

Inteligência Artificial

Docente Joaquim Gonçalves

Licenciatura em Engenharia Informática Médica

2023/2024

**Algoritmos de Pesquisa**

*Profundidade, largura e A\**

23544 - Ana Beatriz Machado Carvalho

23548 – Ana Margarida Maia Pinto

23552 - Diana Alexandra da Costa Dinis

Índice

[1. Introdução 1](#_Toc151729877)

[1.1. Enquadramento 1](#_Toc151729878)

[1.2. Estrutura do documento 1](#_Toc151729879)

[2. Contexto 2](#_Toc151729880)

[2.1. Estratégia 2](#_Toc151729881)

[3. Algoritmos de Pesquisa 3](#_Toc151729882)

[3.1. Largura 3](#_Toc151729883)

[3.2. Profundidade 3](#_Toc151729884)

[3.3. A\* 4](#_Toc151729885)

[3.3.1. Heurística 4](#_Toc151729886)

[4. Testes e Resultados 5](#_Toc151729887)

[5. Conclusão 6](#_Toc151729888)

[6. Bibliografia 7](#_Toc151729889)

**Índice de Figuras**

[Figura 1 - Representação gráfica do algoritmo de pesquisa em largura 3](#_Toc151729890)

[Figura 2 - Representação gráfica do algoritmo de pesquisa em profundidade 4](#_Toc151729891)

**Índice de Tabelas**

**Não foi encontrada nenhuma entrada do índice de ilustrações.**

**Lista de siglas e acrónimos**

**BFS** Algoritmo de pesquisa em largura

**DFS** Algoritmo de pesquisa em profundidade

**FIFO** *First In First Out*

**FILO** *First In Last Out*

**IA** Inteligência Artificial

# Introdução

Nos dias de hoje, há uma crescente adesão à tecnologia, sendo a automação e inteligência artificial inerente.

No contexto de armazéns, a automação deste pode melhorar a produtividade logística, tendo efeitos também na segurança dos trabalhadores, com a diminuição de acidentes de trabalho com a maquinaria pesada utilizada em armazéns de grande porte.

O trabalho terá como plano a automação de um armazém de uma empresa de encomendas, onde um *robot* irá receber um pedido de encomenda de um item e terá de calcular um caminho desde a sua posição até à localização do item encomendado no armazém.

A fase inicial do trabalho será a implementação de um único *robot* com um único objeto de encomenda.

A fase seguinte será a implementação de um único *robot* com dois objetivos.

O trabalho será implementado, em *Python*, uma representação do armazém, assim como os algoritmos que o *robot* terá de modo a conseguir uma solução ao problema. Serão avaliados o tempo que o algoritmo demora a encontrar uma solução e se essa solução será a solução ótima. Este conceito aplicado a este projeto estará explicado posteriormente.

## Enquadramento

Este trabalho enquadra-se da unidade curricular de Inteligência Artificial lecionada pelo docente Joaquim Gonçalves da Licenciatura de Engenharia Informática Médica, no Instituto Politécnico do Cávado e do Ave.

## Estrutura do documento

Este documento encontra-se dividido em essencialmente cinco capítulos. Inicia com o contexto do trabalho, onde estará exposto os objetivos do trabalho. Segue-se com o capítulo dos algoritmos de pesquisa, onde estará uma introdução teórica a cada um dos algoritmos escolhidos. A seguir, o capítulo dos testes e resultados, onde os algoritmos serão aplicados ao problema em Python e, por fim, a conclusão, onde os resultados serão comparados e será escolhido o melhor algoritmo para cada situação.

## Ferramentas utilizadas

***Version Control***

* GitHub

Repositório de versões.

