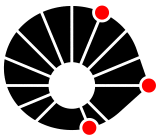

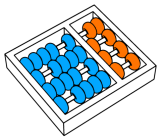

Comparação e Visualização de Algoritmos de Pathfinding em Mapas Dinâmicos em Jogos 2D

Jonathan Junio Guidolino Matos Nunes *Emanuel Felipe Duarte*

Relatório Técnico - IC-PFG-26-20
Projeto Final de Graduação
2026 - Junho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Comparação e Visualização de Algoritmos de Pathfinding em Mapas Dinâmicos em Jogos 2D

Jonathan Junio Guidolino Matos Nunes Emanuel Felipe Duarte*

Resumo

Algoritmos de busca são componentes fundamentais em muitos jogos digitais, responsáveis por mover agentes de um ponto a outro. Em mapas estáticos, algoritmos como Dijkstra e A* oferecem garantias sobre sua otimalidade. Porém, jogos muitas vezes possuem mapas dinâmicos, com objetos que se movem em tempo real, o que exige o replanejamento do caminho e a implementação de algoritmos alternativos, como o HPA e o *Flow Fields*. Este trabalho implementa e compara quatro algoritmos de *pathfinding*: Dijkstra, A*, HPA* e *Flow Fields*, assim como sua visualização em um jogo simples 2D, em diferentes tipos de simulação. Os resultados demonstram que não existe uma solução universal. Em mapas estáticos, o HPA* apresentou um desempenho até 18 vezes superior ao Dijkstra. Contudo, a necessidade de reconstrução integral da hierarquia em ambientes dinâmicos tornou o HPA* até 39 vezes mais lento em tempo de parede. O *Flow Field* mostrou-se o mais eficiente para dezenas de agentes com um destino compartilhado e estável (0,77 ms por chamada), mas seu desempenho degradou drasticamente para 6,1 ms com a movimentação do objetivo. Adicionalmente, confirmou-se que A* e Dijkstra apresentam desempenhos estatisticamente idênticos em mapas com alta heterogeneidade de custos de terreno, invalidando a vantagem da heurística de Manhattan neste contexto.

1 Introdução

1.1 Contextualização

Desde os primeiros jogos de RTS (*Real-Time Strategy*) até os jogos atuais com dezenas de agentes simultâneos, os algoritmos de busca têm evoluído em sofisticação e complexidade. Jogos como *Age of Empires* e *Starcraft* popularizaram o problema de colocar vários agentes se movendo de forma independente e coerente em mapas complexos. O algoritmo A*, proposto por Hart, Nilsson e Raphael em 1968 [6], tornou-se padrão na resolução de problemas de busca por combinar otimalidade e eficiência quando usado com uma boa heurística. Porém, A* e sua variação sem heurística, o algoritmo de Dijkstra [3], são focados na resolução de mapas estáticos, onde o mapa não sofre alterações durante a execução do caminho.

Ambientes dinâmicos, onde objetos se movem, aparecem ou desaparecem em tempo de execução, ou onde o *goal* se move, adicionam complexidade adicional. Nesse contexto, o

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

caminho que foi gerado pode não ser mais válido após a atualização do mapa, exigindo o replanejamento do caminho. A eficiência desse replanejamento afeta diretamente a performance do jogo e, consequentemente, a experiência da pessoa jogadora, podendo ocasionar quedas de FPS (*Frames per Second*) ou comportamento inconsistente dos agentes.

Para lidar com esse problema, foram criados algoritmos como o HPA* e *flow fields*. O HPA* (*Hierarchical Pathfinding A**) [1], projetado por Botea, Müller e Schaeffer em 2004, divide o mapa em múltiplos *clusters* para reduzir o espaço de busca. O algoritmo *flow fields*, usado em vários jogos de estratégia como *Planetary Annihilation* [10], pré-computa as direções do movimento para todo mapa a partir do *goal*, tornando o custo de adicionar novos agentes no mapa praticamente inexistente.

Nenhuma abordagem é necessariamente melhor que a outra; cada uma depende de hipóteses sobre o mapa para funcionar bem, como o número de agentes, o tamanho e a topografia do mapa, e a frequência de mudança.

1.2 Motivação

A motivação para esse tema vem da observação de como diferentes jogos aplicam soluções que, mesmo não sendo perfeitas, resolvem o problema de forma a não atrapalhar a experiência do usuário. Dois exemplos que acho particularmente interessantes são *Age of Empires* e *RollerCoaster Tycoon* (RCT).

Em *Age of Empires* [5], selecionar um conjunto de NPCs para atacar um alvo em movimento, ou ainda para um ponto fixo no mapa, sua IA não é nada precisa. Alguns NPCs vão usar o caminho mais curto, dar voltas sem muito sentido e alguns nem mesmo alcançam o objetivo. Muitas vezes é fácil para nós, jogadores, olhar de cima e identificar qual é o melhor caminho que o agente deveria seguir, mas não é um problema trivial de se resolver computacionalmente em uma fração de segundos.

Ainda mais interessante é o caso de RCT. Lançado em 1999 [2], o jogo simula um parque de diversões com milhares de visitantes, cada um com seu inventário, nível de fome, sede, cansaço e destino, navegando individualmente pelos caminhos do parque. E tudo isso rodando em menos de 16MB de RAM[1]. Chris Sawyer escreveu praticamente todo o jogo em Assembly para otimizar ao máximo a execução. Como algoritmos de busca são normalmente custosos, na maior parte do tempo os NPCs andam quase que aleatoriamente, com apenas algumas regras para evitar *dead ends*, até encontrar o que procuram ou uma atração que desejam visitar. Assim, mesmo com fome ou sede, os visitantes não procuram esses locais; eles andam aleatoriamente até encontrá-los. Há poucos casos em que o A* ainda é usado, como para mecânicos ou quando o NPC quer ir embora do parque. Embora seja uma solução simples, funciona extremamente bem e o jogo foi um grande sucesso e, na opinião dos autores deste relatório, é muito divertido.

O contraste desses dois exemplos é o ponto central deste trabalho: não existe um algoritmo de busca perfeito para todos os cenários. Essa escolha depende do ambiente, do número de agentes, da frequência das mudanças e das restrições de hardware. Tomar essa decisão requer, muitas vezes, uma análise empírica, não apenas teórica; este trabalho busca

[1] Configuração recomendada para o hardware da época. Isso considerando a memória usada por processos do Sistema Operacional e outros processos

fazer experimentos controlados e reproduzíveis em cenários que refletem situações reais de jogos para ver as vantagens e desvantagens de diferentes algoritmos de busca.

1.3 Objetivos

1.3.1 Objetivo Geral

Implementar, comparar e visualizar algoritmos de *pathfinding* em mapas 2D com obstáculos dinâmicos, avaliando o desempenho de cada um em diferentes condições ambientais.

1.3.2 Objetivos Específicos

1. Implementar os algoritmos Dijkstra, A, HPA e Flow Field em C++, com uma interface que permite a troca do solver facilmente.
2. Definir um conjunto de cenários de avaliação que analise ambientes estáticos, obstáculos dinâmicos esparsos e densos (NPCs), e objetivo (player) móvel.
3. Executar os cenários em três tipos de topologia de mapa: labirinto, campo aberto e mapa de salas, coletar métricas de latência, nós expandidos, comprimento do caminho, uso de memória e taxa de sucesso.
4. Construir uma ferramenta de visualização interativa que permita observar o comportamento dos agentes e dos algoritmos em tempo real.
5. Analisar e comparar os resultados, identificando em quais condições cada algoritmo apresenta vantagens e desvantagens práticas.

1.4 Metodologia

1.4.1 Ambiente de Execução e Parâmetros Gerais

Ferramenta / Recurso	**Especificação**	**Uso no Projeto**
Hardware (CPU)	Intel Core Ultra 9 275HX	Processamento das simulações e *benchmarks*
Memória e OS	64 GB / WSL2 (Arch Linux)	Ambiente de execução principal dos testes de performance
Linguagem	C++ (Padrão C++17/20)	Desenvolvimento de todo o *framework* e *solvers*
Compilador	GCC (g++)	Compilação do código fonte (utilizando otimização `-O3`)
Biblioteca Gráfica	SFML V2.6	Renderização da janela interativa a 60 Hz (modo `gui`)

Tabela 1: Ambiente de hardware e ferramentas de software utilizadas nos experimentos.

Todos os experimentos utilizam os parâmetros fixos da Tabela 4, escolhidos para garantir comparabilidade entre os três mapas.

O código-fonte do *framework* está disponível publicamente em https://github.com/jonathan-junio/eng-computacao-tcc.

1.4.2 Abordagem Experimental

Este trabalho tem uma abordagem experimental. Foi desenvolvido em `C++` um *framework* de simulação que carrega mapas em formato customizado (ver Apêndice 9), geração procedural de mapas, múltiplos agentes autônomos e uma interface gráfica feita com SFML. Todos os algoritmos foram implementados com a mesma interface de solver, garantindo que as comparações sejam isoladas da lógica de simulação e renderização.

Os experimentos foram realizados em quatro cenários fixos, elaborados para simular diferentes níveis de estresse e dinamismo: *static* (cenário de controle com um único agente e ambiente imutável), *dyn-sparse* (10 agentes concorrentes com aparecimento periódico de obstáculos), *dyn-dense* (50 agentes sob alta carga de interrupções de rotas) e *dyn-moving-goal* (50 agentes combinando obstáculos dinâmicos e movimentação do destino). Essas condições foram aplicadas sobre três tipos de mapa com topologias estruturais distintas: *maze* (focado em corredores longos e únicos), *rooms* (regiões abertas conectadas por gargalos estreitos) e *open* (campo totalmente livre com áreas de custo variável). Uma descrição detalhada das características de cada cenário e mapa é apresentada na Seção 4. Cada execução processou `50.000 ticks` a `60Hz` virtuais. Os resultados foram compilados em arquivos CSV para análise (ver Apêndice B). As métricas coletadas incluem latência média, percentis (P50, P90, P95, P99) do tempo de computação de caminho, nós expandidos por por chamada, comprimento médio de caminho, pico de uso de memória e número de de chamadas sem resultado válido (*stuck count*).

