UNIVERSIDADE FEDERAL DO PARANÁ
SETOR DE CIÊNCIAS EXATAS
CURSO DE MATEMÁTICA INDUSTRIAL

MURILO STELLFELD DE OLIVEIRA POLOI

RESOLUÇÃO VIA ALGORITMO A* DE LABIRINTOS EM DUAS DIMENSÕES GERADOS PELO ALGORITMO DE *PRIM* SIMPLIFICADO

CURITIBA

2023

MURILO STELLFELD DE OLIVEIRA POLOI

RESOLUÇÃO VIA ALGORITMO A* DE LABIRINTOS EM DUAS DIMENSÕES GERADOS PELO ALGORITMO DE *PRIM* SIMPLIFICADO

Trabalho apresentado como requisito parcial para a obtenção do título de Bacharel em Matemática Industrial, Setor de Ciências Exatas da Universidade Federal do Paraná.

Orientador: Prof. Thiago de Oliveira Quinelato, DSc.

CURITIBA

2023

TERMO DE APROVAÇÃO

MURILO STELLFELD DE OLIVEIRA POLOI

RESOLUÇÃO VIA ALGORITMO A* DE LABIRINTOS EM DUAS DIMENSÕES GERADOS PELO ALGORITMO DE *PRIM* SIMPLIFICADO

Trabalho apresentado como requisito parcial para a obtenção do título de Bacharel em Matemática Industrial, Setor de Ciências Exatas da Universidade Federal do Paraná, pela seguinte banca examinadora:

Prof. Thiago de Oliveira Quinelato, DSc.
Orientador

Prof. Lucas Garcia Pedroso, DSc.
UFPR

Prof. Volmir Eugênio Wilhelm, D.Eng.
UFPR

Curitiba, 8 de Dezembro de 2023.

RESUMO

Este trabalho estuda a eficiência de três algoritmos de busca em grafos quanto à resolução de labirintos bidimensionais, com pontos inicial e final fixos. Os algoritmos são busca em largura, busca em profundidade e A*. Os labirintos utilizados para as comparações entre os algoritmos de busca são gerados pelo algoritmo de *Prim* simplificado. Foi observado durante os testes que adicionar um peso k à distância de *Manhattan*, função heurística escolhida para o algoritmo A*, melhora o seu desempenho em relação ao A* padrão. Em mais de 90% dos testes realizados, o algoritmo A* com peso na função heurística visitou menos nós no labirinto para encontrar a solução em relação aos algoritmos de busca em largura, busca em profundidade e A* padrão.

Palavras-chaves: algoritmos de busca em grafos; resolução de labirintos; geração de labirintos; teoria de grafos.

ABSTRACT

This work studies the efficiency of three graph search algorithms in relation to solving two dimensional mazes with fixed start and ending points. These algorithms are breadth-first search, depth-first search and A*. The mazes utilized for the comparisons between the search algorithms are generated by Prim's simplified algorithm. It was observed from the tests that adding a weight k to the Manhattan distance, the chosen heuristic function for the A* algorithm, improves the performance of the algorithm in relation to the standard A* algorithm. In more than 90% of the tests, the A* algorithm with weight in its heuristic function visited fewer nodes in the maze to find its solution in relation to breadth-first search, depth-first search and standard A*.

Key-words: graph search algorithms; maze solving; maze generation; graph theory.

SUMÁRIO

1 INTRODUÇÃO

O objetivo geral deste trabalho é estudar problemas de busca de caminho, bem como algoritmos e técnicas utilizadas para sua resolução (de acordo com a natureza do algoritmo e do problema em estudo). Deseja-se também estudar a geração de labirintos de maneira algorítmica.

1.1 OBJETIVOS

Especificamente, visa-se implementar o algoritmo A* com a função heurística da distância de Manhattan. O algoritmo será usado na resolução de problemas de labirintos, que consistem em encontrar o caminho entre um ponto inicial e outro final em um labirinto de duas dimensões, composto por paredes e/ou obstáculos.

Quanto à geração dos labirintos, com o algoritmo de *Prim* é possível gerar instâncias de labirintos a serem resolvidos.

1.2 JUSTIFICATIVA

Como mencionado em Yan (2023), o algoritmo A* busca resolver os aspectos negativos dos algoritmos DFS e BFS, combinando boa performance e acurácia, além de ser uma escolha de referência para resolução de problemas de busca de caminho (Wayahdi et al., 2021). Assim, o algoritmo é confiável para a resolução de problemas que podem ser visualizados como um labirinto.

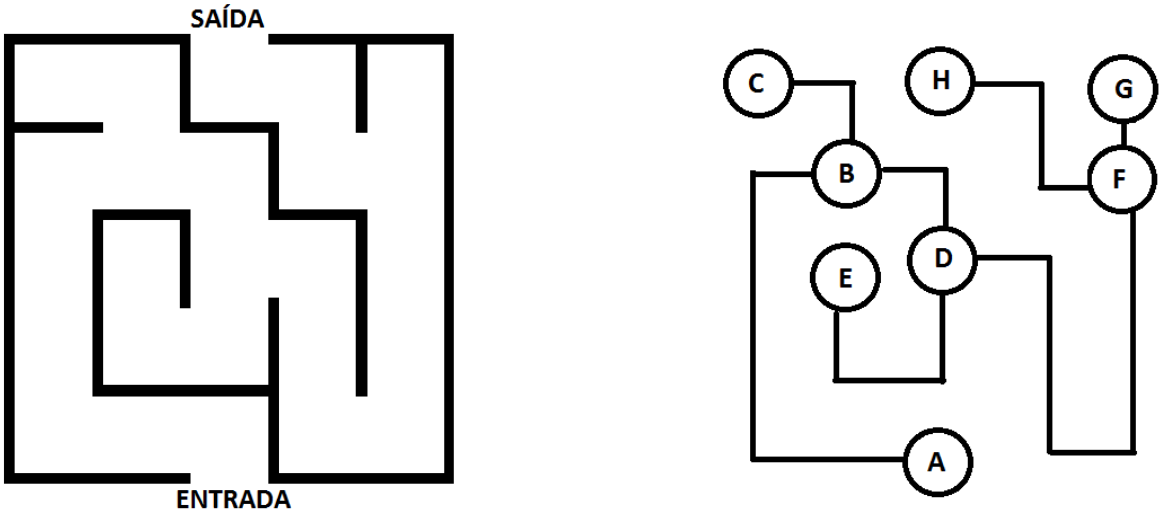
O algoritmo de *Prim* é um algoritmo clássico de geração de labirintos, considerado perfeito em sua geração, isto é, é possível visitar todos os seus espaços e há uma solução única para os pontos inicial e final estabelecidos (Bellot et al., 2021). Assim, é possível que o algoritmo A* encontre a solução dos labirintos gerados, mesmo que não seja a solução ótima.

A escolha de combinar os assuntos de resolução de labirintos via algoritmo A* e geração de labirintos via algoritmo de *Prim* deve-se à relativa simplicidade de implementar o algoritmo A* e ao problema, encontrado antes do início da escrita do trabalho, de ter um meio de gerar os labirintos para que se possam realizar a aplicação e os testes desejados com o algoritmo A*.

1.3 REFERENCIAL TEÓRICO

Uma maneira de resolver um problema de labirinto em duas dimensões casualmente é traçar uma linha contínua que liga o ponto de início do labirinto ao ponto final

FIGURA 1 – UM LABIRINTO E O GRAFO CORRESPONDENTE.



(a) Labirinto. (b) Grafo.

FONTE: Produzido pelo autor.

sem encostar em suas paredes.

É possível resolver labirintos através de algoritmos, sem a necessidade de traçar uma linha manualmente. Para isso, é necessário traduzir as informações do labirinto para algum tipo de estrutura de dados.

Grafos são estruturas de dados úteis para isso. É possível observar o labirinto como um grafo, como discorrido na Seção 5.5 de Skiena (2020). A FIGURA 1 é um exemplo desta visualização.

Ao construir o grafo, têm-se os nós (dados pelas letras A, B, . . . , H, na FIGURA 1b) com cada segmento entre dois nós denominado por aresta, às quais pode-se atribuir um valor que representa o custo de deslocar-se de um nó para outro.

O algoritmo de *Dijkstra* (Dijkstra, 1959) foi desenvolvido a fim de encontrar o menor caminho entre dois nós em um grafo.

O algoritmo A* (Hart et al., 1968) é, essencialmente, uma extensão do algoritmo de *Dijkstra* (Rachmawati; Gustin, 2020). Calcula-se também o menor custo entre um nó inicial e um nó final fixados, porém utiliza-se alguma função heurística para isso.

O funcionamento dos algoritmos de busca de caminhos em grafos abordados nesta Seção é apresentado no Capítulo 3.

Como mencionado anteriormente, labirintos podem ser representados por grafos. Assim, é possível utilizar o algoritmo A* para encontrar a solução dos mesmos. Monteiro (2018) estudou a aplicação dos algoritmos A*, busca em largura (BFS – *Breadth-First Search*) e busca em profundidade (DFS – *Depth-First Search*) à solução de labirintos, a fim de observar a performance da linguagem de programação *Julia* ao

executar os algoritmos de busca mencionados.

1.4 METODOLOGIA

A base da geração dos labirintos será o algoritmo de *Prim* (Prim, 1957), que encontra a árvore geradora miníma em um grafo, isto é, a árvore de caminhos no grafo que possui o menor custo a partir de um nó inicial escolhido. O algoritmo utilizado será uma versão modificada da versão clássica (Buck, 2015), que utiliza aleatoriedade para gerar o labirinto.

A implementação será feita exclusivamente na linguagem de programação *Julia*. Serão apresentados três exemplos de labirintos gerados, de tamanhos que foram considerados pelo autor como pequeno, médio e grande.

A seguir, são feitas comparações utilizando a quantidade de nós visitados pelos algoritmos i) A*; ii) busca em profundidade; iii) busca em largura, para encontrar a solução do labirinto.

2 FUNDAMENTAÇÃO TEÓRICA

Grafos são ferramentas que permitem representar a relação entre diferentes informações dentro de um mesmo contexto. Em termos de geolocalização, por exemplo, podem ser usados para representar quais cidades são vizinhas entre si e a distância entre elas.

Essas estruturas de dados possuem algumas propriedades relevantes para os problemas em estudo como, por exemplo, se a ligação entre duas informações é unidirecional, se há algum valor atribuído entre elas ou se há algum tipo de ciclo entre elas, isto é, se é possível partir de uma informação inicial e retornar a ela em algum momento.

Assim, para poder construir a solução do problema do caminho em um labirinto utilizando as ideias do algoritmo A*, é introduzida neste capítulo uma definição breve de grafos, assim como algumas propriedades importantes.

Quanto à geração de labirintos, existem vários algoritmos disponíveis. É possível usar técnicas simples, como escolher aleatoriamente entre dois possíveis caminhos a serem abertos no labirinto, isto é, o algoritmo da Busca Binária (Buck, 2015). Também, há técnicas que envolvem conceitos baseados na teoria de grafos, como o algoritmo de *Prim* (Buck, 2015). As duas referências citadas neste parágrafo contêm diversas abordagens diferentes para gerar labirintos.

A escolha do algoritmo de *Prim* se dá pelo fato do mesmo usar conceitos da teoria de grafos, área de estudo em comum com as ideias utilizadas pelos algoritmos de busca contemplados nesse trabalho. A compreensão do funcionamento da versão utilizada do algoritmo de *Prim* é simples, pois visa construir de maneira aleatória uma árvore geradora mínima a partir de um grafo com pesos iguais.

Outra versão do algoritmo de *Prim* envolve marcar nós do grafo dentro de três possíveis estados: Interno, Externo ou Fronteira. A partir dessas características, formam-se os caminhos do labirinto. Uma última versão do algoritmo de *Prim* considera que os pesos nas arestas ou nós são definidos aleatoriamente e utiliza esses pesos para escolher os caminhos a serem formados no labirinto.

É necessário falar também dos algoritmos de busca em largura e busca em profundidade, além do algoritmo de *Dijkstra* que possui similaridade com o algoritmo A* (Rachmawati; Gustin, 2020). Porém, somente os dois primeiros terão seus resultados comparados com os resultados do algoritmo A*.

O algoritmo de *Dijkstra* não é atrativo para os testes devido a sua similaridade

com o algoritmo A*. Ao considerar que a função heurística é igual a zero, temos o algoritmo de *Dijkstra* (Rachmawati; Gustin, 2020). Ainda mais, o algoritmo de *Dijkstra* é guloso, no sentido de decidir qual caminho seguir utilizando valores locais, enquanto que a função heurística utilizada no algoritmo A* estima a distância do nó atual ao nó final, priorizando assim caminhos melhores.

