Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação
Inteligência Artificial

Relatório do Jogo dos Garrafões

Lívia Ribeiro Pessamilio - 202165088AC
Gabriel Oliveira Quaresma - 202265178AC

Professor: Saulo Moraes Villela

Inteligência Artifical - DCC014

Juiz de Fora

Julho de 2025

Sumário

1 Introdução

Este trabalho apresenta a aplicação de múltiplos algoritmos à solução do *Jogo dos Garrafões*. A descrição do problema é apresentada na Subseção 1.1 e os objetivos do trabalho, bem como os algoritmos implementados, são detalhados na Subseção 1.2.

1.1 Descrição do Problema

O "Jogo dos Garrafões" (também chamado de "Jogo das Jarras") consiste em um problema cujo estado inicial é um conjunto de jarras, cada uma com sua capacidade máxima e com uma quantidade atual de líquido. As operações permitidas são:

- Esvaziar a jarra
- Encher a jarra
- Transferir líquido de uma jarra para outra

Na operação de transferência, deve-se mover apenas a quantidade de líquido que a jarra de destino acomodar, deixando o restante na jarra de origem. Neste trabalho, por questões de escopo e viabilidade, a transferência foi limitada a jarras adjacentes. A condição de vitória adotada é que todas as jarras possuam o mesmo volume de líquido, valor este especificado antes da execução.

1.2 Objetivos

O objetivo deste trabalho é aplicar os seguintes algoritmos ao problema apresentado:

- Backtracking
- Busca em Largura (BFS)
- Busca em Profundidade Limitada (DFS)
- Busca Ordenada
- Busca Gulosa
- Busca A*
- Busca IDA*

Além disso, mediram-se diversas métricas de desempenho para cada implementação, e, com base nos resultados, realizou-se uma análise comparativa. As métricas obtidas foram:

- Caminho encontrado
- Profundidade da solução
- Custo da solução
- Número total de nós visitados
- Número total de nós expandidos
- Fator médio de ramificação
- Tempo de execução

2 Divisão de Tarefas

A distribuição de responsabilidades entre os integrantes deste trabalho foi organizada da seguinte forma:

- **Gabriel**:
  - Estrutura inicial do projeto (Seção 2)
  - Implementação do algoritmo de Backtracking
  - Implementação dos algoritmos A* e IDA*
  - Refatoração do código e otimização de módulos
  - Elaboração e execução dos testes automatizados
- **Lívia**:
  - Implementação da Busca em Largura (BFS)
  - Implementação da Busca em Profundidade Limitada (DFS)
  - Implementação da Busca Ordenada
  - Implementação da Busca Gulosa (heurística)
  - Redação e formatação do relatório
  - Criação dos slides para a apresentação

3 Estrutura Inicial

Nesta seção, apresentamos a base técnica e organizacional do projeto. Na Subseção 2.1, detalharemos as linguagens de programação e ferramentas utilizadas, bem como o formato de entrada exigido pela aplicação. Em seguida, na Subseção 2.2, descreveremos as principais estruturas de dados e a organização do código, incluindo as classes e funções auxiliares.

3.1 Linguagem, Ferramentas e Entrada

A linguagem escolhida para este trabalho foi C++, devido à familiaridade dos desenvolvedores e ao seu desempenho. Como IDE, utilizou-se o Visual Studio Code e, para controle de versão, o GitHub.

Em relação à entrada de dados, como o problema abordado não exige a leitura de dados externos ou integração com bancos de dados, optou-se por criar conjuntos de jarros dentro da própria `main` do código.

3.2 Estruturas de Dados e Organização do Código

A estrutura de dados do jogo das jarras é composta por duas classes principais: `Jar` e `GameState`. A classe `Jar` representa um jarro individual, com identificador (`id`), quantidade atual de líquido (`current_value`) e capacidade máxima (`max_capacity`). Seu construtor recebe o `id`, a capacidade e um valor inicial (padrão zero). Ela oferece métodos para as operações básicas: `fill()` (encher até a capacidade), `empty()` (esvaziar), `is_full()` e `is_empty()` (testes de estado), `get_capacity()` (retorna a capacidade máxima), `space_left()` (espaço livre) e `transfer(int amount)` (tenta transferir uma quantidade especificada, respeitando limites).