2 Fundamentação teórica

2.1 Representação do Espaço de Busca

Antes de descrever os algoritmos, é importante entender como um mapa 2D pode ser representado na memória. A abordagem mais comum é modelar o mapa como um **grafo**, uma estrutura matemática composta por nós (ou vértices) e arestas que conectam pares de nós [9].

Em um mapa de grade (*grid*), cada célula é um nó e as arestas conectam os nós adjacentes. Dependendo do jogo, o tipo da conectividade pode ser:

- 4-connected: cada célula se conecta apenas às quatro vizinhas ortogonais (cima, baixo, esquerda, direita)
- 8-connected: adiciona também as conexões diagonais

Cada aresta possui um peso que representa o custo de atravessar aquele terreno. Em um campo aberto o custo pode ser 10, em um pântano 40 e uma parede pode ter um custo infinito (célula bloqueada). Essa estrutura com custo é o que permite que algoritmos não apenas achem um caminho até o destino, mas possam também encontrar o caminho com o **menor custo total**.

Formalmente, o problema de *pathfinding* pode ser definido como: dado um grafo `G = (V, E)`, com pesos não negativos nas arestas, um nó de origem `s` e um destino `t`, deve-

se encontrar a sequência de nós de `s` até `t` que minimize a soma dos custos das arestas envolvidas.

2.2 O Algoritmo de Dijkstra

O algoritmo de Dijkstra foi proposto por Edsger W. Dijkstra em 1959 e é considerado um dos algoritmos mais fundamentais da ciência da computação [3]. A ideia é expandir os nós em ordem crescente, acumulando os custos a partir da origem, garantindo que quando um nó é processado pela primeira vez, o custo encontrado da origem até ele seja o menor possível.

1. Inicializa o custo de todos os nós como ∞, exceto a origem (custo $= 0$)
2. Insere a origem em uma fila de prioridade ordenada pelo custo acumulado
3. A cada iteração, retira da fila o nó com menor custo e examina seus vizinhos
4. Se o custo para chegar a um vizinho pelo nó atual for menor que o registrado, atualiza o custo e insere o vizinho na fila
5. Repete até que o nó destino seja retirado da fila ou a fila esvazie.

Uma boa maneira de visualizar o algoritmo de Dijkstra é imaginar como a água se espalharia por um labirinto, preenchendo primeiro as regiões com menor resistência. Porém, sem saber onde é o destino, a água tem que se espalhar em todas as direções até encontrá-lo, o que é ineficiente em mapas grandes por causa da natureza gulosa deste algoritmo [8].

A complexidade de Dijkstra com uma fila de prioridade eficiente, como o heap binário, é $O((V + E) \log V)$, onde V é o número de nós e E o número de arestas. Este algoritmo **garante** o caminho de custo mínimo e é usado como algoritmo base para comparação neste estudo [9].

2.3 Algoritmo A*

O A* (*A-star*) foi proposto por Hart, Nilsson e Raphael em 1968 e resolve o principal problema do Dijkstra: a falta de direcionamento da busca [6]. A ideia é equipar o algoritmo com uma **heurística**, uma estimativa de custo até o destino, para priorizar nós que aparentam ser mais promissores.

A **função de avaliação** de cada nó n é definida como:

$$f(n) = g(n) + h(n)$$

onde $g(n)$ é o **custo real** percorrido do nó de origem até n, e $h(n)$ é a **estimativa heurística** do custo de n até o destino. A fila de prioridade é ordenada por $f(n)$, fazendo o A* explorar primeiro os nós que combinam baixo custo acumulado com boa proximidade estimada do destino.

Para que o A* garanta o caminho ótimo, a heurística precisa ser **admissível**: nunca superestimar o custo real [9]. Em mapas de grade, as heurísticas mais usadas são:

```
função DIJKSTRA(grafo, origem, destino):
    para todo nó n em grafo:
        dist[n]     <- infinito
        anterior[n] <- NULO
    dist[origem] <- 0

    fila <- fila_de_prioridade()
    fila.inserir(origem, prioridade=0)

    enquanto fila não estiver vazia:
        u <- fila.remover_mínimo()
        se u = destino:
            retornar RECONSTRUIR_CAMINHO(anterior, destino)
        para cada vizinho v de u:
            novo_custo <- dist[u] + peso(u, v)
            se novo_custo < dist[v]:
                dist[v]     <- novo_custo
                anterior[v] <- u
                fila.inserir(v, prioridade=novo_custo)
    retornar []   // destino inacessível

função RECONSTRUIR_CAMINHO(anterior, destino):
    caminho <- []
    nó <- destino
    enquanto nó != NULO:
        caminho.prepend(nó)
        nó <- anterior[nó]
    retornar caminho
```

Listing 1: Pseudocódigo do algoritmo de Dijkstra

- **Distância de Manhattan** ($|dx| + |dy|$): correta para movimentos apenas ortogonais
- **Distância Euclidiana** ($\sqrt{dx^2 + dy^2}$): adequada quando movimentos diagonais são permitidos
- **Distância de Chebyshev** ($\max(|dx|, |dy|)$): quando diagonal e ortogonal têm o mesmo custo

Quando $h(n) = 0$ para todos os nós, o A* se comporta exatamente como o Dijkstra. Quando $h(n)$ é perfeita, o A* encontra o caminho sem explorar nenhum nó desnecessário. Na prática, heurísticas admissíveis ficam entre esses extremos e produzem ganhos expressivos de desempenho ao evitar visitar todos os nós possíveis [7].

```
função A_ESTRELA(grafo, origem, destino):
    para todo nó n em grafo:
        g[n]        <- infinito
        anterior[n] <- NULO
    g[origem] <- 0

    fila <- fila_de_prioridade()
    fila.inserir(origem, prioridade=h(origem, destino))

    enquanto fila não estiver vazia:
        u <- fila.remover_mínimo()
        se u = destino:
            retornar RECONSTRUIR_CAMINHO(anterior, destino)
        para cada vizinho v de u:
            g_novo <- g[u] + peso(u, v)
            se g_novo < g[v]:
                g[v]        <- g_novo
                anterior[v] <- u
                f <- g[v] + h(v, destino)
                fila.inserir(v, prioridade=f)
    retornar []

// Heurística: distância de Manhattan (movimentos ortogonais)
função h(nó, destino):
    retornar |nó.x - destino.x| + |nó.y - destino.y|
```

Listing 2: Pseudocódigo do algoritmo A*

2.4 O algoritmo HPA*: Hierarchical Pathfinding A*

Mesmo com a heurística do A*, mapas muito grandes tornam a busca lenta o suficiente para comprometer o desempenho do jogo. O **HPA*** (*Hierarchical Pathfinding A**), proposto por Botea, Müller e Schaeffer em 2004, resolve esse problema introduzindo **abstração hierárquica** do mapa [1].

A ideia central é parecida com a forma como planejamos uma viagem de carro: primeiro decidimos quais cidades passar (nível macro), depois escolhemos as ruas dentro de cada cidade (nível micro). O HPA* faz exatamente isso com o mapa do jogo, em três fases:

1. **Pré-processamento (offline):** O mapa é dividido em regiões retangulares chamadas *clusters*. Para cada par de clusters adjacentes, identificam-se os *entrance nodes* que são células na fronteira entre eles. Esses pontos formam um **grafo abstrato** cujas arestas são calculadas com A* interno a cada cluster. Esse grafo é armazenado e reutilizado em todas as consultas subsequentes

2. **Busca abstrata:** Origem e destino são conectados temporariamente ao grafo abstrato e executa-se um A* sobre ele. O resultado é uma sequência de *entrance nodes*, que seria um caminho no nível macro

3. **Refinamento:** Cada segmento do caminho abstrato é calculado com A* dentro do cluster correspondente, produzindo o caminho final no nível original

O ganho de desempenho vem do fato de que o grafo abstrato tem muito menos nós do que o mapa original. **Em ambientes dinâmicos**, quando um obstáculo muda, apenas os clusters afetados precisam ser reconstruídos enquanto o restante permanece válido [1]. Uma limitação importante é que os caminhos do HPA* podem ser **sub-ótimos**, pois o agente é forçado a passar pelos *entrance nodes* dos clusters. Na prática, esse desvio costuma ser pequeno e imperceptível.

2.5 O Algoritmo Flow Fields

Os algoritmos anteriores calculam um caminho para um agente por vez. Em jogos de estratégia com centenas de agentes se movendo para o mesmo destino, executar A* individualmente para cada um deles seria ineficiente. O algoritmo ***Flow Fields*** resolve esse problema invertendo a lógica: em vez de calcular um caminho por agente, calcula-se um único mapa de direções para todos os agentes ao mesmo tempo [4].

A construção funciona em duas etapas:

1. **Campo de custos (*cost field*):** Executa-se um Dijkstra a partir do **destino**, propagando custos pelo mapa inteiro. Ao final, cada célula contém a distância de custo mínimo até o objetivo

2. **Campo de direções (*flow field*):** Para cada célula, armazena-se um vetor apontando para o vizinho de menor custo, que indica a direção ótima de movimento a partir daquela célula.

A grande vantagem de algoritmos de campo é a escalabilidade. A pré-computação custa $O(N)$ onde N é o número de células, mas a consulta por agente é $O(1)$, apenas uma leitura de tabela. Isso significa que mover 1 ou 10.000 agentes tem praticamente o mesmo custo computacional após a construção do campo [4, 7].

A limitação principal é que o campo inteiro precisa ser reconstruído sempre que o objetivo muda ou quando mudanças de obstáculos afetam rotas já calculadas. Por isso, *Flow Fields* são mais adequados para cenários com **muitos agentes compartilhando um mesmo objetivo** relativamente estável.

2.6 Síntese Comparativa

A Tabela 2 resume as principais características teóricas dos quatro algoritmos, facilitando a compreensão das condições em que cada um tende a se destacar.

C = nós no grafo abstrato ($C \ll V$)

Nenhum algoritmo domina os demais em todas as dimensões. Os experimentos realizados buscam quantificar empiricamente essas diferenças em condições controladas e reproduzíveis [7].

