IGOR HENRIQUE TORATI RUY

UM ESTUDO SOBRE METODOLOGIAS DE BUSCA POR CAMINHOS EM JOGOS DIGITAIS

LAVRAS – MG

ABRIL - 2022

IGOR HENRIQUE TORATI RUY

UM ESTUDO SOBRE METODOLOGIAS DE BUSCA POR CAMINHOS EM JOGOS DIGITAIS

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Graduação em Ciência da Computação, para a obtenção do título de Bacharel.

Prof. DSc. Mayron César de Oliveira Moreira
Orientador

LAVRAS – MG

ABRIL - 2022

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Ruy, Igor H. Torati

Um Estudo sobre Metodologias de Busca por Caminhos em Jogos Digitais / Igor Henrique Torati Ruy. 1^a ed. – Lavras : UFLA, Abril - 2022.

70 p. : il.

Monografia (TCC)–Universidade Federal de Lavras, 2022.
Orientador: Prof. DSc. Mayron César de Oliveira Moreira.
Bibliografia.

1. TCC. I. Universidade Federal de Lavras. II. Pathfinding em Jogos Digitais.

CDD-XXX.XXX

Dedico esse trabalho aos meus pais, por sempre me incentivar a estudar e me ensinar a dar valor ao que realmente importa. Amo vocês dois e lhes devo tudo.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, por sempre me apoiar, e por me incentivar a estudar, sem eles, não estaria onde estou.

Agradeço aos meus colegas por me aturarem e por fazerem parte desta etapa da minha vida, espero que conquistem tudo o que desejam.

Agradeço ao meu orientador por me auxiliar e corrigir intermináveis vezes este trabalho.

Agradeço à Universidade Federal de Lavras e todos os seus colaboradores, agradeço todo o conhecimento que compartilharam e por todo o trabalho que desempenharam ao longo destes anos.

Por fim, mas não menos importante, agradeço ao RPG de mesa por manter minha sanidade em meio às semanas de entrega de trabalho e provas.

It's the questions we can't answer that teach us the most. They teach us how to think. If you give a man an answer, all he gains is a little fact. But give him a question and he'll look for his own answers.

(Patrick Rothfuss)

RESUMO

A busca por caminhos (comumente conhecida pelo termo em inglês: *Pathfinding*) é um desafio existente não apenas em contextos práticos, mas também em jogos digitais. Trata-se de um tema complexo, presente no desenvolvimento de Inteligências Artificiais para agentes em jogos. Embora o problema da busca por caminhos seja tratável por estruturas conhecidas por sua flexibilidade e eficiência, tais como modelos em grafos, várias empresas ainda reportam dificuldades em implementar soluções eficientes. Este trabalho tem como principal contribuição o estudo de alguns dos algoritmos e estratégias mais aplicadas ao contexto de busca de caminhos em jogos digitais, de modo a demonstrar e explicar o benefício que trazem na busca de uma solução. Além de estudar os algoritmos, eles também foram implementados em uma ferramenta desenvolvida ao longo do trabalho, essa ferramenta foi desenvolvida com a finalidade de facilitar a compreensão dos algoritmos e permite criar cenários 2d e verificar o funcionamento do processo de busca de caminhos de cada um dos algoritmos analisados. É esperado que esse trabalho sirva como um guia para pessoas interessadas nessa temática, abrangendo o estado da arte e os desafios existentes, bem como novas estratégias que estão sendo desenvolvidas nessa área.

Palavras-chave: Busca por Caminhos. Grafos. Algoritmos. Jogos Digitais.

ABSTRACT

Pathfinding is a current challenge, not only in practical contexts but also in digital games. It is a complex theme present in the development of Artificial Intelligence for agents in games. Although the pathfinding problem can be dealt with using structures known for flexibility and efficiency, such as graph models, many companies still say it is demanding to implement efficient solutions. This work contributes by studying algorithms and strategies applied to the context of pathfinding in digital games; this will be covered by demonstrating and explaining the benefit that some techniques bring in searching for a solution. In addition to studying the algorithms, they were also implemented in a tool developed throughout the work, this tool was developed with the purpose of facilitating the understanding of the algorithms and allows creating 2d scenarios and verifying the functioning of the path search process of each one of the analyzed algorithms. This work is expected to serve as a guide to those interested in the theme, covering state of the art and some existing challenges and showing some new strategies being developed in this field.

Keywords: Pathfinding. Graphs. Algorithms. Digital Games.

LISTA DE FIGURAS

Figura 3.1 – Exemplo Grade Regular 24
Figura 3.2 – Exemplo Grade Irregular 24
Figura 3.3 – *Unreal Engine 3* etapa de exploração. 28
Figura 3.4 – *Unreal Engine 3*: etapa de exploração, subdivisão da grade 29
Figura 3.5 – *Unreal Engine 3* etapa de simplificação da malha, Passo 1. 30
Figura 3.6 – *Unreal Engine 3* etapa de simplificação da malha, passo 2. 31
Figura 3.7 – *Unreal Engine 3* etapa de simplificação da malha, passo 2. Simplificação de bordas ativada 32
Figura 3.8 – *Unreal Engine 3* malha obtida após o processo de geração da *NavMesh* . . 33
Figura 4.1 – Tela inicial da ferramenta de visualização 36
Figura 4.2 – Ferramenta exibindo caminho encontrado 38

LISTA DE TABELAS

Tabela 4.1 – Comparativo percentual em relação ao melhor custo. 40
Tabela 4.2 – Comparativo percentual em relação à solução com menos vértices visitados. 40

SUMÁRIO

1 Introdução 9
1.1 Objetivos 10
1.2 Motivação e Justificativa 10
1.3 Organização do Trabalho 11
2 Algoritmos de descoberta de caminhos 12
2.1 Algoritmo de Busca Primeiro-Melhor 12
2.2 Algoritmo de Busca em Largura 14
2.3 Algoritmo de Busca em Profundidade 15
2.4 Algoritmo A* 16
2.5 Algoritmo de Dijkstra 20
2.6 Considerações Gerais 21
3 Estratégias Utilizadas 23
3.1 Grades 23
3.2 HPA* 25
3.3 Malha de Navegação 26
3.3.1 Etapas de Geração da *NavMesh* na *Unreal Engine 3*. 27
3.3.1.1 Exploração 28
3.3.1.2 Simplificação da Malha 29
3.3.1.3 Finalização da Malha 32
3.4 Comentários Gerais 33
4 Experimentos Computacionais 35
4.1 Ferramenta de Visualização 35
4.2 Base de Dados 39
4.3 Resultados Obtidos 39
5 CONCLUSÃO 42
REFERÊNCIAS 43
APENDICE A – Cenários de teste e resultados obtidos 46

1 INTRODUÇÃO

A busca por caminhos (comumente conhecida pelo termo em inglês: *Pathfinding*) é um desafio existente não apenas em contextos práticos, mas também em jogos digitais. Esse problema é parte da funcionalidade de jogos eletrônicos, e consiste em encontrar um caminho entre um ponto A e um ponto B presente em mapas que modelam o deslocamento de personagens. Existem algumas restrições relativas ao percurso de entidades envolvidas em um jogo, tal como a existência de obstáculos que podem ser veículos ou mesmo um grupo de soldados, dependendo do estilo e do ambiente representado no jogo em questão.

Neste trabalho, serão abordados os algoritmos e estruturas de dados mais amplamente utilizados para representação e abstração dos mapas de um jogo, tal como definido por (ISKANDAR; DIAH; ISMAIL, 2020) e (ALGFOOR; SUNAR; KOLIVAND, 2015). Entre os assuntos estudados, estão: (i) o Algoritmo A* (HART; NILSSON; RAPHAEL, 1968), para a determinação de caminhos; (ii) o *HPA**, utilizado como uma das estratégias para acelerar o encontro de caminhos em mapas maiores; e (iii) a malha de navegação, ou *NavMesh*, como forma de representação das superfícies e cenários dos jogos.

Os algoritmos de busca de caminhos explorados neste trabalho funcionam para qualquer representação que possa ser explorada através de um grafo não direcionado. Dessa forma, não ficam limitados à exploração de ambientes 2D ou 3D, podendo ser utilizados em ambos os cenários. O funcionamento dos algoritmos também não é diferente, independente se estiverem sendo utilizados em mapas estáticos ou mapas dinâmicos. Porém, em mapas dinâmicos, o HPA* e a NavMesh precisarão de uma implementação mais robusta, que permita esse dinamismo sem que seja necessário recarregar o mapa todo.

O problema de busca de caminhos resolvido ao longo deste trabalho é definido da seguinte forma:

- Instância: Um grafo $G(V,E)$, não direcionado, que represente um território a ser percorrido. O mapa representado pode ser uma grade regular ou irregular. Um vértice $VI \in V$ representa o vértice de início, e o vértice $VD \in V$ representa o vértice de destino em G.

- Solução: Um conjunto de vértices S na ordem que são percorridos da origem ao destino, sendo $S(0)$ o VI, e $S(|S|-1)$ o VD.

Uma outra definição, utilizada especificamente para a estratégia HPA*, é:

- Instância: Um grafo $G(V,E)$, não direcionado, que represente um mapa a ser percorrido e um grafo $Gh(Vh,Eh)$, em que cada vértice $vh \in Vh$ representa uma componente conexa de G, e cada arco $eh \in Eh$ representa que é possível viajar de um vértice $vh1 \in Vh$ a outro vértice $vh \in Vh$, indicando que é viável ir de uma componente conexa à outra. Além disso, tem-se um vértice $VI \in V$ do grafo, representando o vértice de inicio, e um $VD \in V$, representando o vértice de destino em G.

- Solução: Um conjunto de vértices S de tamanho $|S|$ na ordem que são percorridos da origem ao destino, sendo $S(0)$ o VI e $S(|S|-1)$ o VD.

Para facilitar a compreensão do funcionamento dos algoritmos, foi desenvolvida uma ferramenta ao longo deste trabalho. Ela permite a criação de cenários em 2D, definindo obstáculos, ponto de origem e ponto de destino. Com essa ferramenta, é possível analisar o funcionamento de cada algoritmo simultaneamente e comparar visualmente o caminho encontrado por cada algoritmo.

1.1 Objetivos

Por conta do grande desafio e complexidade que o tema de busca de caminhos traz, esse trabalho tem o objetivo de estudar alguns dos algoritmos e estratégias mais aplicados na área, de modo a demonstrar e explicar o benefício que trazem na busca de uma solução.

É esperado que esse trabalho sirva como uma orientação para novos estudantes deste tema, abrangendo a base do que é utilizado atualmente.

1.2 Motivação e Justificativa

Busca de caminhos é um tema complexo, indispensável na criação de Inteligências Artificiais para agentes em jogos digitais (GRAHAM ROSS; MCCABE; SHERIDAN, 2003). Embora o problema da busca por caminhos seja tratável em mapas representados por grafos (CORMEN et al., 2012, Capítulo 24), várias empresas ainda reportam dificuldades em implementar soluções eficientes, como mostrado em palestras do *Game Design Conference* [1].

A busca de caminhos em jogos apresenta inúmeros desafios, desde problemas técnicos e de desempenho até problemas de experiência do usuário. Segundo (ISKANDAR; DIAH;

[1] Palestras do GDC relacionadas à busca de caminhos estão disponíveis em: <https://www.gdcvault.com/search.php#&conference_id=&category=free&firstfocus=&keyword=pathfinding>

ISMAIL, 2020), mesmo sendo amplamente explorado até hoje, a busca de caminhos em jogos também é um problema em aberto, que ainda será estudado e trará novas técnicas e soluções no futuro.

1.3 Organização do Trabalho

No Capítulo 2, são descritos os algoritmos utilizados neste trabalho, bem como uma revisão de cada método e suas características. No Capítulo 3, é explicado sobre o *HPA** e a *NavMesh*, analisando cada uma dessas estratégias e suas características. No Capítulo 4, são apresentados um conjunto de experimentos computacionais, extraídos a partir da execução dos algoritmos de busca considerados. As soluções são ilustradas por meio de uma ferramenta desenvolvida neste estudo. Reporta-se, também, uma explicação sobre as características dos dados testados. No Capítulo 5, consta a conclusão e algumas possibilidades de trabalhos futuros.

2 ALGORITMOS DE DESCOBERTA DE CAMINHOS

Este capítulo apresenta cinco dos algoritmos mais utilizados na literatura de descoberta de caminhos em jogos (ISKANDAR; DIAH; ISMAIL, 2020). A Seção 2.1 apresenta o Algoritmo de Busca Primeiro-Melhor. Os algoritmos elementares de caminhamento em grafos, denominados Busca em Largura (MOORE, 1959; LEE, 1961) e Busca em Profundidade (TARJAN, 1972) são mostrados nas Seções 2.2 e 2.3, respectivamente. O Algoritmo A* (HART; NILSSON; RAPHAEL, 1968) e Algoritmo de Dijkstra (DIJKSTRA, 1959), dois dos algoritmos mais estudados para o cálculo de caminhos mais curtos (RACHMAWATI; GUSTIN, 2020) são apresentados nas seções 2.4 e 2.5, respectivamente. Comentários gerais a respeito dos algoritmos supracitados são apresentados na Seção 2.6.

2.1 Algoritmo de Busca Primeiro-Melhor

O Algoritmo de Busca Primeiro-Melhor é um algoritmo que não garante o encontro do melhor caminho, porém, é significativamente rápido em seu tempo de execução (ISKANDAR; DIAH; ISMAIL, 2020). Dos algoritmos explorados neste trabalho, este é o que encontra um caminho no menor tempo computacional.

> Este algoritmo é essencialmente superior aos algoritmos de Busca em Largura e Busca em Profundidade. É mais eficiente em encontrar uma solução que ambos esses algoritmos e permite trocar o caminho analisado, englobando as vantagens tanto do Algoritmo de Busca em Largura quanto do Algoritmo de Busca em Profundidade. (ISKANDAR; DIAH; ISMAIL, 2020)

Segundo (MEHTA et al., 2015), a Busca Primeiro-Melhor acompanha a fronteira da área explorada, afim de localizar o vértice de destino, ou seja, não considera o vértice inicial ao escolher os vértices explorados, mas sim, os vértices que estão no limite da área explorada. Além disso, faz uso de uma função heurística para determinar aproximadamente a distância até o objetivo, sendo que, os vértices mais próximos do destino possuem uma maior prioridade para serem explorados. Isso auxilia o algoritmo a encontrar um caminho até o vértice alvo rapidamente.

A implementação utilizada neste trabalho para o Algoritmo Primeiro-Melhor está descrita abaixo.

Algoritmo 1: *Busca Primeiro-Melhor*

```
1  Defina E(v) como o avaliador que calcula uma função heurística. Nesse caso, E(v)
     é a distância euclidiana do vértice avaliado v até o destino;
2  Defina PQ como uma fila de prioridade, sendo que o maior valor E(v) tem maior
     prioridade;
3  Defina d(e) como sendo a distância de uma aresta e=(v1,v2) que conecta dois
     vértices v1 e v2;
4  Defina c(v) como sendo o custo do vértice origem (VI) até um vértice 'v'. Assuma
     c(VI) = 0;
5  Defina VD como vértice de destino;
6  início
7      Calcule E(VI) e adicione VI à PQ;
8      enquanto PQ não estiver vazia faça
9          Remova o topo de PQ e defina esse vértice removido como VR;
10         Marque VR como visitado;
11         para cada vértice vizinho de VR, denotado por VV faça
12             se VV ainda não foi visitado então
13                 Defina VR como pai de VV;
14                 Defina e=(VR,VV);
15                 Defina c(VV) = c(VR) + d(e);
16                 se VV for o vértice VD então
17                     retorna VV
18                 senão
19                     se VV não estiver em PQ então
20                         Adicione VV à PQ com o valor E(VV) calculado;
21     se PQ estiver vazia então
22         Não foi encontrado caminho até o destino;
```

Para definir o caminho e o custo, todos os algoritmos apresentados neste trabalho seguem o mesmo padrão. Define-se como parâmetro o vértice retornado pelo algoritmo de busca, e considerando a lógica definida abaixo. O retorno da função abaixo (Algoritmo 2) contém

como dados o tamanho do caminho e uma lista de vértices, na ordem que cada vértice do caminho é visitado a partir da origem até o destino.

Algoritmo 2: *Definição do caminho da origem até o destino*

Entrada: Vértice retornado pelo algoritmo de busca de caminho, denominado VR.

Saída: Estrutura de dados R, contendo o valor c, custo do caminho percorrido e a pilha s, representando os vértices percorridos da origem até chegar ao destino, na ordem que são percorridos.

```
1  Defina uma estrutura de dados R na qual R.c representa o custo do caminho e R.s
     representa uma pilha de vértices percorridos, na ordem que deve ser seguida
     partindo da origem, de modo a chegar ao destino;
2  Defina R.c =0;
3  Defina R.s como vazia;
4  início
5  |  V = VR;
6  |  Faça R.c = R.c + c(VR);
7  |  enquanto V possuir pai faça
8  |  |  Adicione V a R.s;
9  |  |  Atribua o pai de V a V;
10 |  retorna R
```

Com o retorno dessa função, basta desempilhar os valores da pilha para obter o caminho percorrido para chegar ao destino.

2.2 Algoritmo de Busca em Largura

O Algoritmo de Busca em Largura (*BFS*, do inglês: *Breadth-first Search*) realiza sua busca visitando os nós em pré-ordem, explorando primeiro um nó, e então todos os seus filhos (RAHIM et al., 2018). A ideia base desse algoritmo é explorar, a partir da raiz, os nós na ordem que são descobertos, visitando primeiro um vértice, depois os vizinhos dele e em seguida os vizinhos dos vizinhos, até que o objetivo tenha sido encontrado ou todo o grafo tenha sido explorado (FEOFILOFF, 2019).

A implementação utilizada neste trabalho para o Algoritmo de Busca em Largura está descrita abaixo.

Algoritmo 3: *Busca em Largura*

```
1  Defina L como uma fila de vértices;
2  Defina c(v) como sendo o custo do vértice origem (VI) até um vértice 'v'. Assuma
     c(VI) = 0;
3  Defina d(e) como sendo a distância de uma aresta e=(v1,v2), que conecta dois
     vértices v1 e v2;
4  Defina VD como vértice de destino;
5  início
6      Adicione VI a L e o marque como visitado;
7      enquanto L não for vazia faça
8          Remova o vértice VR, com maior prioridade em L;
9          para cada vértice vizinho de VR, denotado por VV faça
10             se VV ainda não foi visitado então
11                 Marque VV como visitado e o adicione a L;
12                 Defina VR como pai de VV;
13                 Defina e=(VR,VV);
14                 Defina c(VV) = c(VR) + d(e);
15                 se VV for VD então
16                     retorna VV
17     se L estiver vazia então
18         Não foi encontrado caminho até o destino;
```

2.3 Algoritmo de Busca em Profundidade

O Algoritmo de Busca em Profundidade (*DFS*, do inglês: *Depth-first Search*), ao contrário do Algoritmo de Busca em Largura, progride sua busca expandindo de um único nó vizinho da raiz. A partir desse nós, o algoritmo aprofunda cada vez mais, analisando apenas um dos vizinhos do vértice, até que um vértice de destino seja encontrado, ou até que um nó sem filho seja alcançado. Nesse último caso, a busca regride ao último nó visitado que ainda possua um vizinho não explorado (KAUR; GARG, 2012).

A implementação utilizada neste trabalho para o Algoritmo de Busca em Profundidade está descrita abaixo.

