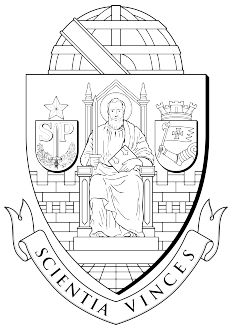
Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação
SCC0530 - Inteligência Artificial
Prof. Dr. Alneu de Andrade Lopes

Algoritmos de Busca

Pathfinder

Eduardo Alves Baratela	10295270
João Vitor dos Santos Tristão	10262652
Vitor Gratiere Torres	10284952



São Carlos, São Paulo, Brasil
Junho de 2020

Conteúdo

1 Introdução

Este estudo consiste no desenvolvimento de cinco algoritmos de busca em cima de labirintos (tanto entrados pelo usuário quanto gerados aleatoriamente), são eles *Depth First Search*, *Breadth First Search*, *Best First Search*, *A* Search* e *Hill Climbing Search*.

Durante suas execuções guardamos métricas para comparação a posteriori, bem como registramos cada iteração dos algoritmos para criar uma animação para visualização deles.

Nesse documento entramos em alguns detalhes de implementações, e discutimos heurísticas e resultados.

2 Descrição das Implementações

Decidimos representar o labirinto por uma matriz de 0s e 1s, onde os 0s representam caminhos livres e os 1s representam as paredes.

As dimensões, entrada e saída do labirinto serão fornecidas através da entrada do usuário ou através do gerador de labirintos aleatório.

2.1 Depth First Search

Um algoritmo de Busca em Profundidade (DFS) realiza uma busca não-informada que progride através da expansão do primeiro nó filho da árvore de busca, e se aprofunda, até que o alvo da busca seja encontrado ou até que ele se depare com um nó que não possui filhos (nó folha). Então a busca retrocede (*backtrack*) e começa no próximo nó.

No caso desta implementação, optamos pela abordagem não-recursiva. Através do uso de uma pilha, todos os nós expandidos (possíveis caminhos) são adicionados a mesma, para realizar a exploração. Enquanto a pilha não estiver vazia ou o local desejado ainda não for encontrado, repete-se o algoritmo. Uma vez que o fim for achado, o caminho retornado é toda a ramificação desde o primeiro nó filho até o nó folha da resposta.

2.2 Breadth First Search

A Busca em Largura (BFS), assim como a DFS, é uma busca não-informada, porém se expande através de uma busca exaustiva, passando por todas as arestas e vértices do grafo. Assim, o algoritmo deve garantir que nenhum vértice ou aresta será visitado mais de uma vez e, para isso, utiliza uma fila para garantir a ordem de chegada dos vértices, ao invés da pilha usada na DFS.

Para a aplicação em questão, além do uso da pilha para marcar os caminhos, por motivos de futuras análises, queríamos saber também qual era o menor caminho. Portanto, adicionamos um dicionário '`parent`' que marcava qual era o nó-pai de determinada posição. Através disso, ao menor caminho chegar ao seu destino, conseguimos retornar até o primeiro nó-filho da árvore, mostrando todo o caminho.

```
1  def __min_path(parent, begin, end):
2      min_path = list()
3      cur_pos = tuple(begin)
4
5      while cur_pos != end:
6          min_path.append(cur_pos)
7          cur_pos = parent[tuple(cur_pos)]
8
9      min_path.append(end)
10
11     return min_path
```

2.3 Best First Search

A busca *best-first search* é um método de busca informada que explora o caminho mais promissor de acordo com a regra estabelecida. Durante a execução do algoritmo, mantém-se uma lista de caminhos possíveis ordenados pela heurística estabelecida. Escolhe-se a cada iteração a célula de menor peso, ou seja, o candidato mais próximo como próximo nó a ser visitado e adiciona-se os novos caminhos que essa célula gera a lista de caminhos.

Para essa implementação utilizamos uma fila de prioridade mínima (*heap*) para ordenar os caminhos possíveis de acordo a heurística estabelecida. Começando pelo ponto de partida, os possíveis caminhos são levantados e juntos com eles a distância até o ponto de saída. Todos

os possíveis caminhos são adicionados à pilha e o de menor distância é escolhido como próximo caminho a ser seguido. Isso se repete até que o destino seja encontrado.