Tabela 2: Comparação teórica dos algoritmos de *pathfinding*

Característica	**Dijkstra**	**A***	**HPA***	**Flow Field**
Heurística	Não	Sim	Sim (A* interno)	Não (Dijkstra)
Caminho ótimo	Sim	Sim (heur. adm.)	Aproximado	Sim
Custo por consulta	$O((V+E)\log V)$	$O((V+E)\log V)$	$O(C \log C)$	$\mathbf{O(1)}$
Pré-computação	Não	Não	Sim (offline)	Sim (por objetivo)
Múltiplos agentes	Ineficiente	Ineficiente	Moderado	**Muito efic.**
Ambiente dinâmico	Replan. total	Replan. total	Replan. parcial	Replan. total

3 Arquitetura do Framework

3.1 Visão Geral das Camadas

O *framework* é organizado em quatro camadas isoladas, conforme o diagrama da Figura 5:

A camada **Core** define as estruturas de dados fundamentais e não tem dependência de nenhuma outra camada do projeto. A camada de **Pathfinding** conhece apenas o Core. As camadas de execução (GUI, Benchmark, CLI) conhecem tanto o Core quanto o Pathfinding, mas são completamente independentes entre si.

3.2 Representação do Mapa

O mapa é representado por três estruturas aninhadas, cada uma adicionando um nível de abstração:

`Grid` é a estrutura mais baixa. Armazena o mapa como dois vetores lineares indexados por $id = y \times largura + x$[2]: `m_tileCost` (custo inteiro de cada célula, -1 para paredes) e `m_tileChar` (caractere original do *tile*).

`MapData` envolve o `Grid` e acrescenta os índices do início e fim: as posições dos marcadores `S` e `G` no arquivo.

`WorldState` é o objeto central de simulação. Ele agrega o `MapData` com a lista de agentes ativos (`m_agents`) e é a estrutura passada por referência a todos os *solvers*. Quando o mapa muda dinamicamente, os *solvers* recebem uma notificação via `onTilesChanged()` sem precisar ser recriados.

Os mapas são armazenados em arquivos texto com três seções:

Essa estrutura permite definir terrenos com custos arbitrários e posicionar início e fim diretamente no *grid*, tornando os mapas legíveis e editáveis manualmente.

3.3 Interface dos Solvers

A decisão de *design* mais importante do *framework* é a abstração `PathSolver`: uma classe base virtual que todos os algoritmos implementam:

O método `compute(from, to)` recebe e retorna índices lineares de células, tornando a interface independente de coordenadas 2D. O método `onTilesChanged()` permite que

[2]Assim evita-se criar `arrays` multidimensionais, melhorando o desempenho.

solvers como o HPA* realizem **replanejamento incremental** atualizando apenas os clusters afetados, enquanto solvers mais simples como Dijkstra e A* simplesmente recalculam o caminho completo na próxima chamada.

A instanciação é feita por uma *factory function*:

Adicionar um novo algoritmo ao *framework* requer apenas criar uma classe que herda de `PathSolver` e registrá-la na *factory*.

3.4 Modos de Execução

O ponto de entrada `Application::run()` lê as opções da linha de comando e despacha para um dos quatro modos de execução disponíveis:

Tabela 3: Modos de execução do *framework*

Modo	`Comando`	**Descrição**
`solve`	`./tcc --solver astar --map mapa.map`	Calcula e imprime um caminho no terminal
`gui`	`./tcc gui --solver astar --map mapa.map`	Abre a visualização interativa com SFML
`generate`	`./tcc generate --type maze --width 200`	Gera um mapa proceduralmente
`bench`	`./tcc bench --map mapa.map --full`	Executa a suíte de benchmarks headless

Essa separação permite que os experimentos de *benchmark* rodem sem qualquer dependência gráfica, essencial para execuções longas e sem interferência de renderização. Por outro lado, o modo GUI oferece uma ferramenta visual para observar o comportamento dos algoritmos em tempo real

3.5 Loop de Simulação (Modo GUI)

O modo gráfico é gerenciado pela classe `Game`, que implementa um loop de atualização a 60 Hz virtuais usando SFML. A cada *frame*, o loop executa três fases:

1. **`process_events()`**: trata entradas do teclado e mouse (pausar, trocar *solver*, mover o objetivo)
2. **`update(dt)`**: avança o estado da simulação. Move agentes ao longo de seus caminhos, dispara mudanças dinâmicas de obstáculos no intervalo configurado e chama `solver->onTilesChanged()` com as células afetadas
3. **`render()`**: desenha o *grid* com as cores de terreno, os caminhos calculados, os agentes e as estatísticas de desempenho em *overlay*.

Cada agente é representado pela classe `Agent`, que armazena seu caminho atual como uma lista de índices e avança um nó por *tick*. Quando o agente chega ao destino ou seu caminho é invalidado por uma mudança de mapa, o *solver* é invocado para replanejamento.

3.6 Infraestrutura de Benchmarks

O modo *benchmark* executa simulações **headless** (sem renderização) de forma determinística, com *seed* de aleatoriedade fixo. Ele opera em dois modos de temporização:

- **Modo por duração** (`--bench-duration 60`): cada *solver* roda por 60 segundos de tempo de parede real, e o número de iterações é variável dependendo do desempenho do algoritmo e do mapa. Esse modo é mais representativo do desempenho em um jogo real, mas pode introduzir variação de hardware entre os resultados.
- **Modo por *ticks*** (`--bench-ticks 50000`): cada *solver* executa exatamente 50.000 iterações da simulação, eliminando variação de hardware dos resultados. Esse modo é o usado para comparações diretas entre algoritmos, garantindo que todos sejam avaliados sob as mesmas condições de mapa e número de eventos dinâmicos.

A função `run_suite()` automatiza a execução dos quatro cenários padronizados: *static*, *dyn-sparse*, *dyn-dense* e *dyn-moving-goal*, garantindo que todos os *solvers* sejam avaliados sob exatamente as mesmas condições de mapa, *seed* e número de *ticks*.

4 Cenários e Mapas dos Experimentos

4.1 Parâmetros Gerais de Simulação

Todos os experimentos utilizam os parâmetros fixos da Tabela 4, escolhidos para garantir comparabilidade entre os três mapas.

Tabela 4: Parâmetros gerais de simulação

Parâmetro	**Valor**
Ticks por execução	50.000
Taxa de simulação	60 Hz virtuais ($dt = 1/60\,\text{s}$)
Semente de aleatoriedade	42 (determinístico)
Intervalo de mudança de obstáculo	3,0 s → a cada 180 *ticks*
Porcentagem de *tiles* bloqueados por evento	5% dos *tiles walkable*
Intervalo de movimentação do objetivo	5,0 s → a cada 300 *ticks*

Com 50.000 *ticks* a 60 Hz cada execução representa **13,9 minutos de tempo simulado**. Nesse período ocorrem cerca de **277 eventos de mudança de obstáculos** ($50.000/180 \approx 277$) e no cenário onde o objetivo é móvel aproximadamente **166 movimentos de objetivo** ($50.000/300 \approx 166$).

4.2 Mapas

Foram utilizados três mapas com topologias diferentes. Todos possuem quatro tipos de terrenos com custos diferentes: piso comum (`.`, custo 10), grama (`g`, custo 15), água (`w`, custo 40) e lama (`m`, custo 80).

4.2.1 Labirinto (*maze*)

O mapa de labirinto tem dimensões 199×99 **células**, ligeiramente menor que os outros tipos de mapas (200×100) devido a uma limitação do algoritmo de geração: O método de *recursive backtracking* percorre o grid em passos compostos por duas células, abrindo passagens entre posições de coordenadas ímpares. Para que esse tipo de mapa seja válido, com células *walkable* em índices ímpares e paredes nos pares, ambas as componentes de dimensão precisam ser ímpares, resultando em 199×99 **células**, sem impacto relevante na análise entre os mapas dado que a diferença é pequena.

Este mapa foi gerado proceduralmente pelo gerador incluído no *framework*. Aproximadamente **51% das células são paredes**, formando um labirinto de corredores estreitos sem becos sem saída. As principais características para fins de avaliação são:

- Caminhos longos e curvas: o caminho entre início e fim tem **806 células** de comprimento
- Alta densidade de obstáculos estruturais reduz o número de *tiles* candidatos para bloqueio dinâmico
- Corredores únicos fazem com que um único obstáculo seja capaz de isolar completamente uma região do mapa, explicando o alto `stuck_count` nos cenários dinâmicos.

4.2.2 Campo Aberto (*open*)

O mapa de campo aberto tem dimensões 200×100 **células** e **nenhuma parede**, 100% dos *tiles* são *walkable*, variando apenas em tipo de terreno. É o mapa mais favorável para todos os algoritmos: sempre existe caminho entre quaisquer dois pontos. O caminho estático tem **260 células**. Nos cenários dinâmicos, o `stuck_count` foi **zero** para todos os algoritmos, confirmando que o campo aberto nunca gera situações de caminho inviável.

4.2.3 Mapa de Salas (*rooms*)

O mapa de salas tem dimensões 200×100 **células** e é o mais denso em paredes: apenas **30% das células são *walkable*** (6.014 de 20.000). O mapa é formado por regiões abertas (salas) conectadas por corredores estreitos, similar a uma planta baixa de edifício. O caminho estático tem **254 células**, mas o percurso obrigatoriamente passa por corredores que conectam salas, tornando o mapa sensível a bloqueios dinâmicos de forma comparável ao labirinto.

4.3 Cenários

Quatro cenários foram definidos para isolar variáveis distintas de dificuldade e refletir situações comuns encontradas em jogos digitais.

Tabela 5: Características dos mapas utilizados

Característica	**Labirinto**	**Campo Aberto**	**Salas**
Dimensões	199×99	200×100	200×100
Total de células	19.701	20.000	20.000
Tiles walkable	9.715 (49%)	20.000 (100%)	6.014 (30%)
Comprimento do caminho estático	806 células	260 células	254 células
Sensibilidade a bloqueios dinâmicos	Alta	Baixa	Alta

4.3.1 Estático (*static*)

Configuração: 1 agente, sem obstáculos dinâmicos, sem movimentação de objetivo. Mede o **custo de uma única computação de caminho** em cada mapa. Serve como linha de base: qualquer *overhead* observado nos cenários dinâmicos é atribuível exclusivamente ao custo de replanejamento.