Algoritmo 4: *Busca em Profundidade*

```
1  Defina P como uma pilha de vértices;
2  Defina c(v) como sendo o custo do vértice origem (VI) até um vértice 'v'. Assuma
    c(VI) = 0;
3  Defina d(e) como sendo a distância de uma aresta e=(v1,v2), que conecta dois
    vértices v1 e v2;
4  Defina VD como vértice de destino;
5  início
6      Adicione VI a P;
7      enquanto P não for vazia faça
8          Remova o topo de P e defina esse vértice removido como VR;
9          se VR não for VD então
10             para cada vértice vizinho de VR, denotado por VV faça
11                 Defina e=(VR,VV);
12                 Defina c(VV) = c(VR) + d(e);
13                 se VV ainda não foi visitado então
14                     Defina VR como pai de VV;
15                     Adicione VV a P;
16         senão
17             retorna VR
18     se P estiver vazia então
19         Não foi encontrado caminho até o destino;
```

2.4 Algoritmo A*

O Algoritmo A* é amplamente utilizado no problema de descoberta de caminhos pela comunidade científica (FOEAD et al., 2021). Sua eficiência, simplicidade, e modularidade são normalmente destacadas como pontos fortes comparado a outras opções (FOEAD et al., 2021).

Segundo (FOEAD et al., 2021), o A* foi exaustivamente testado e estudado ao longo dos anos, e consegue encontrar caminhos com mais eficiência que algoritmos de busca desinformada (*uninformed search*). Porém, não é certo que o caminho encontrado seja o ótimo.

O algoritmo A*, ao explorar um vértice, analisa seus vizinhos, calculando a função heurística para esses vizinhos. A implementação que utiliza-se aqui é semelhante à proposta por (HART; NILSSON; RAPHAEL, 1968), e considera a função heurística, denominada *f(x)*, como a soma da distância do vértice de origem até o vértice atual, valor denominado *g(x)*, e o valor calculado por uma função do vértice atual até o vértice de destino, valor denominado *h(x)*. Neste trabalho, adotou-se a distância Euclidiana para estimar essa distância entre o vértice atual e o destino, considerando a posição de ambos os vértices na grade.

Diferentemente da implementação mencionada no trabalho de (MEHTA et al., 2015), a implementação que utilizamos não encontra o melhor caminho. Isso ocorre, pois, segundo (PATEL, 2010), o algoritmo A* se comporta das seguintes formas em relação ao valor *h(x)* da heurística:

- Se a *h(x)* for zero, o A* se comporta como o algoritmo de Dijkstra.
- Se a *h(x)* for sempre menor que o custo de se mover até o destino, a solução do A* é garantidamente o melhor caminho.
- Se a *h(x)* for exatamente igual ao custo de se mover de um nó até o destino, então o A* seguira apenas esse caminho sem explorar nenhum outro nó.
- Se a *h(x)* é algumas vezes maior que o custo de se mover de um nó até o destino, a solução do A* não é garantidamente o melhor caminho. Porém, o algoritmo é executado mais rapidamente.
- Se a *h(x)* for um valor muito grande em relação à distância percorrida a partir da origem, então *g(x)* colabora muito pouco para o valor de *f(x)*, ao ponto de não afetar a solução e o algoritmo é executado como um *Primeiro-Melhor*.

Na implementação utilizada neste trabalho, o custo calculado pode ser, algumas vezes, maior que o custo real. Dessa forma, o algoritmo não encontra o melhor caminho, mas possui um desempenho melhor que o algoritmo de Dijkstra, como mostram as Tabelas 3.1 e 3.2 no Capítulo 3, Seção 3.

A última afirmação de (PATEL, 2010) é discutível, para a implementação que utilizamos do algoritmo A* e do algoritmo Primeiro-Melhor. Para o algoritmo Primeiro-Melhor não foi considerado a substituição do nó pai de um outro nó, caso o caminho seja menor que um caminho já encontrado até o nó filho. Dessa maneira, a região explorada terá uma grande semelhança, mas não será exatamente igual. O mesmo ocorre para o caminho encontrado.

Algoritmo 5: A*

```
1  Defina E(v) como a distância euclidiana de 'v' até o destino (VD);
2  Defina PQ como uma fila de prioridade, sendo que o menor valor E(v) tem maior
   prioridade;
3  Defina d(e) como sendo o comprimento de uma aresta e=(v1,v2) que conecta dois
   vértices;
4  Defina c(v) como sendo o custo da origem (VI) até um vértice 'v'. Assuma c(VI) =
   0;
5  Defina f(v) = E(v) + c(v);
6  início
7      Defina c(v) como sendo infinito para cada vértice;
8      Adicione VI à PQ e defina f(VI) = c(VI) + E(VI);
9      enquanto PQ não for vazia faça
10         Remova a primeira entrada de P e defina esse vértice removido como VR;
11         Marque VR como visitado;
12         se VR não for VD então
13             para cada vértice vizinho de VR, denotado por VV faça
14                 Defina e=(VR,VV);
15                 Defina temp = c(VR) + d(e) + E(VV);
16                 se temp < c(VV) então
17                     Defina c(VV) = c(VR) + d(e(VR,VV));
18                     Defina f(VV) = temp;
19                     Defina VR como pai de VV;
20                     se VV ainda não foi visitado então
21                         Adicione VV a PQ;
22         senão
23             retorna VR
24     se PQ estiver vazia então
25         Não foi encontrado caminho até o destino;
```

2.5 Algoritmo de Dijkstra

O Algoritmo de Dijkstra encontra o menor caminho a partir de um nó inicial até um nó destino. Isso é feito construindo-se um conjunto de nós que tenham a menor distância até a origem e explorando esses nós (ABLY et al., 2019). Esse algoritmo pode ser utilizado em grafos direcionados ou não, desde que os arcos não possuam peso negativo entre um vértice e outro.

Essa restrição de não permitir pesos negativos ocorre pois o algoritmo segue uma abordagem gulosa. A cada iteração, escolhe o vértice com menor valor de distância até a origem, analisando-se apenas o cenário atual. Dessa forma, um vértice já explorado não será explorado novamente e seu peso jamais será menor que o encontrado até então, uma vez que, sem a existência de arcos negativos, o caminho encontrado já é o menor caminho possível.

> O algoritmo de Dijkstra se baseia em visitar um vértice de cada vez, começando do vértice de início e explorando seus filhos mais próximos que ainda não foram explorados, esse processo segue em um loop que termina quando o vértice examinado é o alvo ou quando todos os vértices existentes são examinados. Do contrário, os vértices mais próximos do vértice examinado são adicionados à coleção para ser examinado. [...] Dijkstra encontra sem falhas o caminho mais curto do ponto de início até o destino. Porém, o uso deste algoritmo para um único alvo ou destino não é recomendado, pois, ele consome tempo e recursos adicionais para o número de vértices analisados pelo algoritmo. [...] (MEHTA et al., 2015)

Algoritmo 6: Dijkstra

```
1  Defina c(v) como sendo o custo da origem (VI) até um vértice 'v';
2  Defina PQ como uma fila de prioridade, sendo que o menor valor c(v) tem maior
     prioridade;
3  Defina d(e) como sendo o comprimento de uma aresta e=(v1,v2) que conecta dois
     vértices v1 e v2;
4  Defina VD como vértice de destino;
5  início
6      Defina, inicialmente, o c(v) de cada vértice como sendo infinito;
7      Adicione VI à PQ, e defina c(VI) = 0;
8      enquanto PQ não for vazia faça
9          Remova a primeira entrada de P e defina esse vértice removido como VR;
10         se VR não foi visitado então
11             Marque VR como visitado;
12             se VR não for VD então
13                 para cada vértice vizinho de VR, denotado por VV faça
14                     Defina e=(Vr,VV);
15                     Defina temp = c(VR) + d(e);
16                     se temp < c(VV) então
17                         Defina VR como pai de VV;
18                         Adicione VV à PQ;
19             senão
20                 retorna VR
21     se PQ estiver vazia então
22         Não foi encontrado caminho até o destino;
```

2.6 Considerações Gerais

Dos algoritmos demonstrados aqui, o Algoritmo Primeiro-Melhor utilizado consegue um resultado que, em geral, é melhor que os Algoritmos de Busca em Largura e Busca em Profundidade (ISKANDAR; DIAH; ISMAIL, 2020). Além disso, o Algoritmo Primeiro-Melhor,

Dijkstra e A* possuem características que os tornam superiores aos outros dois, sendo que o A* é uma união da característica de expansão do Algoritmo de Dijkstra com o acréscimo da função heurística do Algoritmo do Primeiro-Melhor (ISKANDAR; DIAH; ISMAIL, 2020) (PERMANA et al., 2018).

3 ESTRATÉGIAS UTILIZADAS

Ao se realizar a busca de caminhos em jogos digitais, um dos fatores que devem ser levados em consideração, além do tempo para se encontrar a solução e o caminho encontrado, é o gasto de recursos computacionais como memória e processamento. Algoritmos como o A* gastam mais recursos dependendo do tamanho da busca. Dessa forma, a busca de caminhos em mapas muito grandes podem gerar problemas de desempenho (BOTEA; MüLLER; SCHAEFFER, 2004).

Como forma de amenizar problemas de desempenho em mapas grandes, existem duas estratégias comumente utilizadas (PELECHANO; FUENTES, 2016), denominadas: (i) *Busca de Caminhos A* Hierárquica* (conhecida popularmente pelo seu nome em inglês: *Hierarchycal Pathfinding A**, ou HPA*); e a *Malha de Navegação* (do inglês: *Navigation Mesh* - *NavMesh*). Para melhor compreendimento das características apresentadas por grades (*grids*) que modelam jogos digitais, alguns exemplos serão demonstrados na Seção 3.1. As estratégias *HPA** e *NavMesh* são exploradas nas Seções 3.2 e 3.3, respectivamente. Na seção 3.4 constam alguns comentários gerais sobre as estratégias analisadas.

3.1 Grades

Uma grade é composta de vértices conectados por arestas, de modo a representar um grafo (MA et al., 2011), e pode ser Regular ou Irregular. Essas categorias são definidas da seguinte forma:

- Grades Regulares:

> Compõem um dos tipos mais conhecidos e utilizados de grafos em jogos de computador e robótica. Existem vários desenvolvedores de jogos que exploraram esta área, desenvolvendo jogos como *Dawn of War 1 e 2*, *Civilization V* e *Company of Heroes* [...] Em ambientes 2D e 3D, grades regulares descrevem o uso de polígonos regulares (polígonos equiláteros e equiangulares). Hexágonos, quadrados, e triângulos são os únicos polígonos regulares que podem ser utilizados para a criação de ambientes 2D contínuos e grades cúbicas 3D.(ALGFOOR; SUNAR; KOLIVAND, 2015)

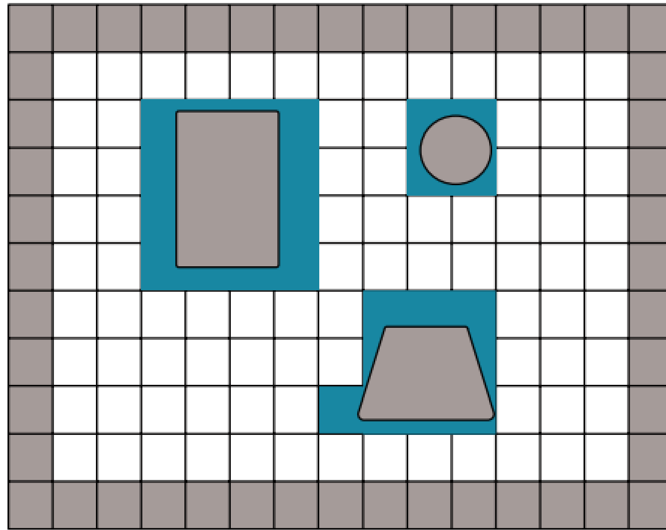
- Grades Irregulares:

> Cada célula é normalmente representada por um círculo ou polígono convexo e representa uma região sem obstáculos, em que as entidades podem percorrer em linha reta quaisquer dois pontos dentro da mesma

> célula (sem necessidade de busca de caminhos). (ALGFOOR; SUNAR; KOLIVAND, 2015)

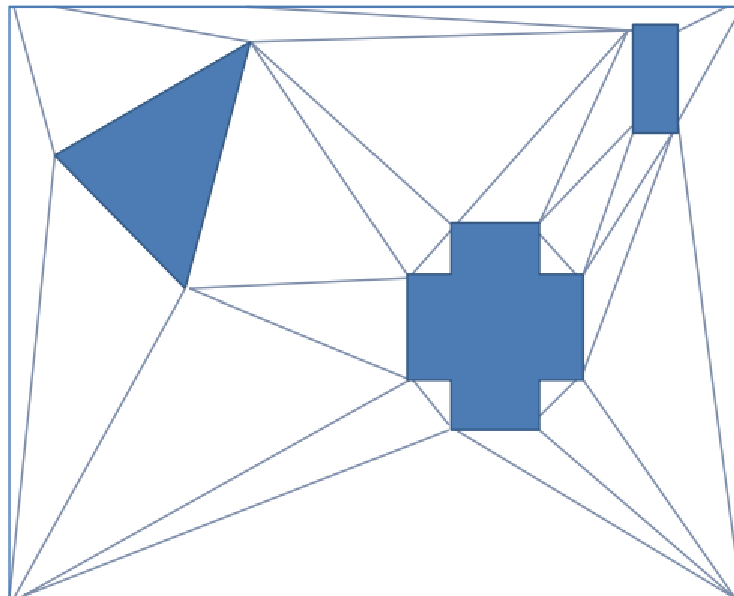
Dessa forma, ambas as demonstrações abaixo representam grades, sendo a Figura 3.1 uma grade Regular e a Figura 3.2 uma grade Irregular.

Figura 3.1 – Exemplo Grade Regular



Fonte: (ALGFOOR; SUNAR; KOLIVAND, 2015)

Figura 3.2 – Exemplo Grade Irregular



Fonte: (ALGFOOR; SUNAR; KOLIVAND, 2015)

3.2 HPA*

Estratégia foi criada por (BOTEA; MüLLER; SCHAEFFER, 2004), definida originalmente pelos autores como:

> O *HPA** é uma abordagem utilizada para reduzir a complexidade de um problema de busca de caminhos em um mapa de grade. Essa técnica abstrai um mapa em grupos locais interligados. Ao nível local, as distâncias para se cruzar cada grupo é pré-computada e armazenada em memória e ao nível global, os clusters são percorridos de uma só vez. Uma hierarquia pode ser estendida para mais de dois níveis e pequenos clusters podem ser agrupados para formar clusters maiores. O cálculo de distâncias para se percorrer um cluster maior usa as distâncias calculadas para os menores clusters contidos nele (BOTEA; MüLLER; SCHAEFFER, 2004).

A estratégia de agrupar cada nó é recomendável, e se baseia na eficiência e flexibilidade em lidar com mapas de jogos e também mapas dinâmicos, sem necessidade do uso de conhecimentos específicos. Sua implementação é considerada de fácil manejo, e pode ser aprimorada se desejado (DUC; SIDHU; CHAUDHARI, 2008).

Segundo (JANSEN; BURO, 2007), esse aumento de eficiência é possível pois o *HPA** é uma combinação de busca de caminhos e clusterização. Seu princípio é dividir um problema de busca em vários problemas menores, solucioná-los individualmente e armazenar as soluções encontradas para cada um desses seguimentos.

(ZAREMBO; KODORS, 2015) define em seu trabalho a estratégia de clusterização, aplicada a esse contexto, da seguinte forma:

> A clusterização do *HPA** é feita de forma simples. Cria-se uma grade $s \times s$, com uma resolução menor que a grade original, sendo s o tamanho da nova grade. A nova grade é então sobreposta à grade antiga e todos os nós contidos em uma célula desta nova grade são considerados membros daquele cluster. Cada cluster é considerado um grafo separado e um grafo abstrato é criado para conectar todos eles. Para se fazer isso, os nós da fronteira de cada cluster são verificados, e, caso seja possível alcançar um nó em um cluster vizinho, ambos são considerados conexos. Essas conexões entre dois clusters são adicionadas ao grafo abstrato (ZAREMBO; KODORS, 2015).

Um exemplo de jogo que utilizou inicialmente o A* e depois aprimorou para o *HPA** é o *Factorio*[1], um jogo de sobrevivência e construção com foco na automação da produção e coleta de recursos. Esse jogo conta com mais de 5 milhões de cópias vendidas, segundo o site *SteamSpy*.[2] No *Factorio*, o terreno é composto por células, formando uma grade e um conjunto

[1] O site do jogo está disponível através do endereço <https://factorio.com/>. Acessado em 12/04/2022
[2] Essa informação está disponível em: <https://steamspy.com/app/427520>. Acessado em 05/03/2022.

de 32×32 células, chamado pelos desenvolvedores de *chunk* (traduzido como *pedaço*, em uma tradução livre). Inicialmente o jogo utilizava o algoritmo A* para realizar a busca de caminho. Porém, problemas com o desempenho desse algoritmo ocorriam ao tentar mover o personagem até uma posição distante do ponto de partida.

Diante disso, houve a necessidade de se aprimorar a estratégia de busca, e a solução escolhida foi uma abordagem via *HPA**.

A geração dos clusters que irão representar o mapa na hierarquia superior foi feito através da geração de componentes conexas de cada *(*chunk). Para cada *(*chunk), foi gerado um ou mais subgrafos, sendo que, para um vértice x estar no mesmo subgrafo que um vértice y, ele deve ser capaz de alcançar esse vértice sem que seja necessário sair do cluster. Se isso não for possível, x e y estão em componentes conexas diferentes.

Cada uma dessas componentes conexas foi representada na hierarquia superior por algo que os desenvolvedores denominaram 'nó abstrato'. Dessa forma, o A* foi utilizado para procurar um caminho no grafo formado por esses nós abstratos e para cada nó abstrato, o caminho percorrido de uma *chunk* à outra era calculado na grade real, caracterizando assim, a estratégia *HPA**.

Em decorrência da não divulgação de dados relativos a testes realizados pela empresa, com a abordagem *HPA**, não foi possível comparar numericamente o ganho obtido. Porém, na publicação original da equipe de desenvolvimento do jogo [3], é mostrado um vídeo de como era feito o encontro do caminho anteriormente. Esse vídeo não foi desacelerado e demorou aproximadamente 15 segundos para o caminho ser encontrado. Posteriormente, foi afirmado que a solução com *HPA** demorou apenas alguns *ticks* [4] para ser encontrado, e o vídeo mostrado do *HPA** foi desacelerado para maior facilidade de visualização.

3.3 Malha de Navegação

A malha de navegação divide o espaço alcançável de uma superfície em um conjunto de polígonos convexos (CUI; HARABOR; GRASTIEN, 2017). Assim, utilizando a definição de (ALGFOOR; SUNAR; KOLIVAND, 2015) sobre grades irregulares, é possível entender que

[3] Essa publicação está disponível em: <https://factorio.com/blog/post/fff-317>. Acessado em 05/03/2022.

[4] *Tick* é uma unidade comum quando o assunto é tempo em jogos digitais, diferente de um *frame*. O *tick* representa um ciclo completo na lógica do jogo, é comum que jogos possuam um número muito maior de *tick*s por segundo do que *frames* por segundo.

a *NavMesh* é de fato uma grade irregular, sendo possível alcançar qualquer posição em um mesmo polígono apenas traçando uma linha reta. Ao contrário de grades regulares, malhas de navegação oferecem representações com maior precisão em relação à geometria, ao mesmo tempo que contém um número menor de células (PELECHANO; FUENTES, 2016).

O uso de malhas de navegação se tornou muito popular para jogos 3D, visto que ambientes em 3D comumente utilizam polígonos para serem representados (ZIKKY, 2016). Assim sendo, a malha de navegação possui a mesma finalidade da grade regular que fora apresentada até o momento: representar o espaço que pode ser percorrido pelas entidades.

Por ser uma técnica comum em jogos 3D, vários motores de desenvolvimento de jogos oferecem a possibilidade de se gerar e trabalhar com malhas de navegação, como é o caso da Unity3d [5] e da *Unreal Engine* [6].

As formas de se gerar as malhas podem variar para cada motor de desenvolvimento. Também podem ser utilizadas técnicas personalizadas criadas pela equipe de projeto. Neste trabalho, iremos explorar uma das formas de se gerar essa malha, utilizando como exemplo a forma que isso era feito na *Unreal Engine 3*[7]. É valido ressaltar que o motor *Unreal Engine* atualmente se encontra em sua quarta versão, lançada em 2014, e a quinta versão já está disponível desde o começo de 2021, apesar de não estar estável.