2.1 CONCEITOS DE GRAFOS

Um grafo pode ser definido como um par ordenado $G = (V, E)$, sendo V um conjunto de vértices, também chamados de nós, e E, do inglês *edges*, um conjunto de arestas, que são as conexões entre dois vértices, representadas por pares ordenados.

Grafos orientados são grafos em que as arestas possuem alguma orientação, geralmente indicada por setas. Por exemplo, na FIGURA 2, é possível sair do nó A aos nós B, C e D, do nó B somente ao nó D, do nó C a nenhum dos outros nós e do nó D aos nós A e C. É possível também ter grafos não orientados, como o da FIGURA 1b.

Grafos com peso, ou ainda, ponderados, possuem valores numéricos associados aos seus vértices ou arestas. Estes valores dependem do problema em estudo, podendo representar distância entre duas cidades, preço de pedágio, entre outros.

Grafos cíclicos contêm ao menos um ciclo, sendo um ciclo um caminho sem vértices repetidos em que os nós inicial e final coincidem. Por exemplo, na FIGURA 2, há um ciclo ao percorrer o caminho (A, B, D, A).

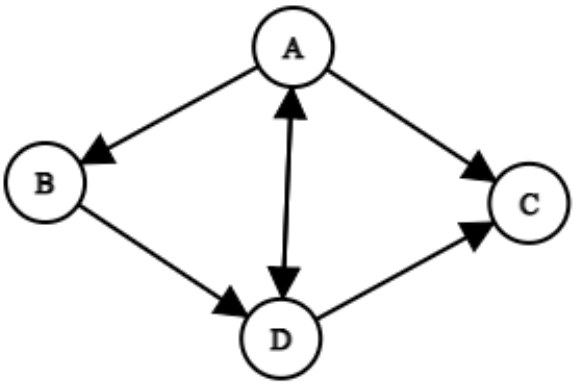
A árvore geradora de um grafo é um subgrafo (contém todos os nós do grafo original, mas nem todas as arestas). Todos os nós da árvore possuem pelo menos uma conexão. Caso o grafo original seja orientado, o grafo da sua árvore geradora também será orientado. A FIGURA 1b mostra uma árvore geradora.

Ainda, pode-se construir uma árvore geradora mínima, que é a árvore geradora com a menor soma total dos pesos dos nós ou arestas incluídos na árvore, isto é, o caminho de menor custo total que pode ser percorrido para acessar todos os nós a partir do nó inicial. Vale ressaltar que um grafo pode ter mais de uma árvore geradora mínima.

Tem-se também a árvore de caminho mais curto, que possui um nó fonte, e a partir dele se constrói o menor caminho entre ele e os nós restantes do grafo. A árvore de caminho mais curto pode ser obtida usando o algoritmo de *Dijkstra*.

Enquanto a árvore geradora mínima busca conectar todos os nós de um grafo com o menor peso total, a árvore de caminho mais curto encontra os menores caminhos entre um nó fonte e todos os outros nós do grafo.

FIGURA 2 – UM GRAFO DIRECIONADO.



FONTE: Produzido pelo autor.

2.2 GERAÇÃO ALGORÍTMICA DE LABIRINTOS

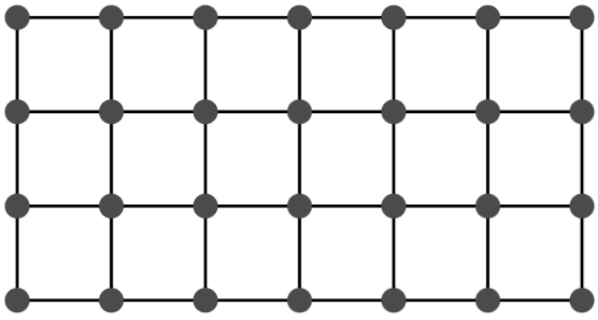
No contexto de geração algorítmica de labirintos, tem-se paredes, espaços e fronteiras.

As paredes são obstáculos dentro do labirinto, não é possível atravessá-las. Os espaços são lugares em que se pode andar pelo labirinto, nas direções norte, sul, leste ou oeste. As fronteiras são paredes que cercam o labirinto, impedindo que a parte interna e a parte externa se conectem por um espaço.

No contexto de representação de labirintos como grafos, os nós representam uma célula ou espaço do labirinto, enquanto que as arestas indicam se é possível percorrer um caminho entre dois nós. Caso não haja uma aresta entre dois nós, tem-se então uma parede.

Caso um labirinto tivesse somente fronteiras, mas nenhuma parede, isso significaria que é possível andar por ele em qualquer direção. A FIGURA 3 representa um labirinto como grafo que possui somente fronteiras. Os nós e arestas do grafo representam informações da parte interna do labirinto.

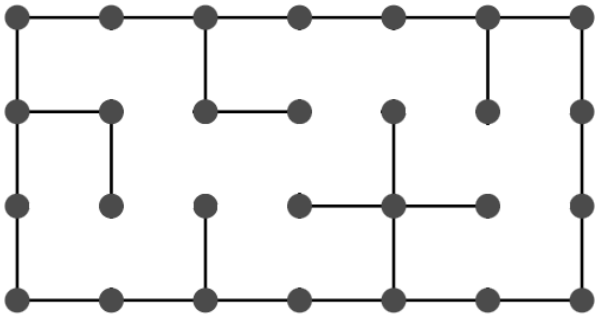
FIGURA 3 – UM GRAFO DE MALHA.



FONTE: Adaptado de (Weisstein, 2023).

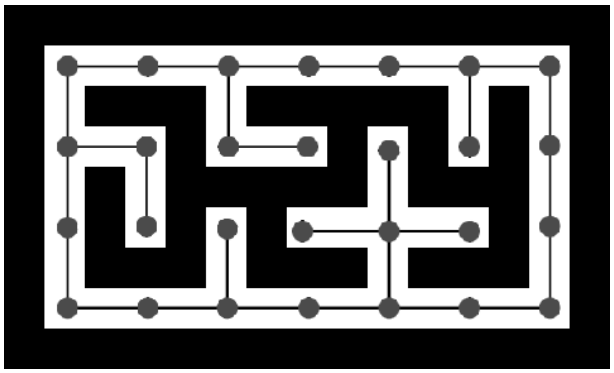
Para acrescentar paredes dentro desse labirinto, baseando-se no grafo de malha da FIGURA 3, é possível remover algumas arestas aleatoriamente, construindo assim o labirinto da FIGURA 4.

FIGURA 4 – UM LABIRINTO EM FORMA DE GRAFO.



FONTE: Adaptado de (Weisstein, 2023).

FIGURA 5 – UM LABIRINTO A PARTIR DE UM GRAFO.



FONTE: Autor.

A FIGURA 5 é o labirinto obtido a partir do grafo da FIGURA 4. A fronteira foi adicionada à parte, pois ela não é representada no grafo.

Para o algoritmo de *Prim* simplificado, utilizam-se árvores geradoras mínimas para gerar os labirintos.

Como apresentado por Buck (2015), há três diferentes versões do algoritmo de *Prim* para geração de labirintos:

a) Simplificado: Para esta versão, considera-se que todos os pesos das arestas de um grafo de malha são iguais.

   Parte-se de um nó inicial aleatório a um vizinho aleatório deste nó inicial, abrindo o caminho entre eles.

Os vizinhos do nó vizinho escolhido são adicionados à lista de nós não visitados e escolhe-se aleatoriamente um nó dessa lista para se tornar o nó atual.

A estratégia de escolher aleatoriamente algum dos vizinhos do nó atual e abrir um caminho entre o nó atual e o nó escolhido da lista de nós não visitados é repetida, até que a lista esteja vazia;

b) Modificado: Seguindo a mesma ideia da versão simplificada, porém, ao invés de considerar arestas, consideram-se os nós de um grafo de malha.

Cada nó é de um tipo: interno, externo ou de fronteira.

Durante a execução do algoritmo, nós internos são aqueles que já são parte do labirinto; nós externos são os que não são parte do labirinto e que não são vizinhos de nenhum nó interno; e nós de fronteira não são parte do labirinto mas são vizinhos de pelo menos um nó interno.

Escolhe-se então um nó para ser interno e seus vizinhos tornam-se nós de fronteira. Sorteia-se um desses nós de fronteira e cria-se um caminho entre o nó de fronteira escolhido e algum nó vizinho dele que seja interno, mudando este nó de fronteira para interno, atualizando o tipo dos nós vizinhos.

Repete-se o processo até que não haja mais nós de fronteira. Consequentemente, não haverá mais nós externos;

c) Verdadeiro: Atribuem-se pesos aleatórios a todas as arestas de um grafo de malha.

Parte-se de algum nó inicial e seleciona-se a aresta de menor peso ligada ao nó inicial.

O nó vizinho que conecta a aresta de menor peso ao nó inicial torna-se o nó atual. O processo é repetido entre as arestas do nó atual até que todos os nós tenham sido explorados.

No final, as arestas selecionadas compõem uma árvore geradora mínima, que representa o labirinto gerado.

Uma característica notável do algoritmo de *Prim* é o fato de gerar labirintos perfeitos, isto é, fixados dois pontos, um inicial e um final, haverá apenas um caminho entre eles.

2.3 MÉTODOS DE BUSCA DE CAMINHOS EM GRAFOS

Um caminho é uma sequência de arestas que conecta dois nós (Skiena, 2020). Há diversos algoritmos que têm como objetivo encontrar o melhor caminho entre dois nós de um grafo.

É possível concluir a partir da FIGURA 1b que o caminho mais curto (e neste caso o único) que deve-se seguir para resolver o labirinto da FIGURA 1a é partir do início, representado pelo nó A ao nó B, então para o nó D, para o nó F e chegar no fim, ao nó H.

A diferença entre algoritmos que encontram caminhos em grafos está na estratégia utilizada, mas tem-se exemplos de algoritmos que são uma modificação de um algoritmo já existente.

Esse é o caso dos algoritmos de *Dijkstra* e o algoritmo A*, pois, em essência, o algoritmo A* é o algoritmo de *Dijkstra* com o uso de alguma função heurística (Rachmawati; Gustin, 2020).

O uso de uma função heurística permite avaliar o problema de maneira global e não local, como o algoritmo de *Dijkstra* o faz ao considerar apenas a distância entre o nó sendo explorado e seus vizinhos. Assim, é possível dizer que o algoritmo de *Dijkstra* é guloso.

2.3.1 Algoritmo de *Dijkstra*

A versão apresentada do algoritmo de *Dijkstra* (Skiena, 2020) é a que parte de um nó fonte e encontra o caminho mais curto entre o nó fonte e um nó final escolhido, juntamente do menor caminho entre os outros nós do grafo. O grafo deve possuir pesos nos vértices ou arestas, gerando assim uma árvore de caminho mais curto.

O Pseudocódigo 1 tem como variáveis de entrada G, s e t (respectivamente, o grafo, o nó fonte e o nó final escolhido).

Inicializa-se um conjunto de nós já visitados e aos ainda não visitados atribui-se o maior peso possível, simbolizado por ∞. Os pesos dos nós conectados por uma aresta ao nó fonte s são então adicionados ao vetor $dist$, que contém informação das distâncias (neste caso, está sendo considerado que os pesos estão nas arestas do grafo). Após terminar de explorar o nó fonte s, marca-se ele como o último explorado.

O laço de repetição da linha quinze termina quando o último nó explorado é o nó final escolhido. Enquanto não se chega nele, escolhe-se o próximo nó a ser explorado, desejando minimizar a distância local entre ele e os nós conectados a ele, e com isso os pesos dos nós são atualizados conforme o algoritmo encontra caminhos mais curtos, atualizando o vetor $dist$.

Pseudocódigo 1 Algoritmo de Dijkstra

```
1: function DIJKSTRAMENORCAMINHO(G, s, t)
2:
3:     explorado = [s]
4:
5:     for i = 1 até n do
6:         dist [i] = ∞
7:     end for
8:
9:     for cada aresta (s, v) do
10:        dist [v] = w (s, v)
11:    end for
12:
13:    ultimo = s
14:
15:    while ultimo ≠ t do
16:        selecione v_next, o vértice que minimiza a distância dist [v]
17:
18:        for cada aresta (v_next, x) do
19:            dist [x] = min [dist [x] , dist [v_next] + w (v_next, x)]
20:        end for
21:
22:        ultimo = v_next
23:        explorado = explorado ∪ [v_next]
24:
25:    end while
26:
27: end function
```

3 ALGORITMOS IMPLEMENTADOS

Dentre as três versões apresentadas na Seção 2.2, foi escolhida a versão simplificada do algoritmo de *Prim*. Será apresentado então o código implementado referente ao algoritmo de *Prim* simplificado.