4.3.2 Dinâmico Esparso (*dyn-sparse*)

Configuração: 10 agentes, obstáculos dinâmicos a cada 3 s, sem movimentação de objetivo. Avalia o comportamento dos algoritmos sob **carga de replanejamento moderada e múltiplos agentes concorrentes**. No campo aberto, os 10 agentes nunca ficam presos (`stuck_count = 0`). No labirinto e no mapa de salas, a maioria das chamadas a `compute()` retorna caminho vazio, resultando em `stuck_count` elevado (≈89% das chamadas).

4.3.3 Dinâmico Denso (*dyn-dense*)

Configuração: 50 agentes, obstáculos dinâmicos a cada 3 s, sem movimentação de objetivo. Avalia o comportamento dos algoritmos sob **carga de replanejamento intensa e alta concorrência entre agentes**, complementando o *dyn-sparse* (10 agentes) para observar como o custo escala com o aumento do número de agentes ativos.

4.3.4 Objetivo Móvel (*dyn-moving-goal*)

Configuração: 50 agentes, obstáculos dinâmicos a cada 3 s, objetivo se move a cada 5 s. Combina os dois tipos de invalidação de caminho, resultando em **442 eventos de replanejamento** contra 276 nos outros cenários dinâmicos. Para o *Flow Field*, mover o objetivo exige reconstruir o campo inteiro a partir do novo destino, o caso mais custoso para esse algoritmo.

5 Resultados e Discussão

Essa seção apresenta e analisa os resultados das execuções dos *benchmarks*. Os valores de `avg_compute_ms` representam a média do tempo de execução de cada chamada de `compute()`, incluindo tanto as chamadas que encontraram um caminho válido, quanto as

Tabela 6: Resumo dos cenários de avaliação

Cenário	**Agentes**	**Obst. dinâmicos**	**Objetivo móvel**	**Eventos replan.**
static	1	Não	Não	0
dyn-sparse	10	Sim (a cada 3 s)	Não	≈276
dyn-dense	50	Sim (a cada 3 s)	Não	≈276
dyn-moving-goal	50	Sim (a cada 3 s)	Sim (a cada 5 s)	≈442

que retornaram vazio (destino inacessível). O `wall_time_ms` representa o tempo total de parede gasto pelo solver durante os 50.000 *ticks* de *clock*, incluindo o custo de reconstrução das estruturas internas como o grafo abstrato do HPA*.

5.1 Cenário Estático

Este cenário serve como referência, cada algoritmo executa exatamente uma chamada a `compute()`, sem nenhum replanejamento posterior. Revelando o custo de uma única consulta em cada topologia.

Tabela 7: Tempo médio de computação por chamada (ms) para cenário estático

Solver	**Labirinto**	**Campo Aberto**	**Salas**
Dijkstra	0,76	6,23	1,61
A*	0,81	6,38	1,82
HPA*	**0,33**	**0,35**	**0,19**
Flow Field	1,83	6,24	1,78

Tabela 8: Nós expandidos por chamada para o cenário estático

Solver	**Labirinto**	**Campo Aberto**	**Salas**
Dijkstra	2.856	19.994	5.721
A*	2.843	19.994	5.612
HPA*	**488**	**381**	**130**
Flow Field	9.715	20.000	6.014

O primeiro resultado que podemos observar é que o **A* e Dijkstra apresentam desempenho quase idêntico** nos três mapas. No campo aberto, por exemplo, ambos expandem exatamente 19.994 nós, praticamente o mapa completo. Isso acontece porque a heurística de Manhattan usada assume um custo uniforme por movimento, mas o mapa possui caminhos com custos variados. Quando o custo das arestas não são uniformes a heurística subestima o custo real e não consegue guiar a busca de forma eficiente, degradando o A* ao custo de Dijkstra [9].

O HPA* é o algoritmo mais rápido em todos os mapas neste cenário estático. Ao operar sobre um grafo abstrato de clusters ele expande entre 130 (salas) até 488 (labirinto) nós contra 2.856 a 19.994 do Dijkstra, uma redução de **duas ordens de grandeza**. Esse ganho é especialmente grande no mapa de campo aberto, onde HPA* leva 0,35 ms contra 6,23 ms do Dijkstra, **18 vezes mais rápido**.

O ***Flow Field*** apresenta o maior custo no cenário estático porque precisa propagar o custo do mapa inteiro a partir do destino para construir o campo de direções. Esse custo de pré-processamento não compensa em caso estático com apenas um agente, pois o campo não vai ser reutilizado.

5.2 Cenários Dinâmicos - Tempo de Computação por Chamada

Nos cenários dinâmicos, o `avg_compute_ms` inclui tanto as chamadas que retornam caminho vazio (rápidas) quanto as que encontram caminho real (custosas). No labirinto e no mapa de salas, a maioria das chamadas falha, o que artificialmente reduz a média. O campo aberto, onde `stuck_count = 0`, oferece a visão mais honesta do custo real por consulta.

Tabela 9: Tempo médio de computação por chamada (ms) — cenários dinâmicos

Cenário	Solver	Labirinto	Campo Aberto	Salas
dyn-sparse	Dijkstra	0,113	3,854	0,325
	A*	0,106	3,650	0,327
	HPA*	0,068	**0,224**	**0,081**
	Flow Field	**0,021**	0,768	0,134
dyn-dense	Dijkstra	0,115	3,967	0,356
	A*	0,108	4,266	0,329
	HPA*	0,081	**0,260**	**0,053**
	Flow Field	**0,062**	6,123	0,304
dyn-moving-goal	Dijkstra	0,110	3,359	0,367
	A*	0,103	3,631	0,429
	HPA*	**0,064**	**0,274**	**0,070**
	Flow Field	0,040	6,170	0,213

No **campo aberto**, o HPA* demonstra a vantagem mais clara: sua latência por chamada fica entre 0,22 e 0,27 ms, enquanto Dijkstra e A* ficam entre 3,4 e 4,3 ms, cerca de **15× mais rápido por consulta**. Esse é o comportamento esperado da hierarquia de clusters operando sobre um espaço de busca reduzido.

O ***Flow Field*** apresenta comportamento oposto nos cenários dinâmicos do campo aberto: no *dyn-sparse* custa apenas 0,77 ms por chamada, mas no *dyn-dense* e *dyn-moving-goal* sobe para 6,1 ms. Isso acontece porque os obstáculos invalidam o campo frequentemente, forçando reconstruções completas com o mesmo custo da computação estática.

5.3 Custo Real de Simulação - Tempo Total de Parede

O `wall_time_ms` revela o custo efetivo de cada *solver* durante toda a simulação, incluindo os custos de atualização de estruturas internas disparados por `onTilesChanged()`.

Tabela 10: Tempo total de parede (ms) — cenários dinâmicos

Cenário	Solver	Labirinto	Campo Aberto	Salas
dyn-sparse	Dijkstra	117	759	249
	A*	112	727	252
	HPA*	**4.593**	**10.741**	**2.613**
	Flow Field	61	236	122
dyn-dense	Dijkstra	58	169	39
	A*	54	177	40
	HPA*	**4.605**	**10.747**	**2.545**
	Flow Field	51	208	36
dyn-moving-goal	Dijkstra	62	223	44
	A*	60	227	44
	HPA*	**4.710**	**10.953**	**2.587**
	Flow Field	49	302	39

O resultado mais expressivo é o **custo de reconstrução do HPA***. Nos cenários dinâmicos, seu tempo total de parede é **39× maior** que o do Dijkstra no labirinto e **14× maior** no campo aberto, invertendo completamente a vantagem que ele demonstrou no cenário estático.

5.4 Análise Detalhada do HPA* em Ambientes Dinâmicos

A estrutura `SolverTiming` registra quatro fases internas do HPA*: busca no grafo abstrato (*search*), atualização de clusters (*cluster_update*), refinamento de segmentos (*refine*) e reconstrução completa do grafo abstrato (*rebuild*).

A fase de *rebuild*, reconstrução total do grafo abstrato após cada mudança de obstáculos, representa **97% a 99% do tempo total do solver** em todos os cenários dinâmicos. A busca abstrata em si é extremamente rápida, confirmando que a hierarquia é eficiente para consultas, no entanto, o gargalo é o custo de manutenção da hierarquia quando o mapa muda.

Esse resultado evidencia uma limitação importante da implementação de HPA* deste trabalho: cada evento de mudança de obstáculos dispara uma **reconstrução total** do grafo abstrato em vez de atualizar apenas os clusters afetados. Implementar uma atualização verdadeiramente incremental é a principal melhoria que poderia viabilizar o HPA* em cenários altamente dinâmicos.

Tabela 11: Decomposição do tempo interno do HPA* (ms acumulados) em cenários dinâmicos

Mapa	Cenário	Search	Clust. upd.	Refine	Rebuild	Total
Labirinto	*dyn-sparse*	7	0	31	4.488	4.525
	dyn-dense	1	0	4	4.537	4.543
	dyn-moving-goal	2	97	5	4.543	4.647
Campo Aberto	*dyn-sparse*	24	0	16	10.576	10.617
	dyn-dense	3	0	2	10.628	10.633
	dyn-moving-goal	5	210	4	10.620	10.840
Salas	*dyn-sparse*	36	0	14	2.516	2.567
	dyn-dense	1	0	0	2.504	2.505
	dyn-moving-goal	2	38	1	2.506	2.546

5.5 Comparação entre Dijkstra e A*

Em todos os cenários e mapas testados, A* e Dijkstra apresentaram desempenho estatisticamente equivalente. A diferença de tempo por chamada raramente supera 10%, e o número de nós expandidos é virtualmente idêntico.

Isso não é um resultado inesperado: a eficácia do A* depende da qualidade da heurística. Nos mapas deste trabalho, os custos de terreno variam entre 10 e 80 (razão de 1:8), fazendo com que a distância de Manhattan, calculada assumindo custo 1 por célula, subestime sistematicamente o custo real. Quanto menor a razão entre a estimativa heurística e o custo verdadeiro, mais o A* se comporta como Dijkstra [8]. Em mapas com custos de terreno uniformes, a vantagem do A* seria muito mais pronunciada.

