UNIVERSIDADE FEDERAL DE SANTA CATARINA
DISCIPLINA DE INTELIGÊNCIA ARTIFICIAL

# Análise Comparativa de Algoritmos de Busca no Problema do PacMan

Professor = Jomi F. Hübner
Aluno = Alexsandro Gehlen,
Yeremie Pando

Florianópolis – SC
Abril, 2025

# 1. Introdução

Este trabalho tem como objetivo implementar e comparar diferentes algoritmos de busca aplicados a um cenário inspirado no jogo PacMan. O agente atua em uma grade modelada como uma matriz em todos os cenários, partindo de uma posição inicial fixa (0,0) e devendo coletar comidas posicionadas em diferentes pontos do mapa, movimentando-se com custo 1 por espaços livres e evitando obstáculos representados por paredes. Foram utilizados três tipos principais de cenário: um com apenas uma comida, outro com várias comidas espalhadas e um terceiro com obstáculos, exigindo que o agente contorne barreiras para atingir os objetivos. A implementação dos algoritmos de busca teve como base a estrutura apresentada no livro *Artificial Intelligence: A Modern Approach (AIMA)*, 4ª edição, de Russell e Norvig, adaptada ao problema do PacMan. Dentre os algoritmos disponíveis, foram testadas abordagens como busca em largura (BFS), aprofundamento iterativo, A*, busca gulosa (Greedy) e best_first_search. A análise considerou critérios como tempo de execução, custo da solução, número de ações realizadas e profundidade da solução. Dessa forma, foi possível observar como cada algoritmo se comporta em diferentes níveis de complexidade.

# 2. Desenvolvimento

## 2.1 Modelagem Pacman

Nesta etapa será explicado a modelagem e codificação do problema/desafio. No código 1 a classe PacManProblem herda da classe abstrata Problem e é responsável por modelar o ambiente em que o agente (PacMan) atua. O ambiente é representado por uma matriz bidimensional de tamanho variável (grid_size), contendo paredes, comidas e uma posição inicial do agente.

O estado inicial é representado como uma tupla que armazena a posição atual do agente e um conjunto imutável (frozenset) com as posições das comidas restantes. O método actions retorna as ações válidas a partir de um estado, considerando os limites da grade e a presença de paredes. Já o método result descreve a transição entre estados: ele calcula a nova posição após a ação e remove a comida caso o agente a tenha alcançado.

O método is_valid_position garante que o agente não se mova para fora da grade nem para posições ocupadas por paredes. A função is_goal define a condição de parada do algoritmo de busca: o problema é resolvido quando todas as comidas foram coletadas.

Além disso, a classe implementa uma heurística baseada na distância de Manhattan entre a posição atual do agente e a comida mais próxima. Essa heurística é admissível e consiste em somar as distâncias absolutas nas direções vertical e horizontal, sem considerar obstáculos.

**Código 1:** Modelagem do problema/desafio Pacman.

```
class PacManProblem(Problem):
def __init__(self, initial_pos, food_positions, walls, grid_size):
        # Estado inicial: (posição do PacMan, posições das comidas
restantes)
        initial_state = (initial_pos, frozenset(food_positions))
        self.walls = walls  # conjunto de posições com parede
        self.grid_size = grid_size  # (altura, largura)
        super().__init__(initial=initial_state, goal=None)

    def actions(self, state):
        pos, food = state
        x, y = pos
        directions = {
            'Up':    (x - 1, y),
            'Down':  (x + 1, y),
            'Left':  (x, y - 1),
            'Right': (x, y + 1)
        }

        valid_actions = []
        for action, new_pos in directions.items():
            if self.is_valid_position(new_pos):
                valid_actions.append(action)
        return valid_actions

    def result(self, state, action):
        pos, food = state
        x, y = pos
        move = {
            'Up':    (x - 1, y),
            'Down':  (x + 1, y),
            'Left':  (x, y - 1),
            'Right': (x, y + 1)
        }[action]
        new_pos = move
        new_food = food - {new_pos} if new_pos in food else food
        return (new_pos, new_food)

    def is_valid_position(self, pos):
        x, y = pos
        h, w = self.grid_size
```

```
        return (0 <= x < h) and (0 <= y < w) and (pos not in
self.walls)

    def is_goal(self, state):
        # Objetivo: comer todas as comidas → food set vazio
        _, food = state
        return len(food) == 0


    def heuristic(node):
        # Heurística: distância de Manhattan até a comida mais
próxima
        pos, food = node.state
        if not food:
            return 0
        return min(abs(pos[0] - fx) + abs(pos[1] - fy) for fx, fy
in food)
```

Os algoritmos de busca foram implementados no livro *Artificial Intelligence: A Modern Approach (AIMA)*, o restante de código foi realizando as chamadas de funções, salvando as informações e gerando os gráficos. Todo código está dísponível no **Github**.

2.2 Análise em Diferentes Cenários.

**2.2.1 Cenário 1:** Apenas uma comida.

No primeiro cenário, foi utilizada uma matriz representando um ambiente livre, sem obstáculos, com uma única comida posicionada no canto inferior esquerdo. O agente PacMan parte sempre da posição inicial fixa no canto superior esquerdo (0, 0) e deve percorrer o caminho até alcançar a comida, deslocando-se por espaços livres, todos com custo 1. Esse cenário tem como objetivo analisar o desempenho dos algoritmos em uma situação simples e direta, com apenas um objetivo a ser atingido e sem interferências externas, como paredes ou múltiplos alvos. Além disso, o tamanho da matriz foi progressivamente aumentado para avaliar o impacto da escala no comportamento e na eficiência dos algoritmos de busca. Na Figura 1 é possível ver que o método iterative_deepening_search já começa descolar dos outros métodos aumentando seu tempo até a chegada da solução *goal*(que é não ter nenhuma comida) com o aumento da matriz(tamanho 9x9).

