Relatório
Algoritmos de Busca

André Luís Mendes Fakhoury (4482145)
David Cairuz da Silva (10830061)
Gustavo Vinicius Vieira Silva Soares (10734428)
Thiago Preischadt Pinheiro (10723801)

SCC0230 - Inteligência Artificial
Prof. Alneu de Andrade Lopes

Introdução

A Inteligência Artificial é a área de estudo que tenta, até certo ponto, replicar a inteligência humana por meio de computadores, criando agentes que agem "racionalmente". Para atingir esse objetivo, os sistemas recorrem à utilização de algoritmos de busca, estratégias que permitem que o computador encontre um valor específico dentro de uma estrutura de dados, seguindo uma sequência de ações.

Algoritmos de busca, em Inteligência Artificial, são definidos como uma forma de identificar e encontrar um determinado estado em uma estrutura de dados, partindo de um estado inicial e gerando como solução uma sequência de passos que seja capaz de atingir o objetivo desejado.

Um problema de busca pode ser definido com base em quatro fatores principais:

- **Espaço de busca:** conjunto de todas as possíveis soluções que o sistema possui.
- **Estado inicial:** o estado do qual o agente inicia a busca.
- **Ações:** conjunto de ações capazes de levar o agente de um estado a outro com um custo bem definido e ≥ 0, por exemplo: distância de um ponto à outro.
- **Teste de objetivo:** uma função que observa o estado atual do agente e verifica se o objetivo foi ou não atingido.

Uma solução válida para um problema de busca é uma sequência de ações que levam o agente do estado inicial até o objetivo. Se a sequência de ações retornada possui custo mínimo - sendo o custo de cada ação definido no problema -, dizemos que essa é uma **solução ótima**.

Os algoritmos de busca possuem quatro propriedades principais, que possibilitam sua análise e comparação. São elas:

- **Completeza:** um algoritmo de busca é dito completo se ele retorna uma solução se ao menos uma solução existe para qualquer entrada aleatória.
- **Complexidade de tempo:** calculada a partir da quantidade de nós que serão gerados para encontrar uma solução.
- **Complexidade de espaço:** calculada a partir da quantidade de nós que vão ocupar a memória ao mesmo tempo.
- **Otimalidade:** um algoritmo de busca é dito ótimo se a solução encontrada por ele é ótima.

Neste documento, serão estudados os dois tipos de algoritmos de busca - informados e não informados -, e serão estudados, analisados e comparados cinco algorítmos específicos, em relação à todas as propriedades mencionadas acima.

Estudo dos algoritmos

Os Algoritmos de Busca são divididos em dois grandes grupos:

- **Algoritmos não informados:** não possuem conhecimento do domínio do problema, buscam de forma cega.
- **Algorítmos informados:** possuem algum conhecimento do domínio do problema, o que pode auxiliar a busca.

Dentro de cada grupo, serão estudados e analisados algoritmos específicos, com exemplos de soluções encontradas, análise de completeza, complexidade de tempo e espaço, e otimalidade.

O problema que os algoritmos de busca deverão resolver nesse estudo pode ser definido da seguinte forma:

- É dado um grid de dimensões MxN, com células livres e células bloqueadas. Para o agente, é um labirinto.
- O estado inicial do agente é pré-definido em uma das células livres.
- O objetivo do agente é chegar à uma célula determinada.
- O agente pode se movimentar para qualquer outra célula livre que esteja acima, abaixo ou aos lados da célula atual, mas não pode entrar em células bloqueadas.
- O custo total da solução é a soma da quantidade de movimentos.

Busca não informada (cega)

Algoritmos de busca cega examinam cada nó sem nenhum conhecimento de domínio - como proximidade do objetivo -, e sem nenhuma heurística, ou seja, buscam cegamente. Esse tipo de algoritmo possui apenas uma forma bem definida de percorrer a estrutura de dados e sabe quando chegou ao nó de destino.

Nesse documento serão estudados dois algoritmos de busca cega: a busca em profundidade e a busca em largura.

Busca em largura

A busca em largura é um algoritmo de busca cega que percorre estruturas de dados, como grafos e árvores, explorando um nível da estrutura de cada vez - ou, "indo para o lado antes de ir para baixo". A implementação desse algorítmo se dá pelo uso de uma *queue*, que irá guardar os próximos nós a serem visitados na ordem correta. O pseudocódigo do algoritmo pode ser observado a seguir.

