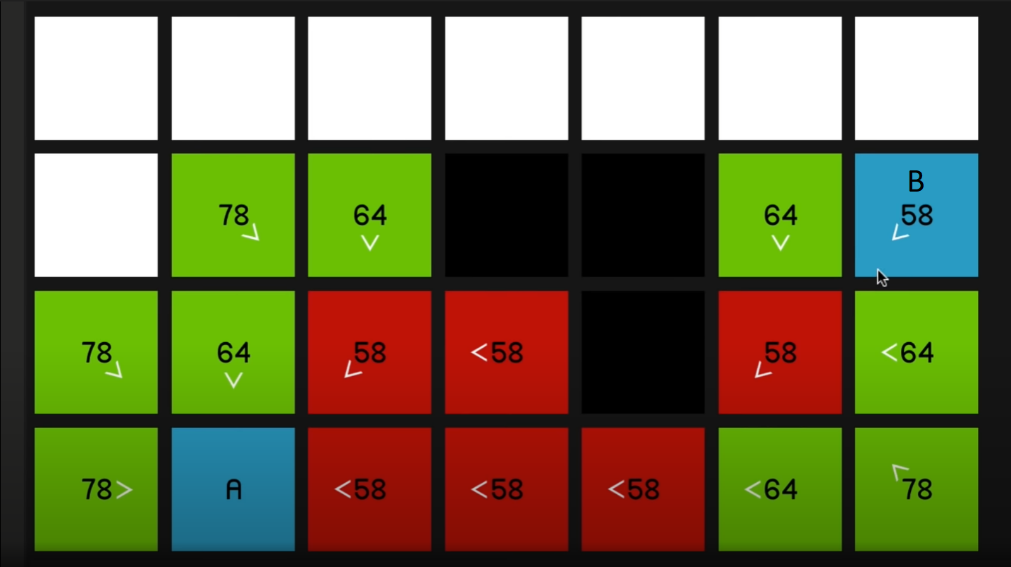
André Felipe Santos Martins

RELATÓRIO: ALGORITMO DE BUSCA A\*

# 1. Explicação teórica

O A\* (lê-se a-estrela) é um algoritmo muito utilizado na área de computação para encontrar caminhos entre dois pontos A e B, com base na introdução de uma heurística no algoritmo de busca, no intuito de planejar com antecedência cada passo a ser dado, de modo que uma decisão mais otimizada possa ser feita. O A\* vem sendo utilizado amplamente para resolver vários problemas de diferentes áreas, como o alinhamento de múltiplas sequências de DNA em biologia, planejamento de caminhos em robótica e jogos digitais (Rios L.H.O., Chaimowicz L., 2010, pg. 1).

Figura 1- Exemplo de execução do A\* (fonte: [A\* Pathfinding](https://www.youtube.com/watch?v=-L-WgKMFuhE) vídeo).

O A\* é como uma extensão do algoritmo de Dijkstra, com adição de uma característica chamada Busca em largura (Breadth-first search), que funciona como uma árvore de caminho de custo ordenada por nós que contém o menor custo do nó inicial até o nó de destino. O que torna o A\* diferente e melhor em alguns casos, é que para cada nó, o A\* utiliza uma função para calcular uma estimativa do custo total de um caminho a partir do nó avaliado. Essa função é o que contém o que chamamos de heurística, pois trata-se mais de uma estimativa e não necessariamente da distância correta.

Durante a execução do algoritmo, o custo é calculado da seguinte maneira:

no qual,

nó avaliado;

custo total estimado do caminho através do nó ;

custo para chegar do nó inicial até ;

custo estimado de para percorrer de até o nó final. Esse cálculo se trata é feito por estimativa, portanto, trata-se da heurística.