5.6 Flow Field - Quando Compensa e Quando Não Compensa

O *Flow Field* demonstra um perfil de desempenho bimodal nos cenários dinâmicos, dependendo fortemente da frequência de mudanças e do número de agentes:

- **Cenário favorável:** muitos agentes com objetivo estável. No *dyn-sparse* com 10 agentes no campo aberto, o *Flow Field* custa 0,77 ms por chamada, mais barato que qualquer outro algoritmo exceto HPA*. O custo de construção do campo é amortizado entre todos os agentes que o consultam simultaneamente.
- **Cenário desfavorável:** objetivo que muda frequentemente. No *dyn-moving-goal* do campo aberto, o *Flow Field* custa 6,1 ms por chamada sendo idêntico ao custo de construção estática do campo. Mesmo com 50 agentes, reconstruir o campo inteiro a cada mudança de objetivo elimina o benefício da amortização.

Essa dualidade confirma a recomendação teórica: *Flow Fields* são indicados quando N agentes compartilham o mesmo destino por tempo suficiente para amortizar o custo de construção, e contraindicados quando o destino muda frequentemente ou o número de agentes é baixo [4].

5.7 Efeito da Topologia sobre o *stuck_count*

O `stuck_count`, número de chamadas a `compute()` que retornaram caminho vazio indicando que o destino está inacessível, varia drasticamente entre mapas:

Tabela 12: Taxa de chamadas sem caminho válido (`stuck_count` / `total_computes`) - Dijkstra

Cenário	**Labirinto**	**Campo Aberto**	**Salas**
dyn-sparse	554/561 (98,7%)	0/169 (0%)	582/653 (89,1%)
dyn-dense	66/68 (97,1%)	0/18 (0%)	16/23 (69,6%)
dyn-moving-goal	107/108 (99,1%)	0/36 (0%)	32/39 (82,1%)

A diferença é grande. No campo aberto, os 5% de *tiles* bloqueados dinamicamente nunca isolam completamente o destino. No labirinto, onde corredores únicos conectam regiões, um único *tile* bloqueado pode isolar o destino, tornando quase todo replanejamento sem sucesso.

Esse resultado tem implicações práticas importantes: em jogos com mapas de corredor (*dungeons*, labirintos), qualquer algoritmo de replanejamento terá desempenho de consulta aparentemente baixo, mas a maioria das chamadas retorna imediatamente com caminho vazio, o custo real está concentrado nos poucos momentos em que o destino é de fato acessível. O comportamento do agente nesses casos (aguardar, tentar nova rota, desistir) é uma decisão de *design* de jogo, não de algoritmo.

5.8 Uso de Memória

Todos os algoritmos apresentaram consumo de memória muito próximo, na faixa de **7,1 MB a 8,3 MB** de RSS de pico. O HPA* acrescenta entre 200 e 600 KB em relação ao Dijkstra para armazenar o grafo abstrato, e o *Flow Field* acrescenta custo similar para manter o campo de direções. Nenhum algoritmo apresenta diferença de memória relevante para aplicações práticas em plataformas modernas.

5.9 Síntese dos Resultados

Tabela 13: Resumo das condições favoráveis a cada algoritmo

Algoritmo	**Melhor condição**	**Pior condição**
Dijkstra	Terreno heterogêneo, implementação simples	Mapas grandes, muitos agentes
A*	Custo de terreno uniforme	Terreno muito variado (aprox. Dijkstra)
HPA*	Mapa estático ou pouco dinâmico	Alta frequência de mudanças
Flow Field	Muitos agentes, destino estável	Poucos agentes ou destino que muda

Os experimentos confirmam que **nenhum algoritmo é universalmente superior**. A escolha ideal depende de três fatores principais: (1) o número de agentes compartilhando o mesmo destino, (2) a frequência de mudanças no ambiente e (3) a topologia do mapa.

6 Conclusão

6.1 Síntese do Trabalho

Este trabalho implementou, comparou e visualizou quatro algoritmos de *pathfinding*: Dijkstra, A*, HPA* e *Flow Field*, em um *framework* de simulação desenvolvido em C++, avaliando seu comportamento em três topologias de mapa distintas (labirinto, campo aberto e mapa de salas) e quatro cenários de dificuldade crescente (estático, dinâmico esparso, dinâmico denso e objetivo móvel). Ao todo, foram executadas 48 configurações de *benchmark*, cada uma com 50.000 *ticks* de simulação determinística.

O *framework* produzido oferece uma interface unificada de *solver*, um sistema de notificação de mudanças de mapa, geração procedural de mapas, uma ferramenta de visualização interativa com SFML e uma infraestrutura de *benchmark headless* capaz de registrar latência por percentil, nós expandidos, uso de memória e métricas internas específicas do HPA*.

6.2 Principais Conclusões

A* e Dijkstra são equivalentes em mapas com custo de terreno variado. O principal resultado desta análise foi a confirmação empírica de que, quando os pesos das arestas variam significativamente, como ocorre nos mapas deste trabalho com custos entre 10 e 80, a heurística de Manhattan perde sua capacidade de guiar a busca de forma eficiente. Esse resultado é relevante para desenvolvedores: a complexidade adicional de implementar e ajustar uma heurística pode não se traduzir em ganho de desempenho se o terreno do jogo for heterogêneo.

HPA* é o algoritmo mais eficiente por consulta em ambientes estáticos, mas se torna o mais custoso em ambientes altamente dinâmicos. No cenário estático, o HPA* foi até 18× mais rápido que Dijkstra no campo aberto. Nos cenários dinâmicos, porém, o custo de reconstrução do grafo abstrato, representando 97% a 99% do tempo total do *solver*, tornou o HPA* 14× a 39× mais lento que Dijkstra em tempo de parede. Esse resultado evidencia que a versão de HPA* implementada neste trabalho não realiza atualizações verdadeiramente incrementais.

Flow Fields têm desempenho dependente do número de agentes e da estabilidade do objetivo. Com 10 agentes no campo aberto, o *Flow Field* apresentou latência por chamada inferior à de Dijkstra no cenário esparso. Quando o destino muda frequentemente, o custo de reconstrução integral do campo elimina essa vantagem independentemente do número de agentes.

A topologia do mapa influencia mais o comportamento dos algoritmos do que o algoritmo em si. No campo aberto, todos os algoritmos encontraram caminho em 100% das chamadas, enquanto no labirinto e no mapa de salas a taxa de falha chegou a 99%. Em jogos com esse tipo de mapa, estratégias de design, como garantir conectividade mínima antes de bloquear um tile (aspecto ignorado neste trabalho, que utilizou mudanças dinâmicas randômicas sem validação), são tão importantes quanto a escolha do algoritmo de *pathfinding*.

6.3 Limitações

Os resultados estão sujeitos a algumas limitações:

- **Implementação incremental do HPA*:** a versão implementada reconstrói o grafo abstrato inteiro a cada mudança de obstáculos, em vez de atualizar apenas os clusters afetados. Uma implementação verdadeiramente incremental poderia reduzir o custo dinâmico do HPA* em uma ou duas ordens de grandeza.
- **Heurística única para o A*:** apenas a distância de Manhattan foi avaliada. Heurísticas adaptadas ao custo médio do terreno poderiam melhorar significativamente o desempenho do A* em mapas heterogêneos.
- **Número fixo de agentes:** os cenários de múltiplos agentes foram testados com 10 e 50 agentes. A vantagem do *Flow Field* em cenários com centenas ou milhares de agentes não foi avaliada nesta escala.
- **Hardware único:** todos os *benchmarks* foram executados em uma única máquina, em ambiente WSL2 sobre Linux. Os valores absolutos de tempo não são diretamente generalizáveis a outros ambientes, embora as proporções relativas entre algoritmos sejam robustas.

6.4 Considerações Finais

O problema de *pathfinding* em jogos digitais continua sendo uma área ativa porque as demandas dos jogos modernos com ambientes cada vez maiores, mais dinâmicos e com mais agentes, continuam empurrando os limites do que os algoritmos clássicos conseguem oferecer. Este trabalho contribui com uma comparação empírica sistemática em condições controladas e reproduzíveis, tornando explícitas as trocas (*trade-offs*) que muitas vezes ficam implícitas na literatura.

O *framework* desenvolvido permanece extensível: a interface `PathSolver` permite adicionar novos algoritmos sem modificar a simulação, os cenários e mapas são configuráveis por linha de comando, e a infraestrutura de *benchmark* produz dados estruturados prontos para análise. O código-fonte está disponível em https://github.com/jonathan-junio/eng-computacao-tcc, licenciado sob a MIT License.

A Exemplo de formato do mapa

B Dados Brutos

B.1 Legenda das Colunas dos Arquivos CSV

Os arquivos CSV gerados pelo benchmark possuem 27 colunas, descritas na Tabela 14.

Tabela 14: Descrição das colunas dos arquivos de resultados

Coluna	Descrição
`scenario`	Cenário executado (*static*, *dyn-sparse*, *dyn-dense*, *dyn-moving-goal*)
`solver`	Algoritmo utilizado (`dijkstra`, `astar`, `hpa`, `flow`)
`num_agents`	Número de agentes na simulação
`fixed_ticks`	1 = modo por ticks fixos
`total_ticks`	Total de ticks executados
`wall_time_ms`	Tempo total de parede (ms)
`sim_fps`	Ticks por segundo de parede
`cpu_time_ms`	Tempo de CPU, usuário + sistema (ms)
`peak_memory_kb`	Pico de memória RSS (KB)
`total_computes`	Total de chamadas a `compute()`
`total_nodes_expanded`	Total de nós expandidos em todas as chamadas
`avg_nodes_expanded`	Média de nós expandidos por chamada
`avg_compute_ms`	Tempo médio por chamada (ms)
`p50_compute_ms`	Percentil 50 da latência de computação (ms)
`p90_compute_ms`	Percentil 90 da latência de computação (ms)
`p95_compute_ms`	Percentil 95 da latência de computação (ms)
`p99_compute_ms`	Percentil 99 da latência de computação (ms)
`max_compute_ms`	Pior tempo de computação registrado (ms)
`stuck_count`	Chamadas que retornaram caminho vazio
`hpa_search_ms`	Tempo acumulado de busca abstrata — HPA* apenas
`hpa_cluster_update_ms`	Tempo acumulado de atualização de clusters — HPA* apenas
`hpa_refine_ms`	Tempo acumulado de refinamento — HPA* apenas
`hpa_rebuild_ms`	Tempo acumulado de reconstrução total do grafo — HPA* apenas
`hpa_total_solver_ms`	Tempo total interno do solver HPA*
`agents_reached`	Agentes que chegaram ao destino
`avg_path_length`	Comprimento médio do caminho (células)
`total_replans`	Total de eventos de replanejamento disparados

B.2 Dados Brutos — Labirinto

B.3 Dados Brutos — Campo Aberto

B.4 Dados Brutos — Mapa de Salas

C Capturas de Tela da Visualização Gráfica

Referências

[1] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):7–28, 2004.