3.3.1 Etapas de Geração da *NavMesh* na *Unreal Engine 3*.

A ideia base da construção de uma *NavMesh* consiste em gerar os polígonos e o grafo que indica qual polígono possui vizinhança com os demais. Na *Unreal Engine 3*, o processo de geração da *NavMesh* é feito em três estágios: (*i*) exploração; (*ii*) simplificação da malha; e (*iii*) finalização da malha.

[5] As instruções de como gerar uma malha de navegação na Unity3d está disponível na documentação desta ferramenta: <https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html>. Acessado em 20/03/2022.

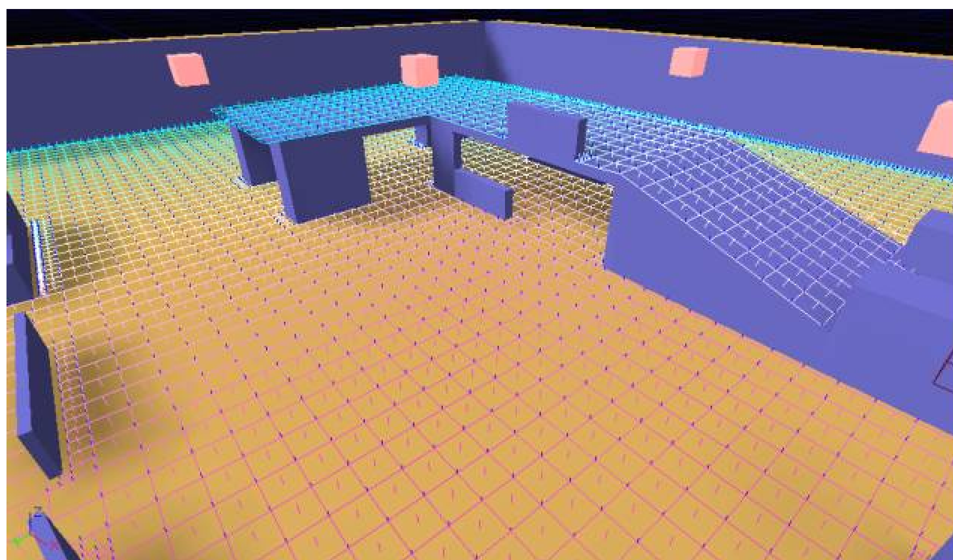
[6] As recomendações de como gerar uma malha de navegação na *Unreal Engine* está disponível na documentação desta ferramenta: <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/NavigationSystem/ModifyingTheNavigationMesh/ModifyingtheNavigationSystemPreparationGuide/>. Acessado em 20/03/2022.

[7] Esse exemplo consta na documentação da *Unreal Engine 3*, e está disponível em: <https://docs.unrealengine.com/udk/Three/NavigationMeshReference.html>. Acessado em 06/03/2022.

3.3.1.1 Exploração

No processo de exploração, o terreno é dividido em quadrados, com tamanho definido nas configurações de geração. Nessa etapa, o mapa se assemelha muito a uma grade, como mostrado na Figura 3.3.

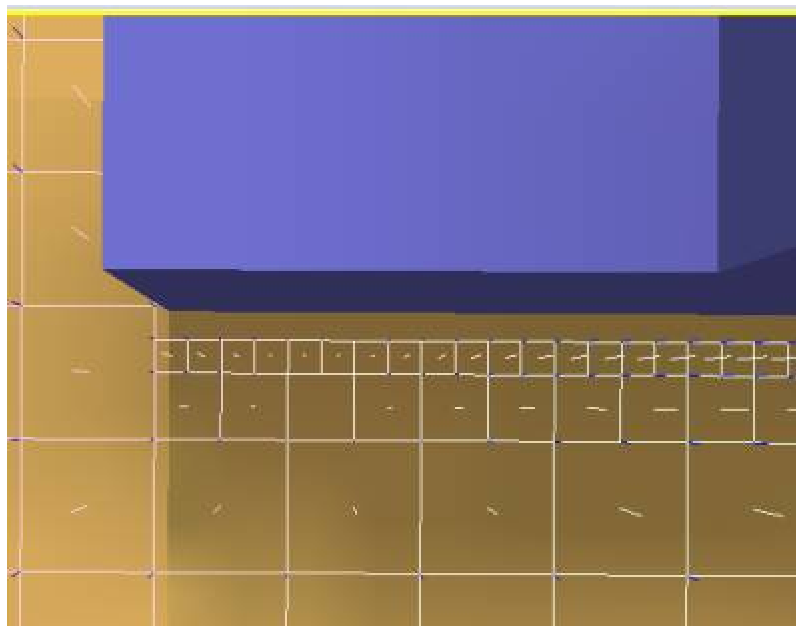
Figura 3.3 – *Unreal Engine 3* etapa de exploração.



Fonte: Unreal Engine 3, ano desconhecido.

O que diferencia essa divisão do terreno de uma grade regular é que, além de dividir o terreno em células do tamanho definido, a exploração tenta obter uma precisão maior. Dessa forma, quando um obstáculo não está exatamente alinhado com a grade, o tamanho da célula é dividida n vezes, sendo esse número de divisões definido nas configurações de geração da malha. Essa divisão alivia um dos fatores desfavoráveis do uso de grades regulares, que é a precisão na representação de uma quina ser sempre a mesma, fixada ao tamanho da célula. Um exemplo do resultado é mostrado na Figura 3.4.

Figura 3.4 – *Unreal Engine 3*: etapa de exploração, subdivisão da grade



Fonte: Unreal Engine 3, ano desconhecido.

Após essa primeira etapa, cada célula dessa grade é adicionada à malha e o processo segue para o próximo passo.

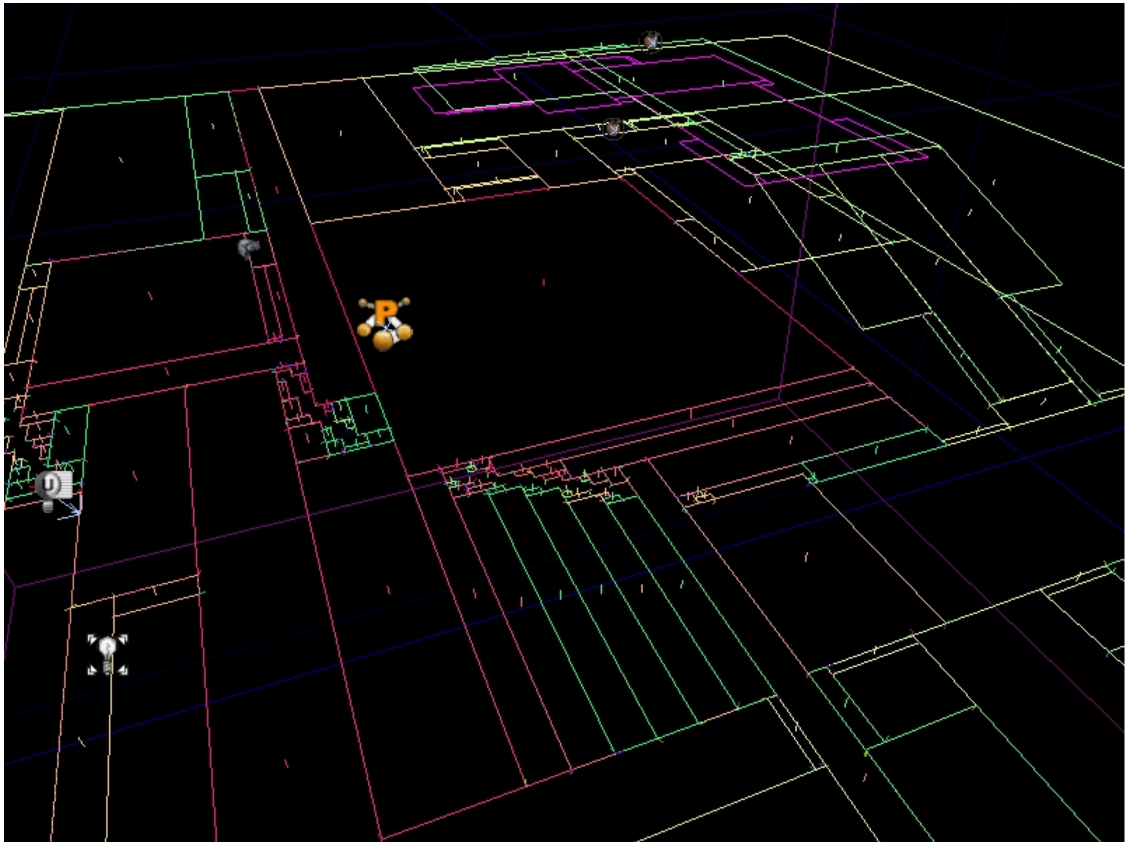
3.3.1.2 Simplificação da Malha

Na segunda etapa, tudo o que foi adicionado à malha será simplificado. Segundo a própria documentação, essa é a parte mais complexa, e foca em reduzir drasticamente o número de polígonos de uma malha. Essa etapa segue os seguintes passos:

1. Fundir os quadrados de modo a reduzir os polígonos e acelerar os passos seguintes.
2. Fundir os polígonos adjacentes com mesma inclinação.
3. Decompor os polígonos resultantes utilizando decomposição convexa (*Convex Decomposition*).

O primeiro passo é rápido e existe apenas para reduzir o número de polígonos dos passos seguintes. Um exemplo do resultado é mostrado na Figura 3.5.

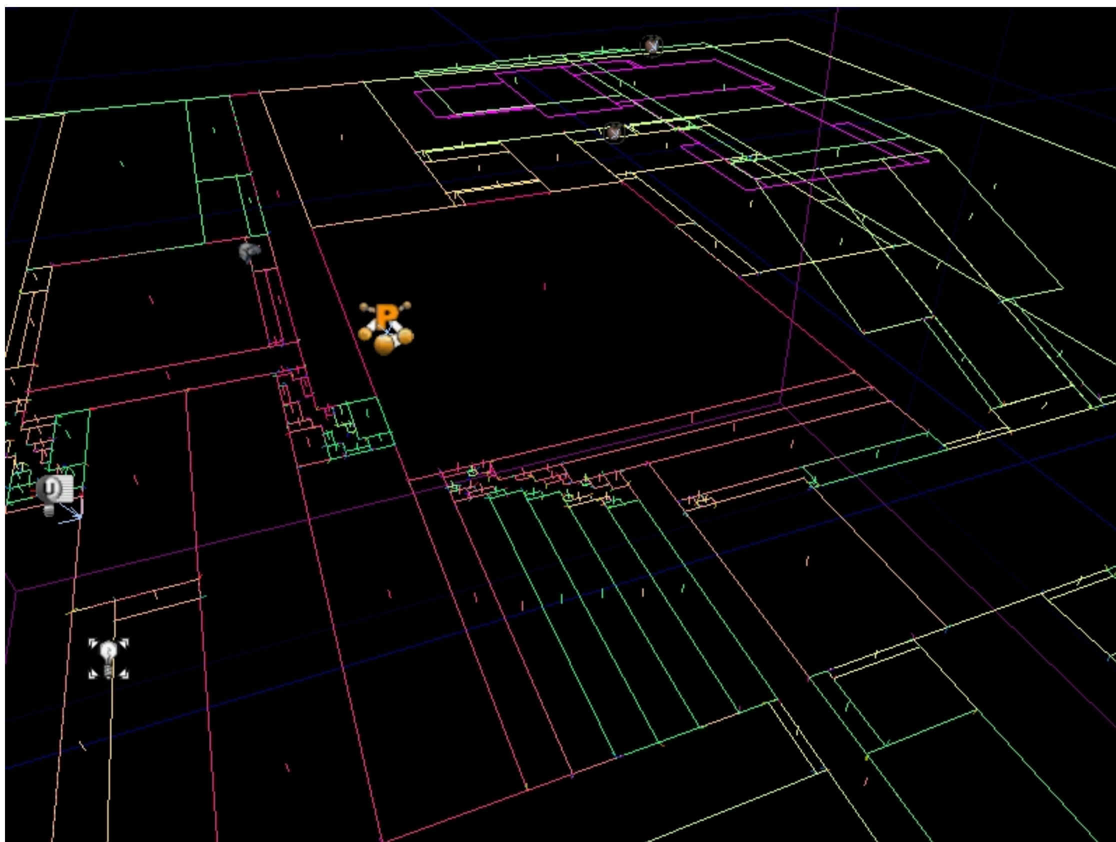
Figura 3.5 – *Unreal Engine 3* etapa de simplificação da malha, Passo 1.



Fonte: *Unreal Engine 3*, ano desconhecido.

A segunda etapa visa formar a maior superfície possível com os polígonos existentes de mesma inclinação. Segue um exemplo do resultado na Figura 3.6.

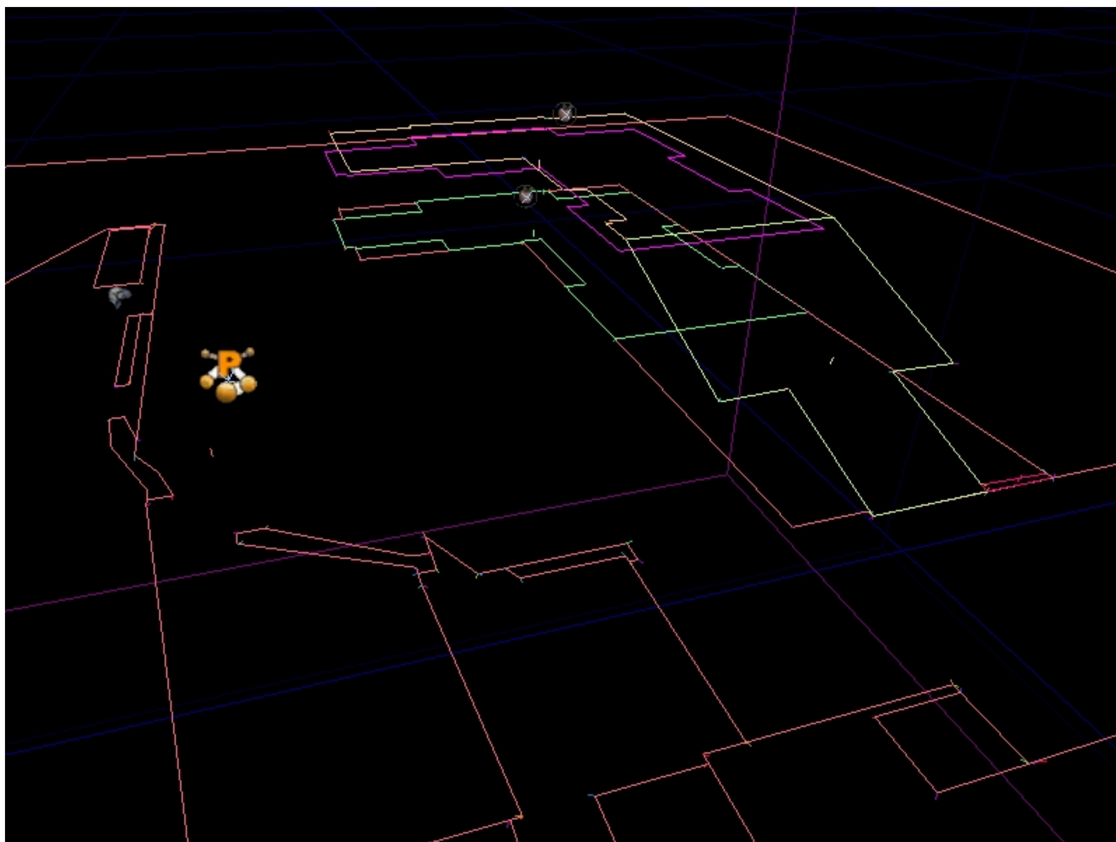
Figura 3.6 – *Unreal Engine 3* etapa de simplificação da malha, passo 2.



Fonte: *Unreal Engine 3*, ano desconhecido.

O segundo passo ainda simplifica as bordas, reduzindo o efeito de serrilhado gerado durante a geração da malha na etapa de Exploração. A Figura 3.7 mostra um exemplo com a simplificação de borda ativada.

Figura 3.7 – *Unreal Engine 3* etapa de simplificação da malha, passo 2. Simplificação de bordas ativada



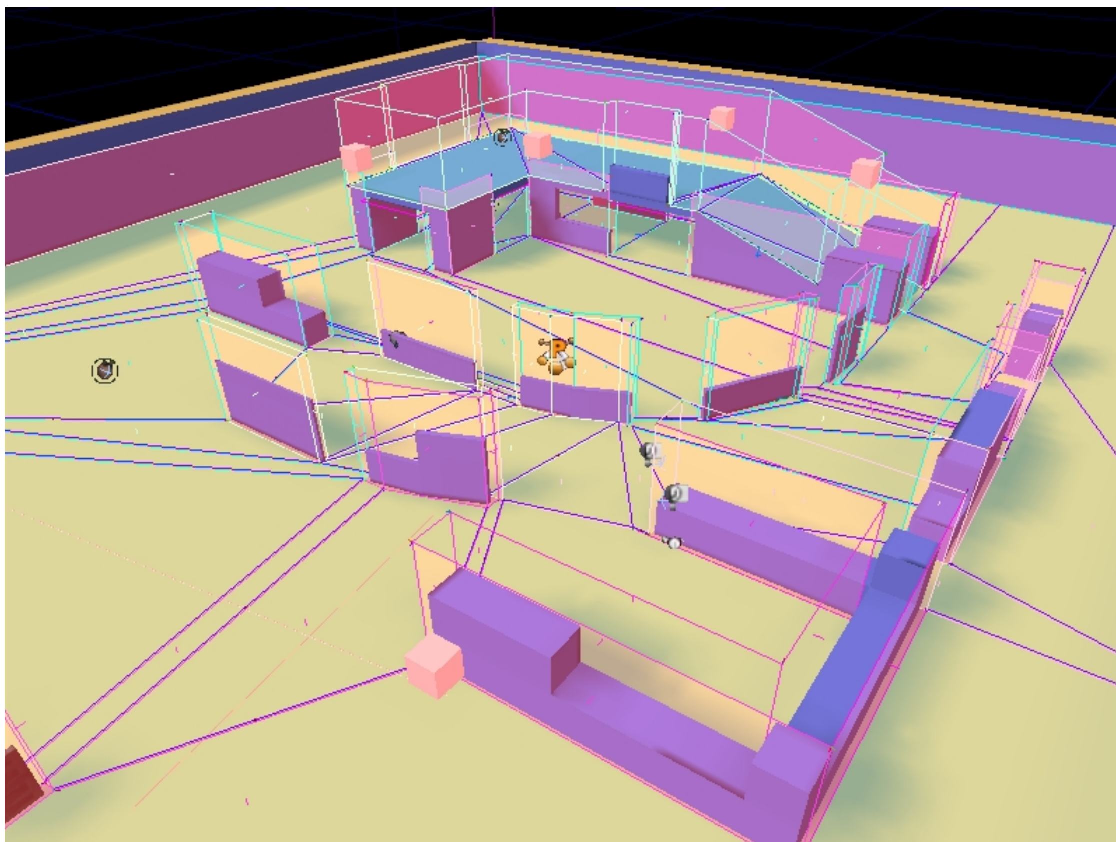
Fonte: *Unreal Engine 3*, ano desconhecido.

Por fim, o terceiro passo foca em subdividir os polígonos obtidos. Essa etapa segue a regra de que toda a superfície de um polígono deve ser alcançável a partir de qualquer ponto no mesmo polígono em linha reta. Para tanto, utiliza a estratégia de decomposição convexa. Para mais detalhes de algoritmos que trabalham com decomposição, sugere-se ao leitor o estudo da obra de (LIEN; AMATO, 2004). Não existe diferença visual na malha antes e depois da etapa de Finalização da Malha. Dessa forma, é possível observar esses polígonos na Figura 3.8 na subseção seguinte.

3.3.1.3 Finalização da Malha

Na finalização da malha, é gerado o grafo através da inserção de arestas entre os vértices, unindo assim cada polígono. Nessa mesma etapa, vértices não utilizados são removidos da malha e os dados são serializados. Além disso, a malha de obstáculos também é gerada, representada pelas superfícies verticais na Figura 3.8.