<https://github.com/xaloftal/IA>

**Planeamento**

* Trello

Ferramenta de organização e planeamento de projetos

**Implementação**

* Visual Studio Code - VSCode

IDE de programação

**Comunicação**

* Discord

Rede social de comunicação por voz

Utilizado para as reuniões do grupo

* Whatsapp

Rede social de mensagens instantâneas

Utilizado para a comunicação mais informal

# Algoritmos de Pesquisa

Este capítulo tratará da introdução teórica e implementação de cada algoritmo escolhido para o desenvolvimento do trabalho prático.

Os algoritmos de pesquisa, de modo a conseguirem prosseguir com a procura, utilizam uma *frontier*, que irá gerir os nós a expandir, de modo que o algoritmo saiba as opções que tem para prosseguir com a procura, sendo um conceito comum dos algoritmos.

## Largura

O algoritmo de pesquisa em largura, ou BFS, irá seguir múltiplas direções ao mesmo tempo, explorando vários caminhos simultaneamente. Para tornar isso possível, a *frontier*, será montada como uma *queue*, tendo como método o FIFO – *First In First Out*.

Assim, todos os novos caminhos possíveis são adicionados à fila e expandidos conforme a ordem que foram adicionados. Montando uma árvore, visível na Figura 1, observa-se que se expande todos os nós do mesmo nível antes de passar para outro nível:

Uma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamenteUma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamente

Uma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamenteUma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamente

Figura 1 - Representação gráfica do algoritmo de pesquisa em largura

Para isto, na definição de *frontier*, que será a *queueFrontier,* definida como a classe deque da biblioteca *collections*, neste caso será feito a eliminação do primeiro item guardado na variável, com recurso ao método popleft(), como se pode observar a seguir:

queueFrontier.popleft()

O par de coordenadas eliminado será armazenado como (x, y).

Caso esse par seja o fim, o caminho será reconstruído a partir do dicionário de pais, definido no início da função de bfs.

Se não, adiciona os novos vizinhos à *frontier* que tenham coordenadas válidas conforme a matriz selecionada, guardado o parente destes.

As vantagens do algoritmo de pesquisa em largura são os seguintes:

* + Garante que a solução é o menor caminho possível;
  + Implementação simples ;
  + Fácil compreensão.

As desvantagens deste algoritmo são as seguintes:

* No pior caso, é o algoritmo que demora o maior tempo possível;
* Ocupa muita memória, mantendo todos os nós explorados.

## Profundidade

O algoritmo de pesquisa em profundidade, ou DFS, é o oposto do algoritmo de pesquisa em largura. Enquanto o algoritmo em largura exausta todos os nós a expandir do mesmo nível, seguindo vários caminhos ao mesmo tempo, o algoritmo em profundidade segue apenas um caminho até ao fim, representado na Figura 2.

Uma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamenteUma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamente

Uma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamenteUma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamente

Uma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamenteUma imagem com texto, captura de ecrã, software, Ícone de computador

Descrição gerada automaticamente

Figura 2 - Representação gráfica do algoritmo de pesquisa em profundidade

Para isso, a estratégia deste algoritmo é programada para agir como uma *stack*, agindo em FILO – *First In Last Out*. Com isso, ao expandir os nós, o último a ser adicionado à *frontier* irá ser o próximo a ser expandido. O último nó adicionado é selecionado através do método pop() da biblioteca *collections*, da seguinte forma:

stackFrontier.pop()

Sendo armazenado também num par ordenado (x, y), seguindo a estratégia já exposta no subcapítulo do BFS para a determinação do caminho com a *frontier*.

As vantagens do algoritmo de pesquisa em profundidade:

* Tendo a sorte de o escolher, encontra o caminho de forma mais rápida;
* Não precisa de tanta memória, apenas guarda os nós do caminho atual;

As desvantagens deste algoritmo são os seguintes:

* A solução encontrada não é garantidamente a ótima;
* No pior dos casos, explora todos os caminhos possíveis até ao fim antes de encontrar a solução, demorando o maior tempo possível.