Os labirintos gerados por esse algoritmo são representados por matrizes contendo apenas uns, que simbolizam paredes, e zeros, que simbolizam caminhos que podem ser percorridos dentro do labirinto. Por exemplo, a matriz que representa o labirinto da FIGURA 5 é:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

É apresentada também uma implementação dos algoritmos referentes a busca de caminhos em grafos, sendo eles os algoritmos A*, busca em largura e busca em profundidade.

Adicionalmente, foram implementados dois algoritmos auxiliares, sendo o propósito de um deles encontrar os nós vizinhos do nó atual, de acordo com o algoritmo de *Prim* simplificado e, do outro, traduzir o labirinto gerado pelo algoritmo de *Prim* (que está na forma de uma matriz) para um dicionário que representa o grafo que descreve o labirinto gerado. Por exemplo, o grafo representado pela matriz

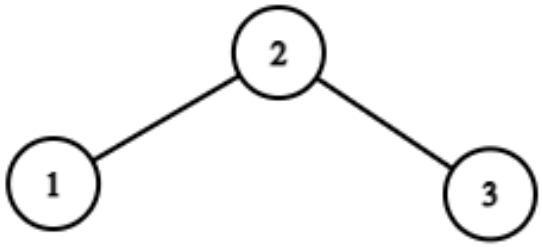
$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

pode ser representado em *Julia* pelo dicionário

```
Dict((2,2) => [(2,3)], (2,3) => [(2,2),(2,4)],
     (2,4) => [(2,3)]),
```

isto é, é possível ir da coordenada $(2,2)$ à $(2,3)$, da coordenada $(2,3)$ à $(2,2)$ e $(2,4)$ e da coordenada $(2,4)$ à $(2,3)$. A FIGURA 6 mostra o grafo correspondente.

FIGURA 6 – GRAFO COM TRÊS NÓS E DUAS ARESTAS.



FONTE: Autor.

A implementação do algoritmo de *Prim* simplificado da Seção 3.1 e do algoritmo da Seção 3.1.1 foram baseadas na biblioteca *mazelib* (Stilley, 2023), implementada em *Python*. O algoritmo dos métodos de busca A*, busca em largura e busca em profundidade foram retirados de Monteiro (2018).

3.1 ALGORITMO DE *PRIM*

O processo de geração de labirintos via algoritmo de *Prim* simplificado inicia-se com um labirinto composto somente de paredes, ou seja, uma matriz de uns. Escolhe-se então um elemento aleatório da matriz que não esteja na fronteira, isto é, que não esteja na primeira ou na última coluna e/ou linha da matriz e que tenha coordenadas pares. Este elemento se torna a célula atual.

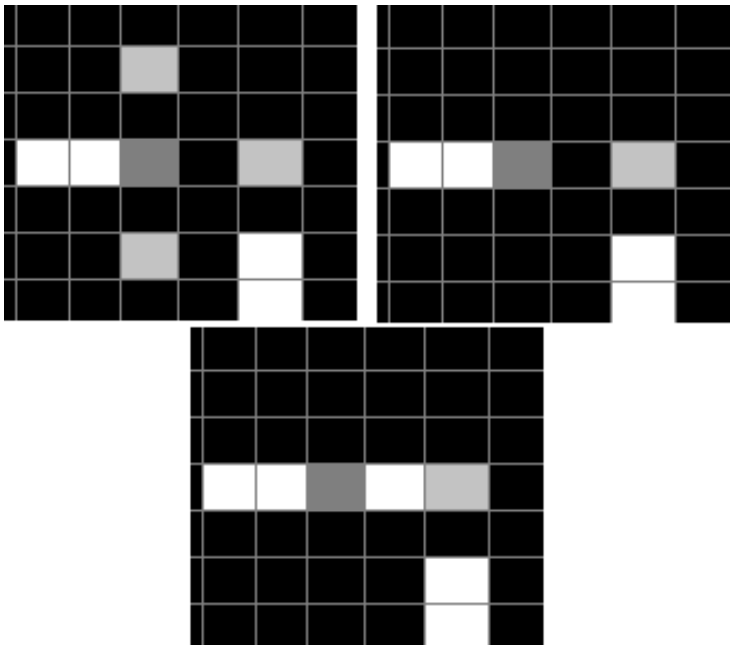
Da célula atual, observam-se células vizinhas do labirinto que estão a duas células de distância que ainda são paredes mas não são fronteiras. Dentro de todas as possibilidades de vizinhos, escolhe-se uma aleatoriamente e então é aberto um caminho entre a célula atual e a célula vizinha escolhida.

Esse processo é exemplificado na FIGURA 7, adaptada de Stenkrona (2019). As células pretas representam paredes; as brancas, caminhos formados. A célula atual tem a cor cinza escuro e os vizinhos desta célula, cinza claro.

Na imagem do canto superior esquerdo observam-se os vizinhos da célula atual; na do canto superior direito, decide-se qual célula vizinha será usada e, na imagem de baixo, abre-se o caminho entre a célula atual e a célula vizinha escolhida.

O algoritmo de *Prim* simplificado é implementado na função $generate$ (Algoritmo 3.1) e baseado em Stilley (2023). A função é inicializada com os inteiros positivos h e w, a altura e a largura desejadas para o labirinto. Caso h ou w for menor que três, o labirinto é considerado muito pequeno e, portanto, não é possível prosseguir com a geração do mesmo.

FIGURA 7 – FUNCIONAMENTO DO ALGORITMO DE *PRIM* SIMPLIFICADO.



FONTE: Adaptado de Stenkrona (2019).

A matriz que representa o labirinto é, na verdade, uma matriz de tamanho $H \times W$. É possível ver isso quando o grafo da FIGURA 4 foi traduzido para o labirinto da FIGURA 5: o grafo da FIGURA 4 tem sete nós na horizontal e quatro na vertical. Para representar esse grafo como um labirinto, a matriz que o representa precisa ter um tamanho 9×15, que é o caso da FIGURA 5.

Realiza-se o processo de inicializar a matriz que representará o labirinto e escolhe-se o ponto inicial para começar o processo de geração do algoritmo de *Prim* simplificado.

Dentro do laço de repetição, repete-se o processo de observar os vizinhos da célula atual, escolher um deles aleatoriamente e abrir um caminho da célula atual até a célula vizinha escolhida, até ter explorado o número total de nós que o labirinto tem em sua forma de grafo. O número total de nós do labirinto é dado por $h \times w$.

No final desse processo, retorna-se uma matriz que representa o labirinto, composta por uns e zeros, sendo uns paredes e zeros caminhos.

Algoritmo 3.1 – Algoritmo de Prim simplificado.

```
function generate(h, w)
    if h < 3 || w < 3
        error("Altura e largura muito pequenos!")
    end

    H, W = 2*h + 1, 2*w + 1
```

```
    grid = Int8.(ones(H,W))
    curr_row = rand(2:2:H)
    curr_col = rand(2:2:W)
    grid[curr_row, curr_col] = 0
    nb = find_nb(H,W, curr_row, curr_col, grid, true)
    visited = 1

    while visited < h*w
        nn = rand(1:length(nb))
        curr_row, curr_col = nb[nn]
        visited += 1
        grid[curr_row, curr_col] = 0
        nb = union(nb[1:nn-1], nb[nn+1:end])

        tmp = find_nb(H, W, curr_row, curr_col, grid, false)
        a, b = tmp[rand(1:length(tmp))]
        grid[div(curr_row + a, 2), div(curr_col + b, 2)] = 0

        unvisited = find_nb(H, W, curr_row, curr_col, grid,
        true)
        nb = unique(append!(nb, unvisited))
    end
    return grid
end
```

3.1.1 Algoritmo para encontrar os vizinhos no algoritmo de *Prim* simplificado

O algoritmo auxiliar do algoritmo de *Prim* simplificado é implementado na função $find\_nb$ (Algoritmo 3.2) e baseado em Stilley (2023).

Suas variáveis de entrada H e W são o tamanho da matriz que representa o labirinto, r e c são as coordenadas da célula atual, $grid$ é a matriz que está sendo usada para armazenar o labirinto e is_wall é uma variável booleana: se for *true*, indica que buscam-se vizinhos que são paredes; caso contrário, vizinhos que são caminhos.

Primeiro, inicializa-se um vetor vazio para armazenar as coordenadas dos vizinhos buscados, caso existam.

Cada bloco *if* olha para uma direção cardinal relativa ao ponto atual (norte, sul, oeste e leste) e verifica se o ponto vizinho naquela direção não está sobre a fronteira. O teste também depende do valor que foi passado no parâmetro *is_wall*, que define se estamos buscando paredes ou caminhos. Satisfeitas essas condições, adiciona-se

uma tupla com as coordenadas do vizinho da qualidade desejada ao vetor ns.

No final, o vetor ns é embaralhado, o que é algo natural de se fazer levando em consideração que está sendo utilizado um método que usa aleatoriedade como base. Por fim, o vetor com a lista de vizinhos é retornado.

Algoritmo 3.2 – Algoritmo auxiliar do algoritmo de Prim simplificado.

```
function find_nb(H, W, r, c, grid, is_wall)
    ns = []
    if r > 2 && grid[r-2,c] == is_wall
        push!(ns, (r-2,c))
    end
    if r < H - 1 && grid[r+2,c] == is_wall
        push!(ns, (r+2,c))
    end
    if c > 2 && grid[r,c-2] == is_wall
        push!(ns, (r,c-2))
    end
    if c < W - 1 && grid[r,c+2] == is_wall
        push!(ns, (r,c+2))
    end
    k = Random.shuffle(ns)
    return k
end
```

3.2 ALGORITMO A*

O que difere o algoritmo A* do algoritmo de *Dijkstra* é o fato de que o algoritmo A* utiliza funções heurísticas para escolher qual caminho seguir. Há diversas possibilidades de escolha para a função heurística; para cada problema haverá alguma melhor. Ainda assim, elas não garantem que o caminho encontrado pelo algoritmo A* seja sempre ótimo ou melhor que outros algoritmos de busca de caminho.

Sejam I o nó inicial, F o nó final, A o nó atual e V algum nó vizinho do nó atual. A distância estimada pelo algoritmo A* entre o nó de início e o nó final é dada por

$$f(I, F) = g(I, A) + h(V, F), \tag{3.1}$$

onde $f(I, F)$ é a distância do nó inicial ao nó final, $g(I, A)$ é o custo total de se ter chegado ao nó atual e $h(V, F)$ é a distância calculada pela função heurística, de algum nó vizinho do nó atual ao nó final.

A função heurística utilizada neste trabalho foi a distância de *Manhattan*, considerando movimento apenas nas direções norte, sul, leste e oeste. Dados dois nós A e B, de coordenadas (a_1, a_2) e (b_1, b_2), respectivamente, a distância de *Manhattan* entre os nós A e B é dada por

$$h(A, B) = |a_1 - b_1| + |a_2 - b_2| . \tag{3.2}$$

O algoritmo A*, implementado na função $astar$ (Algoritmo 3.3) e baseado em Monteiro (2018), tem como variáveis de entrada $graph$, que é um dicionário contendo as informações do grafo que representa o labirinto, $start$, o ponto de início do labirinto, $goal$, o ponto final, e k, um peso colocado na função heurística escolhida.

Inicializa-se uma estrutura de dados do tipo *BinaryHeap*, com uma tupla que contém as informações necessárias relativas ao nó inicial, sendo elas o custo determinado por f na EQUAÇÃO 3.1, o custo acumulado atual, dado por g na EQUAÇÃO 3.1, uma variável do tipo *String* que armazena o caminho que deve ser percorrido para resolver o labirinto e o ponto de início do labirinto. Inicializa-se também um vetor vazio para armazenar a informação dos nós que já foram visitados.

Heaps são estruturas de dados que mantêm com eficiência a ordenação de seus elementos, podendo escolher entre o valor mínimo ou máximo de um conjunto de dados que pode mudar dinamicamente (JuliaCollections, 2023). Neste caso, escolhemos o valor mínimo pois deseja-se minimizar a distância total que é dada por f na EQUAÇÃO 3.1. Este valor é o primeiro elemento das tuplas adicionadas à *BinaryHeap*.

O *while* da função $astar$ é repetido até que a fila esteja vazia; nesse caso a solução não foi encontrada e retorna-se ∞ para simbolizar isso.

O *while* é interrompido pelo *return* caso a solução seja encontrada. Assim, o algoritmo retorna o caminho encontrado e os nós visitados caso o nó atual seja o nó final. O laço *for* adiciona os nós vizinhos do nó atual à fila.