Figura 3.8 – *Unreal Engine 3* malha obtida após o processo de geração da *NavMesh*



Fonte: *Unreal Engine 3*, ano desconhecido.

3.4 Comentários Gerais

Como demonstrado no trabalho de (BOTEA; MüLLER; SCHAEFFER, 2004), é possível perceber que o HPA* começa a encontrar respostas mais rápido que o A* em cenários representados por grades regulares maiores que 20x20. (BOTEA; MüLLER; SCHAEFFER, 2004) também analisa a performance do uso de mais de um nível de hierarquia e avalia a busca de cada um, de modo geral, o uso de mais hierarquias melhora o tempo para se obter um caminho que possua um destino mais distante da origem.

Em relação à *NavMesh*, (CUI; HARABOR; GRASTIEN, 2017) afirma que a malha de navegação é uma técnica poderosa, que oferece as seguintes vantagens ao custo de possuir uma pesada etapa de pré-processamento:

- Oferece um solução completa, onde todos os pontos atrevessaveis constam na malha.
- Normalmente são pouco densos, o que economiza uso de memória.
- É uma abordagem flexivel, em que normalmente é fácil de ser modificada.

Devido essas qualidades, a NavMesh é amplamente utilizada em jogos, especialmente em jogos 3d, devido à sua capacidade de representar cenários complexos (XU; HUANG; ZOU, 2011).

4 EXPERIMENTOS COMPUTACIONAIS

Este capítulo apresenta na Seção 4.1 a ferramenta desenvolvida para visualização dos resultados dos algoritmos de busca mencionados no Capítulo 2. A Seção 4.2 reporta os cenários de teste para os algoritmos. A Seção 4.3 apresenta a comparação dos resultados entre os algoritmos de busca.

4.1 Ferramenta de Visualização

A visualização do funcionamento de cada algoritmo se dá a partir de uma ferramenta[1] desenvolvida nesse trabalho. O visualizador em questão conta com a funcionalidade de se editar uma grade (*grid*, termo em inglês comumente empregado), com as seguintes funcionalidades:

- adição e remoção de obstáculos;
- definição de um ponto de início e um ponto de destino;
- alteração de opções de velocidade da animação;
- alteração no tamanho das células da grade;
- opção de retornar a grade ao seu estado inicial;
- exportar e importar grades;
- salvar o resultado obtido, juntamente com uma imagem do resultado.

A ferramenta foi desenvolvida utilizando Java (JDK 17.0.1), bibliotecas nativas como AWT, Swing e Thread. Considerou-se a biblioteca externa Gson para importação e exportação das grades e resultados em formato Json. Ao executar o programa, a seguinte tela é apresentada.

[1] Essa ferramenta está disponível em <https://github.com/igortorati/TCCFerramenta>.

Figura 4.1 – Tela inicial da ferramenta de visualização

Fonte: Própria (2022)

No lado esquerdo, existem opções para preenchimento da grade, configurações do programa e opções para exportação dos dados. Cada uma dessas funcionalidades são descritas abaixo.

- Obstáculo (*Radio*): quando essa opção está selecionada e o usuário clica com o mouse em uma das grades, a célula é preenchida com a cor preta, representando um obstáculo.
- Apagar (*Radio*): quando essa opção está selecionada e o usuário clica com o mouse em uma das grades, a célula é limpa de qualquer preenchimento, seja ele obstáculo, marcação de início e marcação de destino.
- Início (*Radio*): quando essa opção está selecionada e o usuário clica com o mouse em uma das grades, a célula é limpa de qualquer obstáculo ou marcador de destino, e é preenchida com o marcador de início. Apenas pode existir um início na grade. Se já houver um, ao clicar com o mouse, o marcador existente será apagado e o novo será definido.
- Destino (*Radio*): quando essa opção está selecionada e o usuário clica com o mouse em uma das grades, a célula é limpa de qualquer obstáculo ou marcador de início, e é preenchida com o marcador de destino. Apenas pode existir um destino na grade. Se

já houver um, ao clicar com o mouse o marcador existente será apagado e o novo será definido.

- Heurística do algoritmo A* (Caixa de Seleção): Permite selecionar heurísticas alternativas para o Algoritmo A*. As alternativas existentes são as heurísticas utilizadas pelo Algoritmo Primeiro-Melhor (distância euclidiana até o destino) e pelo Algoritmo de Dijkstra (distância percorrida da origem).

- Buscar Caminho (Botão): quando este botão é pressionado, todos os algoritmos começam a buscar um caminho do ponto de início até o ponto de destino, considerando que os obstáculos não podem ser percorridos. Caso não exista um ponto de início, um ponto de destino ou o destino não seja alcançável a partir do ponto de início, é mostrado abaixo de cada grade a mensagem "Caminho não encontrado!"

- Limpar Matriz (Botão): limpa a grade, removendo obstáculos, marcador de início e marcador de fim. Elimina também qualquer marcação de célula visitada e caminho selecionado.

- Tamanho da Célula (*Scroll*): determina o tamanho da célula mostrada na grade. Com um valor maior, a célula possui um tamanho maior e um menor número de linhas e colunas para que caiba no espaço definido.

- Delay da animação (*Scroll*): determina o intervalo entre uma iteração e outra. Valores mais altos ocasionam em um menor número de atualizações, o que permite visualizar cada passo dos algoritmos. Um valor menor permite que a execução termine mais rapidamente.

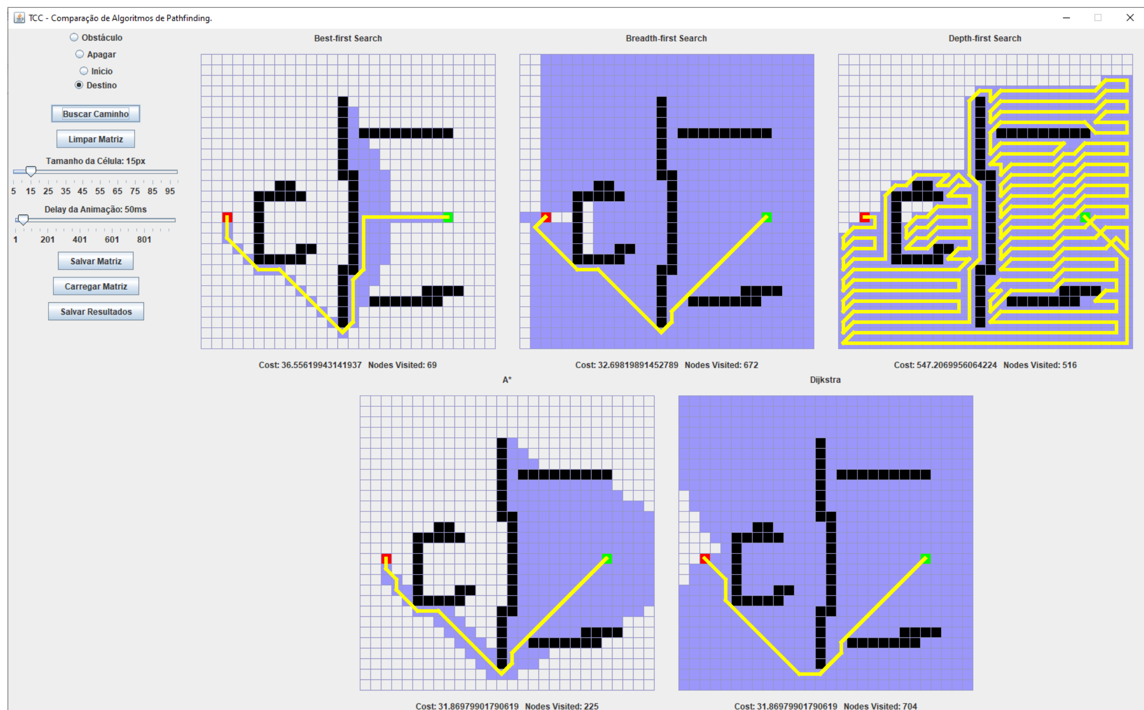
- Salvar Matriz (Botão): permite que a matriz seja salva em um arquivo json, que pode ser importado posteriormente. Além da matriz, também é salvo o tamanho da célula.

- Carregar Matriz (Botão): permite que uma matriz previamente salva seja importada, com os obstáculos, e, opcionalmente, marcador de início e fim, já definidos.

- Salvar Resultados (Botão): exporta os resultados para uma nova pasta. Esse diretório será criado a partir da raiz de onde o código foi executado, e tem o nome igual ao momento que foi clicado (no formato "*yyyy-MM-dd-HH.mm.ss*"). Dentro dessa pasta, é salvo um Json contendo o nome do algoritmo, o número de células visitadas e o custo para se chegar ao

destino. Além do arquivo Json, é salvo também uma imagem PNG, que mostra o grade e o caminho encontrado por cada algoritmo.

Figura 4.2 – Ferramenta exibindo caminho encontrado



Fonte: Própria (2022)

Na Figura 4.2, é possível verificar o resultado obtido em um exemplo de grade. O significado de cada cor é apresentado abaixo.

- Branco: caminho não visitado. Não é um obstáculo, nem um ponto de início ou destino.
- Preto: é um obstáculo e não pode ser percorrido pelo algoritmo.
- Azul: é uma célula visitada pelo algoritmo enquanto buscava um caminho do início ao destino.
- Traço amarelo: é um caminho visitado pelo algoritmo e que faz parte do caminho encontrado do início até o destino.
- Verde: é o marcador de ponto de início. Normalmente já começa marcado como visitado, mas sua cor permanece verde para facilitar a visualização.
- Vermelho: é o marcador de destino, marcado como visitado ao ser encontrado. Porém, permanece vermelho para facilitar a visualização.

4.2 Base de Dados

A base de dados utilizada para comparação dos algoritmos abordados no Capítulo 2 constam no **Apêndice A**. Cada cenário foi gerado de modo a garantir a inexistência de um caminho direto entre o nó de origem e o nó de destino, necessitando que os algoritmos trabalhem para o encontro do caminho, desviando dos obstáculos, de modo a ficar claro a diferença de caminhos encontrada por cada algoritmo. Os pontos de início e destino também foram escolhidos de forma empírica com a mesma finalidade.

Alguns dos cenários possuem variações, criadas com a finalidade de demonstrar como pequenas alterações impactam no funcionamento dos algoritmos. Essas mudanças foram nomeadas com o mesmo nome do cenário original, seguido de um identificador da variação em questão. Por exemplo, o “map7” possui três variações: “map7.1”, “map7.2” e “map7.3”.

4.3 Resultados Obtidos

As Tabelas 4.1 e 4.2 demonstram um comparativo dos resultados obtidos a partir dos cenários de teste para cada algoritmo estudado. Essas tabelas acompanham o valor da melhor solução para o critério avaliado e o desvio dos algoritmos em relação a este melhor resultado. As abreviações M.C., P.M. e N.V representam o “Menor Caminho”, “Algoritmo Primeiro-Melhor” e “Nós Visitados”, respectivamente.

Os dados exibidos em negrito, representam o melhor resultado do cenário e a última linha de cada tabela representa a média da variação de cada algoritmo em relação ao melhor resultado do critério avaliado.

Tabela 4.1 – Comparativo percentual em relação ao melhor custo.

Mapa	M.C.	P.M.	BFS	DFS	A*	Dijkstra
map1	101,53	+ 32,95%	+ 1,63%	+ 388,04%	+ 6,53%	+ **0,00**%
map2	46,63	+ 8,88%	+ 14,21%	+ 249,48%	+ 8,88%	+ **0,00**%
map2.1	46,63	+ 8,88%	+ 14,21%	+ 698,69%	+ 8,88%	+ **0,00**%
map3	72,07	+ 92,98%	+ **0,00**%	+ 138,92%	+ 12,64%	+ **0,00**%
map4	24,90	+ 153,18%	+ 6,65%	+ 2200,31%	+ 6,65%	+ **0,00**%
map4.1	31,97	+ 227,93%	+ 2,59%	+ 1042,51%	+ 7,77%	+ **0,00**%
map4.2	31,97	+ 153,17%	+ 2,59%	+ 1276,51%	+ 7,77%	+ **0,00**%
map5	72,80	+ 227,59%	+ 5,69%	+ 2707,81%	+ 7,97%	+ **0,00**%
map6	164,33	+ 25,19%	+ 18,65%	+ 859,71%	+ 13,11%	+ **0,00**%
map6.1	124,53	+ 149,57%	+ 19,29%	+ 1993,59%	+ 13,11%	+ **0,00**%
map6.2	129,80	+ 43,54%	+ 21,70%	+ 693,94%	+ 15,13%	+ **0,00**%
map7	98,77	+ 30,56%	+ 10,06%	+ 1750,84%	+ 9,23%	+ **0,00**%
map7.1	121,08	+ 27,26%	+ 10,38%	+ 1229,08%	+ 4,91%	+ **0,00**%
map7.2	138,78	+ 117,10%	+ 9,55%	+ 604,48%	+ 4,78%	+ **0,00**%
map7.3	105,50	+ 72,44%	+ 10,21%	+ 1465,63%	+ 6,28%	+ **0,00**%
	0	+ 91,41%	+ 9,83%	+ 1153,30%	+ 8,91%	+ 0,00%

Fonte: Própria(2022)

Tabela 4.2 – Comparativo percentual em relação à solução com menos vértices visitados.

Mapa	N.V.	P.M.	BFS	DFS	A*	Dijkstra
map1	393	+ **0,00**%	+ 37,66%	+ 41,98%	+ 3,82%	+ 49,11%
map2	42	+ **0,00**%	+ 1385,71%	+ 238,10%	+ 888,10%	+ 1390,48%
map2.1	42	+ **0,00**%	+ 1414,29%	+ 1078,57%	+ 888,10%	+ 1428,57%
map3	550	+ 0,73%	+ 13,82%	+ **0,00**%	+ 8,36%	+ 13,82%
map4	124	+ **0,00**%	+ 273,39%	+ 336,29%	+ 40,32%	+ 285,48%
map4.1	183	+ **0,00**%	+ 191,80%	+ 96,72%	+ 80,33%	+ 200,00%
map4.2	330	+ 18,79%	+ 61,82%	+ 61,21%	+ **0,00**%	+ 66,36%
map5	1521	+ 48,98%	+ 95,33%	+ 31,69%	+ **0,00**%	+ 81,07%
map6	889	+ **0,00**%	+ 656,81%	+ 64,57%	+ 506,97%	+ 652,64%
map6.1	1730	+ **0,00**%	+ 242,31%	+ 270,00%	+ 41,04%	+ 240,00%
map6.2	923	+ 230,77%	+ 427,09%	+ **0,00**%	+ 215,06%	+ 414,52%
map7	398	+ **0,00**%	+ 341,46%	+ 475,38%	+ 167,84%	+ 319,60%
map7.1	278	+ **0,00**%	+ 709,35%	+ 779,14%	+ 536,69%	+ 724,46%
map7.2	925	+ 0,76%	+ 173,41%	+ **0,00**%	+ 105,73%	+ 174,05%
map7.3	921	+ **0,00**%	+ 174,59%	+ 69,49%	+ 69,49%	+ 175,24%
	0	+ 20,00%	+ 413,26%	+ 236,21%	+ 236,79%	+ 414,36%

Fonte: Própria(2022)

Os resultados acima corroboram para as afirmações de (MEHTA et al., 2015) e (ABLY et al., 2019), demonstrando que Dijkstra encontrou o melhor caminho existente até o objetivo. Também é possível observar que, de modo geral, o algoritmo que executou mais rápido na

média foi o Algoritmo de Busca Primeiro-Melhor, como afirmado por (MEHTA et al., 2015) e (ISKANDAR; DIAH; ISMAIL, 2020). O Algoritmo A* encontrou uma solução quase tão curta quanto o de Dijkstra, porém, em um tempo melhor, como indicado nos trabalhos de (FOEAD et al., 2021; ISKANDAR; DIAH; ISMAIL, 2020; PATEL, 2010).

Os resultados obtidos para os algoritmos A* e Dijkstra estão condizentes com os resultados obtidos por (ÖSTBERG,), apesar de os resultados do autor estarem em milissegundos, existe equivalência em proporção semelhante ao número de iterações da Tabela 4.2.

5 CONCLUSÃO

O presente trabalho visou explorar algoritmos e estratégias utilizados na busca de caminhos em jogos digitais. Foram apresentados os principais algoritmos aplicados a esse contexto, tendo maior destaque os algoritmos Primeiro-Melhor, Dijkstra e A*. A respeito desses algoritmos, foi desenvolvida uma ferramenta para facilitar a visualização e comparação dos resultados obtidos nos cenários testados.

Também foram abordadas algumas estratégias de representação dos mapas, em especial, o *HPA** e a *NavMesh*. Explorou-se os motivos que tornam essas estratégias populares na indústria e no meio acadêmico. A apresentação desses temas foi feita para facilitar a compreensão do assunto, que, devido à sua complexidade, pode ser uma barreira para que pessoas que desejem se especializar no desenvolvimento e projeto de jogos digitais.

Trabalhos futuros podem estudar, com o mesmo objetivo, problemas de otimização das técnicas reportadas nos Capítulos 2 ao 4, bem como desafios enfrentados na movimentação de grupos de entidades em cenários de jogos, isso é, a movimentação de várias entidades que interagem entre si, por um mesmo mapa (grade regular ou irregular), ao mesmo tempo, como já abordado anteriormente por (DUC; SIDHU; CHAUDHARI, 2008; FATHY; RAOUF; ELKADER, 2015; FOEAD et al., 2021). Um tema que também vale a pena explorar são as novas estratégias que estão surgindo, justamente para suprir movimentações mais complexas em 3D, como saltos, escaladas, voo e *parkour*. Um bom ponto de partida pode ser a publicação feita pela *Ubisoft* [1] em Janeiro de 2022, abordando a nova estratégia que está sendo desenvolvida nos estúdios da empresa para substituir o uso da *NavMesh*.

[1] Está publicação está disponível em: <https://news.ubisoft.com/en-au/article/6Mv4hZqUMJoY1xpf1yiQPi/ubisoft-la-forge-pushing-stateoftheart-ai-in-games-to-create-the-next-generation-of
Acessado em 21/03/2022.

REFERÊNCIAS

ABLY, T. et al. **Dijkstra's Shortest Path Algorithm**. 2019. Acesso em 15/03/2022. Disponível em: <https://brilliant.org/wiki/dijkstras-short-path-finder/#:~:text=One\%20algorithm\%20for\%20finding\%20the,other\%20points\%20in\%20the\%20graph.>

ALGFOOR, Z. A.; SUNAR, M. S.; KOLIVAND, H. A comprehensive study on pathfinding techniques for robotics and video games. **International Journal of Computer Games Technology**, v. 2015, p. 1–11, 04 2015.

BOTEA, A.; MüLLER, M.; SCHAEFFER, J. Near optimal hierarchical path-finding. **Journal of Game Development**, v. 1, p. 7–28, 2004.

CORMEN, T. H. et al. **Algoritmos: teoria e prática**. Campus, 2012. ISBN 9788535236996. Disponível em: <https://books.google.com.br/books?id=6iA4LgEACAAJ>.

CUI, M.; HARABOR, D.; GRASTIEN, A. Compromise-free pathfinding on a navigation mesh. In: . [s.n.], 2017. Disponível em: <https://www.ijcai.org/proceedings/2017/0070.pdf>.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numer. Math.**, Springer-Verlag, Berlin, Heidelberg, v. 1, n. 1, p. 269–271, dec 1959. ISSN 0029-599X. Disponível em: <https://doi.org/10.1007/BF01386390>.

DUC, L.; SIDHU, A.; CHAUDHARI, N. Hierarchical pathfinding and ai-based learning approach in strategy game design. **International Journal of Computer Games Technology**, v. 2008, 01 2008.