Algorithm 1 BFS(G, root)

```
1: let Q be a queue
2: label root as discovered
3: Q.enqueue(root)
4: while Q is not empty do
5:     v := Q.dequeue()
6:     if v is the goal then
7:         return v
8:     end if
9:     for each w ∈ G.adjacentNodes(v) do
10:        if node w is not labeled as discovered then
11:            label w as discovered
12:            Q.enqueue(w)
13:        end if
14:    end for
15: end while
```

Considere a Figura 1: que solução permite que se vá do nó S ao nó G? Na Figura 2 será possível ver como o algoritmo de busca em largura resolve o problema.

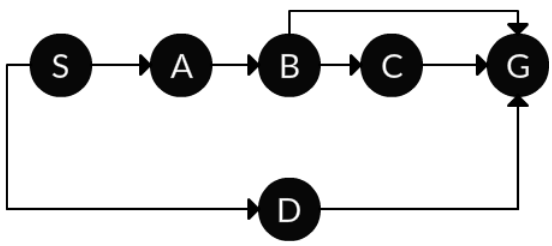

Figure 1: Grafo de um problema de busca

A busca em largura vai ser iniciada no nó de início - o nó S, nesse caso -, e explorar um nível da Figura 2 de cada vez até atingir pela primeira vez o nó objetivo.

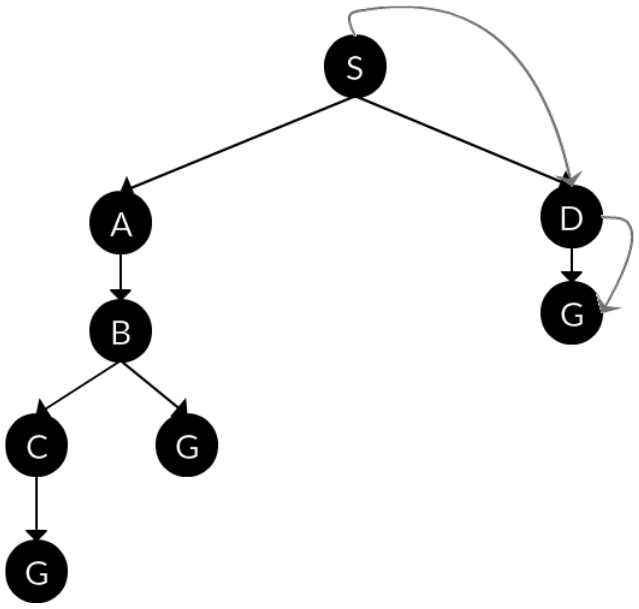

Figure 2: Solução do problema da Figura 1 com busca em largura

Em seguida, o algoritmo será avaliado em relação às quatro propriedades mencionadas na Introdução:

- **Completeza:** a busca em largura é um algoritmo completo e sempre encontrará solução, se ela existir.
- **Complexidade de tempo:** $O(b^d)$, onde b é o fator de ramificação (*branching factor*) - número médio de filhos que um dado nó possui -, e d é a profundidade da árvore.
- **Complexidade de espaço:** $O(b^d)$, no pior caso, pois é necessário guardar todos os nós do próximo nível ao mesmo tempo.
- **Otimalidade:** sempre encontra a solução ótima caso o custo de todas as ações possíveis seja o mesmo. Não garante solução ótima caso contrário.

Na Figura 3, é possível ver o caminho encontrado pela busca em largura para um labirinto gerado aleatoriamente.

Busca em profundidade

A busca em profundidade é um algoritmo de busca cega que percorre estruturas de dados, como grafos e árvores, explorando um ramo inteiro de cada vez antes de retroceder - ou, "indo para baixo até o fim antes de ir para o lado". A implementação desse algorítmo se dá pelo uso de uma *stack*, que irá guardar os próximos nós a serem visitados na ordem correta, isto

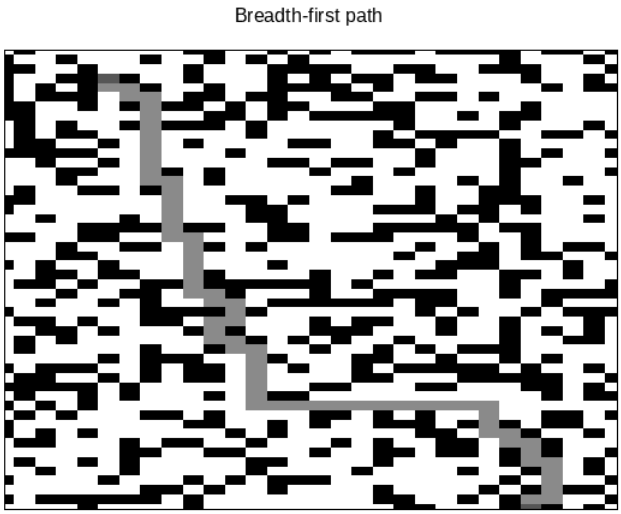

Figure 3: Solução de um labirinto utilizando busca em largura

é, os filhos do nó atual sempre terão prioridade. O pseudocódigo do algorítmo pode ser observado na página seguinte.