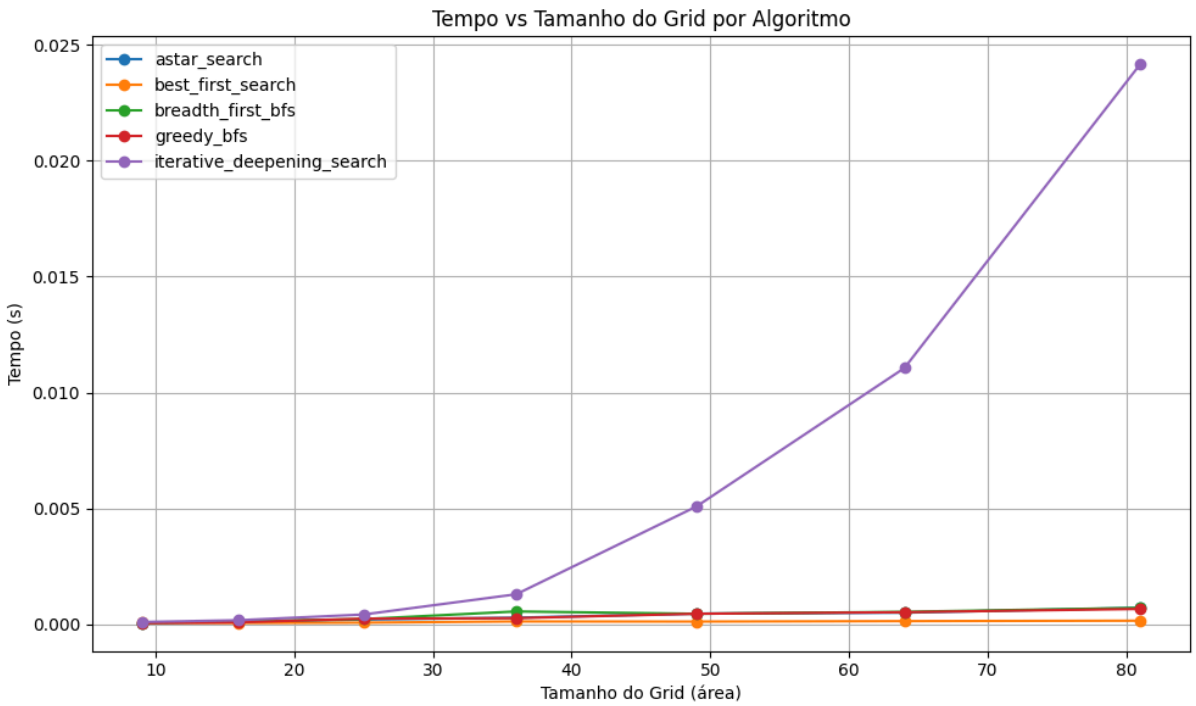


**Figura 1:** Tempo de execução de todos os métodos de busca em matrizes com tamanhos até 9×9.

Devido ao aumento expressivo no tempo de execução do método iterative_deepening_search, esse algoritmo foi retirado do gráfico apresentado na Figura 2.

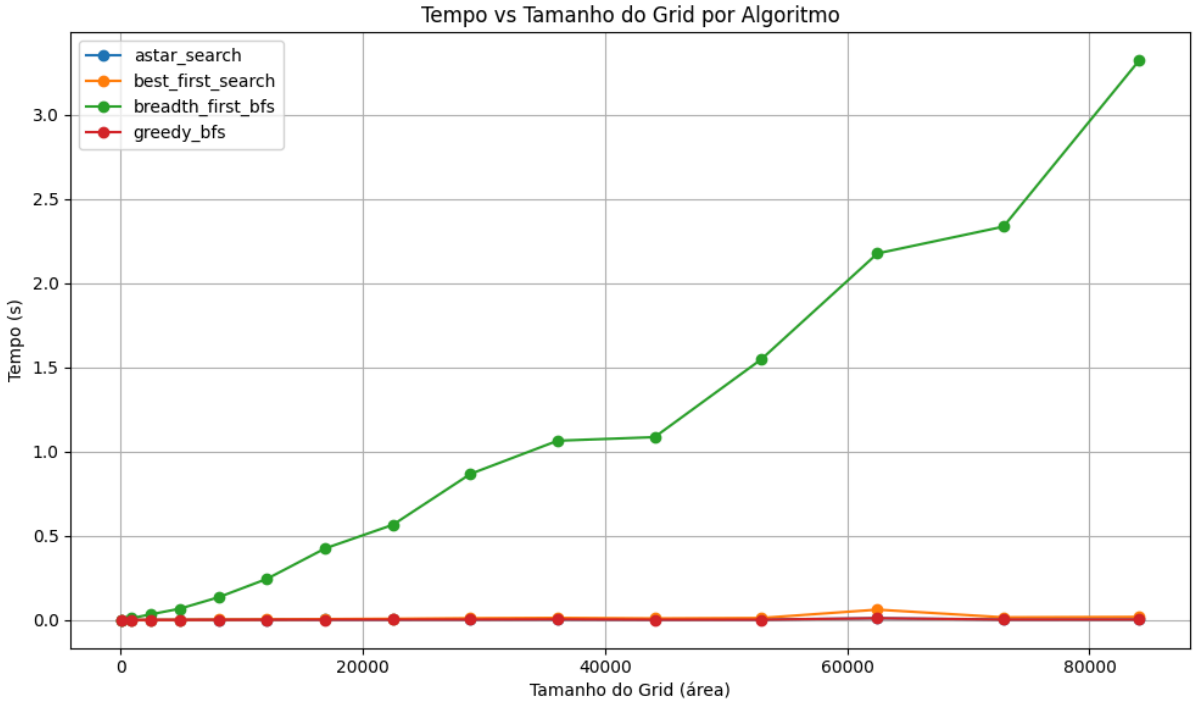


**Figura 2:** Tempo de execução dos quatro métodos de busca em matrizes com tamanhos crescentes até 290×290.

No Gráfico 2, o descolamento observado corresponde ao método de busca em largura, que apresenta um aumento expressivo no tempo de execução a partir de aproximadamente uma matriz de tamanho 140×140. Por esse motivo, esse método foi retirado da Figura 3.