De acordo com a , o algoritmo A\* inicia no ponto ***A***, em azul, e percorre todos os nós adjacentes em busca do nó que contém o menor custo . Uma lista é criada com todos os nós adjacentes e obstáculos são desconsiderados. Então o nó de menor custo é selecionado e todas essas operações são feitas recursivamente até que o nó encontrado seja o ***B***, que é o objetivo final. A função pode ser calculada de diversas formas, mas o método mais comum é o método de Manhattan, é um método padrão para calcular a heurística em uma tabela ou matriz. O cálculo de é feito da seguinte forma:

no qual,

e coordenadas na matriz o nó atual;

e = coordenadas na matriz do nó final (objetivo).

Para calcular a heurística de movimentos não diagonais, o método de Manhattan consegue atender de forma satisfatória, porém quando se trata de movimentos diagonais, pois no método de Manhattan não é considerado que movimentos 45 graus são permitidos. Para isso utilizamos o método Octil. Essa heurística leva em conta o fato que se pode viajar em arestas diagonais, então, se a distância entre um nó (x, y) é x e y, então a heurística pode ser calculada como:

no qual,

e coordenadas na matriz o nó atual;

e = coordenadas na matriz do nó final (objetivo);

e servem para representar que deve ser multiplicado pelo menor valor.

# 2. Problema proposto

O objetivo deste trabalho é implementar ou selecionar um bom algoritmo A\* e fazer uma análise da sua implementação, destacando pontos importantes do código e dando o devido destaque a melhor forma de implementar o algoritmo. Além disso vamos realizar testes com matrizes de diversos e tamanhos e demonstrar os resultados.

Os requisitos que devem ser atendidos neste trabalho são:

1. Utilizar/Testar o mesmo modelo de mapa (grid) disponível no enunciado do trabalho;
2. Fazer entrada de parâmetros por linha de comando;
3. Exibir ao final do programa um mapa com o desenho da trajetória, a lista de coordenadas a serem percorridas e o tempo gasto na execução.

# 3. Implementação

Já que havia a possibilidade de selecionar um algoritmo já implementado, por questões de tempo de priorização de algumas atividades, optamos por selecionar uma implementação feita na biblioteca em Python chamada [python-pathfinding](https://github.com/brean/python-pathfinding), que contém vários algoritmos de busca e encontrar caminhos, mas claramente com ênfase no A\*. Este repositório foi escolhido pois a implementação é feita utilizando diversas classes e estruturas de dados e trabalha de forma paramétrica, ou seja, existe uma classe base genérica chamada Finder, que contém os métodos genéricos de calcular custo, aplicar heurística, encontrar vizinhos etc., e é injetada dentro da classe devida de busca, por exemplo AStarFinder.

Para utilizar a biblioteca eu removi outros arquivos referentes a outras buscas, deixando o repositório mais enxuto. Para os testes, implementei uma função principal que recebe como parâmetro via linha de comando o nome do arquivo, o ponto inicial e o ponto final. O mapa é lido de dentro do arquivo e armazenado dentro de uma classe chamada Grid, que contém toda a estrutura dos nós, ponto inicial e final. Como neste trabalho trata-se de uma análise do algoritmo A\* baseado na biblioteca selecionada, irei demonstrar mais partes do código e fazer os devidos comentários sobre.



Figura 2 - função Fonte: Implementada pelo autor.

A Grid é uma classe que tem o objetivo de representar a matriz do “labirinto. Ela recebe como parâmetro uma matriz (lista) e constrói internamente uma estrutura de nós, que onde cada nó possui suas propriedades de posição (x, y), se é um caminho viável (não se trata de uma barreira ou canto), seus valores de , g(n) e h(n) que serão calculados posteriormente, e uma propriedade auxiliar de aberto ou fechado, que é utilizado para verificar se o nó foi adicionado na lista de caminhos abertos ou fechados. A Grid contém os seguintes métodos:

* - Inicializa a estrutura de nós;
* - Retorna um nó de uma posição (x, y);
* - Verifica se uma posição (x, y) está dentro do mapa;
* - Verifica se um nó (x, y) não é um objeto ou parede;
* - Retorna uma lista de vizinhos de um nó (x, y);
* - Limpa grid;
* - Imprime a grid definida, aceitando como parâmetros os pontos inicial e final e uma lista contendo o caminho descoberto.