Algoritmo 3.3 – Algoritmo A*.

```
function astar(graph, start, goal, k)
    q = BinaryHeap{Tuple{Float64, Int64, String, Tuple{Int64,
    Int64}}, DataStructures.FasterForward}([(h(start, goal, k),
    0, "", start)])
    visited = []
    while isempty(q) != true
        order, cost, path, curr = pop!(q)
        if curr == goal
            push!(visited, curr)
```

```
            return path, visited
        elseif curr in visited
            continue
        end
        push!(visited, curr)
        for (dir, nb) in graph[curr]
            push!(q,
            (cost + h(nb, goal, k), cost + 1, path*dir, nb))
        end
    end
    return Inf
end
```

3.2.1 Algoritmo tradutor de matriz para dicionário

O algoritmo auxiliar ao algoritmo A*, implementado na função $matrix2dict$ (Algoritmo 3.4), converte a matriz que representa o labirinto para um dicionário que contém as informações do grafo que representa este labirinto. A função tem como variável de entrada a matriz A, que representa o labirinto a ser resolvido.

Para isso, todas as entradas da matriz são avaliadas. Quando é encontrado um zero (um caminho), observam-se os seus vizinhos a sul, norte, oeste e leste. Caso o vizinho observado também seja um zero, guardamos a informação da direção, representada por “S”, “N”, “O” e “L” e a coordenada desse vizinho.

No fim, retorna-se um dicionário com as informações armazenadas, em que as chaves são os nós e os valores de cada chave são seus respectivos vizinhos, com a informação da direção relativa ao nó da chave e as coordenadas do vizinho.

Algoritmo 3.4 – Algoritmo auxiliar do algoritmo A*.

```
function matrix2dict(A)
    m, n = size(A)
    keys, values = [], []
    for i = 1:m
        for j = 1:n
            if A[i,j] == 0
                valid_dir = []; push!(keys, (i,j))
                if A[i+1, j] == 0
                    push!(valid_dir, (("S", (i+1,j))))
                end
                if A[i-1, j] == 0
```

```
                    push!(valid_dir, (("N", (i-1,j))))
                end
                if A[i, j-1] == 0
                    push!(valid_dir, (("O", (i,j-1))))
                end
                if A[i, j+1] == 0
                    push!(valid_dir, (("L", (i,j+1))))
                end
                push!(values, valid_dir)
            end
        end
    end
    return Dict(zip(keys, values))
end
```

3.3 ALGORITMOS BFS E DFS

Para o caso do algoritmo BFS é feita uma busca em largura, isto é, quando há mais de um caminho a seguir, estes caminhos são explorados sequencialmente. Exploram-se os vizinhos do nó atual, em sequência os vizinhos dos vizinhos do nó atual e assim por diante até obter-se o caminho entre o começo e o fim. É possível comparar o funcionamento do mesmo a uma tubulação sendo enchida por água, já que a mesma escoa de maneira uniforme pelos caminhos da tubulação.

Quanto ao algoritmo DFS, é feita uma busca em profundidade, isto é, quando há a possibilidade de escolha de mais de um caminho dentro do labirinto, o algoritmo escolhe um dos caminhos possíveis e explora aquele caminho e suas ramificações até chegar em alguma parede ou no fim; caso não encontre a solução nesse caminho, retorna ao ponto com mais de um caminho possível e o processo se repete.

As funções dfs (Algoritmo 3.5) e bfs (Algoritmo 3.6), baseadas em Monteiro (2018), representam, respectivamente, os algoritmos de busca em largura e busca em profundidade. Essas funções possuem apenas uma linha de código diferente; portanto, a descrição será a mesma para ambas. Para o caso do método BFS tem-se uma fila e para o método DFS, uma pilha.

Ambos algoritmos têm como variáveis de entrada $graph$, simbolizando o dicionário construído pela função $matrix2dict$, assim como $start$ e $goal$, respectivamente os pontos de início e fim do labirinto.

Inicializa-se um vetor com a pilha ou fila contendo o ponto de início e um *String* vazio para armazenar o caminho percorrido, assim como um vetor vazio para

armazenar os nós que já foram visitados.

O *while* das funções bfs e dfs é repetido até que a fila/pilha esteja vazia. Nesse caso a solução não foi encontrada e retorna-se ∞ para simbolizar isso. O algoritmo retorna o caminho encontrado e os nós visitados caso o nó atual seja o nó final. O laço *for* adiciona os nós vizinhos do nó atual à fila/pilha.

Ao invés de usar *heaps* para decidir o caminho, o algoritmo de busca em largura explora os nós pela ordem em que foram adicionados à fila, enquanto que o algoritmo de busca em profundidade explora cada vez o último nó que foi adicionado à pilha.

Algoritmo 3.5 – Algoritmo de busca em profundidade.

```
function dfs(graph, start, goal)
    dfss = [(start, "")]
    visited = []
    while length(dfss) != 0
        curr, path = pop!(dfss)
        if curr == goal
            push!(visited, curr)
            return path, visited
        end
        if curr in visited
            continue
        end
        push!(visited, curr)
        for (dir, neigh) in graph[curr]
            push!(dfss, (neigh, path*dir))
        end
    end
    return Inf
end
```

Algoritmo 3.6 – Algoritmo de busca em largura.

```
function bfs(graph, start, goal)
    bfsq = [(start, "")]
    visited = []
    while length(bfsq) != 0
        curr, path = splice!(bfsq,1)
        if curr == goal
            push!(visited, curr)
```

```
            return path, visited
        end
        if curr in visited
            continue
        end
        push!(visited, curr)
        for (dir, neigh) in graph[curr]
            push!(bfsq, (neigh, path*dir))
        end
    end
    return Inf
end
```

4 RESULTADOS E DISCUSSÕES

Todos os resultados foram gerados por um computador com processador Intel® Core™ i7-4790 CPU @ 3.60GHz e 16 GB de memória RAM, utilizando um núcleo. A linguagem de programação utilizada foi *Julia*, versão 1.9.3, com auxílio dos pacotes:

- `Pluto`: visualização dos códigos por meio de *notebooks*;
- `Images`: geração das imagens dos labirintos;
- `DataStructures`: utilização de *heaps* para o algoritmo A*;
- `Random`: aleatoriedade usada pelo algoritmo de *Prim*;
- `StatsBase` e `StatsPlots`: obtenção de resultados estatísticos.

Os resultados do algoritmo de *Prim* se dão por imagens que representam os labirintos gerados.

Para os algoritmos de busca A*, busca em largura e busca em profundidade, são feitos dois tipos de testes: um específico e um geral. No teste específico, são fixados três labirintos de tamanhos 15, 40 e 85. O resultado é dado pelo número de nós visitados pelos algoritmos, assim como a marcação em cinza na imagem do labirinto original dos nós visitados pelos algoritmos.

Note que é possível gerar labirintos retangulares verticais ou horizontais, mas decidiu-se trabalhar somente com labirintos quadrados. Os testes gerais envolvem gerar mil labirintos aleatoriamente e apresentar qual dos três algoritmos de busca visitou menos nós para encontrar a solução. Há ainda a possibilidade de empate entre os algoritmos quanto a essa métrica; nesse caso, todos os algoritmos que empataram são considerados ganhadores.

Também é apresentada uma modificação na função heurística de *Manhattan*, adicionando um peso a ela para que o algoritmo A* tenha menos foco na variável de custo acumulado $cost$. O algoritmo resultante dessa modificação, conhecido como A* ponderado (*weighted* A*), costuma resultar em performance melhor que o algoritmo A*.

4.1 MUDANÇA NA FUNÇÃO HEURÍSTICA

Durante os estudos, resolveu-se fazer uma modificação no algoritmo A* proposto por Monteiro (2018). Considerou-se que somente o valor calculado pela função heurística seria utilizado para decidir o caminho, isto é, a variável $cost$ é igual a zero.

Com isso, considerando o número de nós visitados, foram obtidos melhores resultados que o algoritmo padrão, que soma a variável $cost$ com o valor obtido pela função heurística, como é possível ver na TABELA 1.

TABELA 1 – RESULTADOS DO ALGORITMO A* COM E SEM A VARIÁVEL $COST$.

Tamanho do labirinto / algoritmo	15	40	85
A*	160	158	132
A* sem a variável $cost$	852	842	868

FONTE: Autor.
LEGENDA: Quantidade de vezes que um algoritmo visita menos nós que o outro dentro de mil labirintos aleatórios, com a possibilidade de empate.

Devido a esses resultados, decidiu-se adicionar um peso à função heurística, para que ela tivesse mais influência que a variável $cost$ na hora de decidir um caminho a seguir, ou seja, a função heurística agora é dada por

$$h(x, y, k) = k\left(|x_1 - y_1| + |x_2 - y_2|\right), \tag{4.1}$$

em que x e y são dois nós nos quais se quer avaliar a distância de *Manhattan* e $k \in \mathbb{R}$.

A EQUAÇÃO 4.1 é a estratégia utilizada na versão do algoritmo A* conhecida como A* ponderado, ou *Weighted* A* (Pohl, 1969).

Também foram feitos testes para escolher o valor de k que é mais consistente quanto ao problema da resolução de labirintos, considerando o início do labirinto no canto noroeste e o fim no canto sudeste.

Primeiro, foram gerados cem labirintos de tamanhos fixados 15, 40 e 85, e então verificou-se para qual ou quais valores $k = 0{,}1, 0{,}2, \ldots, 1{,}9, 2, 3, \ldots, 10$ o algoritmo A* visitava menos nós. Está incluso também o caso do algoritmo A* com a variável $cost$ igual a zero. Os resultados desse teste estão na TABELA 2.

TABELA 2 – RESULTADOS DO PRIMEIRO TESTE PARA DEFINIR O VALOR IDEAL DE K.

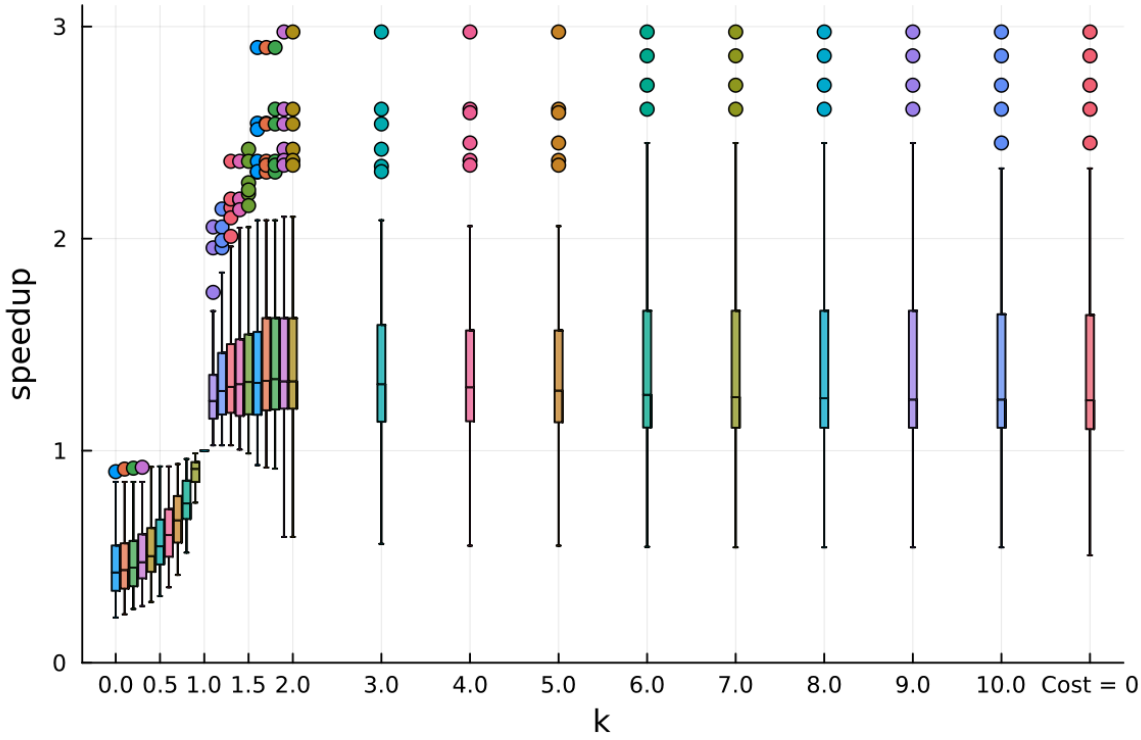
Valor de k	≤ 1	1,1	1,2	1,3	1,4	1,5	1,6
Vezes em que foi melhor ou empatou	0	64	51	53	41	51	36
Valor de k	1,7	1,8	1,9	2	3	4	5
Vezes em que foi melhor ou empatou	51	51	56	61	36	58	51
Valor de k	6	7	8	9	10	$cost = 0$	
Vezes em que foi melhor ou empatou	56	52	60	60	55	62	

FONTE: Autor.
LEGENDA: Número de vezes em que o algoritmo A* visita menos nós ou empata para valores de k fixados.