FATHY, H.; RAOUF, O.; ELKADER, H. A. Flocking behaviour of group movement in real strategy games. **2014 9th International Conference on Informatics and Systems, INFOS 2014**, p. PDC64–PDC67, 02 2015. Disponível em: <https://www.researchgate.net/publication/282379016_Flocking_behaviour_of_group_movement_in_real_strategy_games>.

FEOFILOFF, P. **Busca em largura**. 2019. Acesso em 03/04/2022. Disponível em: <https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html>.

FOEAD, D. et al. A systematic literature review of a* pathfinding. **Procedia Computer Science**, v. 179, p. 507–514, 2021. ISSN 1877-0509. 5th International Conference on Computer Science and Computational Intelligence 2020. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050921000399>.

GRAHAM ROSS; MCCABE, H.; SHERIDAN, S. Pathfinding in computer games. **The ITB Journal**, v. 4, n. 6, 2003. Disponível em: <https://arrow.tudublin.ie/itbj/vol4/iss2/6>.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE Transactions on Systems Science and Cybernetics**, v. 4, n. 2, p. 100–107, 1968. Disponível em: <https://ieeexplore.ieee.org/document/4082128>.

ISKANDAR, U.; DIAH, N.; ISMAIL, M. Identifying artificial intelligence pathfinding algorithms for platformer games. In: . [s.n.], 2020. p. 74–80. Disponível em: <https://www.researchgate.net/publication/342965818_Identifying_Artificial_Intelligence_Pathfinding_Algorithms_for_Platformer_Games>.

JANSEN, M.; BURO, M. Hpa* enhancements. In: . [S.l.: s.n.], 2007. p. 84–87.

KAUR, N.; GARG, D. Analysis of the depth first search algorithms. **Data mining and knowledge engineering**, v. 4, p. 37–41, 2012. Disponível em: <https://www.semanticscholar.org/paper/Analysis-of-the-Depth-First-Search-Algorithms-Kaur-Garg/ac85ed7c59e3d43990d0a510eb037ecd07d2b269>.

LEE, C. Y. An algorithm for path connections and its applications. **IRE Transactions on Electronic Computers**, EC-10, n. 3, p. 346–365, 1961.

LIEN, J.-M.; AMATO, N. M. Approximate convex decomposition of polygons. In: **Proceedings of the Twentieth Annual Symposium on Computational Geometry**. New York, NY, USA: Association for Computing Machinery, 2004. (SCG '04), p. 17–26. ISBN 1581138857. Disponível em: <https://doi.org/10.1145/997817.997823>.

MA, T. et al. Grid task scheduling: Algorithm review. **IETE Technical Review**, Taylor & Francis, v. 28, n. 2, p. 158–167, 2011. Disponível em: <https://www.tandfonline.com/doi/abs/10.4103/0256-4602.76138>.

MEHTA, P. et al. A review on algorithms for pathfinding in computer games. In: . [s.n.], 2015. Disponível em: <https://www.researchgate.net/publication/303369993_A_Review_on_Algorithms_for_Pathfinding_in_Computer_Games>.

MOORE, E. F. The shortest path through a maze. **Proc. International Symposium on the Theory of Switching**, Harvard University Press, p. 285–292, 1959. Disponível em: <https://ci.nii.ac.jp/naid/10010192763/en/>.

ÖSTBERG, O. 28 p.

PATEL, A. **Heuristics**. 2010. Acesso em 09/03/2022. Disponível em: <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>.

PELECHANO, N.; FUENTES, C. Hierarchical path-finding for navigation meshes (hna∗). **Computers & Graphics**, v. 59, p. 68–78, 2016. ISSN 0097-8493. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0097849316300668>.

PERMANA, S. et al. Comparative analysis of pathfinding algorithms a *, dijkstra, and bfs on maze runner game. **IJISTECH (International Journal Of Information System & Technology)**, v. 1, p. 1, 05 2018.

RACHMAWATI, D.; GUSTIN, L. Analysis of dijkstra's algorithm and a* algorithm in shortest path problem. **Journal of Physics: Conference Series**, IOP Publishing, v. 1566, n. 1, p. 012061, jun 2020. Disponível em: <https://doi.org/10.1088/1742-6596/1566/1/012061>.

RAHIM, R. et al. Breadth first search approach for shortest path solution in cartesian area. **Journal of Physics: Conference Series**, IOP Publishing, v. 1019, p. 012036, jun 2018. Disponível em: <https://doi.org/10.1088/1742-6596/1019/1/012036>.

TARJAN, R. Depth-first search and linear graph algorithms. **SIAM journal on computing**, SIAM, v. 1, n. 2, p. 146–160, 1972.

XU, X.; HUANG, M.; ZOU, K. Automatic generated navigation mesh algorithm on 3d game scene. **Procedia Engineering**, v. 15, p. 3215–3219, 2011. ISSN 1877-7058. CEIS 2011. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877705811021059>.

ZAREMBO, I.; KODORS, S. Pathfinding algorithm efficiency analysis in 2d grid. **Environment. Technology. Resources. Proceedings of the International Scientific and Practical Conference**, v. 2, p. 46, 08 2015.

ZIKKY, M. Review of a* (a star) navigation mesh pathfinding as the alternative of artificial intelligent for ghosts agent on the pacman game. **EMITTER International Journal of Engineering Technology**, v. 4, n. 1, p. 141–149, Aug. 2016. Disponível em: <https://emitter.pens.ac.id/index.php/emitter/article/view/117>.

APÊNDICE A – Cenários de teste e resultados obtidos

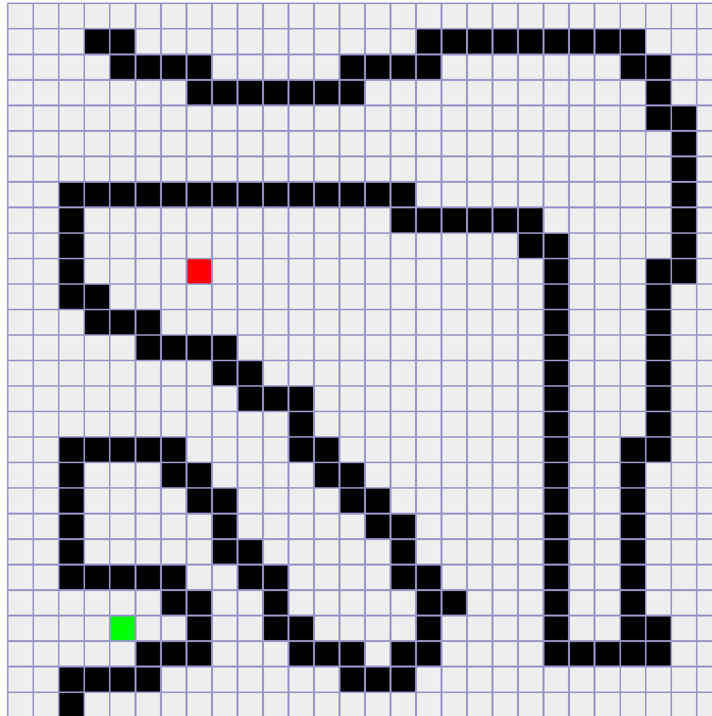
Neste apêndice, constam as imagens dos cenários de teste criados, tal como as tabelas contendo os resultados obtidos, com informações do caminho obtido e número de vértices analisados. Apresenta-se, também, as imagens do caminho encontrado por cada algoritmo.

As colunas da Tabela 1 representam a dimensão, ponto de início, destino e quantidade de obstáculos para cada cenário de teste. As grades são quadradas, e dessa forma, sua dimensão representa o número de células para cada linha e cada coluna. O ponto de início e fim cartesiano consideram a célula na primeira coluna, da esquerda para a direita, e na primeira linha, de baixo para cima, como sendo a célula (0, 0).

	Dimensão	Ponto de Inicio Cartesiano	Ponto de Destino Cartesiano	Obstáculos
map1	28	(4, 3)	(7, 17)	173
map2	28	(0, 27)	(23, 8)	103
map2.1	28	(0, 27)	(23, 8)	101
map3	28	(0, 27)	(26, 24)	157
map4	28	(3, 16)	(16, 16)	38
map4.1	28	(3, 16)	(16, 16)	77
map4.2	28	(3, 16)	(16, 16)	82
map5	61	(4, 37)	(32, 28)	237
map6	86	(0, 85)	(73, 30)	634
map6.1	86	(0, 85)	(73, 30)	661
map6.2	86	(0, 85)	(73, 30)	782
map7	86	(9, 74)	(58, 47)	1299
map7.1	86	(9, 74)	(65, 35)	1299
map7.2	86	(9, 74)	(48, 50)	1360
map7.3	86	(74, 68)	(48, 50)	1360

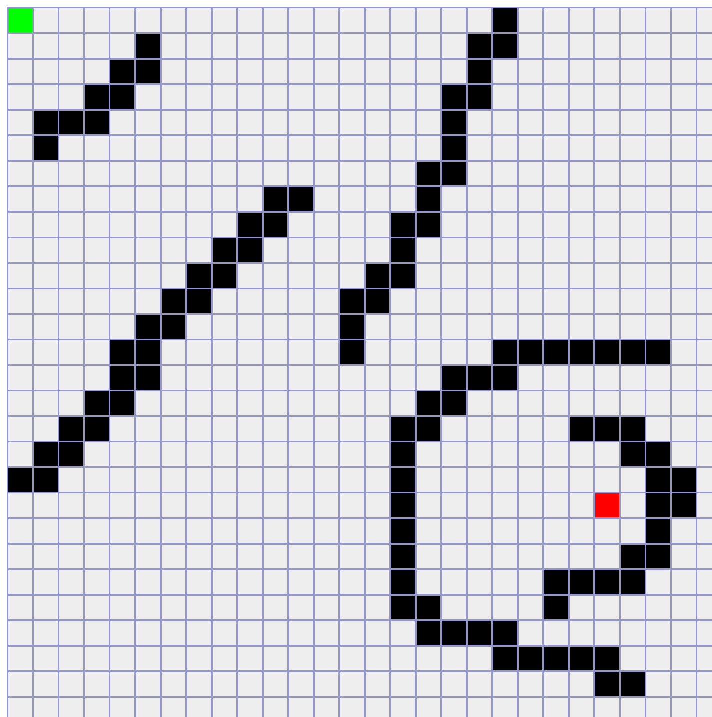
Os cenários utilizados para a comparação dos algoritmos estão descritos nas Figuras 1 a 15 abaixo.

Figura 1 – Cenário de teste "map1"



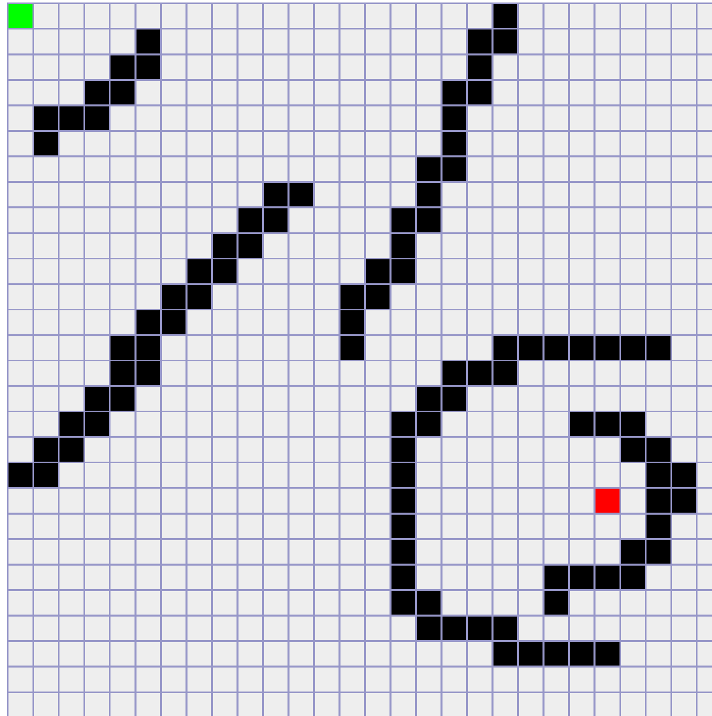
Fonte: Própria (2022)

Figura 2 – Cenário de teste "map2"



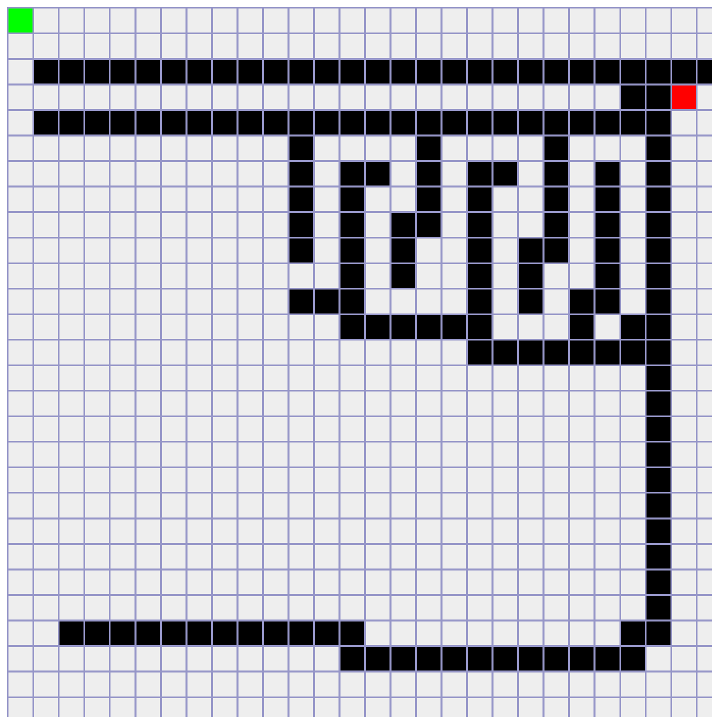
Fonte: Própria (2022)

Figura 3 – Cenário de teste "map2.1"



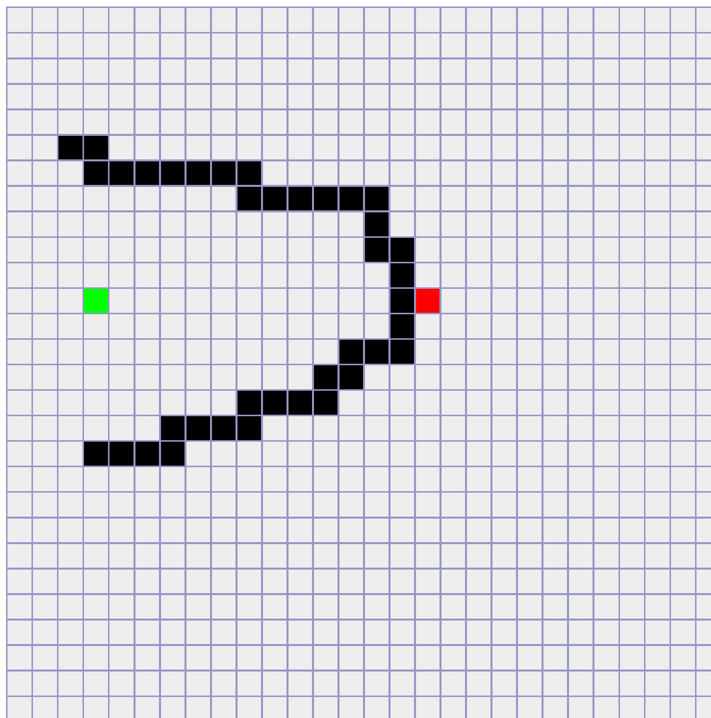
Fonte: Própria (2022)

Figura 4 – Cenário de teste "map3"



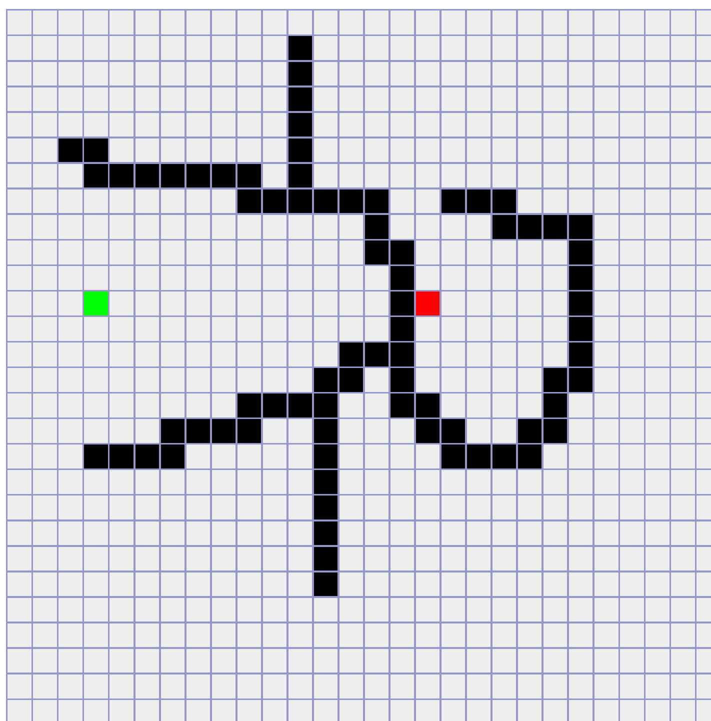
Fonte: Própria (2022)

Figura 5 – Cenário de teste "map4"



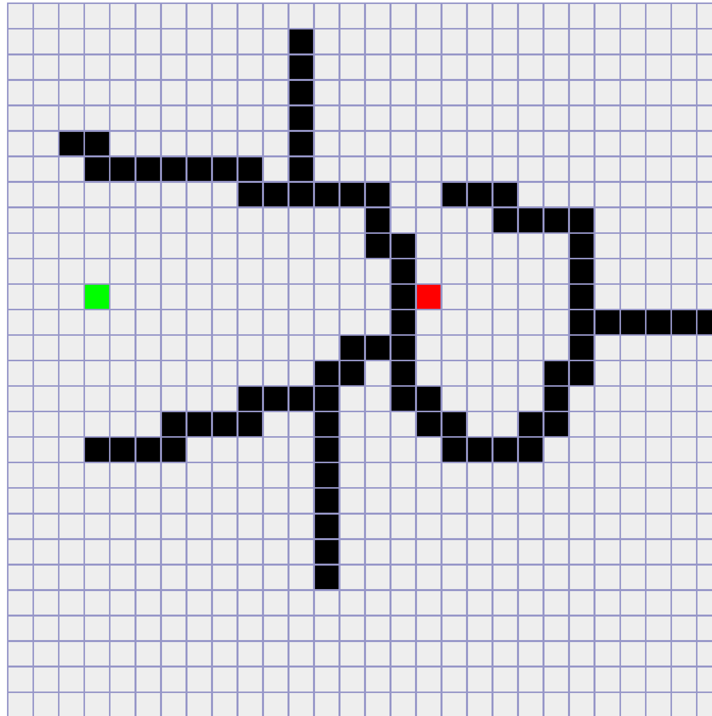
Fonte: Própria (2022)

Figura 6 – Cenário de teste "map4.1"



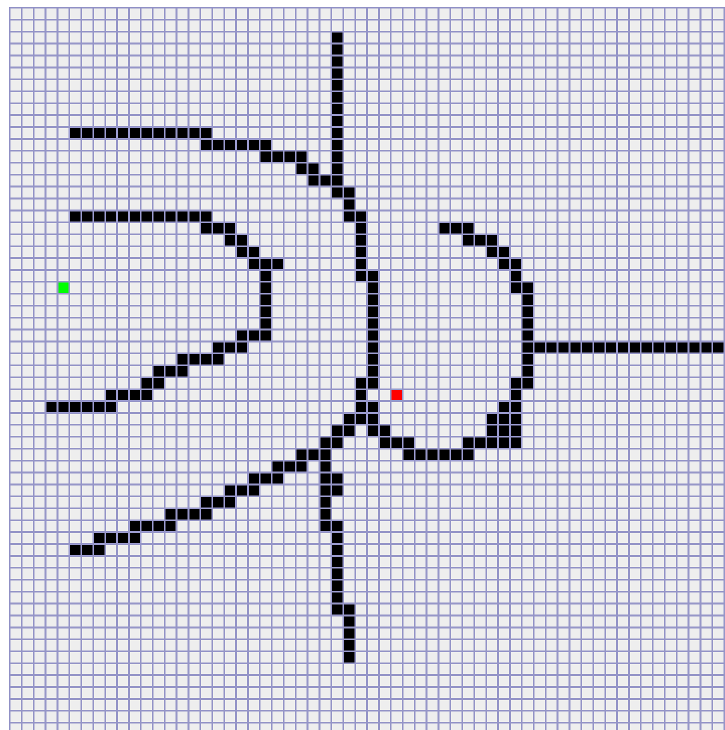
Fonte: Própria (2022)