Como heurística, utilizamos duas funções: a distância euclidiana e a distância absoluta ou Manhattan. Considerando dois pares $I = (x_1, y_1)$ e $F = (x_2, y_2)$, onde I é o ponto atual e F o ponto final, a distância euclidiana considera o peso como uma linha reta traçada entre I e F, já a distância absoluta considera a distância apenas em movimentos horizontais e verticais.

2.4 A* Search

A busca *A* search* é, assim como a *best-first search*, uma busca informada que explora o caminho mais promissor de acordo com uma heurística estabelecida. Ela diferencia-se da *best-first search* ao levar em consideração o caminho já percorrido na sua função heurística.

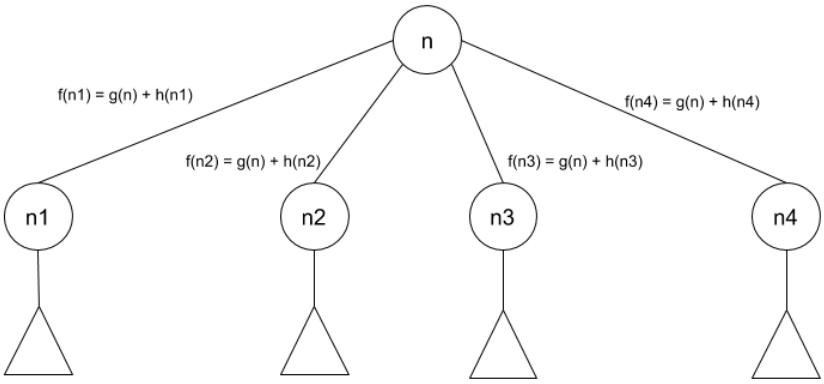


Figura 1: Busca A*

Nesse exemplo, o custo para seguir o caminho n_1 é dado por $f(n_1) = g(n) + h(n_1)$, onde $g(n)$ representa o custo de se chegar no nó n e $h(n_1)$ o custo estabelecido pela heurística para se seguir por n_1.

Já a *best-first search*, tem como custo para seguir n_1 a função $f(n_1) = h(n_1)$.

A implementação é bem pareceida com a de *best-first search*, utilizando uma fila de prioridade mínima (*heap*) para organizar os caminhos a serem escolhidos e armazenando também a distância da origem até o caminho atual. A grande diferença aparece no cálculo heurístico:

```
distance = find_distance(move, maze, heuristic) + height + 1
```

Comparado com o cálculo na busca *best-first search*:

```
distance = find_distance(move, maze, heuristic)
```

Onde `find_distance` é um função para calcular a distância até o ponto final, `move` é o nó atual, `maze` é o labirinto, `heuristic` indica a heurística escolhida (absoluta ou euclidiana) e `height` é o número de passos dados do nó inicial até o nó atual.

2.5 Hill Climbing Search

A busca com *Hill Climbing* é também um método de busca informada que explora caminhos mais promissores de acordo com heurísticas e parte da ideia de construir uma topografia a partir de uma função e percorre o labirinto ponderado pelo gradiente dessa função.

Decidimos utilizar a função distância, consequentemente criamos uma topografia que se assemelha a uma montanha, com o pico dessa montanha centrado no destino. Consequentemente, o peso para se mover nas casas do labirinto são mais leves se mais próximos do pico, dessa forma, subindo a montanha sempre que possível.

3 Heurísticas

Utilizamos apenas duas heurísticas a distância euclidiana e a distância de Manhattan, ambas descritas abaixo.

```
def find_distance(coord, maze, heuristic):
    if heuristic == "euclidean":
        return ((coord[0] - maze.end[0])**2 + (coord[1] - maze.end[1])**2)**0.5
    else:  # Manhattan
        return abs(coord[0] - maze.end[0]) + abs(coord[1] - maze.end[1])
```

3.1 Euclidiana

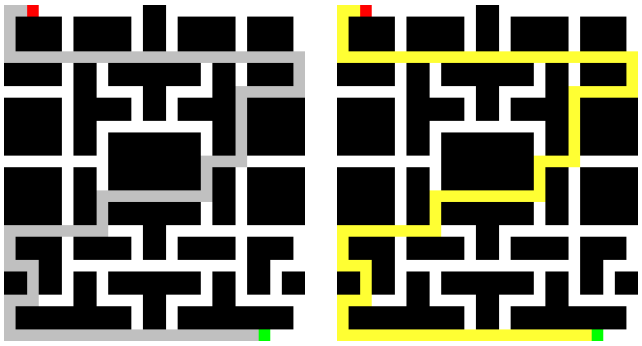
É o comprimento do segmento de linha entre dois pontos em um espaço Euclidiano calculado por $d(c, m) = \sqrt{(c_1 - m_1)^2 + (c_2 - m_2)^2}$.