Foi armazenada a matriz com as informações utilizadas nos testes da TABELA 2, de quantos nós o algoritmo A* visitou para cada valor de k, dentro de cem labirintos gerados aleatoriamente.

Criou-se então uma matriz de índices dividindo todas as colunas pelos resultados obtidos para $k = 1$, obtendo assim um *speedup* de cada caso. Por fim, foram plotados *boxplots* para cada tamanho de labirinto fixado: 15 (FIGURA 8), 40 (FIGURA 9) e 85 (FIGURA 10).

FIGURA 8 – *SPEEDUP* PARA LABIRINTOS DE TAMANHO 15.



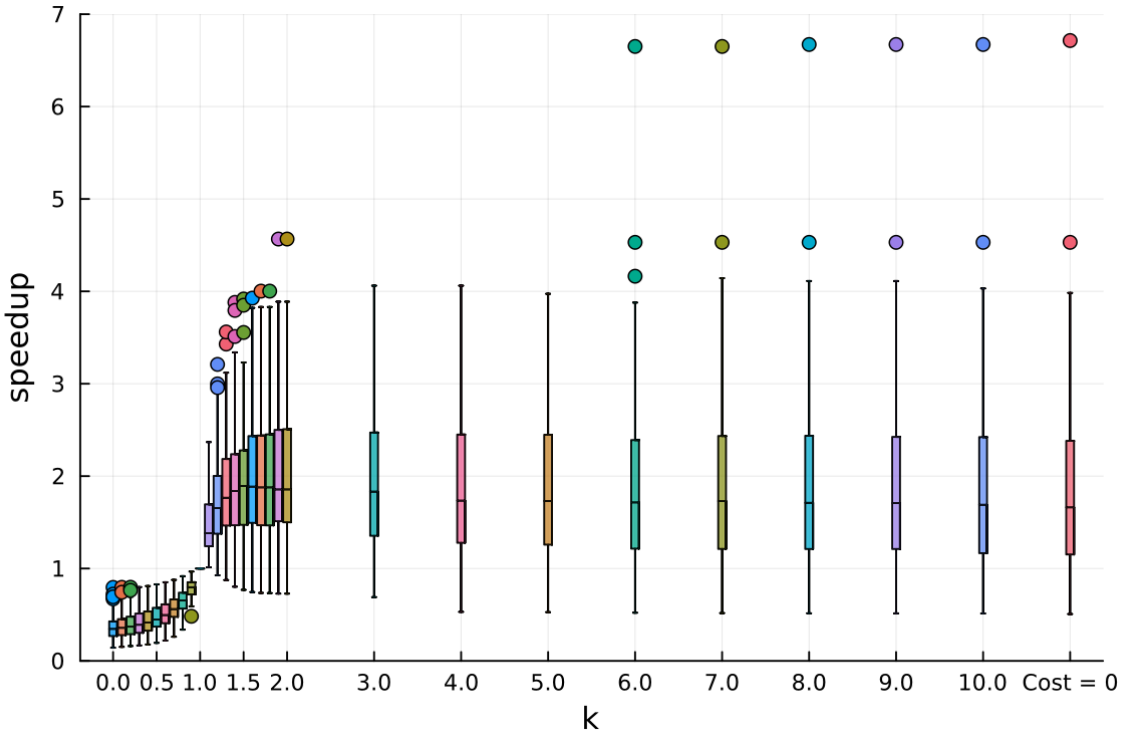
FONTE: Produzido pelo autor.

A escolha de fazer *boxplots* se deu à influência que os *outliers* estavam tendo na média e desvio padrão. Estes *outliers* são vistos nas FIGURAS 8, 9 e 10.

Com *speedup*, os valores são comparados diretamente aos de $k = 1$. É visível que valores de $k < 1$ são piores que o caso base. A partir de algum valor de k, os *boxplots* são visualmente semelhantes, o que impede de escolher peso da função heurística por meio deles.

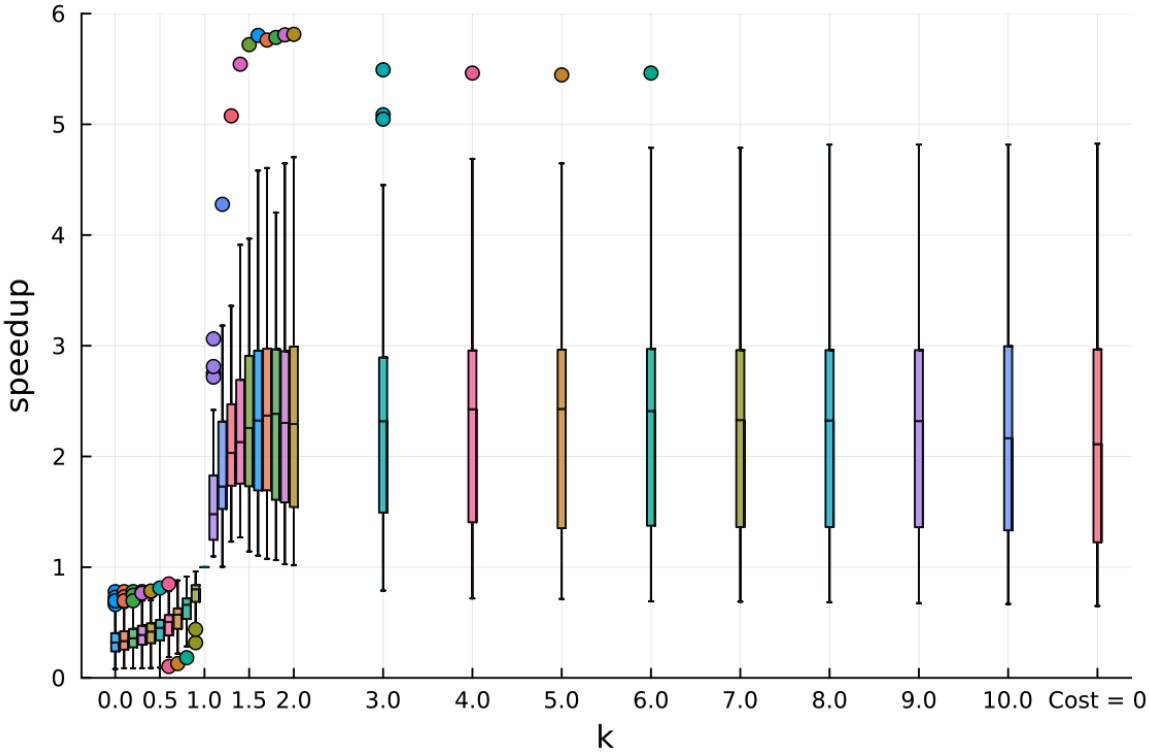
Para o segundo teste, foram escolhidos os oito valores de k da TABELA 2 com maior número de vezes em que foi melhor ou empatou (a saber, $1{,}1, 1{,}9, 2, 4, 6, 8, 9$),

FIGURA 9 – *SPEEDUP* PARA LABIRINTOS DE TAMANHO 40.



FONTE: Produzido pelo autor.

FIGURA 10 – *SPEEDUP* PARA LABIRINTOS DE TAMANHO 85.



FONTE: Produzido pelo autor.

assim como o caso do algoritmo A* em que $cost = 0$ e o caso em que $k = 1$. Esse segundo teste serve como uma filtragem para a escolha do valor de k.

O mesmo procedimento foi realizado: verificou-se qual ou quais valores de k levam ao menor número de nós visitados para encontrar a solução. Neste teste, porém, foram gerados mil labirintos de tamanhos 15, 40 e 85, totalizando três mil labirintos analisados para os valores de k observados. Os resultados obtidos, mostrados na TABELA 3, são a soma dos resultados obtidos para cada tamanho fixado.

TABELA 3 – RESULTADOS DO SEGUNDO TESTE PARA DEFINIR O VALOR IDEAL DE K.

Valor de k	1	1,1	1,9	2	4
Vezes em que foi melhor ou empatou	2	744	1125	980	732
Valor de k	6	8	9	$cost = 0$	
Vezes em que foi melhor ou empatou	516	569	568	723	

FONTE: Autor.
LEGENDA: Número de vezes em que o algoritmo A* visita menos nós ou empata para valores de k fixados.

Finalmente, com os resultados da TABELA 3, o peso escolhido para se usar na função heurística da distância de *Manhattan* foi $k = 1{,}9$.

4.2 RESULTADOS DO ALGORITMO DE *PRIM* SIMPLIFICADO

Para a mostra visual, são gerados três labirintos pelo algoritmo de *Prim* simplificado, que serão utilizados posteriormente para testes com os algoritmos de busca A*, busca em largura e busca em profundidade. Esses labirintos são mostrados nas FIGURAS 11, 12 e 13.

4.3 RESULTADOS DOS ALGORITMOS DE BUSCA

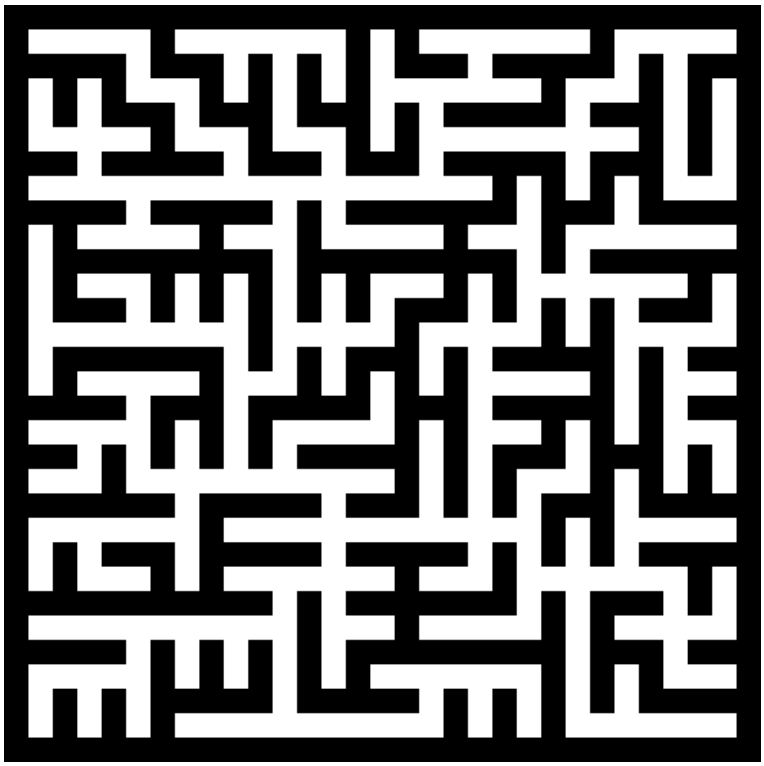
Para mostrar os resultados dos algoritmos de busca, são feitos testes específicos e testes gerais.

4.3.1 Testes específicos

Os testes específicos para os algoritmos de busca A*, busca em largura e busca em profundidade utilizam os labirintos mostrados nas FIGURAS 11, 12 e 13.

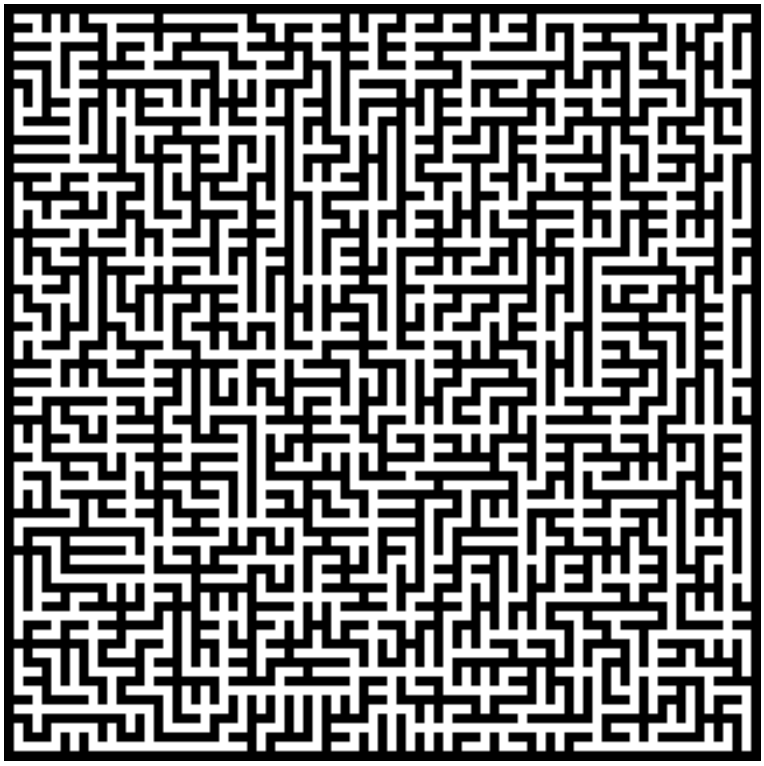
Apesar da escolha do peso da função heurística $k = 1{,}9$, foi considerada também para os testes a função heurística original, isto é, com peso $k = 1$, a fim de evidenciar a influência que o peso tem na função heurística.

