Fundamentos de IA - Jogo do 15

Eduardo Fernandes, Julia Machado, Igor Falco

April 2025

1 Modelagem do Problema

O problema do Jogo do 15 foi modelado como um grafo, onde cada vértice representa uma configuração válida do tabuleiro e as arestas representam movimentos possíveis do espaço vazio (valor 0). Cada movimento (para cima, baixo, esquerda ou direita) gera um novo estado do tabuleiro, conectado ao estado anterior.

A classe `Estado` representa um vértice do grafo. O estado é definido como uma tupla imutável de 16 inteiros, representando a configuração atual do tabuleiro. As operações de comparação e hashing são definidas para que os estados possam ser utilizados em conjuntos e dicionários, o que é fundamental para o controle de estados visitados nos algoritmos de busca.

O método `get_neighbors` gera os estados vizinhos válidos a partir de um estado atual, respeitando as regras de movimentação do quebra-cabeça (por exemplo, não mover para a esquerda se o espaço vazio estiver na primeira coluna).

2 Tarefa 1 - Verificação de Solvabilidade

Para verificar se um tabuleiro aleatório é solucionável, implementamos a função `verificaSolvabilidade`. A lógica baseia-se no número de inversões presentes na configuração e na linha onde se encontra o espaço vazio.

- Uma inversão ocorre quando um número maior aparece antes de um número menor na configuração linear do tabuleiro.
- O número total de inversões e a posição (linha) do zero determinam se o estado é solucionável.

A função percorre todos os pares de peças e conta as inversões. Em seguida, verifica a linha do zero (posição do espaço vazio) e aplica a regra conhecida para determinar a solvabilidade:

> Se o número de inversões for par e o zero estiver em uma linha ímpar (contando de cima para baixo começando em zero), ou se o número de inversões for ímpar e o zero estiver em uma linha par, então o tabuleiro é solucionável.

3 Tarefa 2 - Geração de Estado Inicial

Para gerar estados iniciais aleatórios, duas abordagens foram implementadas:

1. `gerarTabuleiro()`: embaralha aleatoriamente os números de 0 a 15. Como o resultado pode não ser solucionável, utilizamos a função da Tarefa 1 para verificar a validade da configuração.

2. `gerar_estado_por_movimentos(n)`: inicia a partir da configuração objetivo e aplica n movimentos válidos aleatórios, garantindo que todos os estados gerados sejam alcançáveis e, portanto, solucionáveis.

Essa segunda abordagem é preferível, pois garante a geração de configurações válidas sem a necessidade de verificações adicionais de solvabilidade.

4 Tarefa 3 - Agentes de Busca

Foram implementados dois algoritmos clássicos de busca para resolver o Jogo do 15: *Busca em Largura* (BFS) e *Busca em Profundidade* (DFS). Ambos utilizam a estrutura do grafo descrita anteriormente para explorar os estados possíveis a partir de uma configuração inicial.

4.1 Busca em Largura (BFS)

A `bfs` é uma estratégia de busca que utiliza uma fila (FIFO) para explorar os estados em largura. Ela explora todos os nós de uma mesma profundidade antes de avançar para o próximo nível, e garante encontrar o caminho mais curto em termos de número de movimentos. Cada estado visitado tem seus vizinhos adicionados à fila, desde que ainda não tenham sido visitados. Um dicionário de predecessores é utilizado para reconstruir o caminho da solução ao alcançar o estado objetivo.

Uma versão otimizada, `bfs_otimizada`, foi desenvolvida para evitar o uso redundante de estruturas e acelerar a verificação de presença na fila.

4.2 Busca em Profundidade (DFS)

Já a `dfs` explora o caminho mais profundo primeiro, recuando apenas quando não há mais possibilidades. Pode ser mais eficiente em memória, mas não garante a solução ótima. A pilha (LIFO) é usada para simular a recursão. Tal como na BFS, predecessores são armazenados para posterior reconstrução da solução. Um controle de profundidade impede que a busca entre em ciclos infinitos ou consuma memória excessiva.

Ambos os métodos registram o número de nós expandidos e o tempo de execução, importantes para a análise comparativa.