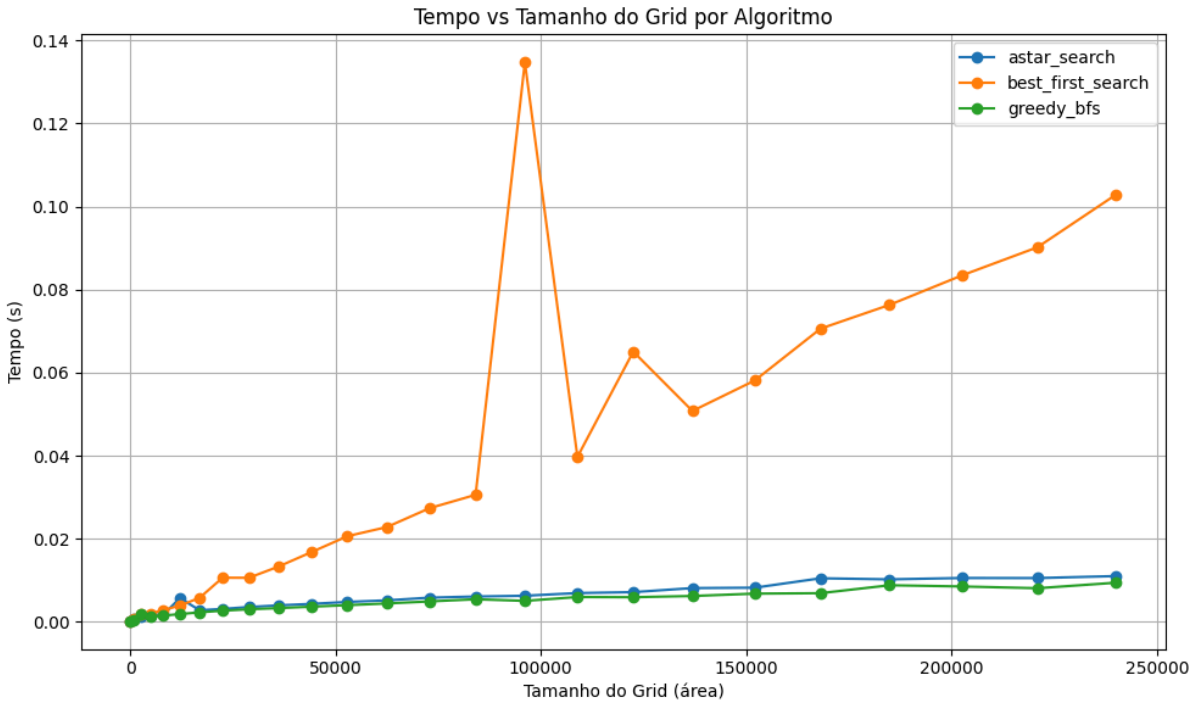


**Figura 3:** Tempo de execução dos 3 métodos de busca em matrizes com tamanhos crescentes até 290×290.

Nos testes com apenas uma comida posicionada no canto oposto à posição inicial do agente, foi possível observar diferenças significativas no tempo de execução entre os algoritmos. O greedy_bfs apresentou o melhor desempenho em todos os tamanhos de grid, sendo o mais rápido por utilizar apenas a heurística para guiar a busca, mesmo sem garantia de solução ótima. O astar_search manteve tempos estáveis e baixos, equilibrando custo real e heurístico, sendo uma opção eficiente e completa. O best_first_search mostrou tempos instáveis, com picos de execução em alguns tamanhos, possivelmente devido ao comportamento da função de avaliação utilizada. Já o breadth_first_bfs teve crescimento constante no tempo com o aumento da matriz, demonstrando limitação em escalabilidade. Por fim, o iterative_deepening_search foi o mais lento, mesmo em grids menores, devido à repetição sucessiva de buscas com profundidades limitadas.

**2.2.2 Cenário 2:** Comida nos canto inferior esquerdo, superior direito e ponto central.

Neste cenário, diferentemente do primeiro, há mais de uma comida para o PacMan coletar. Os métodos utilizados e a expansão progressiva do grid permanecem os mesmos, permitindo observar o impacto do aumento do número de objetivos no desempenho dos algoritmos. Na Figura 4 é possível ver novamente o descolamento do método iterative_deepening_search e um grid de tamanho aproximado de 8x8.