Para instanciar a classe AStarFinder passamos apenas o parâmetro , que é um enumerador e permite que o programador decida se é permitido realizar caminhos usando diagonais. Enfim, para rodar o buscador são passados pelo menos três parâmetros, o primeiro é a Grid e os dois últimos são instância de um nó para o início e o fim, que pode ser retirado diretamente da Grid com o comando .

A partir do momento que a função é chamada, todo algoritmo é realizado internamente pela biblioteca. A partir daqui todo código é referente ao repositório utilizado. Como citado anteriormente, tudo foi desenvolvido de uma forma genérica para que funcionasse com outros algoritmos de busca, então a apenas chama a sua equivalente dentro da classe Finder, que foi injetada no momento que a AStarFinder foi instanciada. Em seguida, temos a função de busca em si.



Figura 3 - função da classe AStarFinder.



Figura 4 - função da classe Finder.

A função nada mais faz do que o que foi citado na 1. Explicação teórica: é criada uma lista aberta com o primeiro nó, e enquanto houver nós dentro dessa lista, o loop continua o looping. Um dos parâmetros opcionais da função é o tempo máximo de execução do programa, para isso serve a linha que contém , que verifica que o tempo limite foi excedido, porém não foi considerada essa funcionalidade neste trabalho.



Figura 5 - função na classe AStarFinder.

A função deve ser implementada individualmente, por isso ela se encontra dentro da classe AStarFinder com sua própria implementação. É nela que é selecionado o nó de menor custo, retirada da lista como numa estrutura de pilha utilizando o comando , e todos seus vizinhos são verificados e processados dentro do looping, removendo o nó atual da lista aberta, fechando-o e desconsiderando vizinhos que são obstáculos.

Dentro da função é chamada a função , que retorna a lista de vizinhos válidos. Cada vizinho verificado se ele já foi para lista fechada. A parte mais importante é , no qual o vizinho é processado, calculando seu custo checando se se ele é o objetivo final. Ignore a função , pois ela é utilizada apenas no algoritmo A\* bidirecional.



Figura 6 - função na classe Finder.

Na função é verificado se o nó faz parte do caminho através do cálculo do seu custo e baseado nisso ele é adicionado ou removido do caminho final. O valor de do nó enviado como parâmetro é calculado através da função , que basicamente verifica a distância entre o nó atual e seu vizinho, também verificando se é um vizinho direto ou um vizinho indireto (diagonal).

Se o nó ainda não foi processado em nenhum momento ou seu custo de movimento medido no momento for menor (, o valor de é mantido, e então calcula-se a heurística, caso ainda não calculada, utilizando . Nesta função é chamada a o cálculo da heurística, que nesse caso é o método Octil, citado na1. Explicação teórica. Calculado o e o , é somado em e armazenado. Se o nó não foi adicionado a lista aberta, ele é adicionado. Caso esteja aberto, a posição dele é atualizada dentro da lista aberta.

Seguindo a partir da Figura 5, quando o nó final é encontrado, é chamada a função , que percorre o último nó e concatena todos os nós pais até chegar ao nó inicial.

# 4. Resultados

Para execução e teste foram criados dois arquivos de diferentes, são eles:

* main.py – Arquivo que recebe parâmetros através do terminal, sendo eles o nome do arquivo e duas coordenadas no labirinto. Um exemplo de uso:
* mainV2.py – Arquivo utilizado para testar vários pontos aleatórios em massa. Todos os parâmetros estão atribuídos dentro do código, não são aceitos parâmetros via linha de comando.