A classe `GameState` modela um estado completo do jogo, contendo um vetor de `Jar` (`jars`), um vetor de inteiros (`values`) com os volumes atuais para rápida comparação e geração de chaves, e metadados para busca informada: `parent` (índice do estado anterior), `visited` (marcação de nó visitado), `closed` (nó completamente explorado), `g_cost` (custo acumulado do caminho), `f_cost` (`g_cost + heuristic()`), `index` (posição no vetor global), `target_Q` (volume-alvo a ser alcançado), e `num_jars` (número de jarros). Entre seus métodos, destacam-se `to_key()` (gera string única, e.g. "3|0|5|"), `is_goal()` (verifica se atingiu o estado de vitória), `heuristic()` (estimação do custo restante), `calculate_action_cost()` (custo de encher, esvaziar ou transferir), `transfer_from_jars()` e `get_transfer_value_from_jars()` (execução e cálculo de transferência), além de `print()` e `print_path()` para exibição de estados e reconstrução de caminhos.

3.3 Heurísticas

Os algoritmos Guloso, A* e IDA* necessitam de uma heurística para guiar suas escolhas. Para o nosso problema, a heurística utilizada considera dois fatores: o total de líquido que precisa ser ajustado para atingir a soma ideal entre todos os jarros e a maior diferença individual entre um jarro e o valor alvo.

Por exemplo, considere o estado $(3, 7, 2)$ com valor objetivo 5 para cada jarro e capacidade máxima de transferência igual a 4. A soma atual dos jarros é 12, e a soma alvo seria $5 \times 3 = 15$. A diferença total é 3, e o jarro mais distante do valor desejado é o de valor 2 (distância 3 de 5). Portanto, para ajustar a soma total em no máximo 4 unidades por operação, seriam necessários $\lceil 3/4 \rceil = 1$ passo. Para o pior jarro individual, também seriam necessários $\lceil 3/4 \rceil = 1$ passo. A heurística então retorna o maior desses valores, neste caso, 1.

Essa heurística foi construída com base em um relaxamento das regras do problema. Ela assume que é possível distribuir o líquido de forma ideal em cada passo, utilizando a capacidade máxima possível de transferência (`max_cap`) para ajustar tanto a soma total quanto o valor de cada jarro individualmente. Isso permite estimar, com segurança, o número mínimo de passos necessários para alcançar o objetivo.

Dessa forma, a heurística é admissível, pois ela **nunca superestima o custo real**: no mínimo essa quantidade de líquido precisa ser movida, mesmo que não se consiga fazer isso de forma perfeita em cada passo. Assim, ela mantém a correção e a optimalidade dos algoritmos de busca informada como o A* e o IDA*.

Contudo, embora essa heurística ofereça uma estimativa mais refinada em relação à versão simples baseada apenas na soma das diferenças individuais, ela também pode apresentar limitações de desempenho dependendo da estrutura do espaço de estados, principalmente se muitos estados tiverem valores de soma já próximos do alvo mas com uma má distribuição interna entre os jarros.

Assim, uma nova heurística foi formatada, levando esses fatores em conta. A heurística estima quantos movimentos são necessários para atingir a configuração alvo considerando dois critérios: (1) o desvio total de água, calculado pela diferença entre a soma atual de todos os jarros e o volume desejado ($Q \times n$), dividido pela maior capacidade de um jarro e arredondado para cima; e (2) o desvio máximo de um único jarro em relação a Q, igualmente dividido e arredondado. Retornamos o maior desses dois valores, garantindo uma estimativa admissível que reflete tanto o esforço coletivo quanto o esforço mínimo para corrigir o jarro mais fora do alvo.