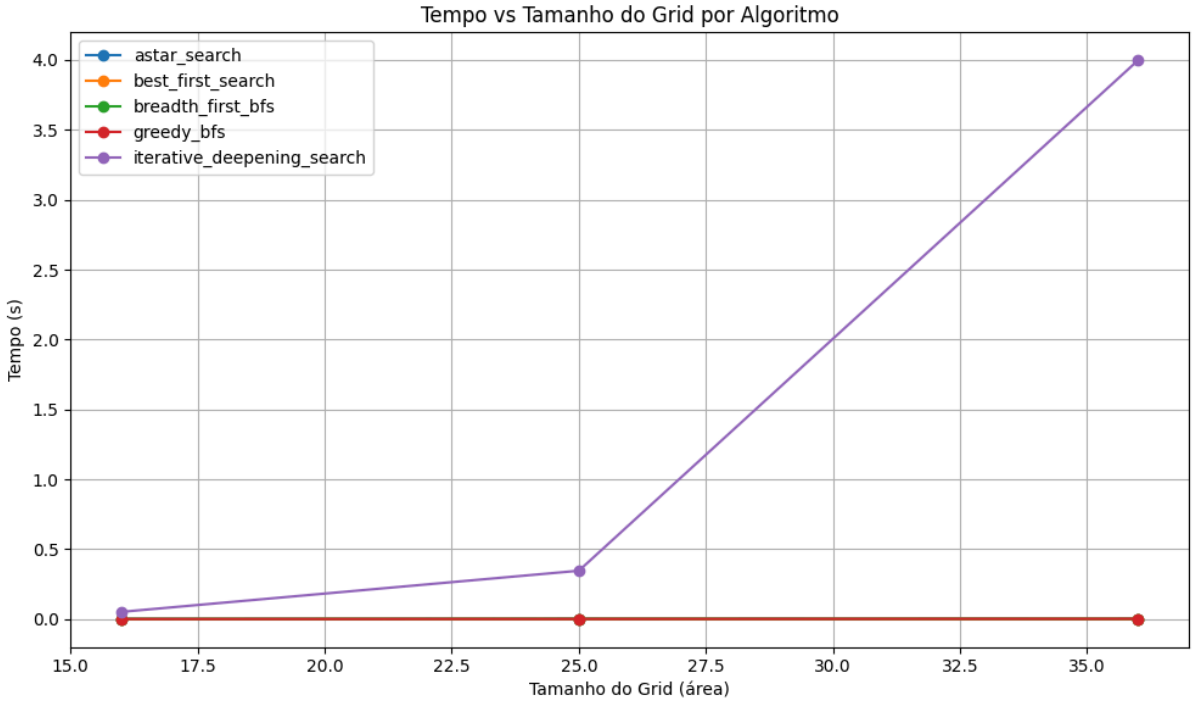


**Figura 4:** Tempo de execução de todos métodos de busca em matrizes com tamanhos crescentes até 8x8 com 3 comidas.

Diferente do cenário 1 começou-se ter diferentes custos para os métodos, como mostrado na Figura 5.

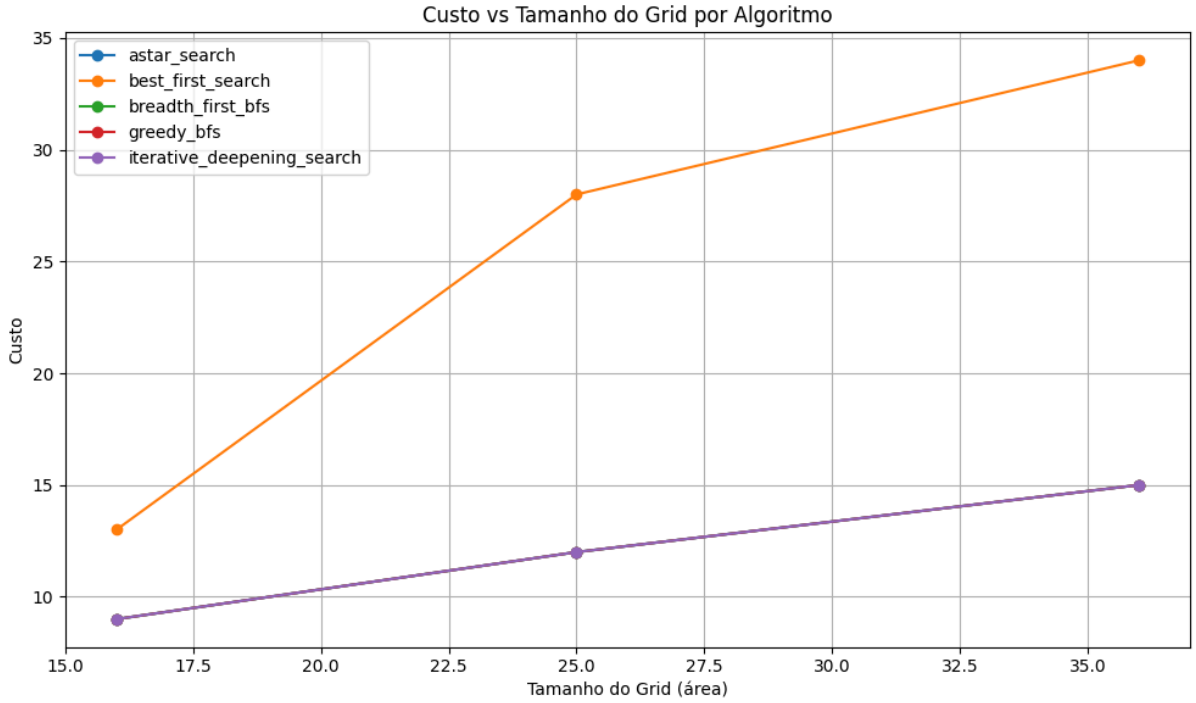


**Figura 5:** Custo de todos métodos de busca em matrizes com tamanhos crescentes até 8x8 com 3 comidas.

Na Figura 5, é possível ver que os algoritmos apresentaram diferenças no custo total das soluções ao lidar com várias comidas no cenário. O best_first_search teve um custo bem maior que os outros, o que significa que ele precisou dar mais passos para completar a tarefa. Já os demais algoritmos encontraram caminhos mais curtos e eficientes.

Na Figura 6 vou excluído o método iterative_deepening_search na mesma é possível ver que o método best_first_search aumentou o tempo para resolução do grid com um tamanho aproximado de 25x25.