Figura 7 – Cenário de teste "map4.2"



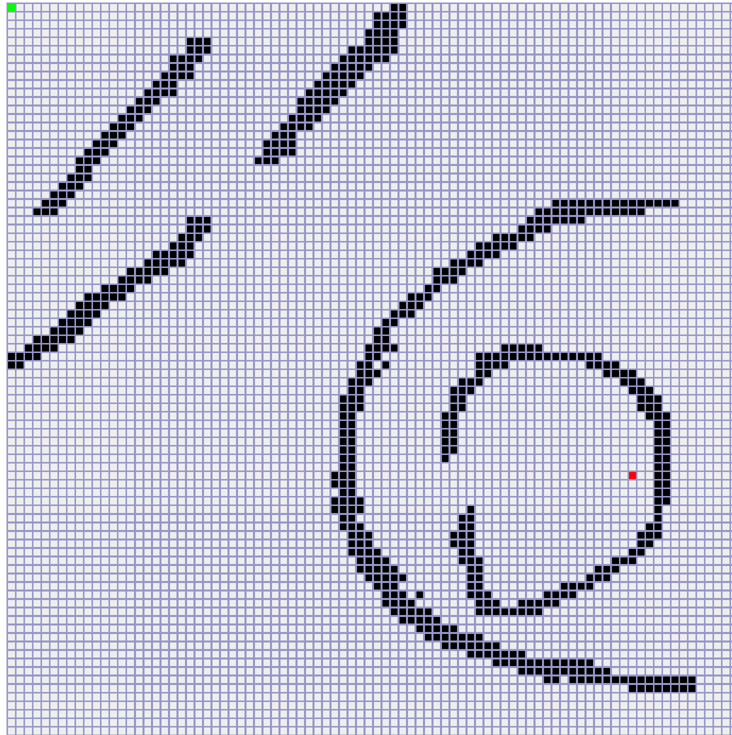
Fonte: Própria (2022)

Figura 8 – Cenário de teste "map5"



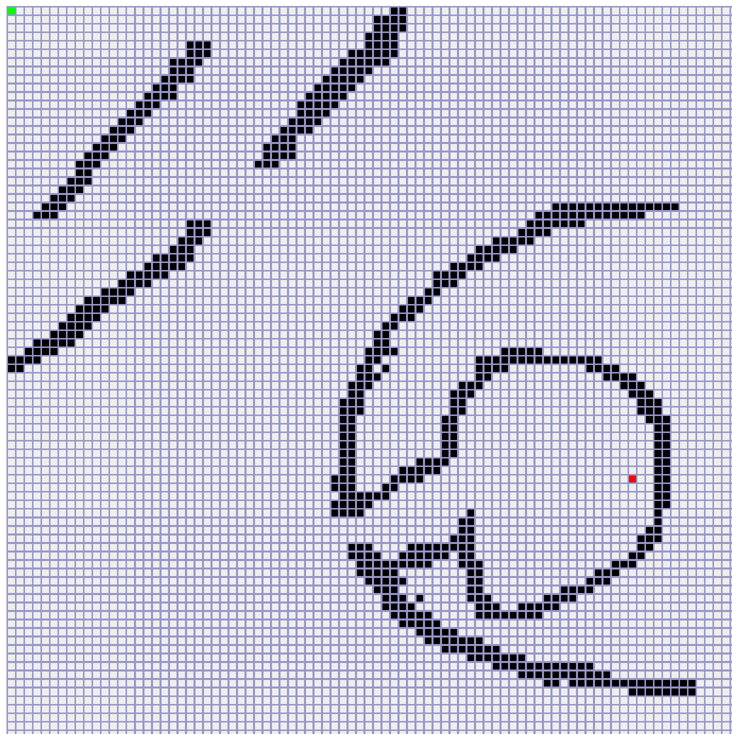
Fonte: Própria (2022)

Figura 9 – Cenário de teste "map6"



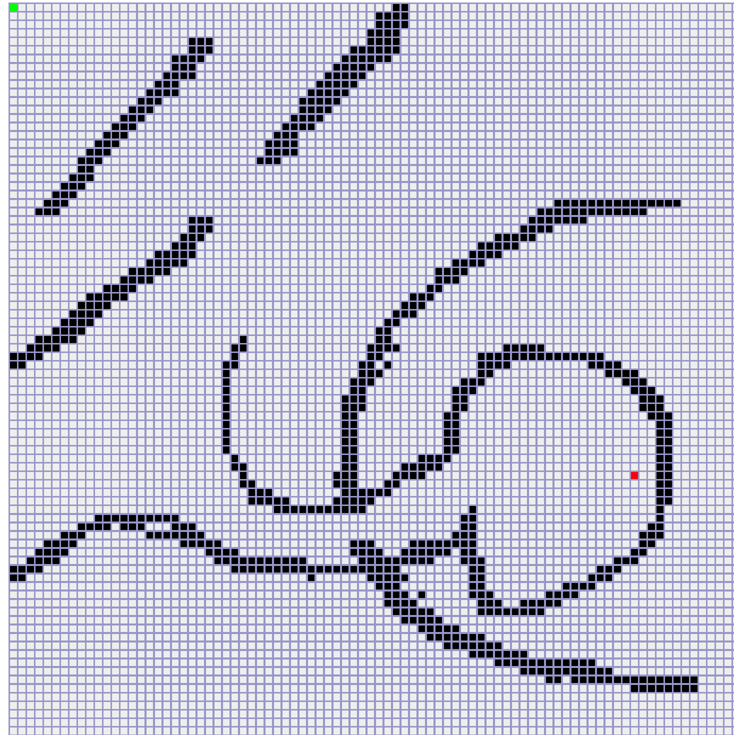
Fonte: Própria (2022)

Figura 10 – Cenário de teste "map6.1"



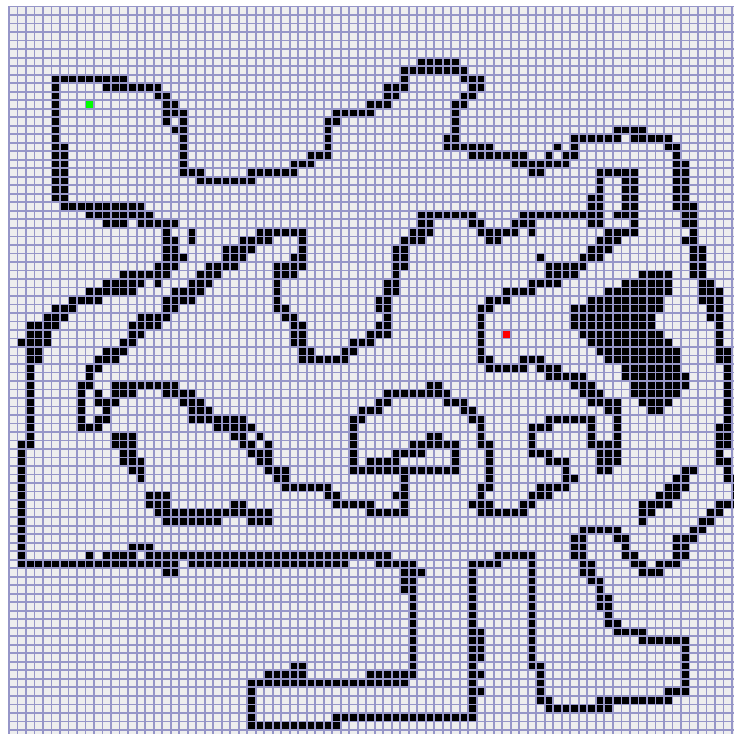
Fonte: Própria (2022)

Figura 11 – Cenário de teste "map6.2"



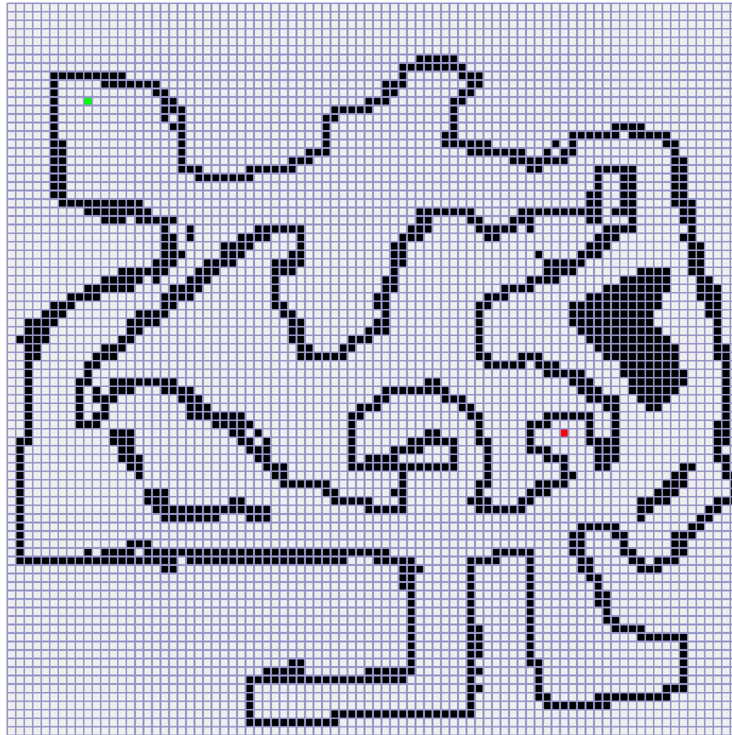
Fonte: Própria (2022)

Figura 12 – Cenário de teste "map7"



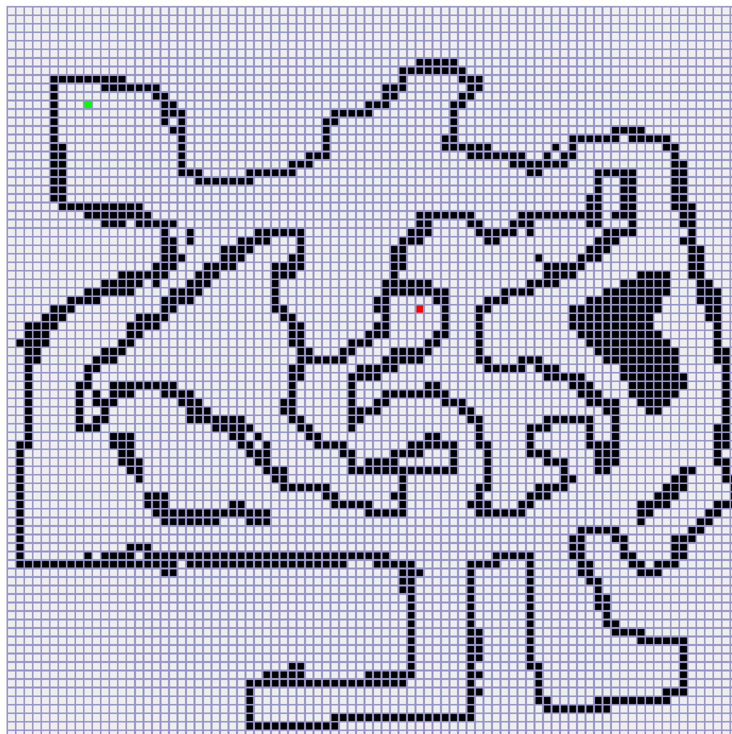
Fonte: Própria (2022)

Figura 13 – Cenário de teste "map7.1"



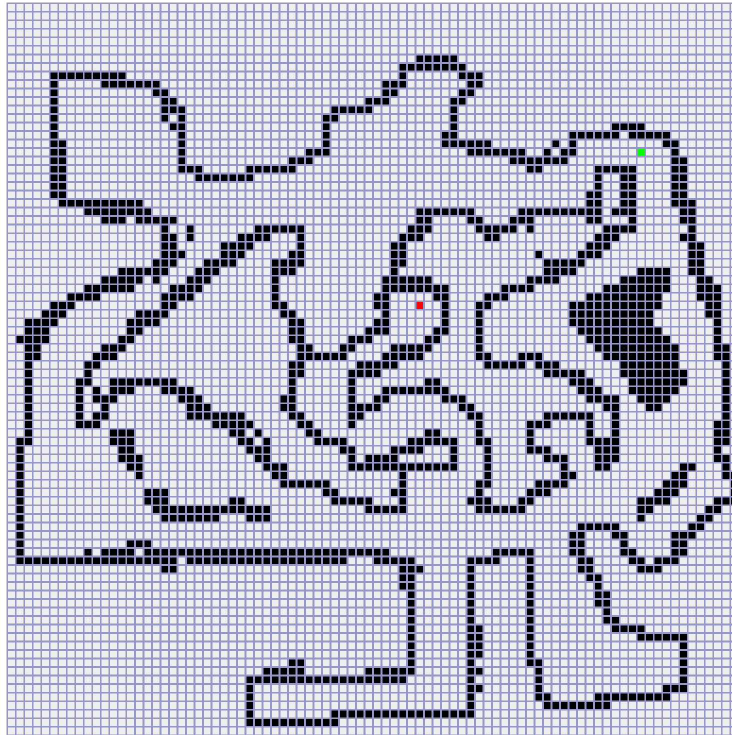
Fonte: Própria (2022)

Figura 14 – Cenário de teste "map7.2"



Fonte: Própria (2022)

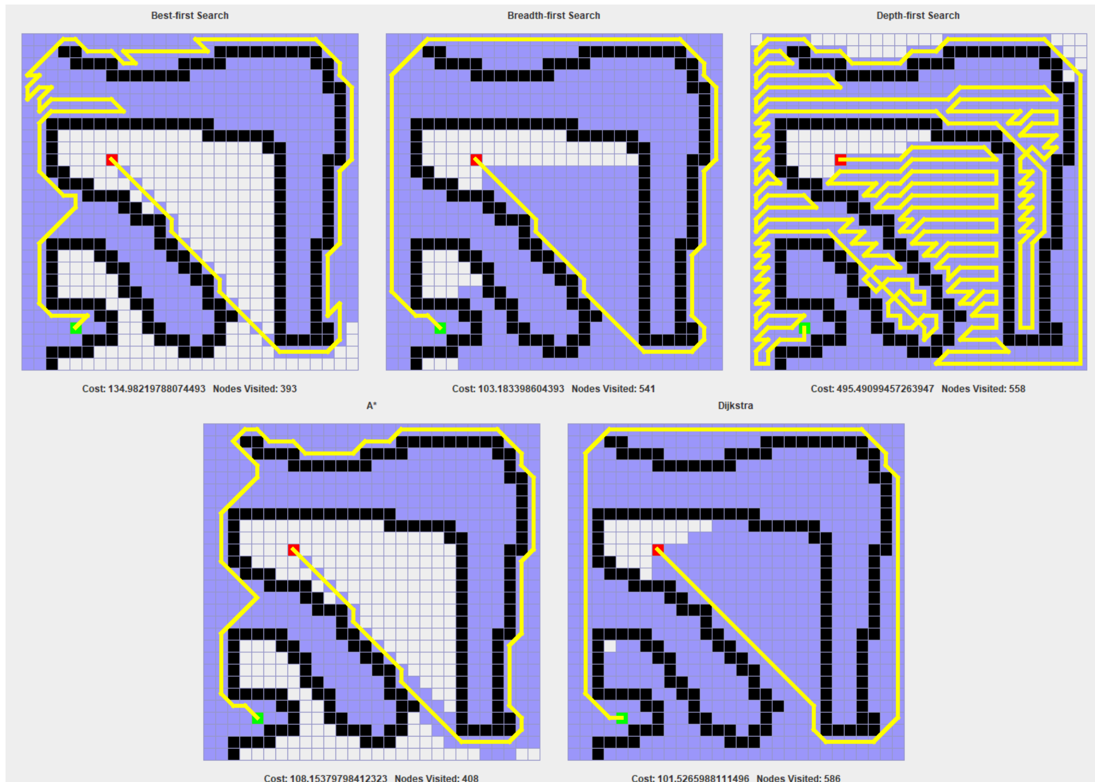
Figura 15 – Cenário de teste "map7.3"



Fonte: Própria (2022)

Os resultados obtidos em cada uma dos cenários estão descritos nas Tabelas 1 a 15, juntamente com as imagens dos caminhos encontrados nas Figuras 16 a 30 abaixo.

Figura 16 – Resultados cenário de teste "map1"



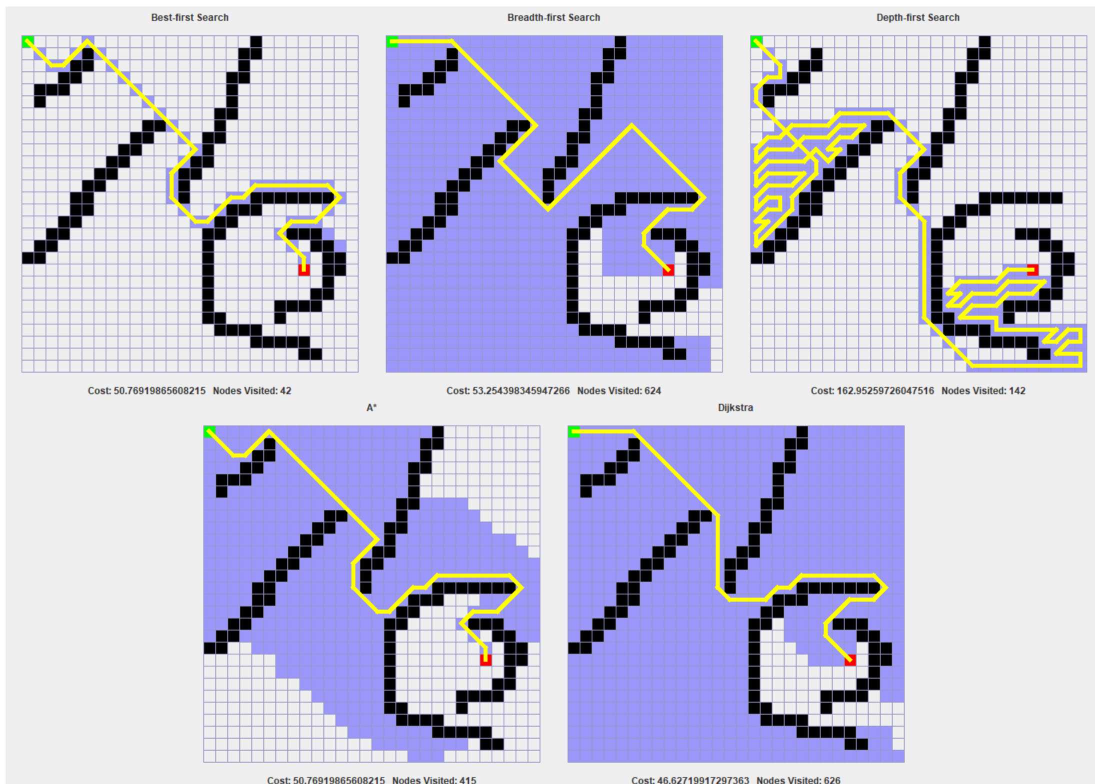
Fonte: Própria (2022)

Tabela 1 – Resultados obtidos para "map1"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	134.98	393
Breadth-first Search	103.18	541
Depth-first Search	495.49	558
A*	108.15	408
Dijkstra	101.53	586

Fonte: Própria(2022)

Figura 17 – Resultados cenário de teste "map2"