FIGURA 11 – EXEMPLO DE LABIRINTO DE TAMANHO 15 GERADO PELO ALGORITMO DE *PRIM* SIMPLIFICADO.



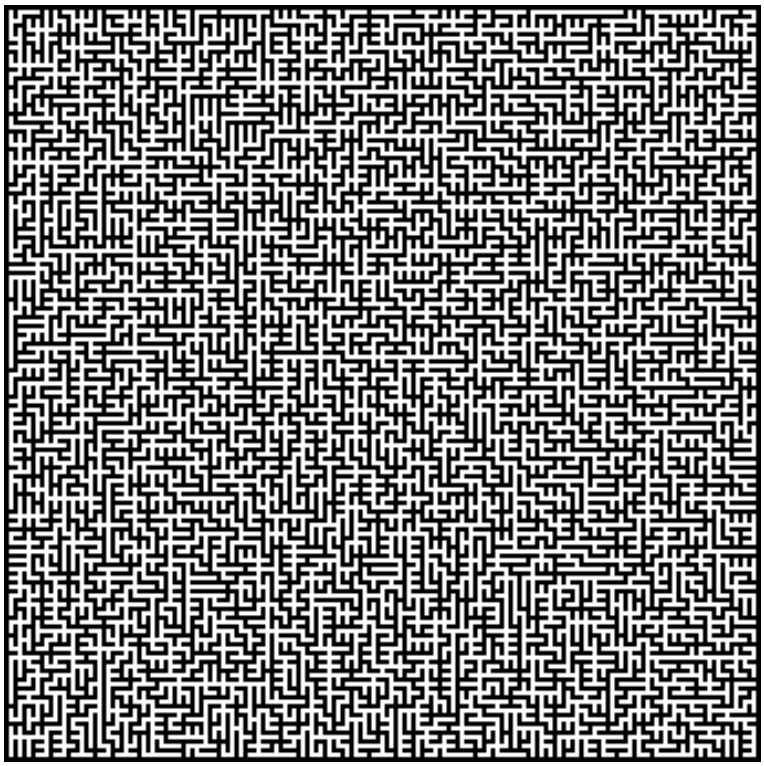
FONTE: Produzido pelo autor.

FIGURA 12 – EXEMPLO DE LABIRINTO DE TAMANHO 40 GERADO PELO ALGORITMO DE *PRIM* SIMPLIFICADO.



FONTE: Produzido pelo autor.

FIGURA 13 – EXEMPLO DE LABIRINTO DE TAMANHO 85 GERADO PELO ALGORITMO DE *PRIM* SIMPLIFICADO.



FONTE: Produzido pelo autor.

Na TABELA 4 é apresentado o número de nós visitados por cada um dos algoritmos. A representação visual dos nós visitados é feita por espaços de cor cinza nas demais figuras desta Seção.

TABELA 4 – RESULTADOS DOS TESTES ESPECÍFICOS.

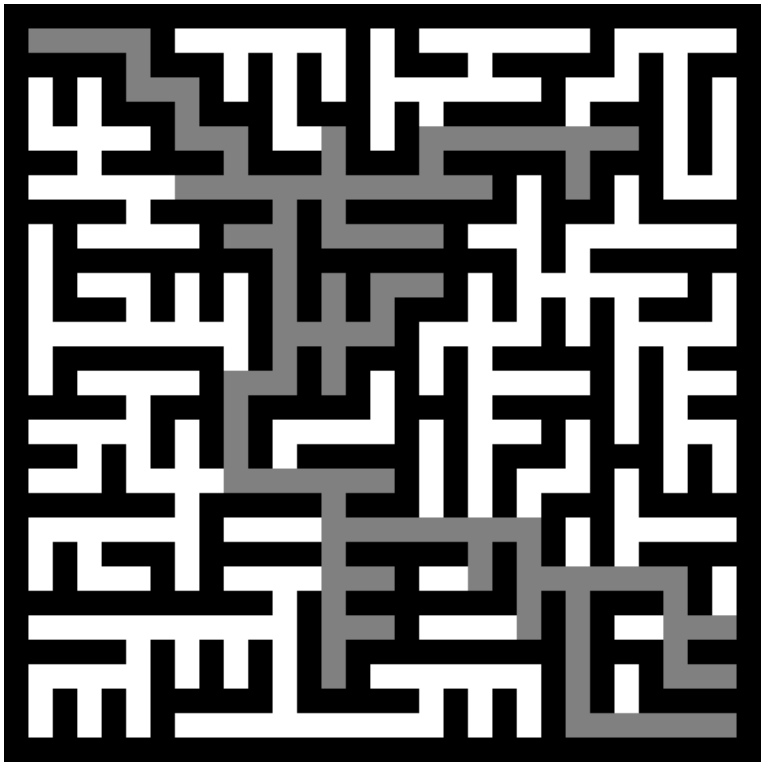
Algoritmo / Tamanho	A*	A* com $k = 1{,}9$	BFS	DFS
15	135	105	421	219
40	781	362	2520	1783
85	4571	4153	14442	7335

FONTE: Autor.
LEGENDA: Quantidade de nós visitados para os testes específicos.

4.3.1.1 Tamanho 15

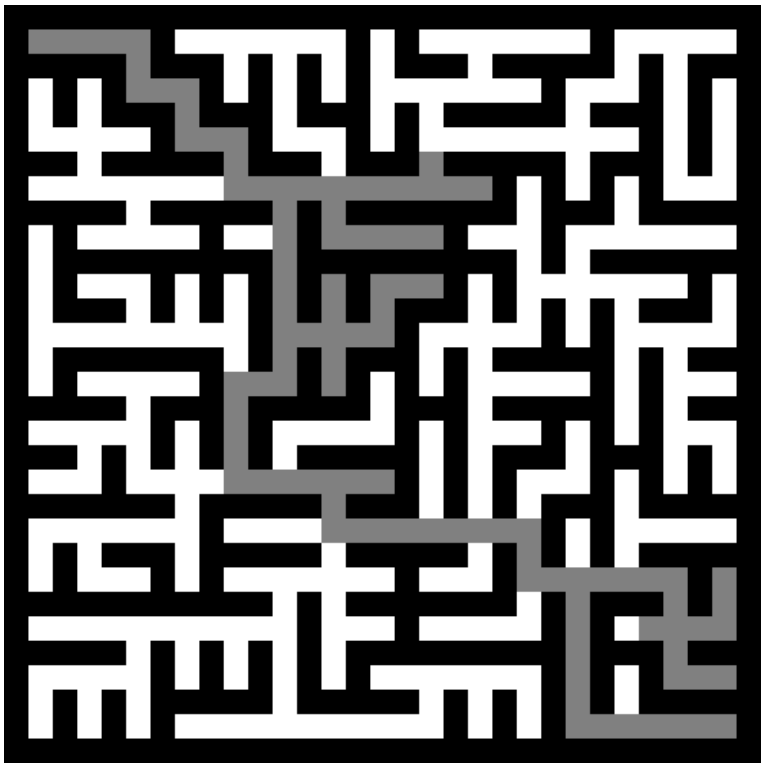
Os nós visitados pelos algoritmos de busca no labirinto de tamanho 15 mostrado na FIGURA 11 podem ser vistos nas FIGURAS 14, 15, 16 e 17.

FIGURA 14 – NÓS VISITADOS PELO ALGORITMO A* COM $K = 1$.



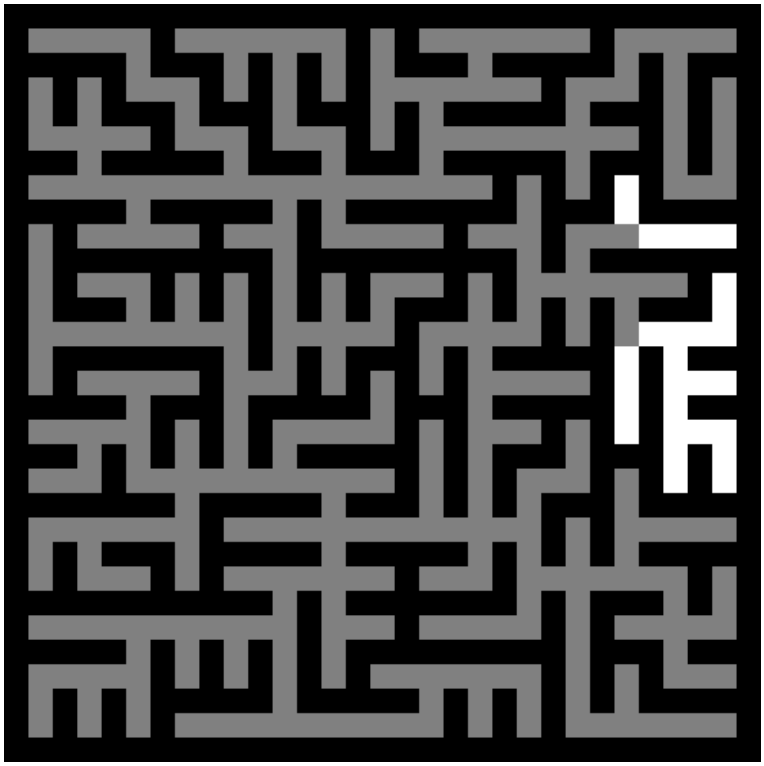
FONTE: Produzido pelo autor.

FIGURA 15 – NÓS VISITADOS PELO ALGORITMO A* COM $K = 1{,}9$.



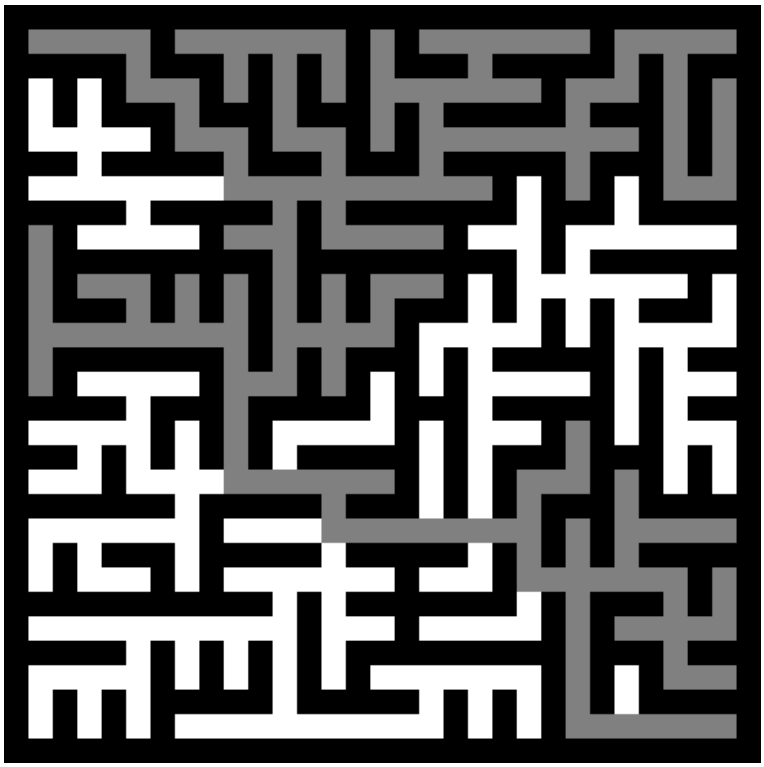
FONTE: Produzido pelo autor.

FIGURA 16 – NÓS VISITADOS PELO ALGORITMO DE BUSCA EM LARGURA.



FONTE: Produzido pelo autor.

FIGURA 17 – NÓS VISITADOS PELO ALGORITMO DE BUSCA EM PROFUNDIDADE.