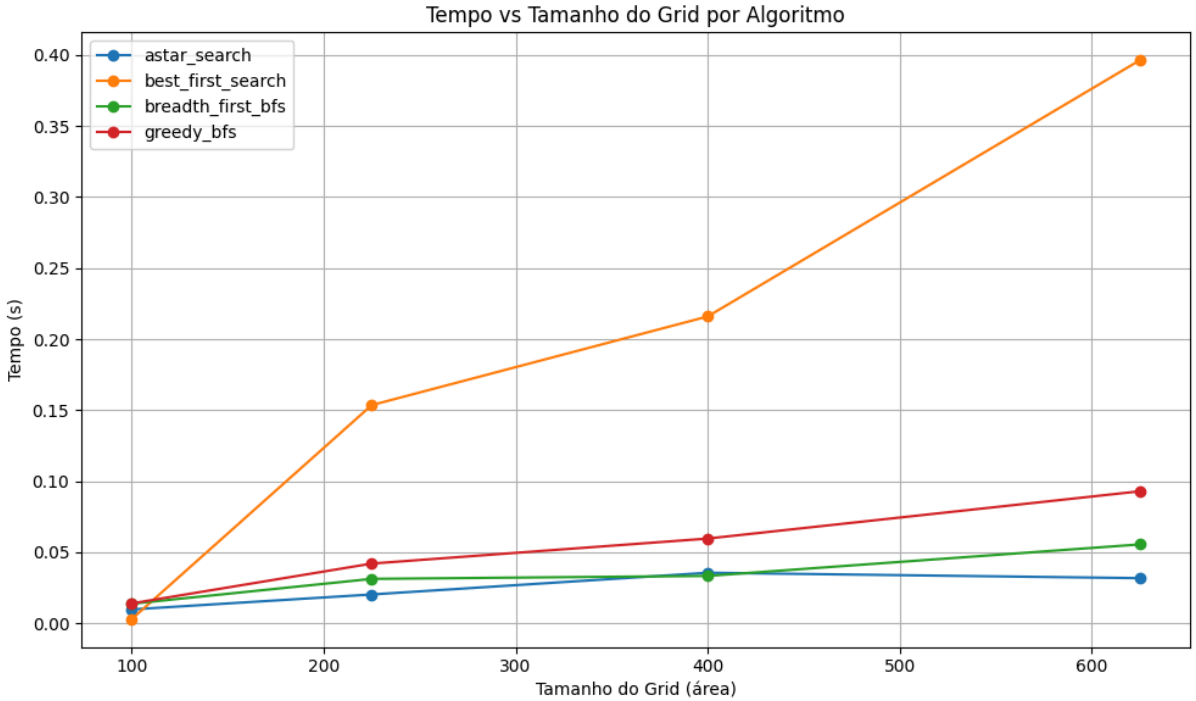


**Figura 6:** Tempo de execução de 4 métodos de busca em matrizes com tamanhos crescentes até 25x25 com 3 comidas.

Já na Figura 7 mostra os 3 métodos que tiveram melhor desempenho de tempo.

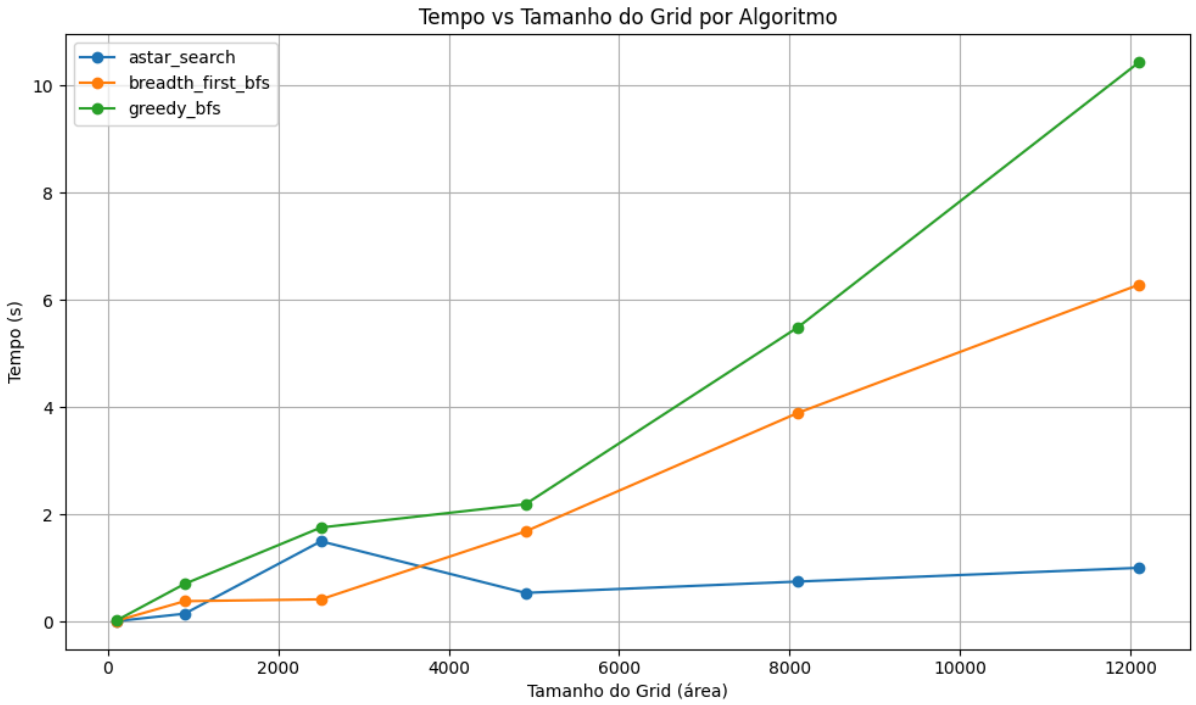


**Figura 7:** Tempo de execução de 3 métodos de busca em matrizes com tamanhos crescentes até 25x25 com 3 comidas.

Neste cenário com mais comidas e espalhadas diferente do cenário 1, percebeu-se maior diferença no desempenho dos algoritmos. Apesar de o algoritmo greedy_bfs apresentar um tempo de execução relativamente baixo nos grids menores, seu desempenho degrada rapidamente à medida que o tamanho da matriz cresce. O breadth_first_bfs apresenta desempenho intermediário, com um tempo de execução crescente com a expansão da matriz, ele encontra soluções com menor custo, por garantir caminhos mais curtos. Já o astar_search demonstrou o melhor equilíbrio entre tempo e qualidade da solução. Mesmo com um leve aumento no tempo à medida que o grid cresce, ele manteve-se como o método mais rápido em matrizes maiores e com o menor custo entre os algoritmos. O algoritmo iterative_deepening_search apresenta um crescimento à medida que o tamanho do grid aumenta. Isso ocorre porque ele combina a profundidade limitada com repetição crescente, resultando em exploração redundante de estados.