4 Algoritmos de Busca

Nesta seção, apresentamos os métodos de busca implementados para resolver o problema. Nas Subseções 3.1 a 3.7, descreveremos cada algoritmo (Backtracking, Busca em Largura (BFS), Busca em Profundidade Limitada (DFS), Busca Ordenada, Busca Gulosa, Busca A* e Busca IDA*) detalhando sua lógica e referenciando os trechos de código relevantes. Em seguida, na Subseção 3.8, abordaremos as heurísticas utilizadas, justificando sua admissibilidade e eficácia.

4.1 Backtracking

A implementação de backtracking começa com a rotina `solve_with_backtracking`, que inicializa o vetor de estados com a configuração inicial dos jarros e índices auxiliares para ações e jarros. Em cada iteração do laço principal, verifica-se se o estado corrente é objetivo (`is_goal()`); em caso afirmativo, reconstrói-se o caminho até a solução usando os ponteiros `parent` e encerra-se a busca. Caso contrário, se o estado ainda não estiver "fechado", chama-se `generate_one_child` para tentar gerar um único filho aplicando cada ação possível (esvaziar, encher, transferir à esquerda ou à direita) ao jarro indicado. Para cada ação válida, cria-se um novo `GameState`, atualizam-se seus valores, atribuem-se `parent` e `index`, e adiciona-se ao vetor de estados.

Antes de aceitar um filho, a função `checkParentLoopBck` percorre recursivamente os ancestrais pelo campo `parent`, comparando as chaves geradas por `to_key()` para detectar e rejeitar ciclos. Quando todas as ações de todos os jarros de um estado são exauridas, marca-se `closed = true` e retrocede-se ao estado pai, reiniciando o processo até encontrar a meta ou esgotar o espaço de estados. Dessa forma, a busca em profundidade é realizada de maneira sistemática, evitando repetições e garantindo a completude da estratégia dentro do espaço explorável.

A tabela 1 a seguir mostra as métricas capturadas com diferentes quantidades de jarros:

Tabela 1: Métricas de desempenho para o algoritmo Backtracking

Métrica	**3 garrafões**	**4 garrafões**	**5 garrafões**
Profundidade da solução	46	414	1343
Custo da solução	144	1074	3694
Número de nós visitados	650	1637	1852
Número de nós expandidos	651	1638	1853
Fator médio de ramificação	1.280	1.468	1.125
Tempo de execução (ms)	4.574	670.445	1636.24

4.2 Busca em Largura (BFS)

A busca em largura inicia com a criação de um estado inicial (`GameState`) baseado nos jarros fornecidos. Este estado é adicionado ao vetor de estados, marcado como visitado por meio de sua chave (`to_key()`) em um conjunto, e inserido em uma fila que representa os estados abertos a serem explorados. Um temporizador é iniciado para medir o tempo total de execução, e uma estrutura auxiliar (`MedidasBusca`) acompanha estatísticas como número de nós visitados, expandidos e profundidade máxima. O laço principal da busca prossegue enquanto houver estados na fila.

A cada iteração, o estado no início da fila é removido, impresso e contabilizado como visitado. Se este estado satisfaz o objetivo, o algoritmo imprime os dados do caminho, tempo de execução e encerra. Caso contrário, tenta-se gerar até quatro filhos a partir de cada jarro usando as ações possíveis: encher (`FILL`), esvaziar (`EMPTY`), transferir para a esquerda (`TRANSFER_LEFT`) e para a direita (`TRANSFER_RIGHT`). Cada filho gerado passa por uma verificação de unicidade usando o conjunto de estados visitados, evitando repetição. Se for inédito, é adicionado à fila de abertos e ao vetor de estados, incrementando o contador de nós expandidos.