5 Tarefa 4 - Busca A* e Heurística

A busca A* foi implementada na função `a_estrela`. Ela é uma estratégia de busca informada que utiliza uma heurística para estimar o custo até o objetivo. Assim, o algoritmo utiliza uma fila de prioridade (heap) para expandir os nós com menor custo estimado total $f(n) = g(n) + h(n)$, onde:

- $g(n)$ é o custo acumulado do caminho desde o estado inicial até o estado atual (número de movimentos);
- $h(n)$ é a heurística de custo estimado até o objetivo.

5.1 Heurística de Distância de Manhattan

Foi adotada como função heurística $h(n)$ a **distância de Manhattan**, definida como a soma das distâncias absolutas entre as posições atuais e as posições objetivo de cada peça (excluindo o 0). Essa heurística é admissível e consistente para o problema, garantindo a optimalidade da A*.

6 Tarefa 5 - Comparação dos Métodos

Para comparar os algoritmos, implementamos a função `avaliar_algoritmo`, que executa múltiplos testes com diferentes configurações iniciais geradas por `gerar_estado_por_movimentos`. Para cada execução, são computados:

- Número de movimentos necessários até a solução;
- Número total de nós expandidos;
- Tempo de execução.

Os experimentos foram realizados variando a dificuldade dos estados iniciais, controlada pela quantidade de movimentos aleatórios aplicados a partir da configuração objetivo. Para o BFS e DFS, os testes foram limitados a estados com até 50 movimentos de distância, devido a restrições de tempo e memória. Para o A*, foram realizados testes também com distâncias maiores. Os resultados são apresentados a seguir.

6.1 Resumo da Análise Comparativa

6.1.1 Movimentos na Solução

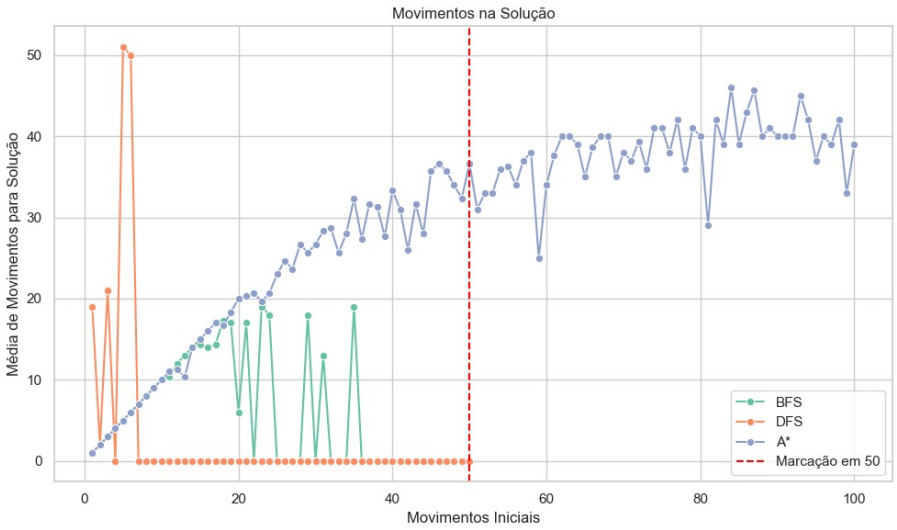

Figure 1: Média de movimentos para resolver o problema conforme a dificuldade inicial aumenta.

O gráfico da Figura 1 mostra que o algoritmo A* mantém um crescimento relativamente suave, lidando bem com maiores dificuldades. Já o BFS e o DFS, limitados a 50 movimentos, apresentam maior instabilidade, com o DFS exibindo várias falhas (média zero), indicando que, em muitos casos, o algoritmo não conseguiu encontrar uma solução.