O best_first_search foi crescendo de forma mais acentuada à medida que o tamanho do grid aumentou. Além disso, conforme observado no gráfico de custo da Figura 5, esse método apresenta um custo consideravelmente mais alto em comparação aos demais, o que indica que pode estar tomando decisões menos eficientes, mesmo sendo mais rápido em alguns casos. Por outro lado, os algoritmos astar_search, greedy_bfs, breadth_first_bfs e iterative_deepening_search mantiveram custos mais baixos e semelhantes, sugerindo que encontraram caminhos mais curtos e eficientes para visitar todos os pontos de comida.

**2.2.3 Cenário3:** Comidas e obstáculos(paredes).

O terceiro cenário avalia o desempenho dos algoritmos de busca em ambientes mais complexos, contendo comidas e paredes. Nos cenários construídos, foram utilizados grids de diferentes tamanhos(5x5, 7x7, 9x9, 15x15, 29x29 e 49x49) com uma estrutura padronizada para representar desafios crescentes para o agente. Em todos eles, a posição inicial do PacMan é fixada no canto superior esquerdo do mapa, na coordenada (0, 0). As paredes foram colocadas estrategicamente ao longo das duas diagonais principais: a diagonal principal (do canto superior esquerdo ao inferior direito) e a diagonal secundária ou de retorno (do canto superior direito ao inferior esquerdo). Essa configuração cria uma barreira em forma de "X" no centro do grid, dificultando o acesso direto ao meio. As 4 comidas foram posicionadas propositalmente próximas ao ponto de interseção das diagonais exigindo que o agente encontre caminhos alternativos para alcançar o objetivo, contornando os obstáculos impostos pelas paredes dispostas nas diagonais principal e secundária.

A Figura 8 evidencia que o algoritmo iterative_deepening_search apresenta um crescimento expressivo no tempo de execução à medida que o tamanho do grid aumenta, tornando-se significativamente mais lento em grids maiores devido à sua natureza de repetir buscas com profundidades progressivamente maiores. Em contraste, os demais algoritmos — astar_search, best_first_search, breadth_first_bfs e greedy_bfs — mantêm tempos de execução muito baixos e praticamente constantes, mesmo com o aumento da complexidade do cenário, demonstrando maior eficiência computacional em mapas maiores.

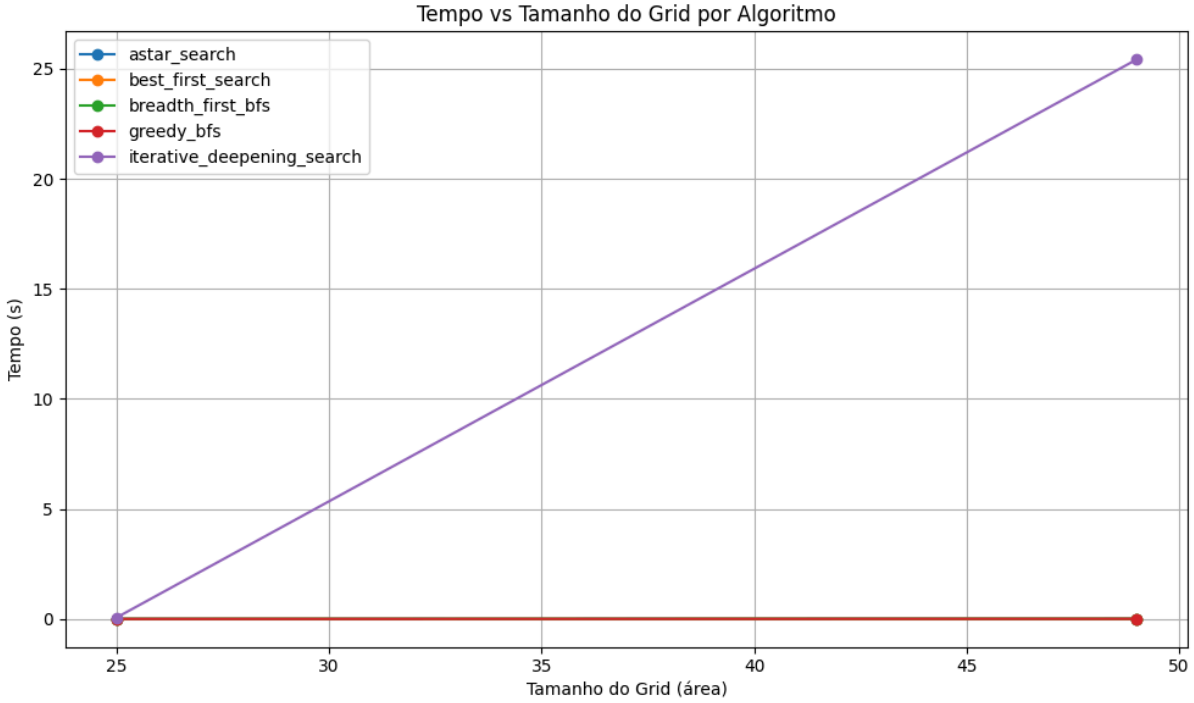


**Figura 8:** Tempo de execução de 5 métodos de busca em mapas 1 e 2.

A Figura 9 mostra o custo de todos os algoritmos, o best_first_bfs, apesar de também encontrar soluções corretas, apresenta um custo significativamente maior à medida que o grid cresce, o que sugere que ele percorre caminhos mais longos ou realiza mais repetições desnecessárias durante a busca, como, mostrado no cenário 2.