Considere novamente o problema apresentado na Figura 1. Na Figura 4 será possível ver como o algoritmo de busca em profundidade o resolve.

A busca em profundidade vai ser iniciada no nó de início - o nó S, nesse caso -, e vai explorar um ramo da Figura 4 de cada vez até atingir pela primeira vez o nó objetivo.

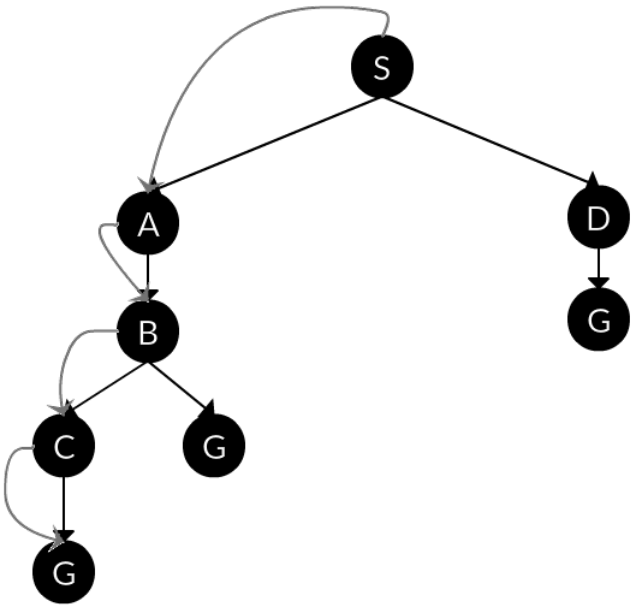

Figure 4: Solução do problema da Figura 1 com busca em profundidade

Após o pseudocódigo, o algoritmo será avaliado em relação às quatro propriedades mencionadas na Introdução.

Algorithm 2 DFS(G, root)

```
1: let S be a stack
2: label root as discovered
3: S.push(root)
4: while S is not empty do
5:     v := S.pop()
6:     if v is the goal then
7:         return v
8:     end if
9:     for each w ∈ G.adjacentNodes(v) do
10:        if node w is not labeled as discovered then
11:            label w as discovered
12:            S.push(w)
13:        end if
14:    end for
15: end while
```

- **Completeza:** a busca em profundidade é um algoritmo completo e sempre encontrará solução, se ela existir.
- **Complexidade de tempo:** $O(b^d)$, onde b é o fator de ramificação (*branching factor*) - número médio de filhos que um dado nó possui -, e d é a profundidade da árvore.
- **Complexidade de espaço:** $O(b \cdot d)$.
- **Otimalidade:** Não garante solução ótima.

Na Figura 5, é possível ver o caminho encontrado pela busca em profundidade para um labirinto gerado aleatoriamente.

Busca informada

Algoritmos de busca informada examinam cada nó contando com algum conhecimento de domínio - como a proximidade do nó de objetivo -, e utilizando heurísticas para auxiliar na busca. É importante notar que isso não os torna necessariamente melhores que os algoritmos de busca cega.

Nesse documento serão estudados três algoritmos de busca informada: a busca Best-First, a busca A* e a busca Hill Climbing.

Busca Best-First

A busca Best-First é uma busca informada que se move sempre em direção ao nó mais próximo do nó de objetivo, com base em uma heurística. O valor de cada nó nesse método é calculado utilizando a heuristica $h(x)$, definida da seguinte forma:

- $h(x)$: distância estimada do nó atual x até o nó de objetivo.

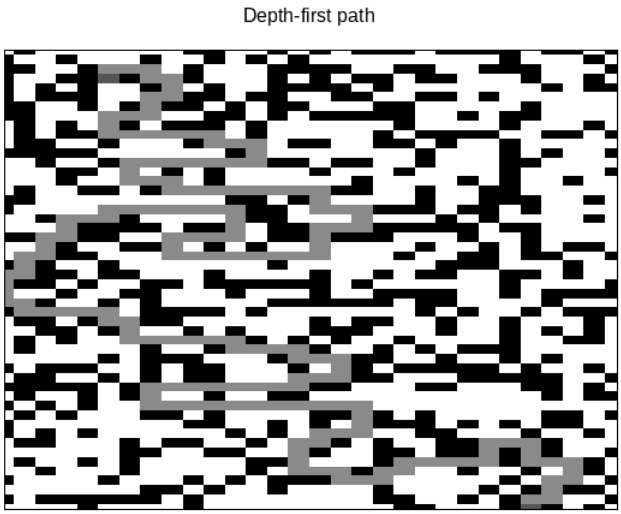

Figure 5: Solução de um labirinto utilizando busca em profundidade

O algoritmo é muito similar à uma busca em largura, no entanto, faz uso de uma *priority queue* para olhar antes para os nós mais próximos do objetivo. É conhecido por unir as vantagens da busca em profundidade e da busca em largura em um único algoritmo.