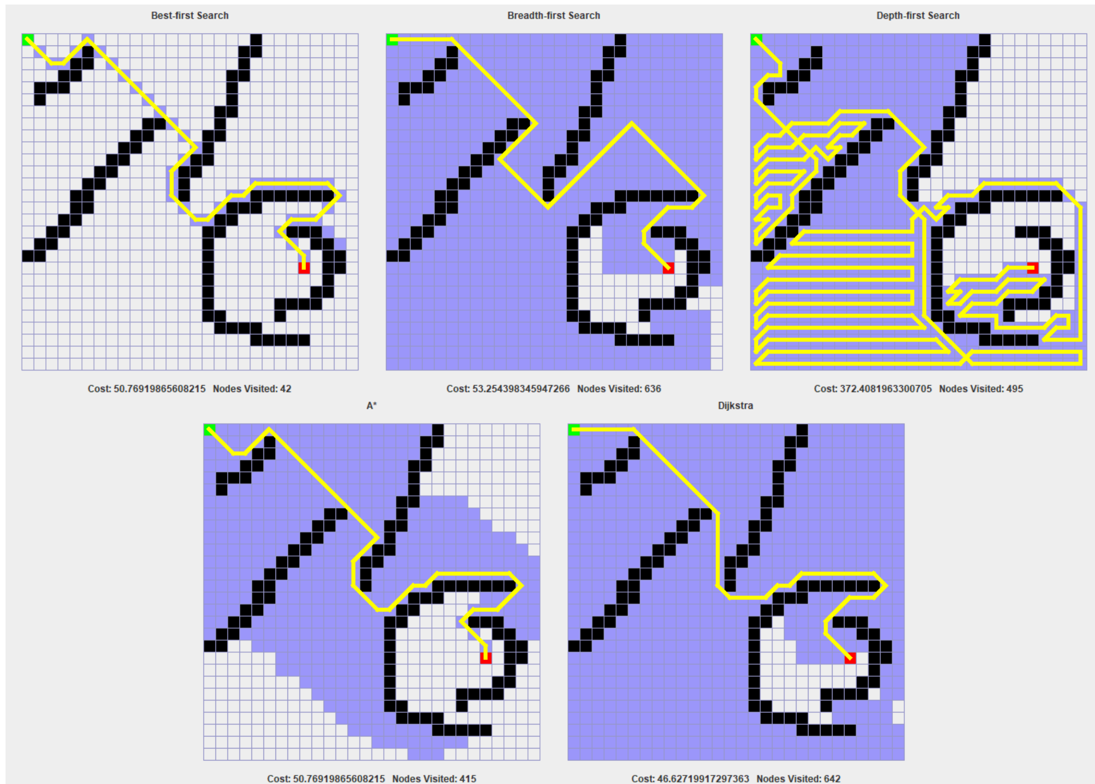
Fonte: Própria (2022)

Tabela 2 – Resultados obtidos para "map2"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	50.77	42
Breadth-first Search	53.25	624
Depth-first Search	162.95	142
A*	50.77	415
Dijkstra	46.63	626

Fonte: Própria(2022)

Figura 18 – Resultados cenário de teste "map2.1"



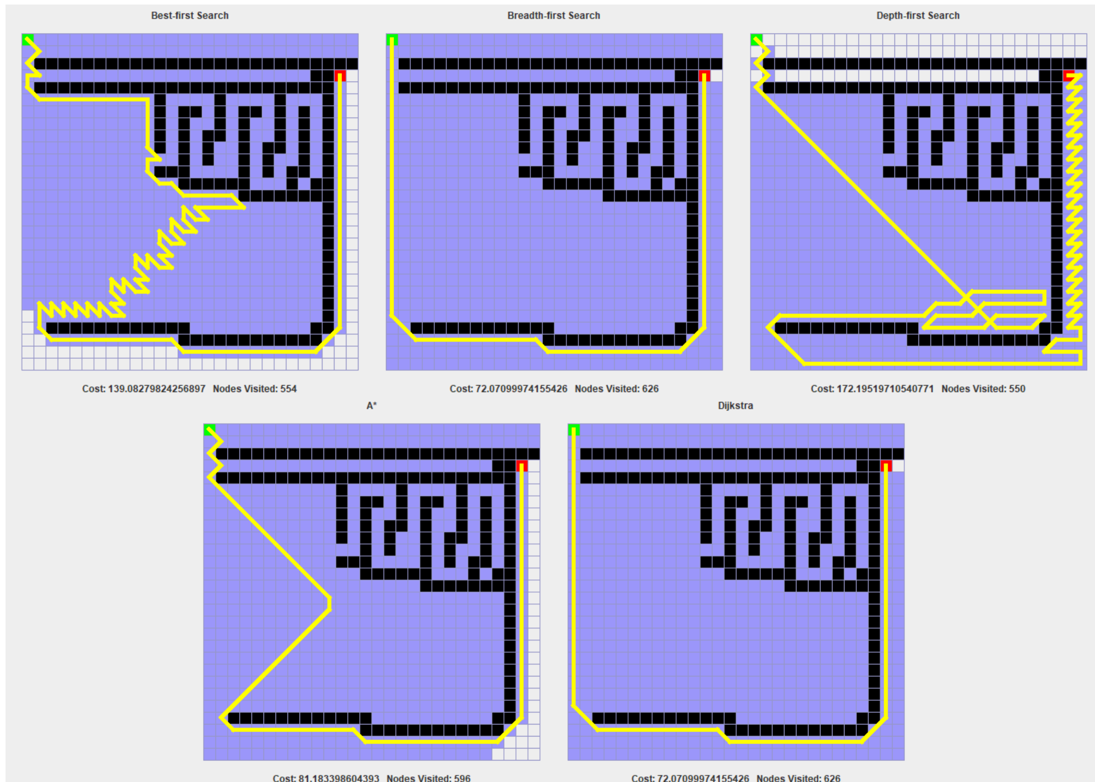
Fonte: Própria (2022)

Tabela 3 – Resultados obtidos para "map2.1"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	50.77	42
Breadth-first Search	53.25	636
Depth-first Search	372.41	495
A*	50.77	415
Dijkstra	46.63	642

Fonte: Própria(2022)

Figura 19 – Resultados cenário de teste "map3"



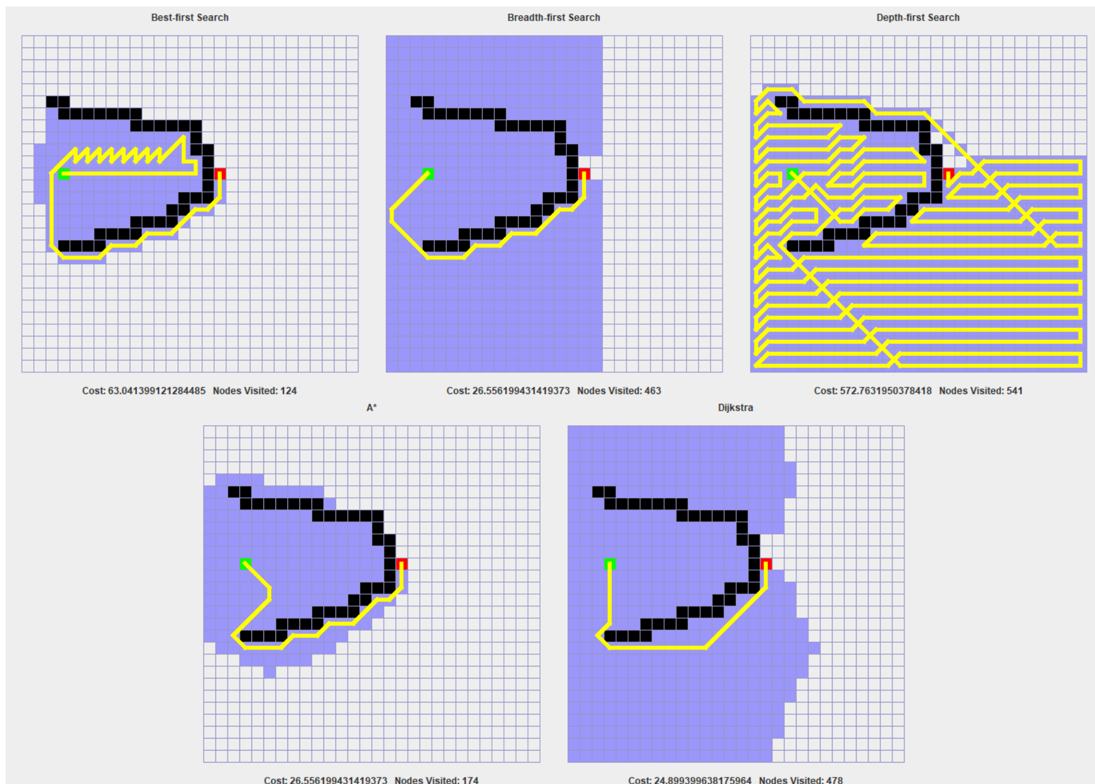
Fonte: Própria (2022)

Tabela 4 – Resultados obtidos para "map3"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	139.08	554
Breadth-first Search	72.07	626
Depth-first Search	172.20	550
A*	81.18	596
Dijkstra	72.07	626

Fonte: Própria(2022)

Figura 20 – Resultados cenário de teste "map4"



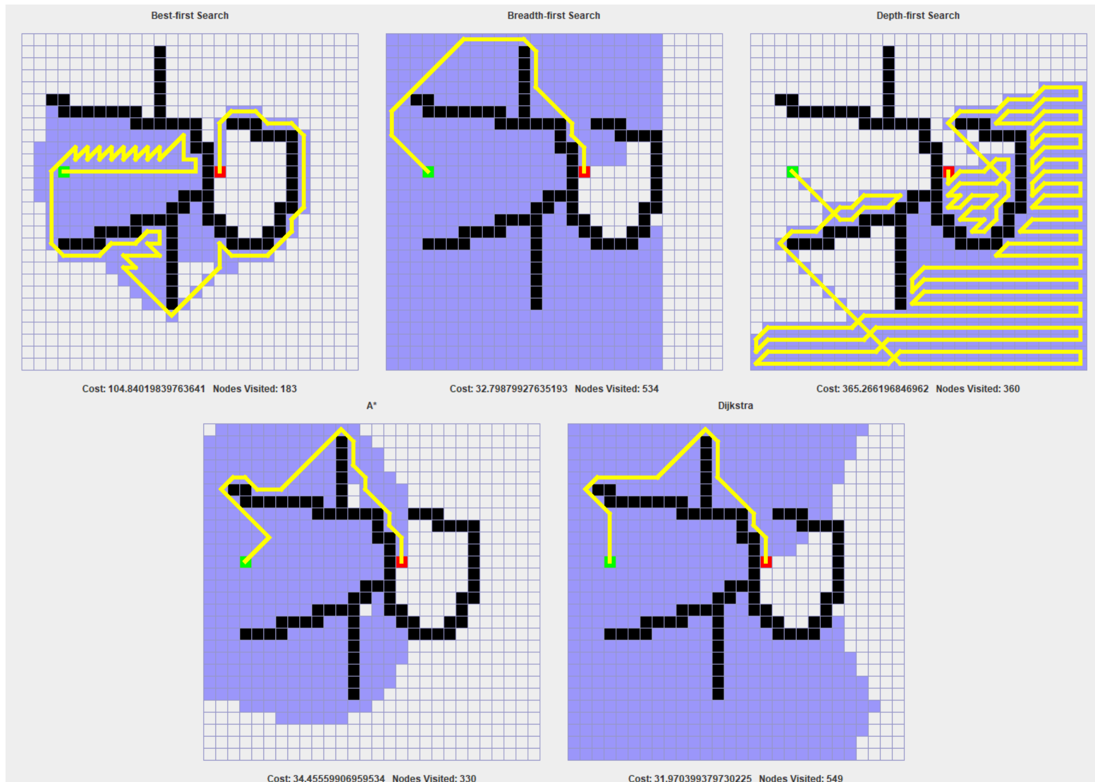
Fonte: Própria (2022)

Tabela 5 – Resultados obtidos para "map4"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	63.04	124
Breadth-first Search	26.56	463
Depth-first Search	572.76	541
A*	26.56	174
Dijkstra	24.90	478

Fonte: Própria(2022)

Figura 21 – Resultados cenário de teste "map4.1"



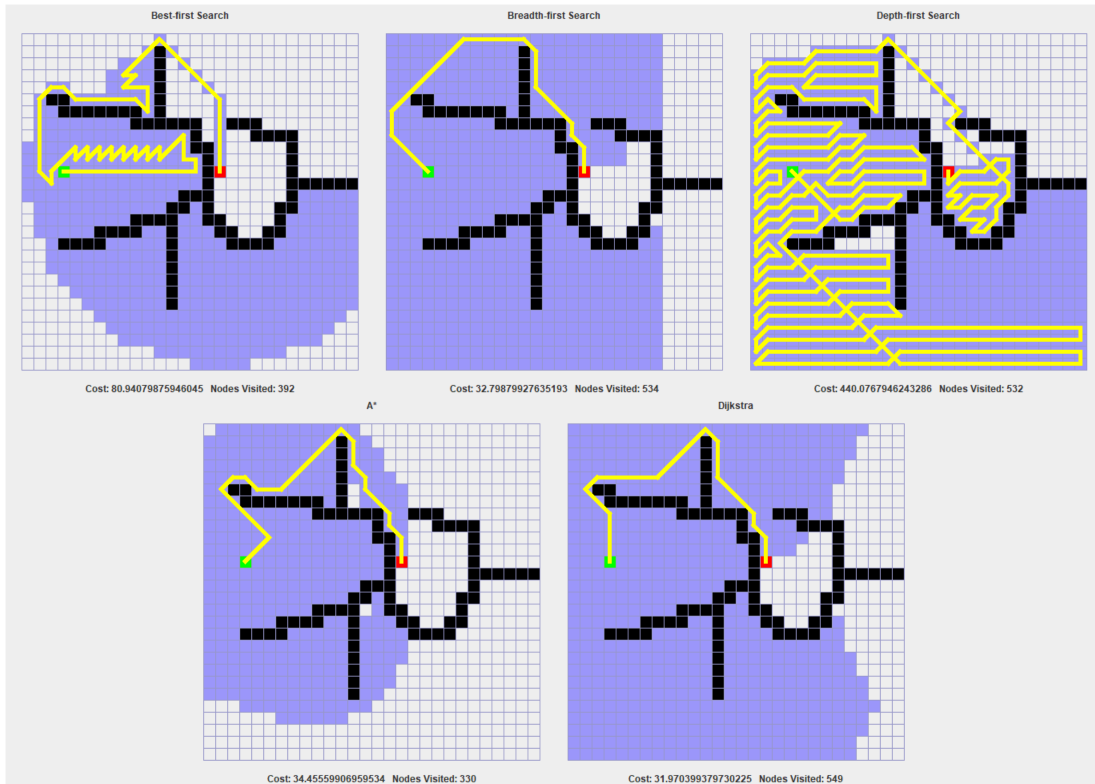
Fonte: Própria (2022)

Tabela 6 – Resultados obtidos para "map4.1"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	104.84	183
Breadth-first Search	32.80	534
Depth-first Search	365.27	360
A*	34.46	330
Dijkstra	31.97	549

Fonte: Própria(2022)

Figura 22 – Resultados cenário de teste "map4.2"



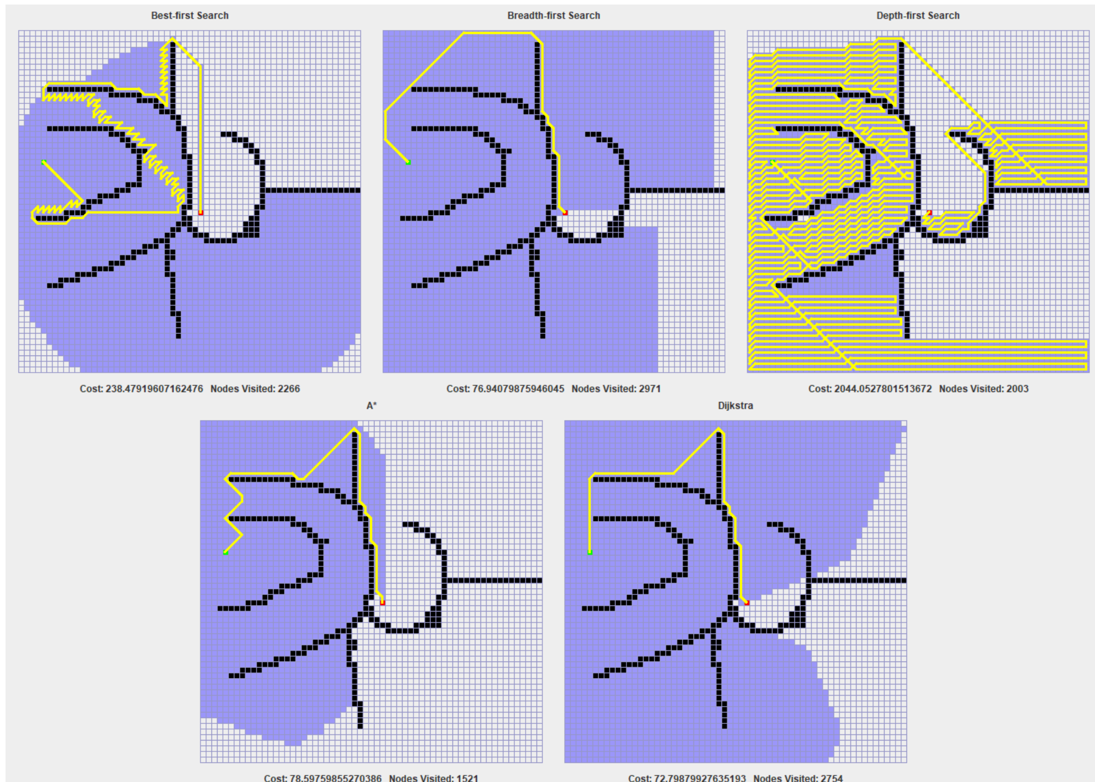
Fonte: Própria (2022)

Tabela 7 – Resultados obtidos para "map4.2"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	80.94	392
Breadth-first Search	32.80	534
Depth-first Search	440.08	532
A*	34.46	330
Dijkstra	31.97	549

Fonte: Própria(2022)

Figura 23 – Resultados cenário de teste "map5"



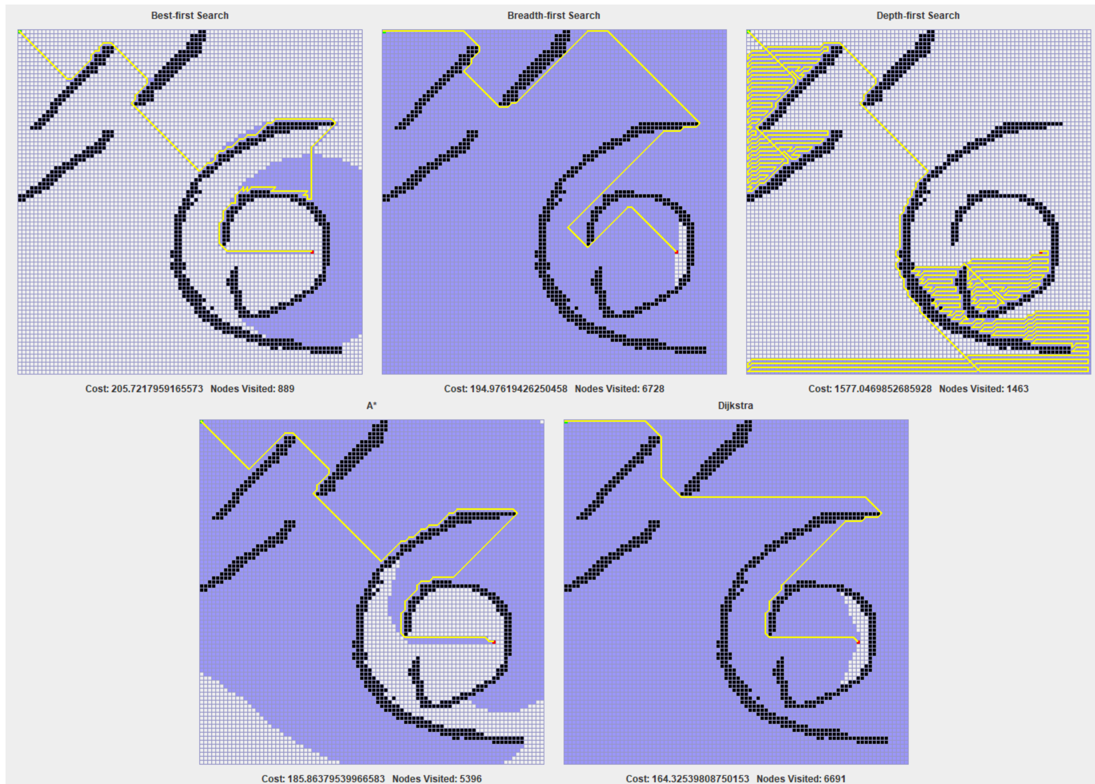
Fonte: Própria (2022)