[2] Chris Sawyer Productions. Rollercoaster tycoon. Jogo eletrônico, 1999. Plataforma: PC.

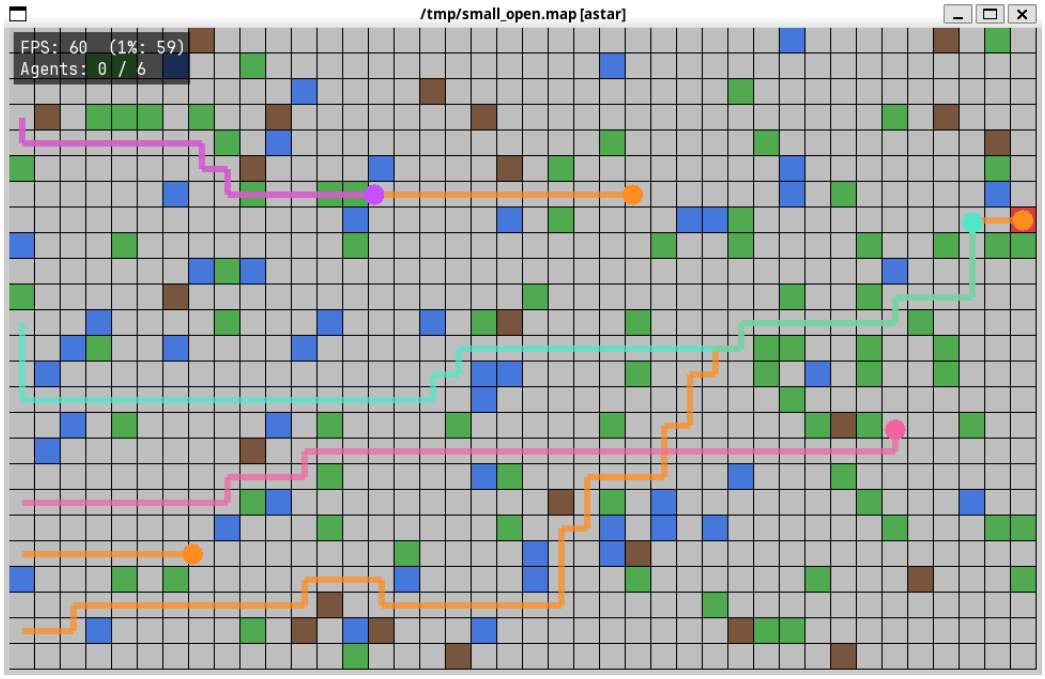

Figura 1: Visualização do mapa de campo aberto (open).

[3] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[4] Elijah Emerson. Crowd pathfinding and steering using flow field tiles. In Steve Rabin, editor, *Game AI Pro: Collected Wisdom of Game AI Professionals*, pages 67–76. CRC Press, 2013.

[5] Ensemble Studios. Age of empires. Jogo eletrônico, 1997. Plataforma: PC.

[6] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[7] Ian Millington. *Artificial Intelligence for Games*. CRC Press, 3 edition, 2019. ISBN 978-1138483972.

[8] Amit Patel. Amit's a* pages - red blob games, 2024. URL https://www.redblobgames.com/pathfinding/a-star/introduction.html. Acessado em: 26 fev. 2026.

[9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4 edition, 2020. ISBN 978-0134610993.

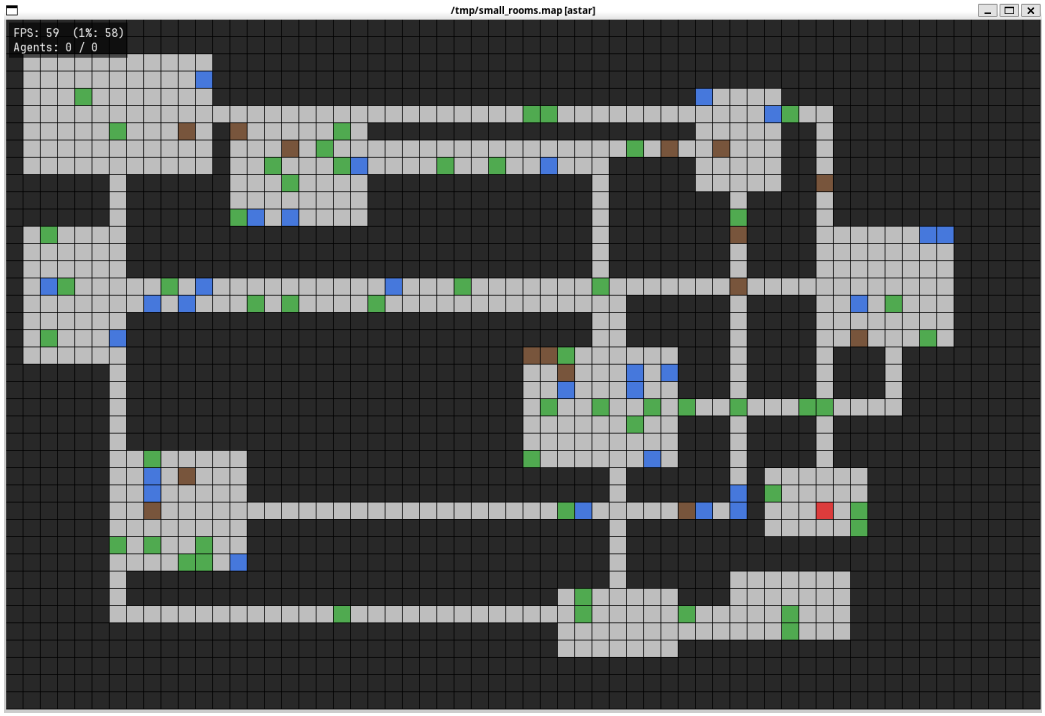

Figura 2: Visualização do mapa de salas (rooms).

[10] Uber Entertainment. Planetary annihilation: March 22nd livestream. Vídeo do YouTube, 2013. URL https://www.youtube.com/watch?v=5Qyl7h7D1Q8. Acesso em: 16 de junho de 2026.

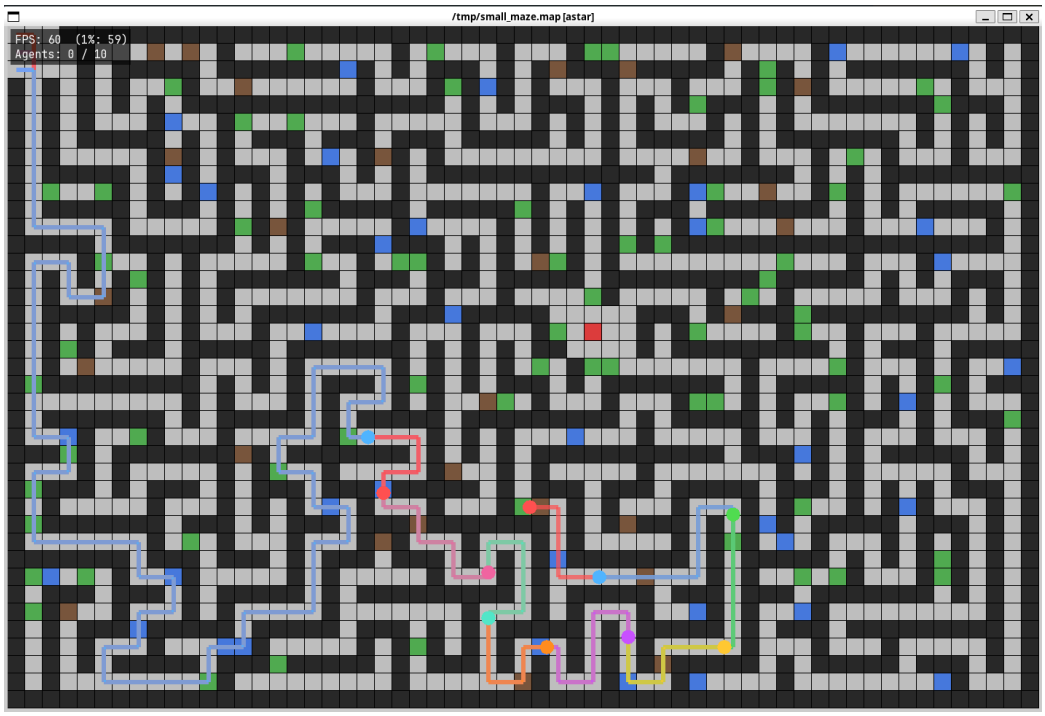

Figura 3: Visualização do mapa de labirinto (maze).