Considere a Figura 6: que solução permite que se vá do nó S ao nó G? Na Figura 7 será possível ver como o algoritmo de busca Best-First resolve o problema.

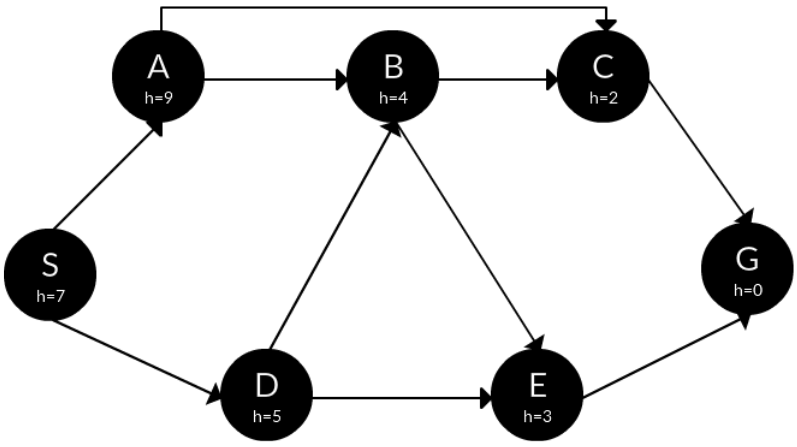

Figure 6: Grafo de um problema de busca com heurística

A Busca Best-First vai ser iniciada no nó de início - o nó S, nesse caso -, e vai se mover sempre em direção ao nó com menor h(x) na Figura 7 até atingir pela primeira vez o nó objetivo.

Em seguida, o algoritmo será avaliado em relação às quatro propriedades mencionadas na Introdução:

- **Completeza:** a busca Best-First é um algoritmo completo e sempre encontrará solução,

Algorithm 3 Best-First(G, root)

```
1: let PQ be a priority queue
2: label root as discovered
3: PQ.enqueue({0, root})
4: while Q is not empty do
5:     v := Q.dequeue()
6:     if v is the goal then
7:         return v
8:     end if
9:     for each w ∈ G.adjacentNodes(v) do
10:        if node w is not labeled as discovered then
11:            label w as discovered
12:            cost := h(w)
13:            Q.enqueue({cost, w})
14:        end if
15:    end for
16: end while
```

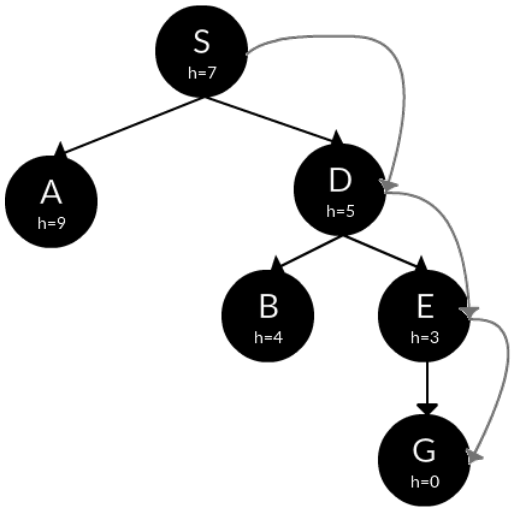

Figure 7: Solução do problema da Figura 6 com busca Best-First

mesmo que precise explorar todos os nós para isso.

- **Complexidade de tempo:** $O(b^d \cdot log(b^d))$, onde b é o fator de ramificação (*branching factor*) - número médio de filhos que um dado nó possui -, e d é a profundidade da árvore. O *log* se dá por conta da complexidade da inserção em uma *priority queue*.
- **Complexidade de espaço:** $O(b^d)$.
- **Otimalidade:** Não garante solução ótima.

Na Figura 8, é possível ver o caminho encontrado pela busca Best-First para um labirinto gerado aleatoriamente.