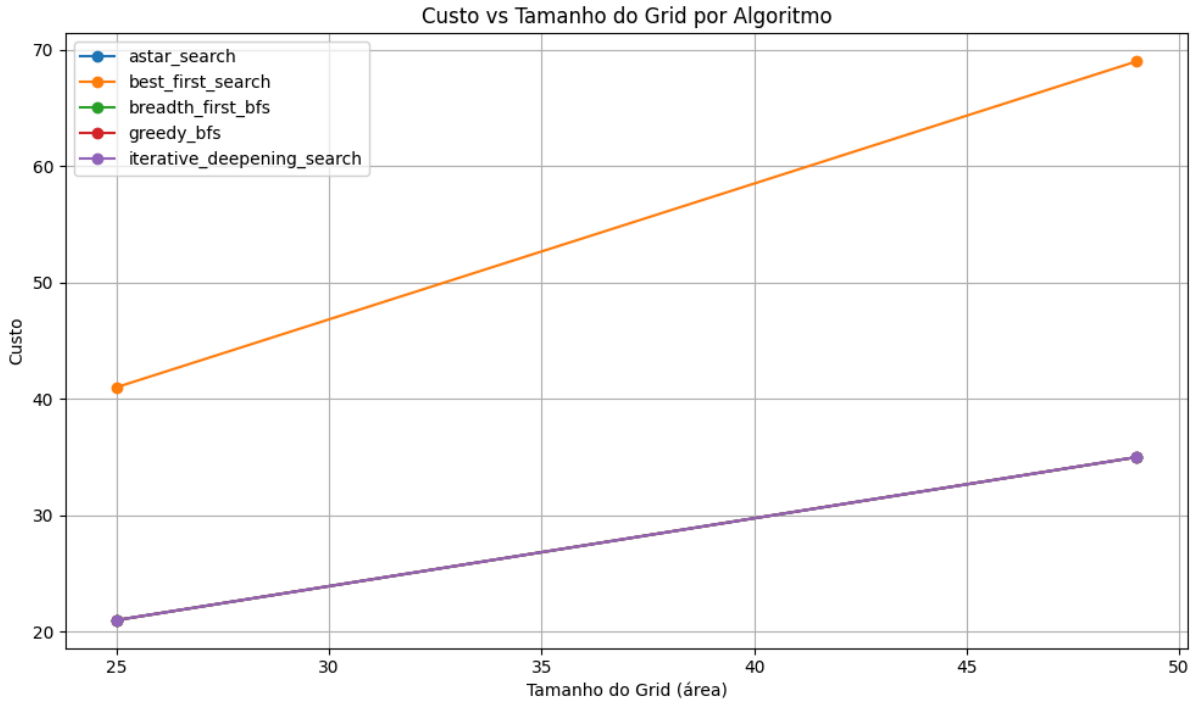


**Figura 9:** Custo de todos métodos de busca nos mapas 1 e 2.

Quando inserido o mapa número 3 em diante é necessário retirar o método iterative_deepening_search devido ao tempo de execução. A Figura 10 mostra que o algoritmo breadth_first_bfs apresenta o maior tempo de execução nos grids maiores, com uma curva de crescimento acentuada, indicando menor eficiência temporal à medida que o espaço de busca aumenta. O best_first_search também apresenta aumento significativo de tempo, embora em menor proporção. Em contraste, os algoritmos greedy_bfs e astar_search mantêm tempos consideravelmente mais baixos e com crescimento mais controlado, mesmo em ambientes maiores.