A tabela 2 a seguir mostra as métricas capturadas com diferentes quantidades de jarros:

Tabela 2: Métricas de desempenho para o algoritmo BFS

Métrica	**3 garrafões**	**4 garrafões**	**5 garrafões**
Profundidade da solução	6	16	11
Custo da solução	12	8	22
Número de nós visitados	43	323	2007
Número de nós expandidos	251	2756	23447
Fator médio de ramificação	4.302	7.152	10.336
Tempo de execução (ms)	0.444	5.254	53.225

4.3 Busca em Profundidade Limitada (DFS)

A busca em profundidade começa construindo um estado inicial a partir dos jarros fornecidos e mantendo um vetor de estados explorados e um conjunto de chaves já visitadas. A função auxiliar `busca_profundidade_aux` marca o estado atual como visitado, incrementa o contador de nós visitados e exibe o estado.

Se o estado satisfaz a condição de meta (`is_goal()`), sinaliza sucesso, registra a profundidade atingida e o custo acumulado, e interrompe a recursão (essa interrupção também ocorre caso chegue na profundidade limite). Caso contrário, aumenta o nível de profundidade e, para cada jarro, tenta gerar filhos usando as quatro ações possíveis (`FILL`,

EMPTY, TRANSFER_LEFT, TRANSFER_RIGHT) por meio de geraFilhoPL. Esta função copia o estado atual, verifica a validade da ação (por exemplo, não encher um jarro já cheio), calcula o custo da ação, atualiza g_cost e f_cost, atribui índices e adiciona o novo estado ao vetor de estados.

Cada filho válido e ainda não visitado aciona chamada recursiva, contando-o como nó expandido. Após percorrer todas as ações de todos os jarros ou encontrar a meta, a função decrementa a profundidade e retorna, retrocedendo pelos ramos ainda não explorados até exaurir o limite de profundidade ou resolver o problema. No final, a função principal calcula o tempo de execução e imprime estatísticas de nós visitados, expandidos, profundidade máxima e custo do caminho.

A tabela 3 a seguir mostra as métricas capturadas com diferentes quantidades de jarros:

Tabela 3: Métricas de desempenho para o algoritmo DFS

Métrica	**3 garrafões**	**4 garrafões**	**5 garrafões**
Profundidade da solução	10	10	10
Custo da solução	22	22	22
Número de nós visitados	3876	694749	69500000
Número de nós expandidos	12198	3710706	371000000
Fator médio de ramificação	3.147	5.342	5.338
Tempo de execução (ms)	60.956	18455.1	1.850.000

4.4 Busca Ordenada

A busca ordenada começa com a criação de um estado inicial baseado na configuração dos jarros, sendo esse estado inserido em um vetor de estados, marcado como visitado e adicionado à estrutura de dados que representa os nós abertos (deque). A função de geração de filhos leva em conta o custo acumulado do caminho (g_cost), e cada novo estado tem também seu custo total estimado (f_cost) atualizado com base em uma heurística. O tempo de execução é monitorado desde o início da busca e os dados estatísticos são registrados em uma estrutura auxiliar.

Durante a execução, o algoritmo remove o estado de menor custo acumulado do início da fila, imprime o estado atual e verifica se é o objetivo. Se for, imprime o caminho, o custo e o tempo total. Caso contrário, gera todos os filhos possíveis a partir do estado atual — enchendo, esvaziando ou transferindo entre jarros — e armazena apenas os válidos. Esses filhos são então ordenados pelo custo do caminho acumulado (função comparaPorCusto) e inseridos de forma que o de menor custo permaneça na frente da fila. Isso é feito para que a busca pegue sempre o filho do nó atual que possua menor custo.

A tabela 4 a seguir mostra as métricas capturadas com diferentes quantidades de jarros:

Tabela 4: Métricas de desempenho para o algoritmo Busca Ordenada

Métrica	**3 garrafões**	**4 garrafões**	**5 garrafões**
Profundidade da solução	25	148	811
Custo da solução	51	292	1668
Número de nós visitados	55	179	1660
Número de nós expandidos	366	1642	22158
Fator médio de ramificação	6.636	9.168	13.348
Tempo de execução (ms)	0.672	3.581	46.162

4.5 Busca Gulosa

A função `busca_gulosa` começa com o estado inicial dos jarros, custo acumulado zerado e o insere na fila de abertos (`std::deque`). A cada passo, retira da frente o estado com menor valor heurístico (obtido por `heuristic()`), marca-o como visitado e testa se é meta. Caso contrário, gera sucessores aplicando as ações FILL, EMPTY, TRANSFER_LEFT e TRANSFER_RIGHT a cada jarro, atualiza `g_cost` e recalcula `f_cost` = `g_cost` + heurística, e só adiciona estados nunca antes vistos (verificado em `jaVisitados`) ao início do deque.