3.2 Manhattan

Para distância entre dois pontos é utilizado a soma das diferenças absolutas de suas coordenadas cartesianas calculado por $d(c, m) = |c_1 - m_1| + |c_2 - m_2|$.

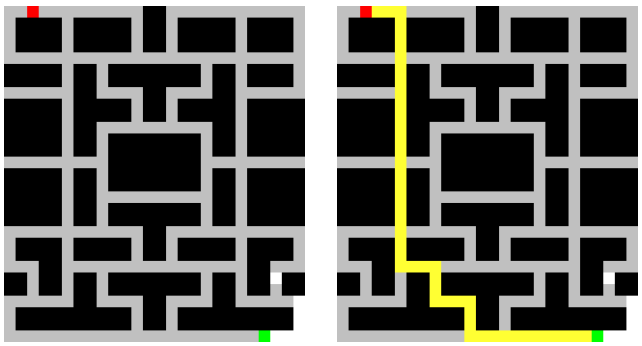
4 Exemplo

Usando como exemplo o labirinto disponibilizado na descrição do projeto, temos os seguintes exemplos:



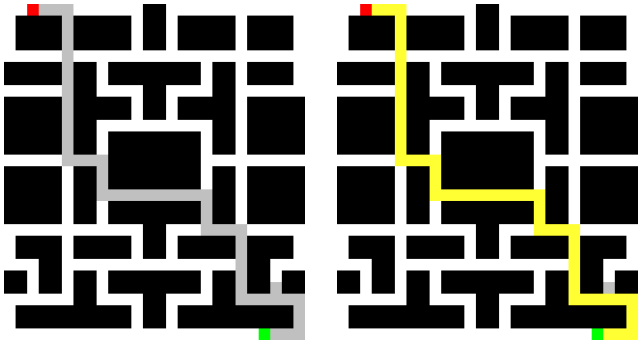
(a) Espaço explorado (b) Caminho mínimo

Figura 2: Depth-first search



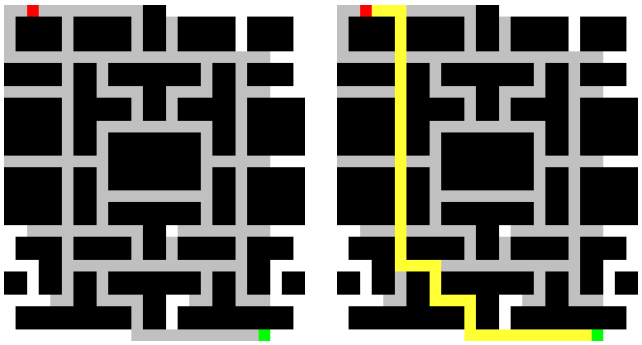
(a) Espaço explorado (b) Caminho mínimo

Figura 3: Breadth-first search



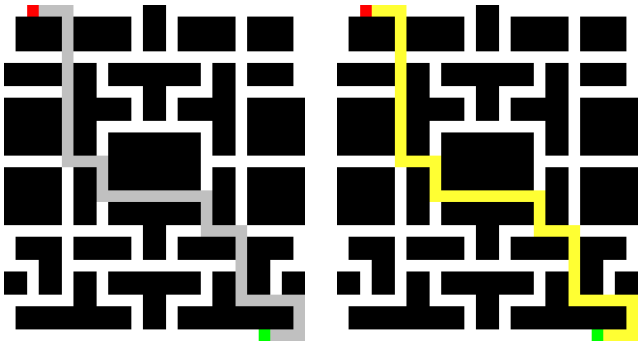
(a) Espaço explorado (b) Caminho mínimo

Figura 4: Best-first search



(a) Espaço explorado (b) Caminho mínimo

Figura 5: A* search



(a) Espaço explorado (b) Caminho mínimo

Figura 6: Hill Climbing search

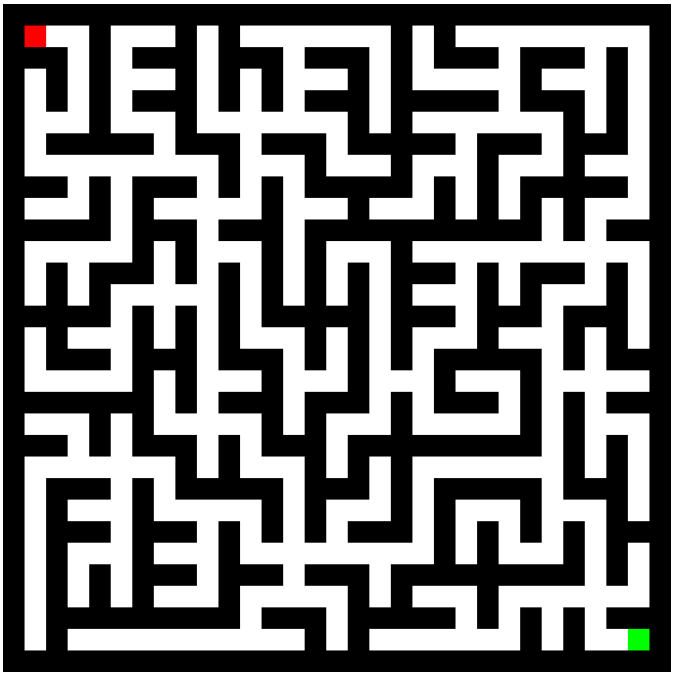


Figura 7: Labirinto Artificial

5 Resultados

5.1 Número de Iterações e Caminho Mínimo

Com os algoritmos em mãos, gerou-se labirintos de tamanho 50, 100 e 250. Para cada configuração de dimensão, geraram-se 10 labirintos aleatórios diferentes e mediu-se o o número de iterações necessários para se obter um caminho com cada algoritmo. Os resultados são apresentados em 8.