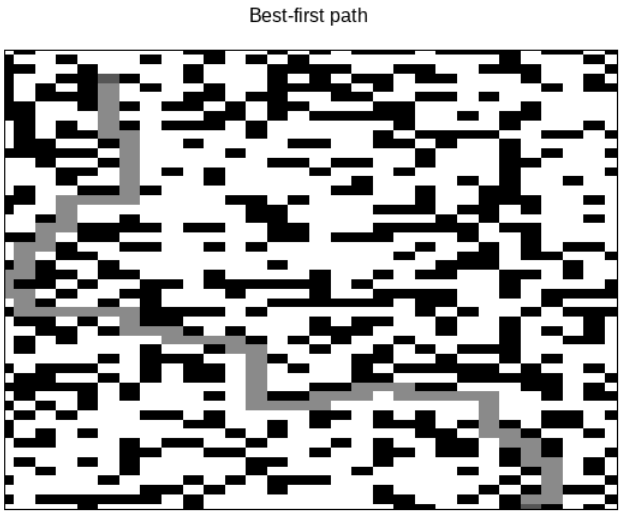

Figure 8: Solução de um labirinto utilizando busca Best-First

Busca A*

A busca A* é uma busca informada que se move sempre em direção ao nó mais próximo do nó de objetivo, com base em uma heurística. O valor de um nó nesse método é calculado utilizando a heuristica $f(x) = h(x) + g(x)$, um pouco diferente da heurística vista na Busca Best-First, e definida da seguinte forma:

- $h(x)$: chamado de *forward cost*, estimativa da distância do nó atual x até o nó de objetivo.
- $g(x)$: chamado de *backward cost*, é o custo cumulativo do caminho percorrido do nó inicial até o atual.

Um pseudocódigo pode ser encontrado na página seguinte.

Considere a Figura 9, na página seguinte: que solução permite que se vá do nó S ao nó G? Na tabela abaixo será possível ver como o algoritmo de busca A* resolve o problema.

Caminho	h(x)	g(x)	f(x)
S	7	0	7
S → A	9	3	12
S → D ✓	5	2	7
S → D → B ✓	4	3	7
S → D → E	3	6	9
S → D → B → C ✓	2	5	7
S → D → B → E ✓	3	4	7
S → D → B → C → G	0	9	9
S → D → B → E → G ✓	0	7	7

Algorithm 4 A-Star(G, root)

```
let PQ be a priority queue
let dist[x] be the cumulative distance from the root to the node x
label root as discovered
PQ.enqueue({0, root})
while Q is not empty do
    v := Q.dequeue()
    if v is the goal then
        return v
    end if
    for each w ∈ G.adjacentNodes(v) do
        if node w is not labeled as discovered then
            label w as discovered
            dist[w] := dist[v] + G.distanceBetween(v, w)
            cost := h(w) + dist[w]
            Q.enqueue({cost, w})
        end if
    end for
end while
```

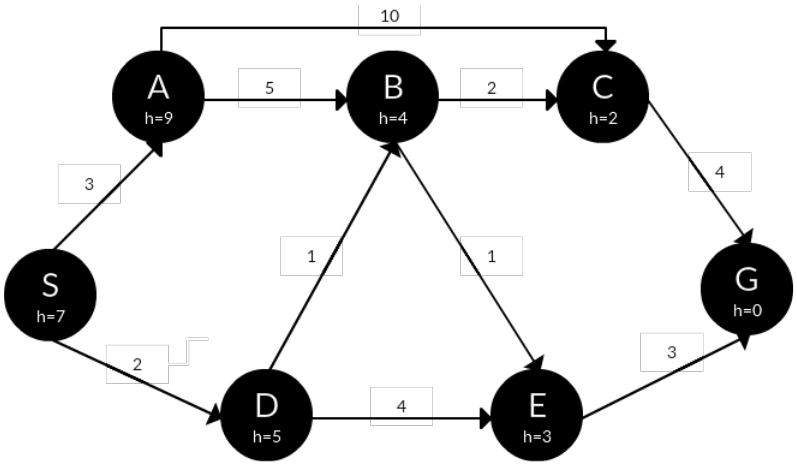

Figure 9: Grafo de um problema de busca com heurística para A*

A cada passo, o algoritmo calcula a heurística $f(x) = h(x) + g(x)$ e escolhe qual o melhor nó para ir. No exemplo dado, o melhor caminho foi $\mathbf{S} \rightarrow \mathbf{D} \rightarrow \mathbf{B} \rightarrow \mathbf{E} \rightarrow \mathbf{G}$ com um custo de 7.