Após a inserção de filhos, o deque é reordenado, para que a heurística seja sempre a melhor no início. Assim, a cada iteração, a busca explora sempre o sucessor mais promissor segundo a heurística, sem considerar o custo já gasto além de impedir revisitas. O processo repete-se até encontrar o o objetivo ou esgotar as opções, exibindo mensagem de fracasso se não houver solução.

Tabela 5: Métricas de desempenho para o algoritmo Busca Gulosa

Métrica	**3 garrafões**	**4 garrafões**	**5 garrafões**
Profundidade da solução	14	57	514
Custo da solução	38	168	1379
Número de nós visitados	16	58	552
Número de nós expandidos	92	460	6763
Fator médio de ramificação	5.688	7.914	12.250
Tempo de execução (ms)	0.251	1.345	17.78

4.6 Busca A*

A busca A* (A-estrela) é uma estratégia que combina o custo real do caminho percorrido até o estado atual (`g_cost`) com uma estimativa do custo restante até o objetivo (`heuristic`), formando o custo total estimado `f_cost = g + h`. A implementação começa criando o estado inicial, cujo custo g é zero e h é calculado com base em uma heurística apropriada para o problema das jarras. Esse estado é armazenado em um vetor global e inserido em uma fila de prioridade (`priority_queue`) que organiza os estados com base no menor `f_cost`, garantindo que os mais promissores sejam explorados primeiro. O algoritmo mantém também um mapa para rastrear os estados visitados, permitindo detectar repetições e otimizar o percurso.

Durante a execução, o estado com menor `f_cost` é retirado da fila e verificado quanto à condição de objetivo. Se atingir o estado desejado, o caminho é reconstruído e impresso. Caso contrário, o algoritmo gera filhos por meio das ações possíveis: encher, esvaziar ou transferir entre jarros adjacentes. Para cada filho válido, calcula-se o novo `g_cost`, a heurística e, consequentemente, o novo `f_cost`. Se o filho ainda não foi visitado ou se um caminho melhor até ele foi encontrado, ele é atualizado ou adicionado à fila de prioridade. A estratégia A* é completa e ótima quando a heurística utilizada é admissível, o que a torna particularmente eficaz para problemas como o das jarras, onde é necessário minimizar o número de operações ou o custo total para atingir um volume alvo.

A tabela 6 a seguir mostra as métricas capturadas com diferentes quantidades de jarros:

Tabela 6: Métricas de desempenho para o algoritmo A*

Métrica	**3 garrafões**	**4 garrafões**	**5 garrafões**
Profundidade da solução	6	8	11
Custo da solução	12	16	22
Número de nós visitados	30	152	858
Número de nós expandidos	42	254	1361
Fator médio de ramificação	1.783	2.108	2.064
Tempo de execução (ms)	0.339	2.979	18.256

4.7 Busca IDA*

A busca IDA* (Iterative Deepening A*) combina a profundidade limitada com as vantagens da heurística de A*, realizando sucessivas profundidades máximas (limites de custo `f_cost`) até encontrar a solução. Inicialmente, define-se um limiar (`threshold`) igual ao valor heurístico do estado inicial, e prepara-se a estrutura de estados, marcando o estado raiz com custo g zero e f igual à heurística. Em cada iteração, reinicializa-se esse vetor

apenas com o estado inicial e zera-se o conjunto de visitas, mantendo também um conjunto `discarded` para acumular os valores de `f_cost` de nós que forem "podados" por excederem o limiar.

