

Descrição: A Busca em Largura (BFS) explora todos os vizinhos de um nó antes de avançar para o próximo nível, garantindo o menor caminho em grafos não ponderados.

Implementação: A BFS utiliza uma fila para gerenciar os nós a serem visitados. No método run, o nó inicial é inserido na fila, e a cada iteração, o primeiro nó da fila é removido para explorar seus vizinhos. A BFS mantém um conjunto de nós visitados para evitar explorar o mesmo nó mais de uma vez. Se o nó final for encontrado, a busca termina com sucesso.

Cenários de Uso: Ideal para grafos onde é necessário encontrar o menor caminho em termos de número de arestas.

Limitações: Em grafos ponderados, BFS pode não ser eficiente.

2.3 Algoritmo A*

Descrição: O A* combina o custo acumulado até o nó atual com uma heurística que estima o custo restante até o objetivo, priorizando caminhos promissores. A* é ideal para encontrar o menor caminho em grafos ponderados.

Implementação: Na classe `AEstrela`, uma fila de prioridade organiza os nós pelo custo total $f(n) = g(n) + h(n)$, onde $g(n)$ é o custo até o nó atual e $h(n)$ é a heurística. O algoritmo remove o nó com menor custo $f(n)$ da fila e explora seus vizinhos. Cada nó é expandido com base na menor estimativa de custo total até o objetivo, fornecida pela heurística.

Cenários de Uso: Indicado para grafos ponderados onde o objetivo é encontrar o caminho mais curto.

Limitações: O desempenho do A* depende da qualidade da heurística escolhida.

2.4 Algoritmo de Dijkstra

Descrição: O algoritmo de Dijkstra encontra o caminho mais curto em grafos ponderados sem o uso de heurísticas.

Implementação: Dijkstra é implementado como uma variação do A* com heurística zero. A fila de prioridade organiza os nós pelo custo acumulado $g(n)$ até o nó atual, e a busca termina ao atingir o objetivo com o menor custo.

Cenários de Uso: Ideal para grafos ponderados onde a heurística é desconhecida.

Limitações: Em grafos grandes, costuma ser mais lento que A*.

2.5 Busca Best-First

Descrição: A Busca Best-First explora o nó que parece estar mais próximo do destino com base na heurística.

Implementação: Na classe `BestFirstSearch`, os nós são armazenados em uma fila de prioridade com base em uma heurística. A cada passo, o algoritmo escolhe o nó com menor valor heurístico, sem considerar o custo acumulado. Isso permite uma busca rápida, mas não garante o caminho mais curto.

Cenários de Uso: Adequado para buscas rápidas com uma boa estimativa do objetivo.

Limitações: Pode ficar preso em caminhos subótimos, pois não considera o custo acumulado.

2.6 Algoritmo Hill Climbing

Descrição: O Hill Climbing escolhe o vizinho com a melhor heurística, tentando alcançar o objetivo através de melhorias incrementais.

Implementação: A classe `HillClimb` implementa a versão mais básica do algoritmo, onde o próximo nó é escolhido com base na melhor heurística entre os vizinhos. O algoritmo prossegue até encontrar o objetivo ou até que não existam vizinhos melhores.

Cenários de Uso: Útil para otimizações rápidas em problemas simples.

Limitações: Suscetível a ficar preso em máximos locais.

3 Heurísticas

As heurísticas ajudam a otimizar a busca nos algoritmos que utilizam estimativas de distância, aproximando o custo ou a distância até o objetivo e guiando o caminho explorado. Utilizamos três heurísticas principais: Euclidiana, Manhattan e Chebyshev.