Em seguida, o algoritmo será avaliado em relação às quatro propriedades mencionadas na Introdução:

- **Completeza:** a busca A* é um algoritmo completo e sempre encontrará solução.
- **Complexidade de tempo:** $O(b^d \cdot log(b^d))$, onde b é o fator de ramificação (*branching factor*) - número médio de filhos que um dado nó possui -, e d é a profundidade da

árvore. O *log* se dá por conta da complexidade da inserção em uma *priority queue*. Uma boa heurística pode reduzir essa complexidade significativamente.

- **Complexidade de espaço:** $O(b^d)$.
- **Otimalidade:** Garante solução ótima quando, para todos os nós, h(x), o *forward cost*, é menor ou igual ao valor real de h*(x). O nome dessa propriedade é admissibilidade e ela pode ser escrita como: $0 \leq h(x) \leq h^*(x)$

Na Figura 10, é possível ver o caminho encontrado pela busca A* para um labirinto gerado aleatoriamente.

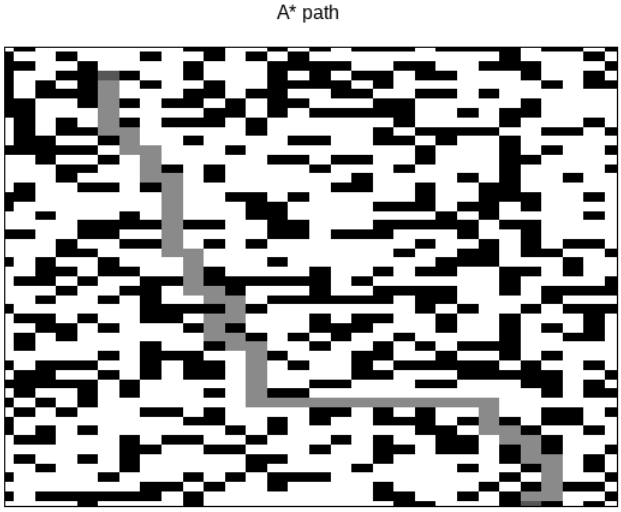

Figure 10: Solução de um labirinto utilizando busca A*

Busca Hill Climbing

A busca Hill Climbing é uma busca informada que se move sempre em direção ao nó com maior valor próximo ao nó atual, sem armazenar uma árvore de busca e sem fazer *backtracking* - o que faz com que muitas vezes não encontre caminho algum. O valor de cada nó nesse método é calculado utilizando a heuristica $h(x)$, definida da seguinte forma:

- $h(x)$: distância estimada do nó atual x até o nó de objetivo.

Na página seguinte, é possível ver um pseudocodigo do algoritmo de busca Hill Climbing.

Na Figura 11, é possível ver o comportamento da busca Hill Climbing em um labirinto gerado aleatoriamente. A busca rapidamente bateu em uma parede e ficou presa, pois a heurística assumia um valor mais baixo em todos os vizinhos.

Em seguida, o algoritmo será avaliado em relação às quatro propriedades mencionadas na Introdução:

Algorithm 5 Hill-Climbing(G, root)

```
1: let goal be the goal node
2: current := root
3: while current is not equals goal do
4:     next := a highest-valued successor of current
5:     if next.VALUE < current.VALUE then
6:         return current
7:     end if
8:     current := next
9: end while
```

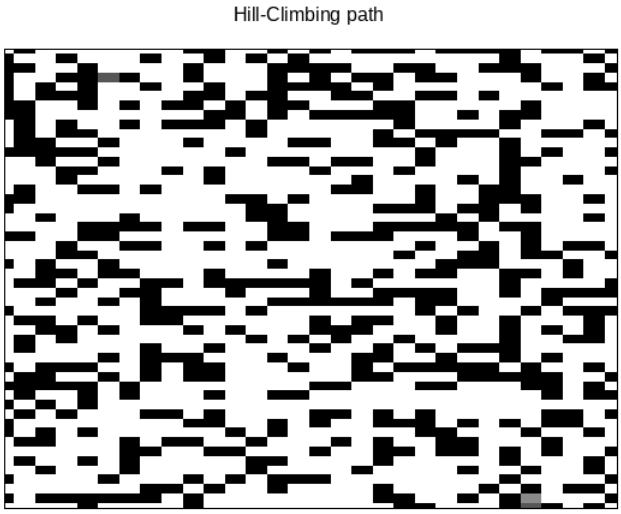

Figure 11: Solução de um labirinto utilizando busca Hill Climbing

- **Completeza:** a busca Hill-Climbing não é um algoritmo completo e nem sempre encontrará solução, mesmo que ela exista.
- **Complexidade de tempo:** $O(d)$.
- **Complexidade de espaço:** $O(b)$.
- **Otimalidade:** Não garante solução ótima.