Dentro de cada iteração, a função principal faz um backtracking semelhante ao DFS: para o estado corrente, imprime-se valores de g, h e f, e se f exceder o limiar, marca-se como visitado e insere-se f em `discarded`, efetivamente podando aquele ramo. Se o estado for meta e estiver dentro do limiar, reconstrói-se o caminho e encerra-se tudo. Caso contrário, para cada combinação de jarro e ação (esvaziar, encher, transferir esquerda ou direita), chama-se `generate_one_child`, que cria um filho válido, atualiza g e f, e rejeita ciclos ancestrais via `checkParentLoopIDA`. Filhos novos ou melhores caminhos a estados já vistos são explorados recursivamente, enquanto o algoritmo retrocede pela árvore (marcando estados "fechados" e retornando ao pai) quando esgota ações ou atinge nós além do limiar.

Ao final de cada iteração em que não se encontrou solução, o limiar é atualizado para o menor valor de `f_cost` registrado em `discarded`, garantindo que a próxima profundidade de busca A* iterativa consiga avançar pelo ramo mais promissor que foi podado anteriormente. Se, em algum momento, não houver valores em `discarded` maiores que o limiar antigo, conclui-se que não há solução e o processo termina.

A tabela 7 a seguir mostra as métricas capturadas com diferentes quantidades de jarros:

Tabela 7: Métricas de desempenho para o algoritmo IDA*

Métrica	**3 garrafões**	**4 garrafões**	**5**
Profundidade da solução	6	8	11
Custo da solução	12	16	22
Número de nós visitados	40	219	1255
Número de nós expandidos	41	220	1256
Fator médio de ramificação	1.538	1.973	2.156
Tempo de execução (ms)	3.153	34.167	370.152

5 Comparação de Resultados

Nesta seção, apresentamos a metodologia e os resultados obtidos na avaliação dos algoritmos implementados. Na Subseção 4.1, descrevemos a configuração dos testes, incluindo o procedimento de geração de instâncias do Jogo dos Garrafões e os parâmetros utilizados (número de iterações e quantidade de jarras). Em seguida, na Subseção 4.2, comparamos o desempenho dos métodos por meio de gráficos e tabelas, destacando as diferenças nas métricas coletadas e discutindo as vantagens e limitações de cada abordagem.

5.1 Configuração dos Testes

Para os testes de desempenho, foram montados três cenários distintos, contendo respectivamente 3, 4 e 5 jarros. Em cada um desses cenários, coletaram-se as métricas pertinentes (como tempo de execução e custo da solução) a partir da aplicação de cada algoritmo aos mesmos conjuntos.

5.2 Comparação de Desempenho entre Algoritmos

Para melhor visualização dos dados mais importantes, os gráficos 1 e 2 apresentados a seguir mostram, respectivamente, os dados de tempo de execução e os custos registrados nas tabelas.