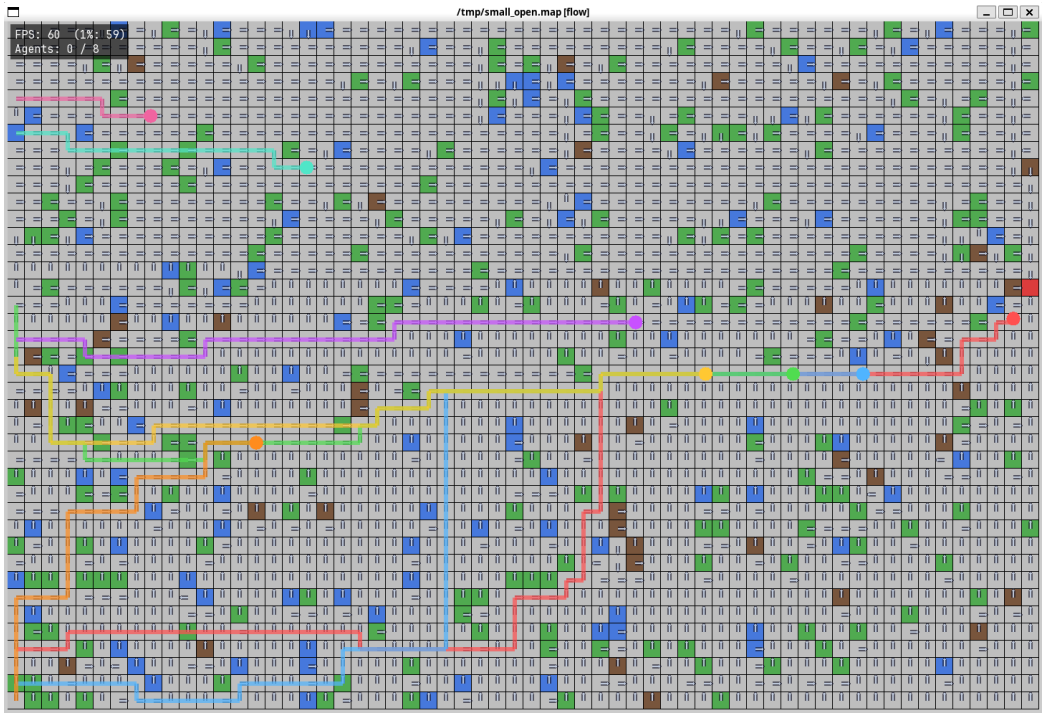

Figura 4: Visualização do Flow Field — setas indicam a direção do fluxo calculado.

```
// Pré-processamento, executado uma vez, offline
função CONSTRUIR_GRAFO_ABSTRATO(mapa, tamanho_cluster):
    clusters <- dividir mapa em regiões de tamanho_cluster x tamanho_cluster
    G_abs <- grafo vazio

    para cada par de clusters adjacentes (C1, C2):
        entradas <- células na fronteira entre C1 e C2
        para cada entrada e em entradas:
            G_abs.adicionar_nó(e)
        para cada par (e1, e2) de entradas no mesmo cluster C:
            caminho <- A_ESTRELA(mapa restrito a C, e1, e2)
            se caminho não vazio:
                G_abs.adicionar_aresta(e1, e2, custo=len(caminho))
    retornar G_abs

// Consulta, executada a cada pedido de caminho
função HPA_ESTRELA(mapa, G_abs, origem, destino):
    // Conectar temporariamente origem e destino ao grafo abstrato
    para cada entrada e no cluster de origem:
        caminho <- A_ESTRELA(mapa restrito ao cluster, origem, e)
        G_abs.adicionar_aresta_temp(origem, e, custo=len(caminho))
    para cada entrada e no cluster de destino:
        caminho <- A_ESTRELA(mapa restrito ao cluster, e, destino)
        G_abs.adicionar_aresta_temp(e, destino, custo=len(caminho))

    caminho_abstrato <- A_ESTRELA(G_abs, origem, destino)
    se caminho_abstrato vazio:
        G_abs.remover_arestas_temp()
        retornar []

    caminho_final <- []
    para cada par consecutivo (u, v) em caminho_abstrato:
        segmento <- A_ESTRELA(mapa restrito ao cluster, u, v)
        caminho_final.append(segmento)

    G_abs.remover_arestas_temp()
    retornar caminho_final
```

Listing 3: Pseudocódigo do HPA*

```
// Construção do campo, uma vez por objetivo
função CONSTRUIR_FLOW_FIELD(mapa, destino):
    para toda célula n em mapa:
        custo[n] <- infinito
    custo[destino] <- 0

    fila <- fila_de_prioridade()
    fila.inserir(destino, prioridade=0)

    enquanto fila não estiver vazia:
        u <- fila.remover_mínimo()
        para cada vizinho v de u (não bloqueado):
            c <- custo[u] + custo_terreno(v)
            se c < custo[v]:
                custo[v] <- c
                fila.inserir(v, prioridade=c)

    para toda célula u não bloqueada:
        melhor <- vizinho de u com menor custo[vizinho]
        direção[u] <- vetor_unitário(u -> melhor)

    retornar direção

// Consulta por agente O(1)
função PRÓXIMO_PASSO(agente, direção):
    retornar direção[agente.posição_atual]
```

Listing 4: Pseudocódigo do *Flow Field*

```
+-----------------------------------------------------+
|                  Aplicacao (main)                   |
|         Application::run() -- despacha o modo       |
+--------------+--------------+-----------------------+
|  GUI / Game  |  Benchmark   |  CLI (solve/generate) |
|  (SFML)      |  (headless)  |  (terminal)           |
+--------------+--------------+-----------------------+
|              Pathfinding (solvers)                  |
|   Dijkstra | A* | HPA* | FlowField  <- PathSolver   |
+-----------------------------------------------------+
|                   Core                              |
|         Grid | MapData | WorldState | Agent         |
+-----------------------------------------------------+
```

Listing 5: Diagrama de camadas do *framework*

```
-- meta
WIDTH=200
HEIGHT=100
START=S
GOAL=G

-- costs
.=10   g=15   w=40   m=80   S=5   G=5   #=-1

-- grid
########...
##S.....#..
...
..........G
```

Listing 6: Estrutura simplificada do formato de mapa

```
class PathSolver {
public:
    virtual std::string_view    name() const = 0;
    virtual std::vector<int>    compute(int from, int to) = 0;
    virtual void                onGridChanged();
    virtual void                onTilesChanged(const std::vector<int>& tiles);
    virtual long                lastNodesExpanded() const;
    virtual SolverTiming        lastTiming() const;
};
```

Listing 7: Interface `PathSolver`

```
std::unique_ptr<PathSolver> create_solver(const std::string& nome,
                                          const WorldState& world);
```

Listing 8: *Factory function* dos *solvers*

```
-- 02.map
-- meta
WIDTH=50
HEIGHT=20
START=S
GOAL=G

-- costs per tile type
-- path
.=10
-- grass
g=15
-- water
w=40
-- mountain
m=80
-- start
S=5
-- goal
G=5
-- wall
#=-1

-- grid
S....ggggg.....................mmmmmmmmmmmmmm...#
.....g...g.....................m...........m...#
######...g..##################..m..########..m...#
#........g..#..............#..m..#......#..m...#
#..#######..#..wwwwwwwwwww..#..m..#.mmmm.#..m...#
#..#.....#..#..w.........w..#..m..#.m..m.#..m...#
#..#..#..#..#..w..######..w..#..m..#.m..m.#..m...#
#..#..#..#..#..w..#....#..w..#..m..#.mmmm.#..m...#
#..#..#..#..#..w..#....#..w..#..m..#......#..m...#
#..#..#..#..#..w..######..w..#..m..########..m...#
#..####..#..#..w.........w..#..m...........m...#
#........#..#..wwwwwwwwwww..#..mmmmmmmmmmmmmm...#
##########..#..............#...................#
...........##################..##################
mmmmmmmmmm..................g..................#
m........m..wwwwwwwwwwwww...g...wwwwwwwwwwww...#
m..####..m..w...........w...g...w...........w...#
m..#..#..m..w..########..w...g...w..#######..w...#
m..####..m..w..#......#..w...g...w..#.....#..w...G
m........m..w..########..w...g...w..#######..w...#
```

Listing 9: Exemplo da estrutura de mapa carregada pelo framework

```
scenario,solver,num_agents,fixed_ticks,total_ticks,wall_time_ms,sim_fps,cpu_time_ms,peak_memory_kb,total_computes,total_nodes_expanded,avg_nodes_expanded,avg_compute_ms,p50_compute_ms,p90_compute_ms,p95_compute_ms,p99_compute_ms,max_compute_ms,stuck_count,hpa_search_ms,hpa_cluster_update_ms,hpa_refine_ms,hpa_rebuild_ms,hpa_total_solver_ms,agents_reached,avg_path_length,total_replans
static,dijkstra,1,1,50000,2.39359,2.08891e+07,2.403,7104,1,2856,2856,0.762299,0.762299,0.762299,0.762299,0,0,0,0,0,0,1,806,0
static,astar,1,1,50000,2.37831,2.10233e+07,2.38,7104,1,2843,2843,0.809198,0.809198,0.809198,0.809198,0,0,0,0,0,0,1,806,0
static,hpa,1,1,50000,1.93837,2.57948e+07,1.94,7488,1,488,488,0.329218,0.329218,0.329218,0.329218,0.329218,0.329218,0,0.225563,0,0.101229,0,0.327368,1,806,0
static,flow,1,1,50000,3.67985,1.35875e+07,3.682,7680,1,9715,9715,1.83336,1.83336,1.83336,1.83336,1.83336,1.83336,0,0,0,0,0,0,1,806,0
dyn-sparse,dijkstra,10,1,50000,117.42,425821,117.369,7680,561,33818,60.2816,0.112998,0.095959,0.182266,0.205482,0.237394,1.162,554,0,0,0,0,0,2,243,276
dyn-sparse,astar,10,1,50000,111.636,447883,111.639,7680,561,33794,60.2389,0.106298,0.095729,0.126922,0.179579,0.222522,0.850198,554,0,0,0,0,0,2,243,276
dyn-sparse,hpa,10,1,50000,4593.17,10885.7,4591.47,7680,561,5845,10.4189,0.0682174,0.062278,0.095705,0.11063,0.160626,0.297123,554,6.97228,0,30.5232,4487.64,4525.38,2,243,276
dyn-sparse,flow,10,1,50000,60.7885,822524,60.793,7680,561,40216,71.6863,0.0210195,0.000247,0.041674,0.048137,0.065567,1.85282,554,0,0,0,0,0,2,243,276
dyn-dense,dijkstra,50,1,50000,57.9257,863174,57.93,7680,68,6094,89.6176,0.114899,0.101326,0.133939,0.140353,0.836235,0.836235,66,0,0,0,0,0,1,408,276
dyn-dense,astar,50,1,50000,53.5994,932847,53.604,7680,68,6081,89.4265,0.108136,0.094184,0.115772,0.125507,0.784666,0.784666,66,0,0,0,0,0,1,408,276
dyn-dense,hpa,50,1,50000,4605.31,10857,4605.29,7680,68,1145,16.8382,0.0804818,0.068956,0.137946,0.149538,0.28414,0.28414,66,1.15523,0,4.19529,4537.37,4542.76,1,408,276
dyn-dense,flow,50,1,50000,51.4424,971961,51.447,7680,68,17234,253.441,0.061675,0.034135,0.052286,0.057722,1.78266,1.78266,66,0,0,0,0,0,1,408,276
dyn-moving-goal,dijkstra,50,1,50000,61.7441,809794,61.75,7680,108,8230,76.2037,0.109718,0.09334,0.123899,0.14365,0.23698,1.05127,107,0,0,0,0,0,1,806,442
dyn-moving-goal,astar,50,1,50000,59.8816,834980,59.886,7680,108,8217,76.0833,0.103133,0.094533,0.10922,0.120033,0.135908,0.816532,107,0,0,0,0,0,1,806,442
dyn-moving-goal,hpa,50,1,50000,4710.16,10615.4,4710.09,7680,108,1593,14.75,0.0641429,0.059577,0.091154,0.105212,0.161204,0.329133,107,1.59992,97.0523,5.18039,4543.41,4647.3,1,806,442
dyn-moving-goal,flow,50,1,50000,49.0532,1.0193e+06,49.056,7680,108,15553,144.009,0.0403606,0.022698,0.035168,0.038806,0.056363,1.74625,107,0,0,0,0,0,1,806,442
```