Comparando os resultados, as buscas informadas foram muito mais eficientes quando comparadas com as não informadas. Dentre `breadth-first search` e `depth-first search`, a primeira apresentou um número menor de iterações, o que esta dentro do esperado, uma vez que a BFS explora um conjunto maior de possibilidades ao mesmo tempo.

Comparando as buscas informadas, `best-first search` apresentou o melhor resultado, sendo seguida de perto por `hill climbing search`. `A* search` apresentou resultados piores em termos de número de iterações, uma vez que assim como BFS, considera um número maior de

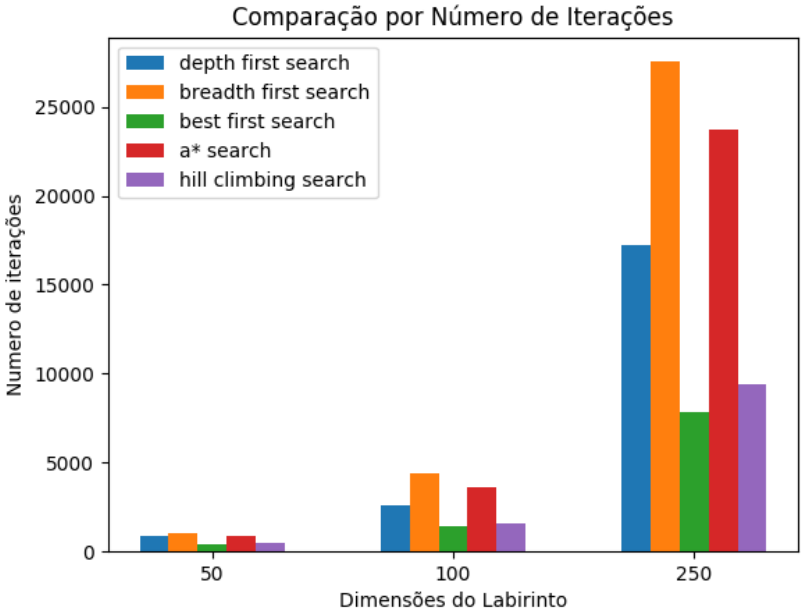


Figura 8: Número médio de iterações para se chegar ao ponto final

possibilidades ao mesmo tempo.

Estabeleceu-se então comparações entre os menores caminhos encontrados dentro do espaço de busca estabelecido em cada algoritmo(9. Os algoritmos `breadth-first search` sempre apresenta o caminho mínimo, mas para isso, precisa de um número maior de iterações. Já `best-first search`, `A* search` e `hill climbing search` aproximam-se bastante do caminho mínimo. Apenas `depth-first search` não gera um caminho mínimo próximo.