3.1 Heurística Euclidiana

A heurística Euclidiana calcula a distância em linha reta entre dois pontos, ideal para cenários onde o movimento é contínuo em um plano. A fórmula é:

$$h_{\text{euclidiana}}(x_1, y_1, x_2, y_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{1}$$

3.2 Heurística Manhattan

A heurística Manhattan mede a distância ao longo dos eixos em uma grade, onde os movimentos são restritos às direções ortogonais (cima, baixo, esquerda, direita). A fórmula é:

$$h_{\text{manhattan}}(x_1, y_1, x_2, y_2) = |x_2 - x_1| + |y_2 - y_1| \tag{2}$$

3.3 Heurística Chebyshev

A heurística Chebyshev calcula a distância considerando movimentos diagonais, útil para cenários que permitem movimentação em todas as 8 direções, como tabuleiros. A fórmula é:

$$h_{\text{chebyshev}}(x_1, y_1, x_2, y_2) = \max(|x_2 - x_1|, |y_2 - y_1|) \tag{3}$$

4 Rede de Mundo Pequeno

Utilizamos uma rede de mundo pequeno como base para os experimentos de busca. Redes de mundo pequeno combinam alta conectividade local com algumas conexões aleatórias, criando atalhos que encurtam distâncias entre regiões distantes. Esse modelo reflete estruturas de redes reais, como redes sociais e de comunicação, onde a maioria das conexões é local, mas existem ligações ocasionais que reduzem a distância entre nós afastados.

4.1 Implementação da Rede de Mundo Pequeno

A rede de mundo pequeno foi gerada usando a classe `MundoPequeno`, que simula redes complexas reais, com conectividade densa e caminhos curtos. O processo de construção ocorre em três etapas:

1. **Geração dos nós**: Cada nó é posicionado aleatoriamente em um plano bidimensional. A posição é controlada por uma semente aleatória para garantir a reprodutibilidade.
2. **Cálculo das distâncias**: A matriz de distâncias entre pares de nós é calculada usando a métrica Euclidiana ou a similaridade cosseno, permitindo ajustar a rede ao tipo de experimento.
3. **Conexões entre os nós**:
   - **Conexões locais**: Cada nó conecta-se aos k vizinhos mais próximos, formando clusters locais de alta conectividade.
   - **Conexões aleatórias**: Para criar atalhos, uma fração p dos nós recebe conexões com nós distantes, selecionados aleatoriamente, mantendo a estrutura de mundo pequeno.

4.2 Parâmetros de Configuração

A classe `MundoPequeno` permite ajustar os seguintes parâmetros para simular diferentes redes:

- n: Número total de nós.
- k: Número de vizinhos locais para cada nó.
- p: Fração de nós com conexões aleatórias.
- `metric`: A métrica de distância utilizada (Euclidiana ou cosseno).

Esses parâmetros foram ajustados conforme os requisitos dos experimentos, criando redes com diferentes distribuições de conexões.

4.3 Aplicação nos Algoritmos de Busca

A rede de mundo pequeno é ideal para testar algoritmos de busca, pois combina complexidade realista com a eficiência dos atalhos. Em nossos experimentos, aplicamos os algoritmos sobre pares de nós aleatórios, analisando a distância média percorrida e o tempo de execução.

Os atalhos beneficiam especialmente algoritmos heurísticos como o A*, que aproveitam estimativas para encontrar caminhos curtos. Em contraste, algoritmos como DFS e BFS não exploram esses atalhos, mostrando-se menos eficientes na rede de mundo pequeno.

5 Análise dos Resultados

Nesta seção, analisamos os resultados obtidos com diferentes algoritmos de busca em redes de mundo pequeno, considerando as configurações de cada rede nas imagens apresentadas.

5.1 Configurações e Parâmetros

Cada imagem representa uma execução dos algoritmos em uma rede de "mundo pequeno" com os seguintes parâmetros:

- **Número de nós** (n): 2000
- **Número de vizinhos próximos** (k): 7
- **Probabilidade de conexão aleatória** (p): Variando entre 0.1, 0.05, e 0.01

A variação no parâmetro p altera a quantidade de atalhos na rede. Quanto maior o valor de p, maior a probabilidade de existirem conexões aleatórias entre nós distantes, o que reduz o diâmetro da rede e, teoricamente, facilita a navegação.