6.1.2 Nós Expandidos

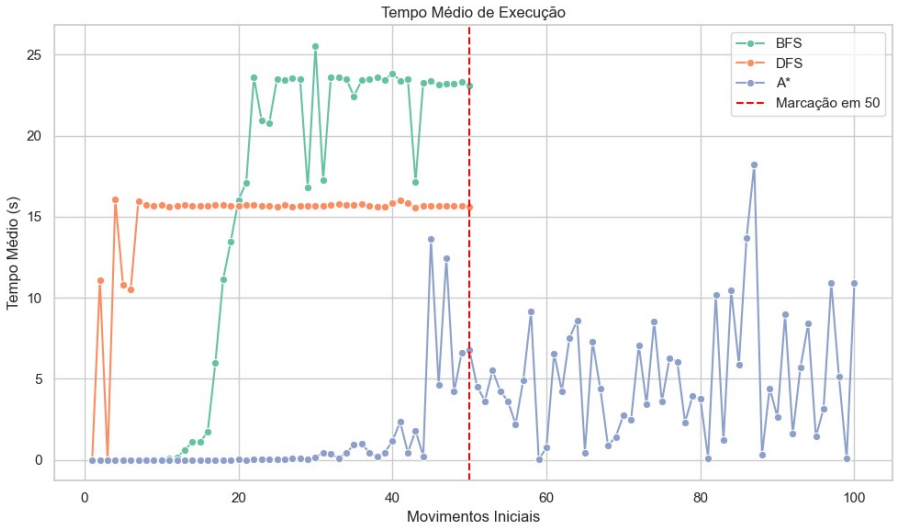

Figure 2: Número médio de nós expandidos pelos algoritmos.

No gráfico da Figura 2, o A* novamente se destaca, expandindo muito menos nós do que os outros métodos, mesmo para estados iniciais mais difíceis. BFS e DFS atingem rapidamente o limite de 2 milhões de nós expandidos, tornando-se inviáveis para instâncias mais complexas. A linha vermelha no gráfico, em 50 movimentos, marca o ponto onde as análises de BFS e DFS foram interrompidas.

6.1.3 Tempo Médio de Execução

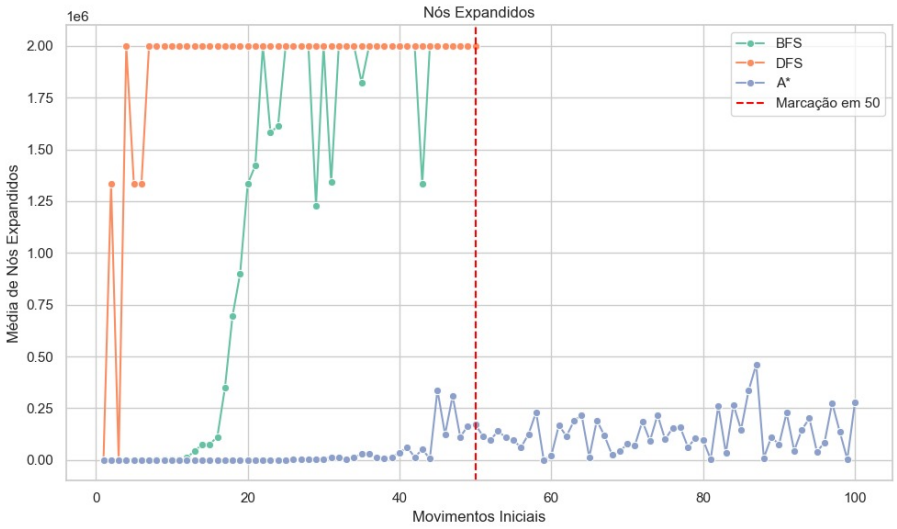

Figure 3: Tempo médio de execução dos algoritmos.

O gráfico da Figura 3 confirma a superioridade do A*. Seu tempo permanece baixo mesmo em instâncias mais complexas, enquanto BFS e DFS rapidamente alcançam o teto de tempo imposto para os testes. Esse comportamento reforça a necessidade de limitar os experimentos com BFS e DFS a problemas menos difíceis.

6.2 Discussão dos Resultados

Com base nos resultados apresentados, podemos destacar:

- A **Busca A*** com heurística de distância de Manhattan foi a abordagem mais eficiente, apresentando o menor número de nós expandidos e menor tempo de execução, mesmo em problemas mais difíceis.
- A **Busca em Largura** (BFS) garantiu encontrar as soluções ótimas, porém com grande custo computacional (tempo e número de nós).
- A **Busca em Profundidade** (DFS) foi mais rápida em alguns casos iniciais, mas, devido à possibilidade de entrar em ciclos e à falta de garantia de otimalidade, apresentou muitas falhas conforme a dificuldade aumentava.

Esses resultados reforçam a importância do uso de heurísticas bem definidas e métodos de busca informada em problemas de grande espaço de estados como o Jogo do 15.

References

[1] POOLE, D. L.; MACKWORTH, A. K. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010.

[2] RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2009.