Comparação dos algoritmos

Tempo de execução

Foram gerados labirintos de dimensões até 1000x1000 para comparação dos algoritmos estudados neste documento. Na Figura 12 é possível observar o comportamento de cada um dos algoritmos para labirintos de diferentes tamanhos e complexidades.

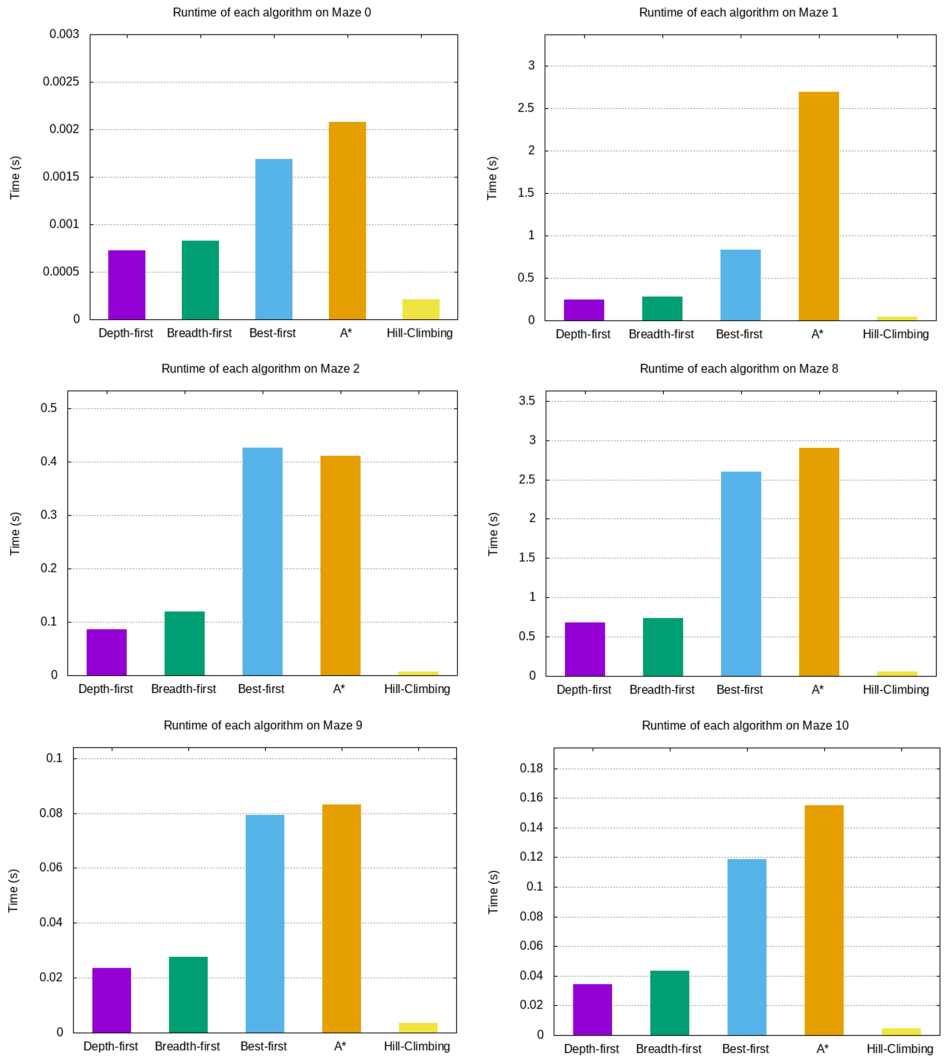

Figure 12: Comparação de tempo de execução dos algoritmos

É possível perceber que os algoritimos de busca informada possuem tempos muito maiores de execução, isso se dá por conta da complexidade de inserir elementos na *priority queue* afetar a complexidade do código de forma mais intensa do que a utilização de uma heurística. Vale notar que o Hill Climbing possui tempo de execução baixo pois parou sua execução muito rapidamente sem encontrar caminho algum.

Caminhos encontrados

Na Figura 13 é possível ver o caminho encontrado por cada um dos algoritmos. O caminho ótimo foi encontrado pelos algoritmos de busca em profundidade e busca A* - os caminhos são diferentes, porém possuem o mesmo tamanho.