Listing 10: Resultados completos — labirinto (`results_maze.csv`)

```
scenario,solver,num_agents,fixed_ticks,total_ticks,wall_time_ms,sim_fps,cpu_time_ms,peak_memory_kb,total_computes,total_nodes_expanded,avg_nodes_expanded,avg_compute_ms,p50_compute_ms,p90_compute_ms,p95_compute_ms,p99_compute_ms,max_compute_ms,stuck_count,hpa_search_ms,hpa_cluster_update_ms,hpa_refine_ms,hpa_rebuild_ms,hpa_total_solver_ms,agents_reached,avg_path_length,total_replans
static,dijkstra,1,1,50000,7.82259,6.39174e+06,7.81,7104,1,19994,19994,6.22914,6.22914,6.22914,6.22914,6.22914,0,0,0,0,0,0,1,260,0
static,astar,1,1,50000,7.9226,6.31106e+06,7.924,7296,1,19994,19994,6.38383,6.38383,6.38383,6.38383,6.38383,6.38383,0,0,0,0,0,0,1,260,0
static,hpa,1,1,50000,2.01153,2.48567e+07,2.014,7356,1,381,381,0.345317,0.345317,0.345317,0.345317,0.345317,0.345317,0,0.277579,0,0.065194,0,0.3441,1,264,0
static,flow,1,1,50000,7.83272,6.38348e+06,7.782,7356,1,20000,20000,6.24168,6.24168,6.24168,6.24168,6.24168,6.24168,0,0,0,0,0,0,1,260,0
dyn-sparse,dijkstra,10,1,50000,758.78,65895.2,758.652,7356,169,2045262,12102.1,3.85387,4.94877,6.26616,7.10689,8.25136,8.45338,0,0,0,0,0,0,10,127,276
dyn-sparse,astar,10,1,50000,726.958,68779.7,726.962,7356,170,1899969,11176.3,3.64973,4.59473,5.9690,4,6.21429,8.18378,8.75573,0,0,0,0,0,0,10,126.441,276
dyn-sparse,hpa,10,1,50000,10740.8,4655.17,10740.6,8124,181,27046,149.425,0.223844,0.208021,0.411959,0.449159,0.562809,0.571061,0,23.9078,0,16.4054,10576.4,10616.8,10,134.552,276
dyn-sparse,flow,10,1,50000,235.518,212298,235.523,8124,169,419016,2479.38,0.767972,0.001781,5.55836,5.69252,6.97999,9.19081,0,0,0,0,0,0,10,127.024,276
dyn-dense,dijkstra,50,1,50000,169.434,295101,169.424,8124,18,232962,12942.3,3.96724,5.22951,7.31941,7.55568,7.55568,7.55568,0,0,0,0,0,0,1,132.333,276
dyn-dense,astar,50,1,50000,177.276,282045,177.281,8124,18,221589,12310.5,4.26553,5.54963,6.74066,9.24457,9.24457,9.24457,0,0,0,0,0,0,1,132.333,276
dyn-dense,hpa,50,1,50000,10747.1,4652.42,10747,8124,19,3166,166.632,0.259843,0.201972,0.551552,0.600885,0.600885,0.600885,0,3.02266,0,1.86485,10627.8,10632.7,1,135.737,276
dyn-dense,flow,50,1,50000,207.55,240906,207.525,8316,18,343016,19056.4,6.12269,5.87748,7.29984,7.97424,7.97424,7.97424,0,0,0,0,0,0,1,132.333,276
dyn-moving-goal,dijkstra,50,1,50000,223.371,223843,223.375,8316,36,393410,10928.1,3.35874,4.17644,5.76711,6.90065,7.92766,7.92766,0,0,0,0,0,0,1,109.75,442
dyn-moving-goal,astar,50,1,50000,226.632,220622,226.637,8316,36,386368,10732.4,3.63051,4.70956,6.03929,8.83689,8.86423,8.86423,0,0,0,0,0,0,1,112.917,442
dyn-moving-goal,hpa,50,1,50000,10952.9,4565.02,10952.8,8316,34,5108,150.235,0.273792,0.25198,0.436802,0.564345,0.733016,0.733016,0,5.26097,210.223,3.97119,10620.2,10839.7,1,126.294,442
dyn-moving-goal,flow,50,1,50000,301.533,165819,301.537,8316,33,628031,19031.2,6.17039,5.72154,8.5767,8.93876,9.26146,9.26146,0,0,0,0,0,0,1,122.818,442
```

Listing 11: Resultados completos — campo aberto (`results_open.csv`)

```
scenario,solver,num_agents,fixed_ticks,total_ticks,wall_time_ms,sim_fps,cpu_time_ms,peak_memory_kb,total_computes,total_nodes_expanded,avg_nodes_expanded,avg_compute_ms,p50_compute_ms,p90_compute_ms,p95_compute_ms,p99_compute_ms,max_compute_ms,stuck_count,hpa_search_ms,hpa_cluster_update_ms,hpa_refine_ms,hpa_rebuild_ms,hpa_total_solver_ms,agents_reached,avg_path_length,total_replans
static,dijkstra,1,1,50000,3.18336,1.57067e+07,3.193,7104,1,5721,5721,1.61417,1.61417,1.61417,1.61417,1.61417,0,0,0,0,0,0,1,254,0
static,astar,1,1,50000,3.34276,1.49577e+07,3.344,7296,1,5612,5612,1.81864,1.81864,1.81864,1.81864,1.81864,0,0,0,0,0,0,1,254,0
static,hpa,1,1,50000,2.04594,2.44386e+07,2.049,7352,1,130,130,0.194114,0.194114,0.194114,0.194114,0.194114,0,0.145103,0,0.045303,0,0.191823,1,258,0
static,flow,1,1,50000,3.29524,1.51734e+07,3.297,7544,1,6014,6014,1.78057,1.78057,1.78057,1.78057,1.78057,0,0,0,0,0,0,1,254,0
dyn-sparse,dijkstra,10,1,50000,249.417,200468,249.286,7544,653,549881,842.084,0.324719,0.128207,1.0262,1.16929,1.91603,2.23109,582,0,0,0,0,9,110.859,276
dyn-sparse,astar,10,1,50000,251.673,198670,251.677,7544,657,522789,795.721,0.326749,0.126191,1.072,1.20578,1.62551,2.38298,574,0,0,0,0,10,107.88,276
dyn-sparse,hpa,10,1,50000,2613.44,19131.9,2613.42,7544,634,56815,89.6136,0.0805316,0.034074,0.271574,0.302875,0.444947,0.608214,560,36.2942,0,14.064,2516.29,2566.88,9,111.784,276
dyn-sparse,flow,10,1,50000,122.187,409209,122.181,7544,633,310133,489.942,0.134189,0.000157,0.5484,1.00301,1.21393,2.05075,560,0,0,0,0,9,111.438,276
dyn-dense,dijkstra,50,1,50000,38.7466,1.29043e+06,38.738,7544,23,17147,745.522,0.355794,0.234288,0.967925,0.942585,1.60839,1.60839,16,0,0,0,0,1,76.1429,276
dyn-dense,astar,50,1,50000,39.644,1.26122e+06,39.641,7544,20,15956,797.8,0.329223,0.175238,1.02599,1.7042,1.7042,1.7042,12,0,0,0,0,1,82.875,276
dyn-dense,hpa,50,1,50000,2544.97,19646.6,2544.94,7544,20,1063,53.15,0.0528251,0.036805,0.152106,0.21102,0.21102,0.21102,12,0.640221,0,0.387162,2503.8,2504.83,1,86.875,276
dyn-dense,flow,50,1,50000,35.5292,1.4073e+06,35.534,7544,23,27071,1177,0.303786,0.060237,0.845503,0.997729,1.67058,1.67058,16,0,0,0,0,1,76.1429,276
dyn-moving-goal,dijkstra,50,1,50000,43.9761,1.13698e+06,43.969,7544,39,28100,720.513,0.367201,0.206247,1.05526,1.30721,1.6242,1.6242,32,0,0,0,0,1,86.2857,442
dyn-moving-goal,astar,50,1,50000,44.2062,1.13106e+06,44.192,7544,33,28180,853.939,0.429305,0.212734,1.02117,1.79498,1.97716,1.97716,30,0,0,0,0,1,113.667,442
dyn-moving-goal,hpa,50,1,50000,2586.82,19328.8,2586.75,7544,31,2690,86.7742,0.0701022,0.034768,0.234224,0.252218,0.254244,0.254244,28,1.56837,38.4481,0.565238,2505.71,2546.31,1,102.333,442
dyn-moving-goal,flow,50,1,50000,39.2311,1.2745e+06,39.235,7544,39,26294,674.205,0.213152,0.039265,0.526606,1.70463,2.37156,2.37156,32,0,0,0,0,1,86.2857,442
```

Listing 12: Resultados completos — mapa de salas (`results_rooms.csv`)