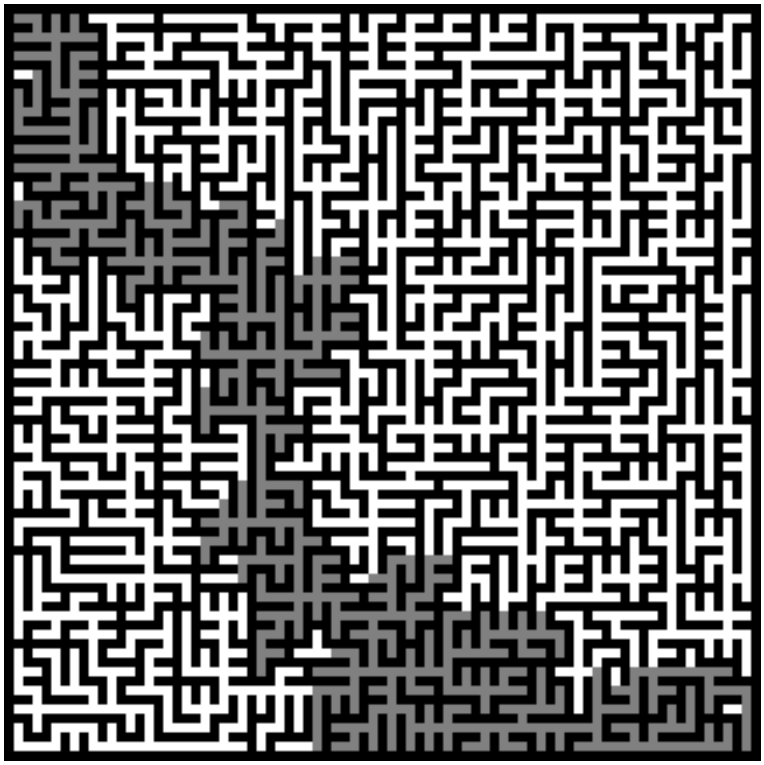


FONTE: Produzido pelo autor.

4.3.1.2 Tamanho 40

Os nós visitados pelos algoritmos de busca no labirinto de tamanho 40 mostrado na FIGURA 12 podem ser vistos nas FIGURAS 18, 19, 20 e 21.

FIGURA 18 – NÓS VISITADOS PELO ALGORITMO A* COM $K = 1$.

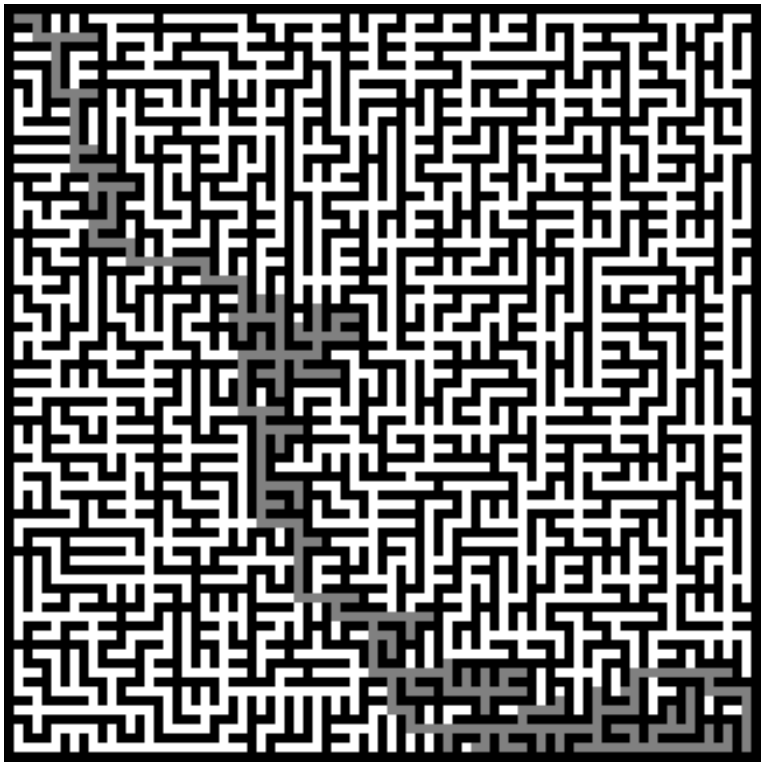


FONTE: Produzido pelo autor.

4.3.1.3 Tamanho 85

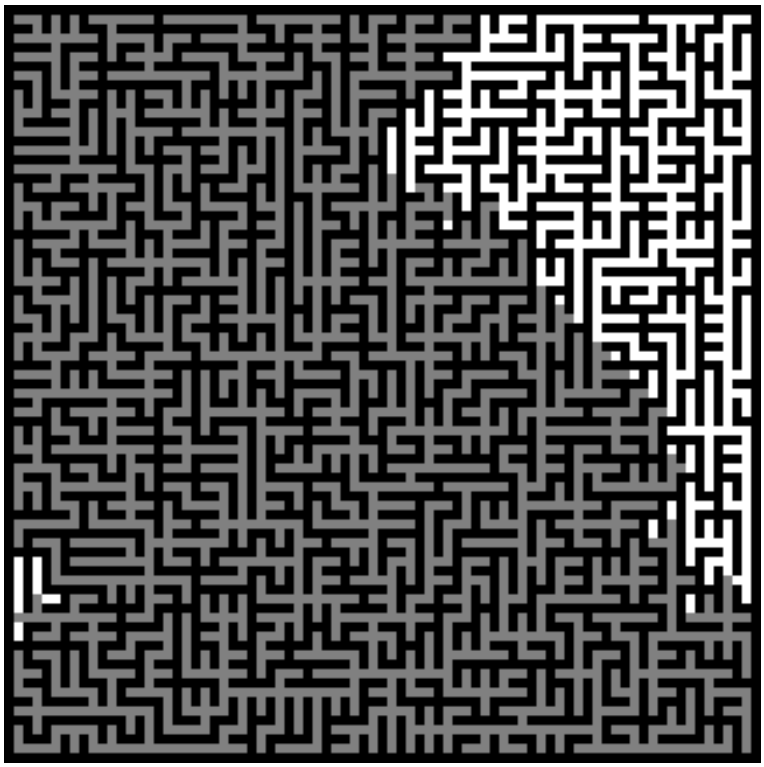
Os nós visitados pelos algoritmos de busca no labirinto de tamanho 85 mostrado na FIGURA 13 podem ser vistos nas FIGURAS 22, 23, 24 e 25.

FIGURA 19 – NÓS VISITADOS PELO ALGORITMO A* COM $K = 1{,}9$.



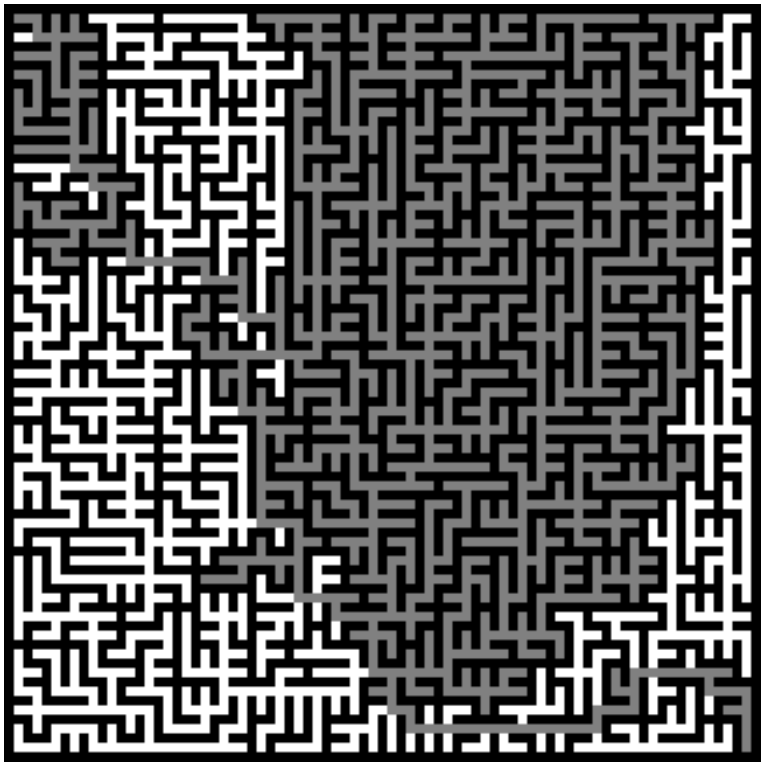
FONTE: Produzido pelo autor.

FIGURA 20 – NÓS VISITADOS PELO ALGORITMO DE BUSCA EM LARGURA.



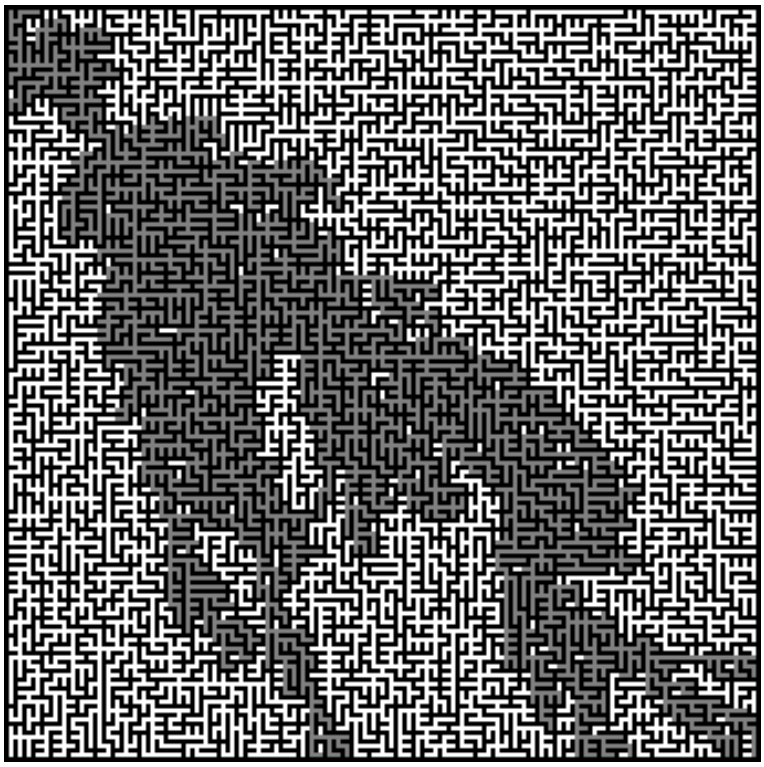
FONTE: Produzido pelo autor.

FIGURA 21 – NÓS VISITADOS PELO ALGORITMO DE BUSCA EM PROFUNDIDADE.



FONTE: Produzido pelo autor.

FIGURA 22 – NÓS VISITADOS PELO ALGORITMO A* COM $K = 1$.

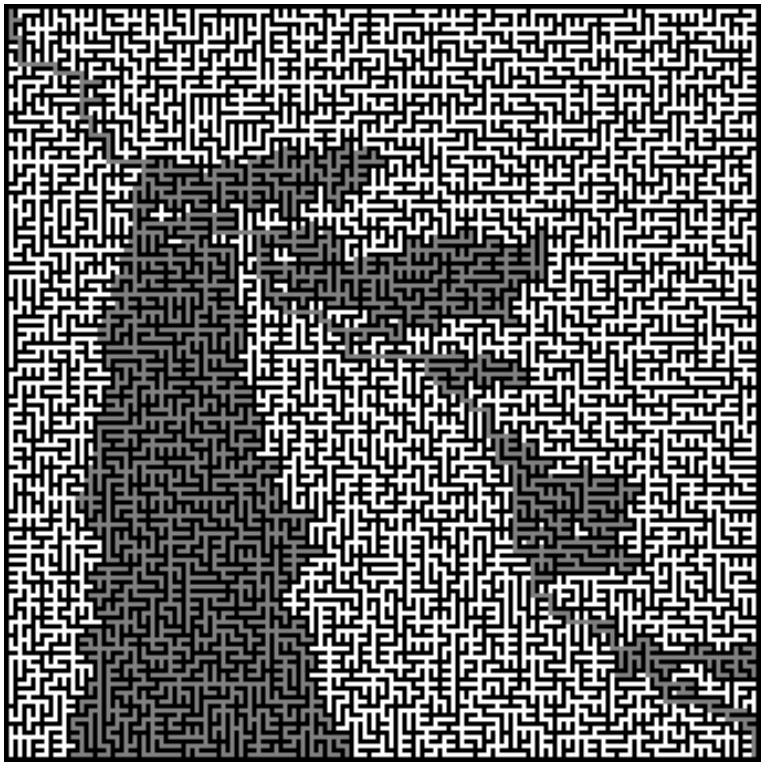


FONTE: Produzido pelo autor.