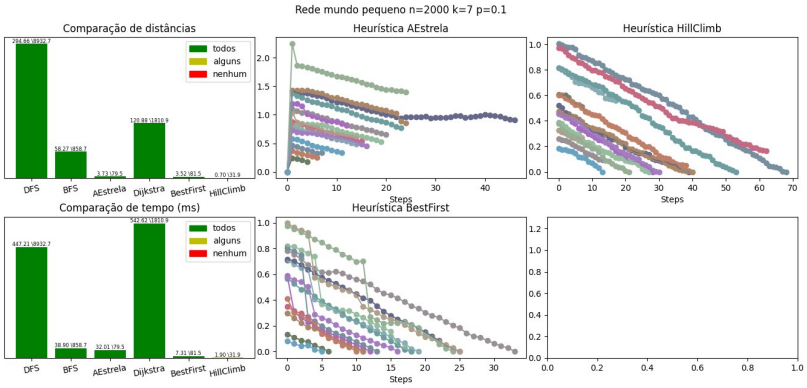

Figure 1: p = 10%

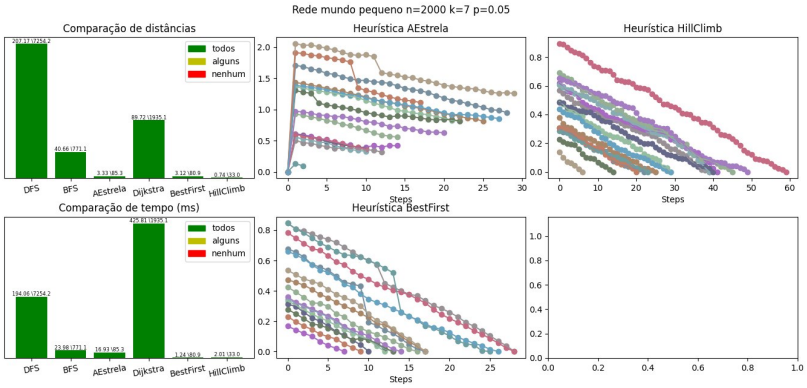

Figure 2: p = 5%

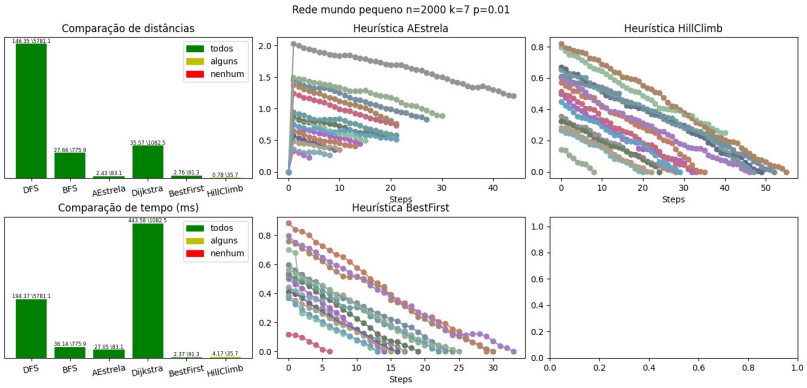

Figure 3: p = 1%

Estes gráficos estão disponíveis ao rodar o arquivo experiments.py. Assim como explicado no README.md do projeto, confira também o arquivo main.py e a geração de gifs.

5.2 Comparação de Desempenho: Distância Percorrida e Tempo de Execução

Os algoritmos de busca heurística (A*, Best-First) no geral demonstram um desempenho significativamente melhor em termos de distância percorrida e tempo de execução em comparação com DFS e BFS. Entre esses, o **A*** destaca-se como o mais eficiente, pois combina custo acumulado e heurística, encontrando caminhos mais curtos com menos tempo de execução. Já o **Dijkstra** é similar ao A*, mas explora mais nós devido à falta de uma heurística, resultando em distância e tempo de execução superior entre os três.