## A\*

### Heurística

# Testes e Resultados

Neste capítulo estarão expostos os testes a cada algoritmo e a comparação entre os resultados de cada.

A solução para este projeto será o caminho encontrado pelos algoritmos desde a posição inicial até ao objetivo, ou seja, até ao item no armazém. Uma solução pode ser considerada a solução ótima quando esta for o menor caminho, com o menor custo, tendo também peso o tempo de processamento do algoritmo.

## Preparação

De modo a representar o armazém, com objetivo de testar os algoritmos, surgiu a necessidade de criar matrizes, onde 0 representa os caminhos livres e 1 as paredes. Estas serão sempre quadradas, com o mesmo número de linhas e colunas.

Com isso, definiu-se uma forma de gerar matrizes de forma automática, no ficheiro randomMatrices.py, com o método *random.choice* da biblioteca *NumPy*:

np.random.choice([0, 1], size=(10, 10), p=[0.7, 0.3])

Onde o 0.7 representa a probabilidade de aparecer 0 e 0.3 a probabilidade de 1, ou seja, a matriz gerada aqui terá o tamanho 10x10 com probabilidade de obstáculos a 30%. A inserção do tamanho da matriz e as probabilidades tem de ser manipulada diretamente no código.

A matriz, após ser gerada, é armazenada no ficheiro outputMatrix.txt e, manualmente, tem de ser guardada no ficheiro fixedMatrices.py, onde estarão todas as matrizes armazenadas, com um nome que permite identificar o seu tamanho e a probabilidade de obstáculos, da seguinte forma:

matrix (tamanho)(probabilidade de obstáculos)

Ou seja, a matriz do exemplo de cima terá o nome matrix1030.

## Comparações

A primeira matriz a ser testada foi a matriz2530.

Com o início nas coordenadas (0,0) e o objeto a (9,2), da Figura 3 à figura estão representados os resultados no terminal do *VSCode* para os algoritmos em profundidade, largura e A\*, respetivamente:

Uma imagem com texto, captura de ecrã, software, Software de multimédia

Descrição gerada automaticamente

Figura 3 - DFS para a matriz2530 com fim a (9,2)

Uma imagem com texto, captura de ecrã, software, ecrã

Descrição gerada automaticamente

Figura 4.2 - BFS para a matriz2530 com fim a (9,2) com tempo de espera

No fim do processamento, de modo a conseguir apresentar o resultado sem ser como 0,0, foi introduzido um tempo de espera de 1 segundo, sendo, então, o resultado real é dado como 0,0080556869506836s.

Na tabela está representado todos os testes efetuados aos algoritmos, onde P é o tempo de processamento do algoritmo, em segundos, e C o custo, o número de passos dados, sem contar com a orientação.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Matriz | Objetivo | Métrica | Algoritmos | | |
| BFS | DFS | A\* |
| Matriz2530 | 9,2 | P | 0,00805568 | 0,00099992 |  |
| C | 13 | 37 |  |
| 20,24 | P |  |  |  |
| C |  |  |  |
| Matriz5030 |  | P |  |  |  |
| C |  |  |  |
|  | P |  |  |  |
| C |  |  |  |

Tabela 1 - Tabela de resultados

# Conclusão

# Bibliografia

https://www.mecalux.pt/blog/automatizacao-armazens

<https://www.geeksforgeeks.org/depth-first-traversal-dfs-on-a-2d-array/>

<https://www.baeldung.com/cs/dfs-vs-bfs-vs-dijkstra>

<https://www.pygame.org/tags/algorithm>

https://www.pygame.org/docs/ref/time.html

<https://cs50.harvard.edu/ai/2023/notes/0/>

https://www.techwithtim.net/tutorials/game-development-with-python/snake-pygame/snake-tutorial-1

https://www.inf.ed.ac.uk/teaching/courses/inf2d/timetable/05\_SearchB\_wbg\_4up.pdf

https://en.wikipedia.org/wiki/Breadth-first\_search