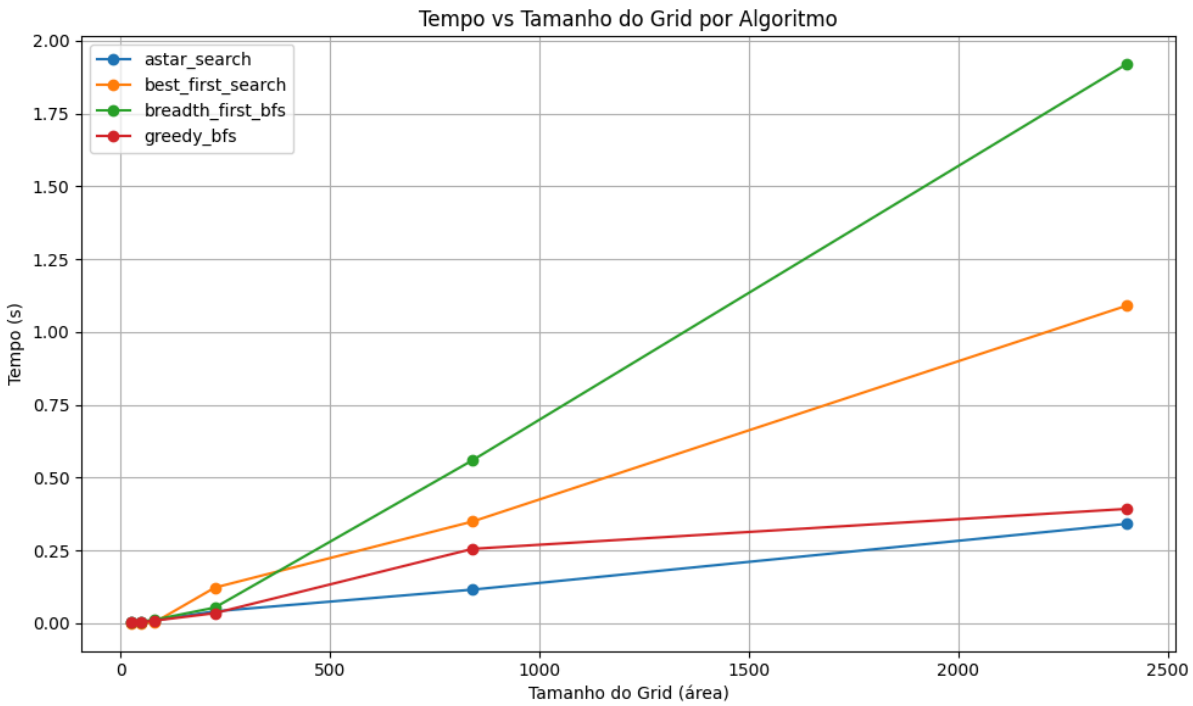


**Figura 10:** Tempo de execução de 4 métodos de busca em mapas 1, 2, 3, 4, 5 e 6.

Na Figura 11 observa-se que o custo do greedy_bfs aumenta porque ele prioriza a proximidade da meta sem considerar o caminho percorrido. Em mapas com obstáculos centrais como os seus, isso leva a caminhos mais longos do que o necessário, especialmente em grids maiores.

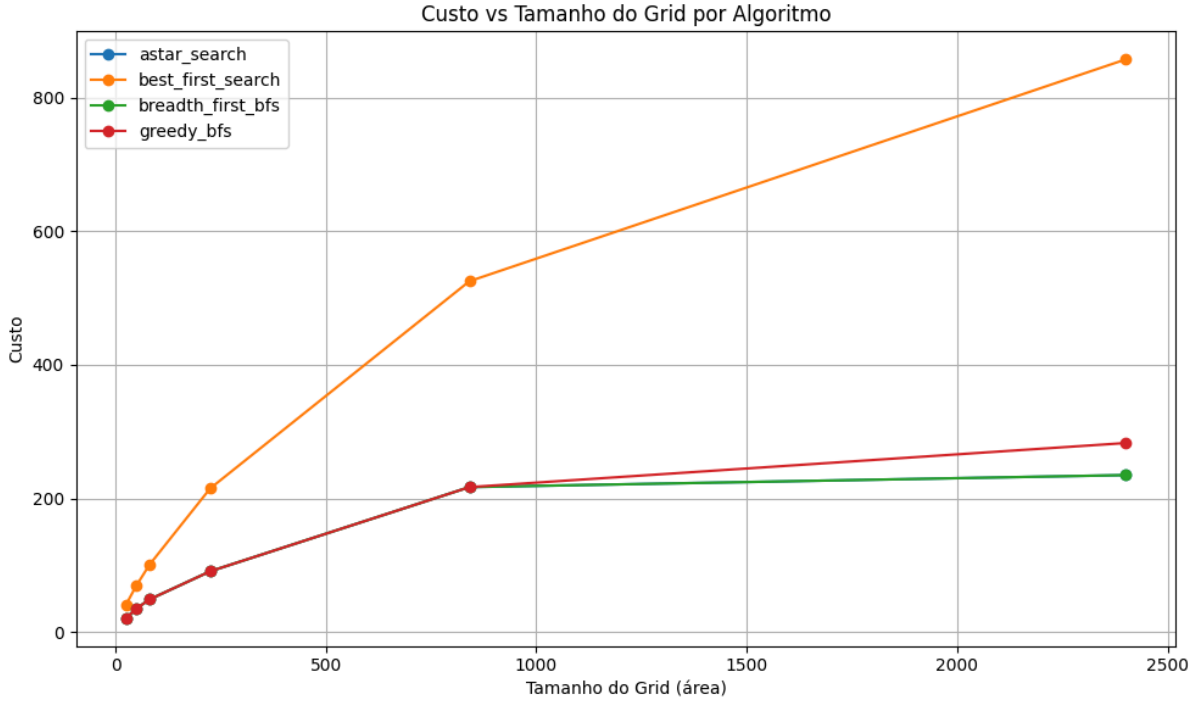


**Figura 11:** Custo de 4 métodos de busca nos 6 mapas.

## 3. Conclusão

Com base nos experimentos realizados, foi possível observar diferenças significativas entre os métodos de busca aplicados ao problema do PacMan, especialmente à medida que os cenários se tornaram maiores e mais complexos. O algoritmo A* mostra o melhor desempenho temporal geral, superando os demais algoritmos em escalabilidade e velocidade de execução. Esses resultados reforçam a vantagem dos métodos heurísticos, como A*, quando se busca eficiência em tempo em cenários de maior complexidade espacial. O breadth_first_bfs, apesar de garantir soluções ótimas, apresentou um crescimento de tempo mais acentuado com o aumento do grid, o que limita sua escalabilidade. O **greedy_bfs**, embora eficiente em cenários menores, mostrou baixa robustez em ambientes maiores. Com o aumento do grid e a complexidade causada por obstáculos diagonais, o algoritmo passou a tomar decisões menos eficazes, já que considera apenas a heurística e ignora o custo real do caminho. Isso levou a tempos de execução maiores e soluções mais custosas, evidenciando que sua estratégia gananciosa pode ser enganosa em contextos mais complexos. Já o **best_first_search** apresentou grande variação de desempenho e custo conforme o tamanho do grid aumentou, especialmente em função da heurística utilizada. Por considerar apenas a estimativa de distância ao objetivo, sem levar em conta o custo acumulado, o algoritmo frequentemente seguiu caminhos aparentemente promissores, mas que se mostraram ineficientes diante de obstáculos centrais. Com isso, teve crescimento acentuado no custo e no tempo de execução, revelando-se menos confiável e escalável em comparação aos demais métodos analisados. Por fim, o iterative_deepening_search teve bom desempenho em pequenas instâncias, mas sofreu com crescimento exponencial de tempo em grids maiores devido à repetição de estados causada por sua natureza iterativa.