Best-First é rápido, priorizando a proximidade ao objetivo, mas pode se desviar em caminhos subótimos em redes menos conectadas (valores de p mais baixos). **Hill Climbing** exibe o menor tempo de execução, mas tende a se prender em máximos locais, o que limita seu uso em redes com poucos atalhos.

Para **DFS** e **BFS**, observa-se uma maior distância percorrida e um tempo de execução mais alto. Esses algoritmos não-heurísticos exploram extensivamente o grafo sem considerar custos acumulados ou heurísticas, tornando-os menos eficientes em redes grandes e com conexões aleatórias.

5.3 Efeito do Parâmetro p (Probabilidade de Atalhos)

As variações no valor de p afetam o desempenho dos algoritmos, especialmente os de busca heurística:

- **Para $p = 0.1$ (mais atalhos)**: Algoritmos heurísticos como A* e Best-First encontram caminhos mais curtos com menor tempo de execução, aproveitando as conexões aleatórias para navegação rápida.
- **Para $p = 0.05$**: Com menos atalhos, a distância média entre nós aumenta ligeiramente. Ainda assim, os algoritmos heurísticos mantêm vantagem sobre BFS e DFS, embora com um pequeno aumento em distância e tempo.
- **Para $p = 0.01$ (menos atalhos)**: A redução de atalhos impacta todos os algoritmos, especialmente os não-heurísticos (DFS e BFS), que percorrem distâncias maiores. Os algoritmos heurísticos perdem eficiência, mas ainda apresentam melhor desempenho.

5.4 Análise das Heurísticas no A*, Best-First e Hill Climbing

Nos gráficos de passos das heurísticas, observamos o comportamento das heurísticas nos algoritmos A*, Best-First e Hill Climbing:

- **A* com Heurística**: A heurística do A* mantém uma trajetória consistente, minimizando o custo acumulado. Com valores maiores de p, o A* aproveita melhor os atalhos, resultando em uma convergência mais rápida.

- **Best-First com Heurística**: Em redes com muitos atalhos (alto p), Best-First mostra uma descida gradual. Em redes menos conectadas (baixo p), ele perde eficiência e se torna mais propenso a seguir caminhos subótimos.

- **Hill Climbing com Heurística**: Hill Climbing apresenta uma descida linear nos gráficos, priorizando sempre o nó com a melhor heurística. Em redes menos conectadas, ele se prende facilmente em máximos locais, devido à falta de exploração de alternativas.

5.5 Conclusões

- **Eficácia dos Algoritmos Heurísticos**: A* é o algoritmo mais robusto para encontrar o menor caminho de maneira eficiente, com Best-First também apresentando bons resultados. Em redes de mundo pequeno, o A* se beneficia muito dos atalhos.

- **Desempenho de DFS, BFS e Dijkstra**: Esses algoritmos não-heurísticos não são ideais para redes grandes e com conexões aleatórias, como as redes de mundo pequeno. Eles são muito menos eficientes em termos de distância e tempo de execução.

- **Impacto do Parâmetro p**: Conforme p aumenta (mais atalhos), a eficiência dos algoritmos heurísticos aumenta, especialmente A*, enquanto a diferença entre os algoritmos de busca tradicionais (DFS, BFS) e os heurísticos torna-se mais pronunciada. Em redes com poucos atalhos (p baixo), todos os algoritmos enfrentam uma dificuldade maior para encontrar caminhos curtos.

- **Hill Climbing**: Embora rápido e eficiente em distâncias, o Hill Climbing é limitado pela possibilidade de se prender em máximos locais, o que é potencializado em redes de mundo pequeno com poucos atalhos.

Esses resultados sugerem que, em redes de mundo pequeno, algoritmos como A* e Best-First são mais adequados para navegação rápida e eficiente, especialmente em redes com maior conectividade aleatória (maior p), enquanto DFS e BFS são menos indicados para esse tipo de estrutura.