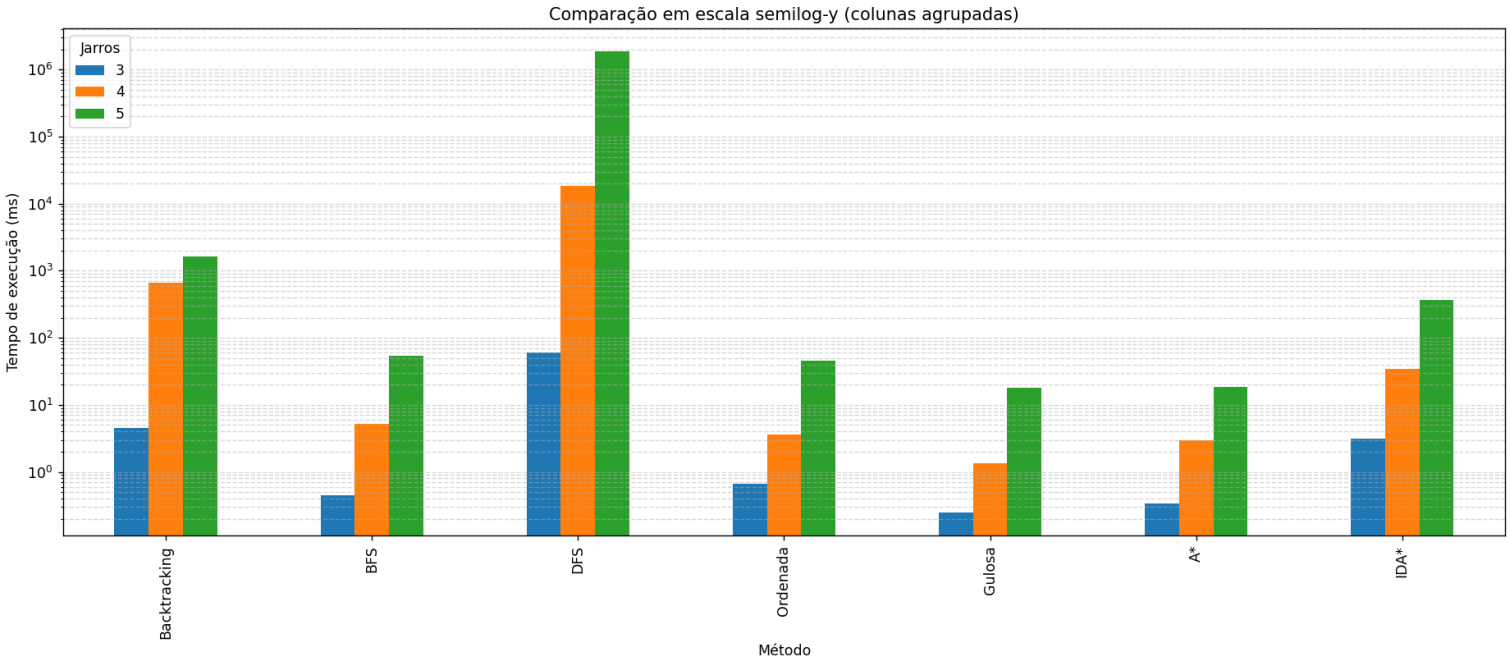

Figura 1: Gráfico do Tempo de Execução (ms)