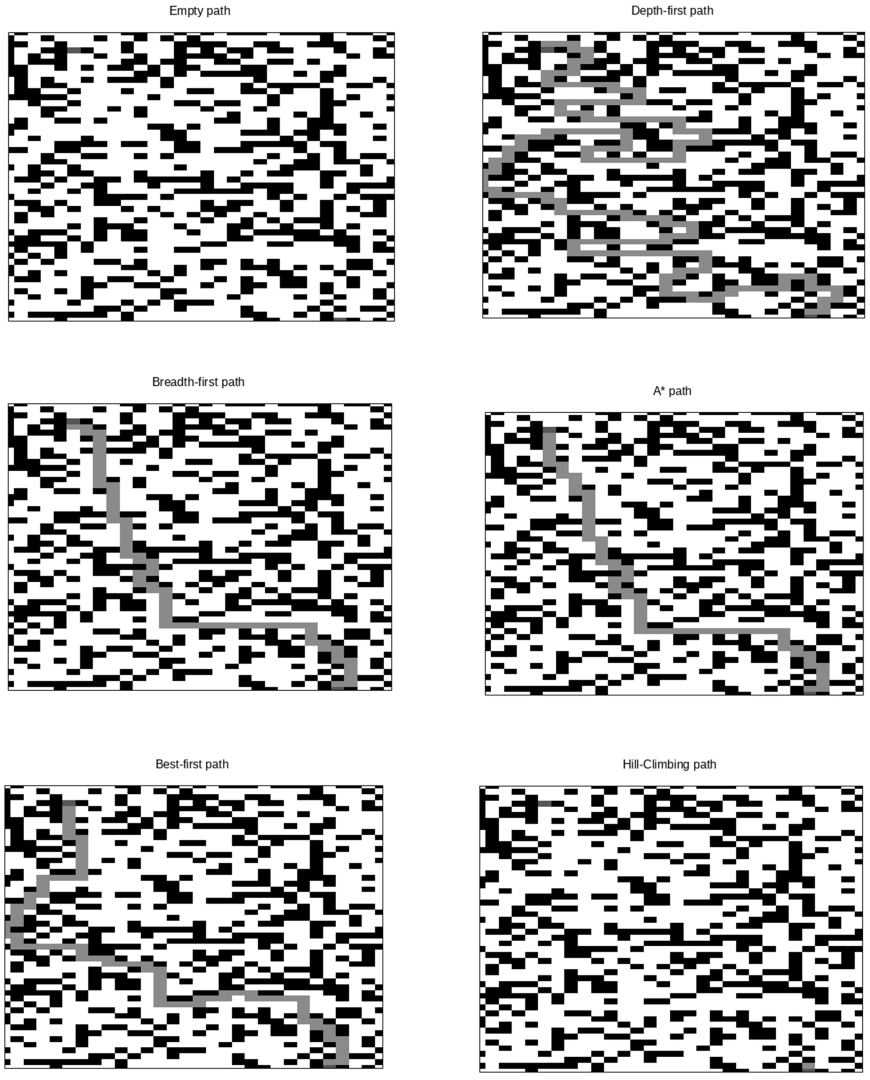

Figure 13: Caminho encontrado por cada algoritmo

Propriedades

Algoritmo	Tipo	Tempo	Espaço	Completo	Ótimo
Busca em largura	Cego	$O(b^d)$	$O(b^d)$	✓	✓
Busca em profundidade	Cego	$O(b^d)$	$O(bd)$	✓	
Busca Best-First	Informado	$O(b^d \cdot log(b^d))$	$O(b^d)$	✓	
Busca A*	Informado	$O(b^d \cdot log(b^d))$	$O(b^d)$	✓	✓
Busca Hill Climbing	Informado	$O(d)$	$O(b)$		

Executando o código

Detalhes sobre como executar o código e gerar os gráficos podem ser encontrados no arquivo *README* no repositório do projeto.

Referências bibliográficas

[1] Educba Contributors. Search algorithms in ai, 2019. [Online; accessed 20-October-2020].

[2] Fernando Gomide. Estruturas e estratégias de busca, 2018. [Online; accessed 20-October-2020].

[3] Java T Point. Informed search algorithms, 2019. [Online; accessed 20-October-2020].

[4] Rafi Akhtar. Search algorithms in ai, 2019. [Online; accessed 20-October-2020].

[5] Wikipedia contributors. Best-first search — Wikipedia, the free encyclopedia, 2019. [Online; accessed 20-October-2020].

[6] Wikipedia contributors. A* search algorithm — Wikipedia, the free encyclopedia, 2020. [Online; accessed 20-October-2020].

[7] Wikipedia contributors. Breadth-first search — Wikipedia, the free encyclopedia, 2020. [Online; accessed 20-October-2020].

[8] Wikipedia contributors. Depth-first search — Wikipedia, the free encyclopedia, 2020. [Online; accessed 20-October-2020].

[9] Wikipedia contributors. Hill climbing — Wikipedia, the free encyclopedia, 2020. [Online; accessed 20-October-2020].