4.4 TESTES GERAIS

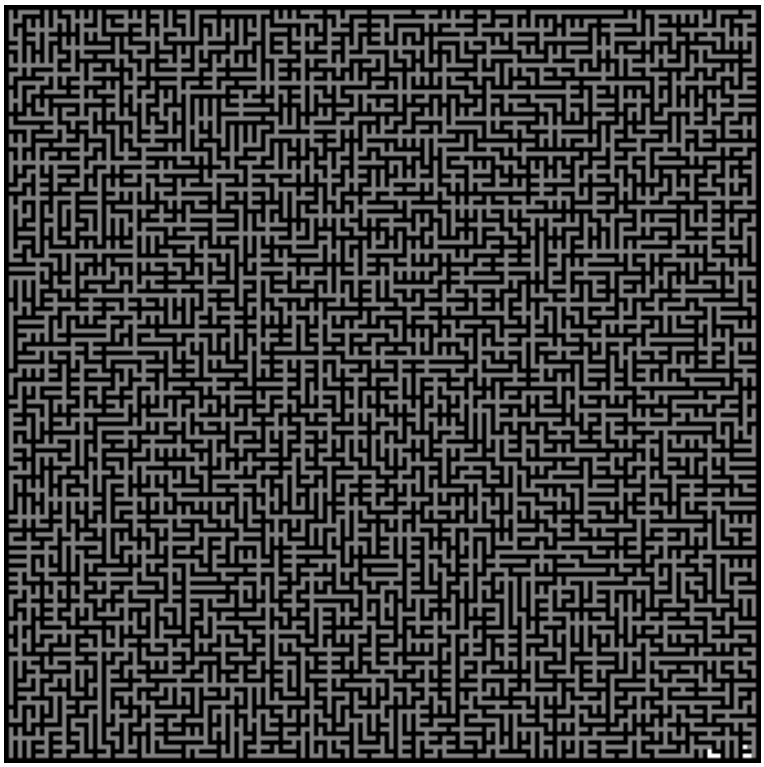
Para os testes gerais, geraram-se mil labirintos para cada tamanho fixado (15, 40 e 85) e verificou-se qual dos quatro algoritmos de busca, A*, A* com $k = 1{,}9$, BFS e

FIGURA 23 – NÓS VISITADOS PELO ALGORITMO A* COM $K = 1{,}9$.



FONTE: Produzido pelo autor.

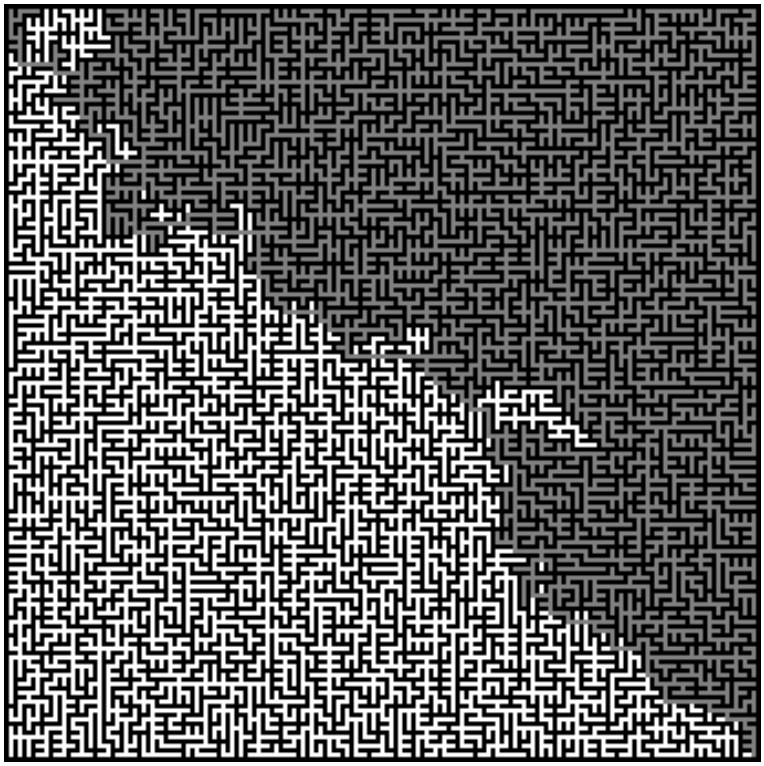
FIGURA 24 – NÓS VISITADOS PELO ALGORITMO DE BUSCA EM LARGURA.



FONTE: Produzido pelo autor.

DFS visitou menos nós, considerando também possíveis empates entre os algoritmos. Os resultados desses testes estão na TABELA 5.

FIGURA 25 – NÓS VISITADOS PELO ALGORITMO DE BUSCA EM PROFUNDIDADE.



FONTE: Produzido pelo autor.

TABELA 5 – RESULTADOS DOS TESTES GERAIS.

Algoritmo / Tamanho	A*	A* com $k = 1{,}9$	BFS	DFS
15	45	904	0	56
40	39	904	0	57
85	37	919	0	44

FONTE: Autor.
LEGENDA: Quantidade de vezes que um algoritmo visitou menos nós ou empatou durante os testes gerais.

4.5 DISCUSSÕES

Pelas FIGURAS 11, 12 e 13, é possível perceber que uma das características de labirintos gerados pelo algoritmo de *Prim* simplificado é a existência de muitos impasses.

Para os labirintos em estudo dos testes específicos, é notável visualmente e pela quantidade de nós visitados que o algoritmo A* com $k = 1{,}9$ tem um desempenho melhor em relação aos outros.

Ainda assim, para o labirinto de tamanho 85 da FIGURA 23 é possível ver que o algoritmo A* com peso $k = 1{,}9$ decidiu explorar boa parte do lado esquerdo ao invés

de ir na direção da solução. Para este caso, utilizar algum dos outros valores de k considerados pode evitar esse problema.

Quanto ao comportamento do algoritmo A* com $k = 1$, na FIGURA 14 e na FIGURA 22, é perceptível que o algoritmo decide explorar algumas regiões mais a fundo do que o algoritmo A* com $k = 1{,}9$.

O algoritmo de busca em largura foi o que mais visitou nós em todos os casos. Nos experimentos mostrados nas FIGURAS 16 e 24, quase todos os nós do labirinto são explorados.

Uma explicação do desempenho do algoritmo de busca em largura é o fato de que os labirintos gerados pelo algoritmo de *Prim* possuem muitos impasses. Mesmo encontrando a solução, ao considerar o número de nós visitados, o algoritmo BFS não é uma boa escolha para labirintos com essa característica.

A quantidade de impasses também é influente para o algoritmo de busca em profundidade. A quantidade de nós visitados depende da ordem em que os nós são explorados.

Pela maneira que o algoritmo de busca em profundidade decide explorar os nós, é preciso ter algum tipo de sorte na ordem em que eles são adicionados à fila, para evitar explorar um caminho longo, com muitas ramificações e que não chegam ao ponto final. Isso é visível em todos os exemplos: o algoritmo DFS explora, como seu nome diz, profundamente uma região do labirinto.

Em relação aos testes gerais e considerando a quantidade de nós visitados, foi possível evidenciar que o algoritmo BFS não é adequado para resolver o problema de encontrar o caminho de labirintos gerados pelo algoritmo de *Prim* simplificado, sendo o início no canto noroeste do labirinto e o final no canto sudeste.

Para os três tamanhos observados, o algoritmo A* com $k = 1{,}9$ manteve-se consistentemente melhor que os outros algoritmos de busca quanto à quantidade de nós visitados. Os algoritmos A* e DFS necessitaram visitar menos nós em alguns casos somente.

5 CONCLUSÃO

O algoritmo de *Prim* simplificado gerou os labirintos que foram usados para comparar o desempenho dos algoritmos de busca.

Considerando o problema de encontrar o caminho entre um ponto inicial e um ponto final de um labirinto em duas dimensões, os algoritmos de busca estudados mostraram-se capazes de encontrar a solução para os labirintos gerados pelo algoritmo de *Prim* simplificado.

A efetividade dos algoritmos A*, BFS e DFS se deu pela observação da quantidade de nós que cada algoritmo explorou até encontrar a solução.

A estratégia de adicionar um fator peso k à função heurística da distância de *Manhattan* acelerou a convergência do algoritmo A*. Para os testes realizados, o melhor resultado foi observado com o peso $k = 1{,}9$.

O algoritmo A* com peso $k = 1{,}9$ mostrou-se mais consistente dentre os avaliados e para os testes realizados; porém, há casos observados na TABELA 2 em que tanto o algoritmo A* com $k = 1$ como o algoritmo de busca em profundidade possuem um desempenho igual ou melhor. Isto é, o algoritmo A* com peso $k = 1{,}9$ visitou menos nós ou empatou com os algoritmos A* com peso $k = 1$, BFS e DFS em aproximadamente $90{,}74\%$ dos testes.

Os resultados dos algoritmos de busca devem levar em consideração as características dos labirintos, isto é, labirintos perfeitos mas com muitos impasses.

O desempenho dos algoritmos A*, BFS e DFS pode ser diferente para labirintos com outras características. Para o caso do algoritmo A*, outros valores de k (ou outra função heurística) podem ser melhores para labirintos com características distintas dos estudados neste trabalho.

REFERÊNCIAS

BELLOT, V.; CAUTRÈS, M.; FAVREAU, J.-M.; GONZALEZ-THAUVIN, M.; LAFOURCADE, P.; LE CORNEC, K.; MOSNIER, B.; RIVIÈRE-WEKSTEIN, S. How to generate perfect mazes? **Information Sciences**, v. 572, p. 444–459, 2021. DOI: 10.1016/j.ins.2021.03.022. Citado 1 vez na página 6.

BUCK, J. **Mazes for Programmers**. Texas, USA: The Pragmatic Programmers, LLC, 2015. Citado 4 vezes nas páginas 8, 9, 12.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numerische Mathematik**, v. 1, p. 269–271, 1959. DOI: 10.1007/BF01386390. Citado 1 vez na página 7.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. **IEEE Transaction on Systems Science and Cybernetics**, v. 4, p. 100–107, 2 1968. DOI: 10.1109/TSSC.1968.300136. Citado 1 vez na página 7.

JULIACOLLECTIONS. **Heaps · DataStructures.jl**. [S.l.: s.n.], 2023. Acesso em 08 nov 2023. Disponível em: https://juliacollections.github.io/DataStructures.jl/stable/heaps/. Citado 1 vez na página 21.

MONTEIRO, A. d. F. **Performance of the Julia Programming Language in Different Search Methods**. 2018. Trabalho de Conclusão de Curso – Universidade Federal do Estado do Rio de Janeiro. Acesso em 23 ago 2023. Disponível em: https://bsi.uniriotec.br/wp-content/uploads/sites/31/2020/05/201807AliceMonteiro.pdf. Citado 5 vezes nas páginas 7, 17, 21, 23, 26.

POHL, I. First Results on the Effect of Error in Heuristic Search. In: MELTZER, B.; MICHIE, D. (Ed.). **Machine Intelligence**. [S.l.]: Edinburgh University, Department of Machine Intelligence and Perception, 1969. P. 219–236. Citado 1 vez na página 27.

PRIM, R. C. Shortest connection networks and some generalizations. **The Bell System Technical Journal**, v. 36, n. 6, p. 1389–1401, 1957. DOI: 10.1002/j.1538-7305.1957.tb01515.x. Citado 1 vez na página 8.

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. 012–061, 2020. DOI: 10.1088/1742-6596/1566/1/012061. Citado 4 vezes nas páginas 7, 9, 10, 14.

SKIENA, S. S. **The Algorithm Design Manual**. New York, USA: Springer Cham, 2020. Citado 3 vezes nas páginas 7, 13, 14.

STENKRONA, A. **MazeFun**. [S.l.: s.n.], 2019. Acesso em 10 nov 2023. Disponível em: https://github.com/ArneStenkrona/MazeFun. Citado 1 vezes nas páginas 17, 18.

STILLEY, J. **mazelib**. [S.l.: s.n.], 2023. Acesso em 14 set 2023. Disponível em: https://github.com/john-science/mazelib. Citado 3 vezes nas páginas 17, 19.

WAYAHDI, M. R.; GINTING, S. H. N.; SYAHPUTRA, D. Greedy, A-Star, and Dijkstra's Algorithms in Finding Shortest Path. **International Journal of Advances in Data and Information Systems**, v. 2, n. 1, p. 45–52, 2021. DOI: 10.25008/ijadis.v2i1.1206. Citado 1 vez na página 6.

WEISSTEIN, E. W. **"Grid Graph." From MathWorld–A Wolfram Web Resource**. [S.l.: s.n.], 2023. Acesso em 09 nov 2023. Disponível em: https://mathworld.wolfram.com/GridGraph.html. Citado 0 vezes nas páginas 11, 12.

YAN, Y. Research on the A Star Algorithm for Finding Shortest Path. **Highlights in Science, Engineering and Technology**, v. 46, p. 154–161, 2023. DOI: 10.54097/hset.v46i.7697. Citado 1 vez na página 6.