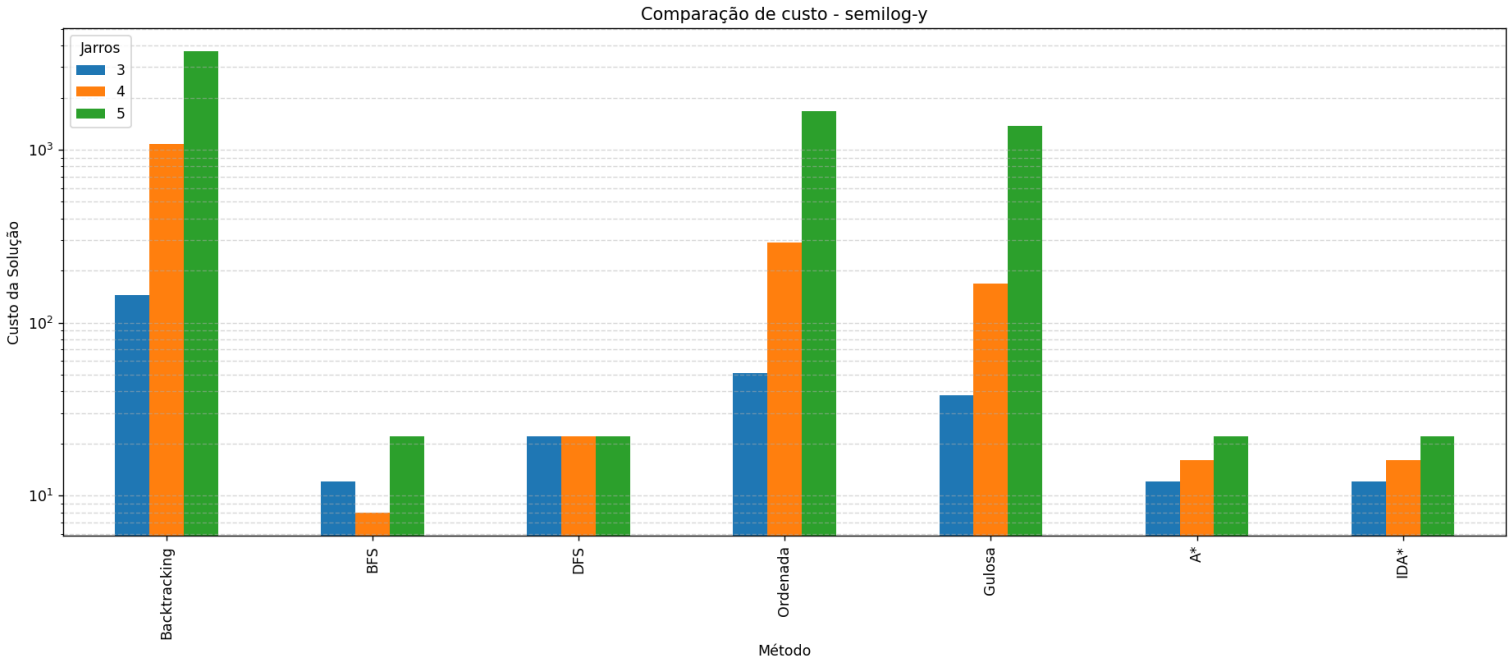

Figura 2: Gráfico do Custo da Solução

A partir dos resultados observados nos gráficos de tempo de execução e de custo da solução, fica evidente que existe um compromisso a ser tomado entre a qualidade do caminho encontrado e a escalabilidade do algoritmo. Os métodos de busca não informada pura, como o Backtracking e profundidade(DFS), mostram bom custo, mas tornam-se rapidamente impraticáveis à medida que o número de jarros cresce. Em cinco jarros, por exemplo, o DFS chega a demandar quase dois milhões de milissegundos, enquanto o Backtracking ultrapassa a casa dos milhares, além de produzir soluções com custo muito elevado em relação ao ótimo.

A razão especulada para o tempo elevado de execução da busca DFS, mesmo limitada à profundidade 10, se encontra na estrutura da função. Nesse trabalho, a sua implementação ocorreu de forma recursiva, com uma função principal chamando a DFS auxiliar durante toda a execução. Essa escolha de arquitetura pode ser uma das principais causas da lentidão desse algoritmo em particular.

Já a Busca em Largura (BFS), ainda que não partilhando do mesmo nível de problemas do DFS, seu tempo de execução cresce mais para 5 jarras do que a Gulosa ou A, por exemplo. A Busca Ordenada (Uniform Cost) reduz um pouco esse esforço de expansão, mas, sem informação heurística, ainda é incapaz de entregar soluções com custo competitivo quando comparada às abordagens informadas.

Nos métodos informados, podemos observar que a Busca Gulosa apresenta desempenho impressionante em tempo — sendo o mais rápido dentre todos — porém, por descartar informações do custo já percorrido, encontra caminhos de qualidade muito in-

ferior ao mínimo. Por sua vez, o A demonstra ser o melhor compromisso, pois, graças à heurística admissível, consegue manter a optimalidade do custo (sempre igual ao menor possível) e escalonar muito melhor do que as buscas não informadas, apresentando tempos moderados mesmo em cinco jarros. O IDA, embora também ótimo em custo, sofre com a sobrecarga das iterações sucessivas de aprofundamento, crescendo em tempo de maneira menos favorável em problemas maiores.

6 Conclusão

Em síntese, para o "Jogo dos Garrafões", o algoritmo A se destaca como a melhor escolha prática: ele une garantia de solução ótima a um desempenho eficiente, se beneficiando da heurística para controlar a expansão de estados e mantendo tanto o tempo de execução quanto o custo da solução em níveis aceitáveis, mesmo à medida que a complexidade do problema aumenta.