	Tamanho do labirinto		
Algoritmo	50x50	100x100	250x250
Depth First Search	760.8	2553.9	18113.5
Breadth First Search	1159.3	4217.6	26719.8
Best First Search	511.9	1294.2	8656.4
A* Search	951.6	3441.2	21558.2
Hill Climbing Search	586.4	1442.8	8271.4

Tabela 1: Número de iterações necessárias para completar a execução

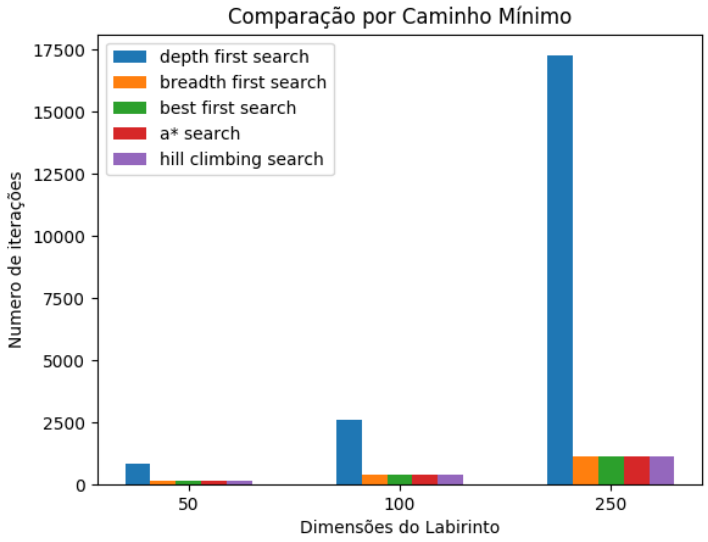


Figura 9: Número de iterações dos algoritmos por dimensão do labirinto buscando o caminho mínimo

	Tamanho do labirinto		
Algoritmo	50x50	100x100	250x250
Depth First Search	760.8	2553.9	18113.5
Breadth First Search	148.6	339.4	1139.0
Best First Search	148.6	339.4	1139.0
A* Search	148.6	339.4	1139.0
Hill Climbing Search	148.6	339.4	1139.0

Tabela 2: Número de passos do caminho mínimo encontrado

6 Guia para Compilação e Execução

6.1 Organização de pastas

```
./
    Maze/
        Algorithm.py (Implementacao dos algoritimos de busca)
        Animation.py (Gerador de animacoes para as iteracoes dos algoritmos)
        Maze.py (Definicao da classe Maze, seus atributos e algumas funcoes)
```

```
        input.in (Entrada de um labirinto descrito na definicao do trabalho)
        pymaze/ (Gerador de labirintos)
        stats.py (Stript para gerar analises estatisticas dos algoritmos)
        video_generator.py (Gerador de animacoes de todos os algoritmos)
        maze_solver.py (Resolvedor de labirintos, retornando os caminhos
    percorridos).
    Videos/ (Pasta contendo execucoes dos algoritmos para 'input.in')
        a_star_search.avi
        best_first_search.avi
        depth_first_search.avi
        hill_climbing_search.avi
    README.md
    requirements.txt (Modulos necessarios para execucao)
```

6.2 Módulos necessários

Os módulos necessários para a plena execução do nosso trabalho podem ser instalados com o comando `python3 -m pip install -r requirements.txt`.

6.3 Gerando vídeos de execuções

O comando `python3 Maze/video_generator.py path/to/maze` executa nosso projeto e gera vídeos das execuções.

Note que `'path/to/maze'` é um arquivo de texto com o labirinto descrito nele com as especificações definidas pelo trabalho, ou seja, as duas dimensões na primeira linha, seguidas pelo labirinto em si.

6.4 Gerando estatísticas

Executando `python3 Maze/stats.py` este programa retornará uma comparação média entre todos os algoritmos para labirintos gerados aleatoriamente com tamanhos 50, 100, 250.

6.5 Obtendo caminhos

O comando `python3 Maze/maze_solver.py path/to/maze` executa os cincos algoritmos predefinidos e imprime as células visitadas na ordem em que foram visitadas, bem como o caminho mínimo que pode ser obtido nesse conjunto de células.

Note que `'path/to/maze'` é um arquivo de texto com o labirinto descrito nele com as especificações definidas pelo trabalho, ou seja, as duas dimensões na primeira linha, seguidas pelo labirinto em si.