Tabela 8 – Resultados obtidos para "map5"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	238.48	2266
Breadth-first Search	76.94	2971
Depth-first Search	2044.05	2003
A*	78.60	1521
Dijkstra	72.80	2754

Fonte: Própria(2022)

Figura 24 – Resultados cenário de teste "map6"



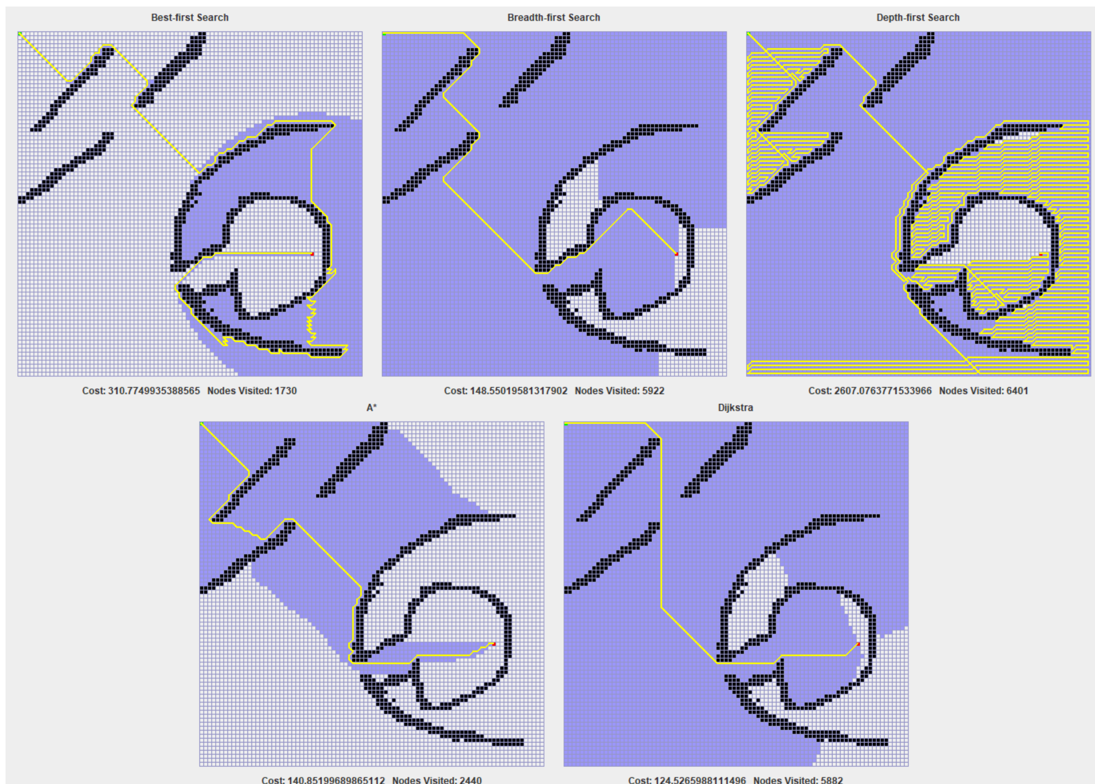
Fonte: Própria (2022)

Tabela 9 – Resultados obtidos para "map6"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	205.72	889
Breadth-first Search	194.98	6728
Depth-first Search	1577.05	1463
A*	185.86	5396
Dijkstra	164.33	6691

Fonte: Própria(2022)

Figura 25 – Resultados cenário de teste "map6.1"



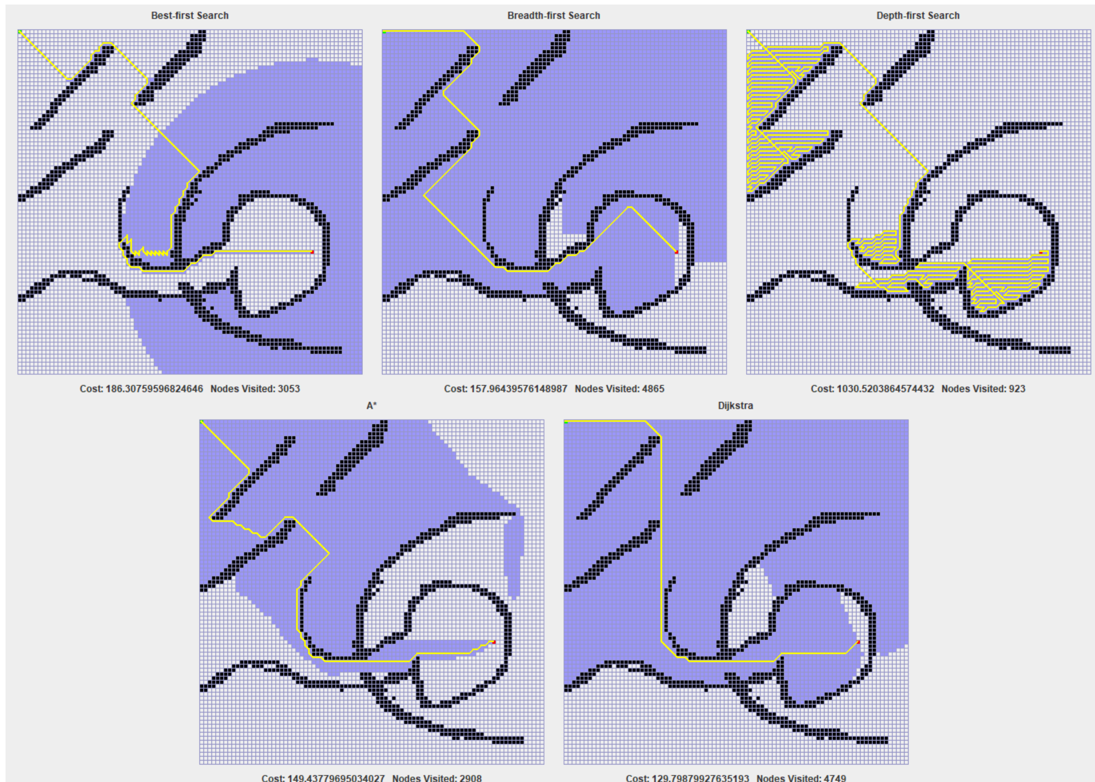
Fonte: Própria (2022)

Tabela 10 – Resultados obtidos para "map6.1"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	310.77	1730
Breadth-first Search	148.55	5922
Depth-first Search	2607.08	6401
A*	140.85	2440
Dijkstra	124.53	5882

Fonte: Própria(2022)

Figura 26 – Resultados cenário de teste "map6.2"



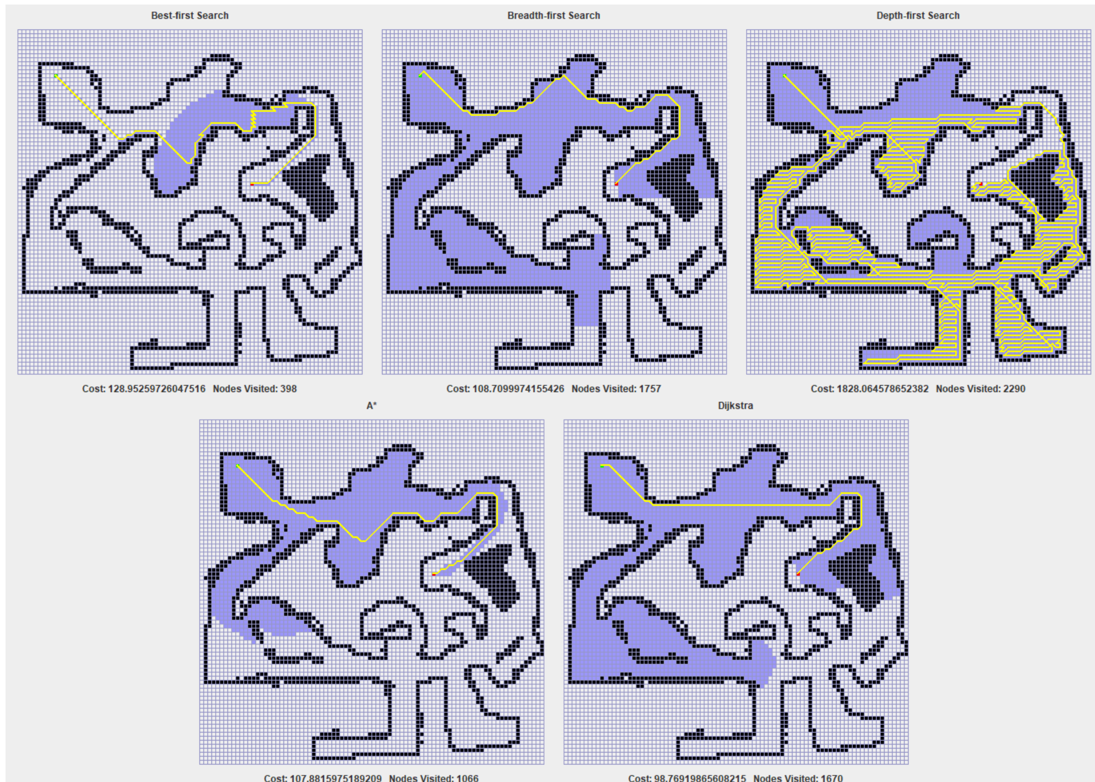
Fonte: Própria (2022)

Tabela 11 – Resultados obtidos para "map6.2"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	186.31	3053
Breadth-first Search	157.96	4865
Depth-first Search	1030.52	923
A*	149.44	2908
Dijkstra	129.80	4749

Fonte: Própria(2022)

Figura 27 – Resultados cenário de teste "map7"



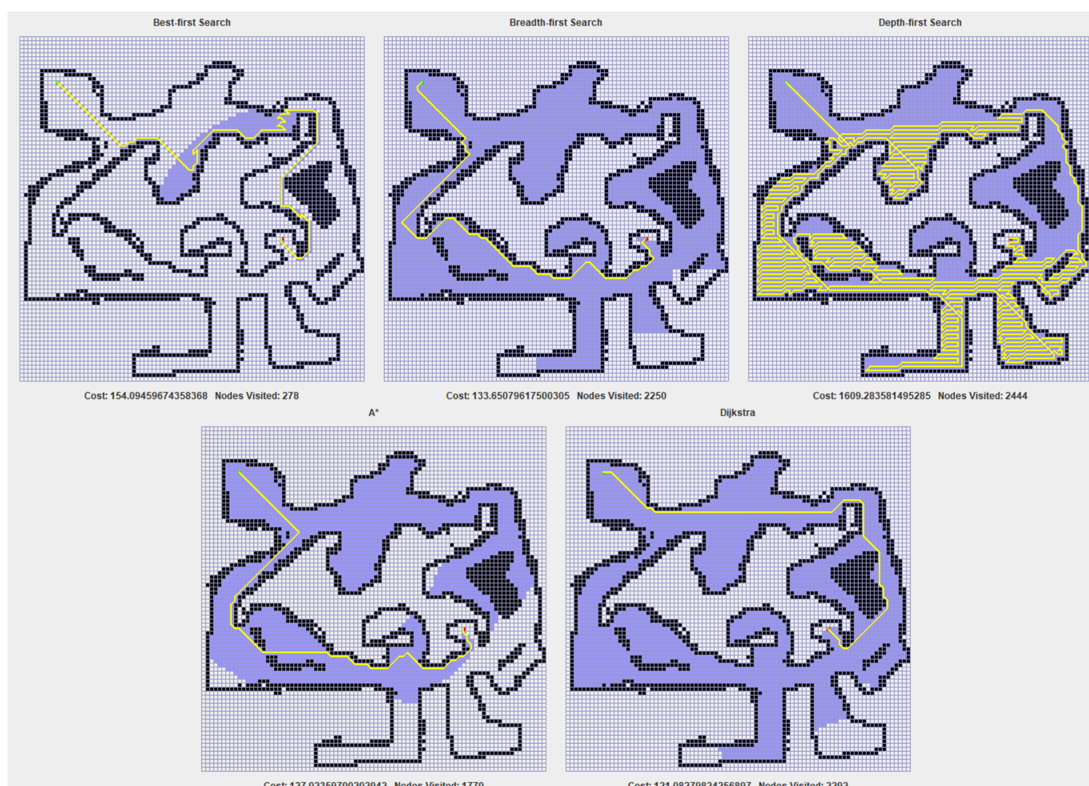
Fonte: Própria (2022)

Tabela 12 – Resultados obtidos para "map7"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	128.95	398
Breadth-first Search	108.71	1757
Depth-first Search	1828.06	2290
A*	107.88	1066
Dijkstra	98.77	1670

Fonte: Própria(2022)

Figura 28 – Resultados cenário de teste "map7.1"



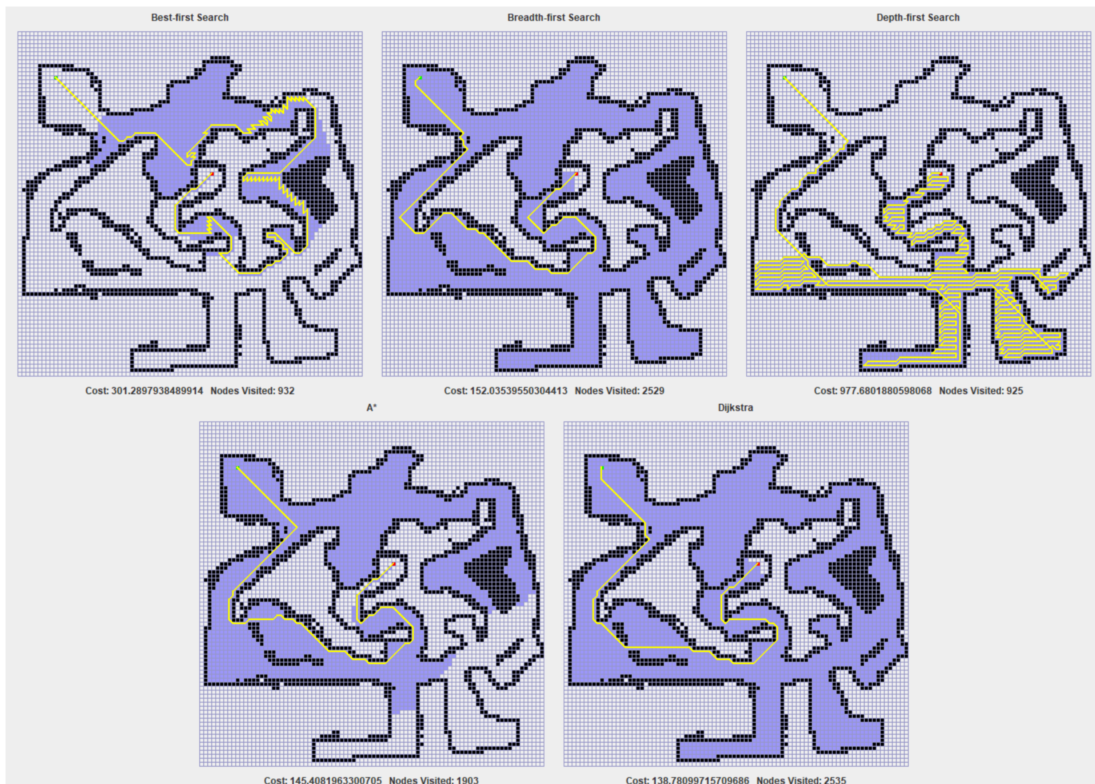
Fonte: Própria (2022)

Tabela 13 – Resultados obtidos para "map7.1"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	154.09	278
Breadth-first Search	133.65	2250
Depth-first Search	1609.28	2444
A*	127.02	1770
Dijkstra	121.08	2292

Fonte: Própria(2022)

Figura 29 – Resultados cenário de teste "map7.2"



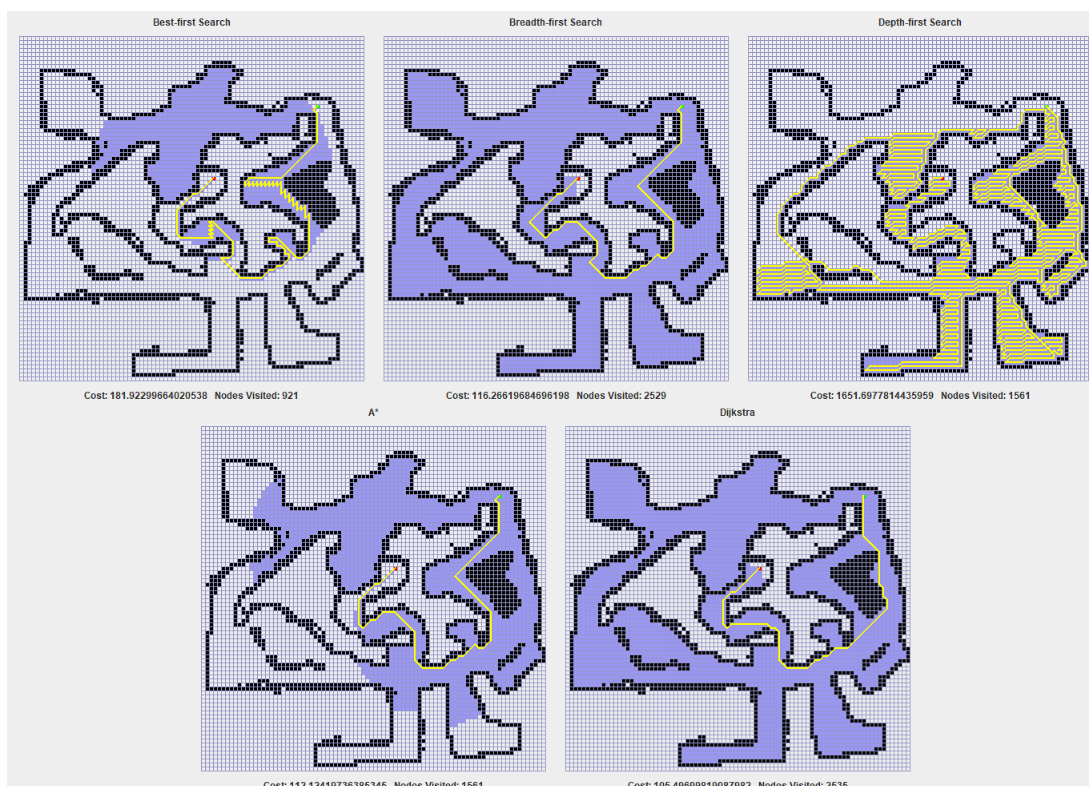
Fonte: Própria (2022)

Tabela 14 – Resultados obtidos para "map7.2"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	301.29	932
Breadth-first Search	152.04	2529
Depth-first Search	977.68	925
A*	145.41	1903
Dijkstra	138.78	2535

Fonte: Própria(2022)

Figura 30 – Resultados cenário de teste "map7.3"



Fonte: Própria (2022)

Tabela 15 – Resultados obtidos para "map7.3"

Algoritmo	Custo Caminho	Vértices Visitados
Best-first Search	181.92	921
Breadth-first Search	116.27	2529
Depth-first Search	1651.70	1561
A*	112.12	1561
Dijkstra	105.50	2535

Fonte: Própria(2022)