Os dois arquivos são tem a mesma funcionalidade e possuem funções para tratamento e preparação da entrada de dados para serem passados como parâmetros. Um exemplo de execução o arquivo main.py sobre o labirinto no arquivo Mapa\_B.txt, é o seguinte:

Comando executado:



Figura 7- labirinto do arquivo Mapa\_B.txt

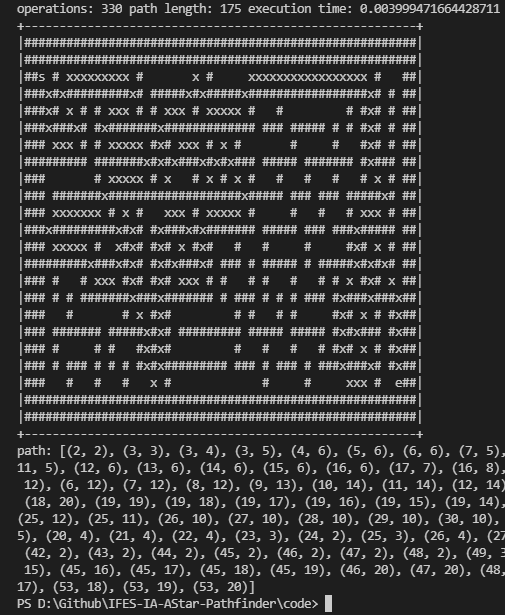


Figura 8 - Saída do programa.

Como podemos ver, no resultado temos o número de operações sobre a lista aberta, a quantidade de passos para chegar ao objetivo, o tempo de execução, o labirinto representando por X onde é o caminho e a lista inteira de coordenada dos caminhos. Como pode ver esse algoritmo pode performar em muito menos de um segundo. Mas para realmente testá-lo que foi criado o arquivo mainV2.py. Nele é feito testes em massa sobre os labirintos de diferentes tamanhos em pontos aleatórios do mapa.

Para efetuar os testes e armazenas os resultados, a saída do arquivo mainV2.py não consta com o mapa do labirinto nem o caminho percorrido, mas sim com dados referentes para análise a seguir. Comando executado: python .\mainV2.py > output.csv;



Figura 9 - exemplo de saída do arquivo mainV2.py.

A saída do teste da figura 9 foi feito com 100 coordenadas diferentes de início e fim. Com base no arquivo de CSV gerado, podemos gerar o seguinte gráfico:

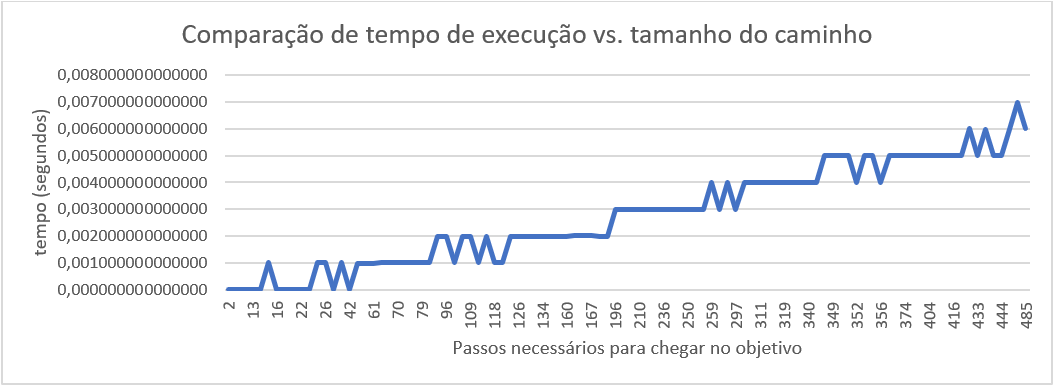


Figura 10 - Gráfico de comparação de tempo de execução e o tamanho do caminho percorrido.

Na Figura 10 é possível observar que o algoritmo performa muito bem até quase 500 passos, levando alguns milésimos para encontrar o objetivo final. Considerando que este labirinto, como é exibido na saída CSV, possui um tamanho de 22 linhas e 55 colunas, vamos fazer um teste em um labirinto maior.

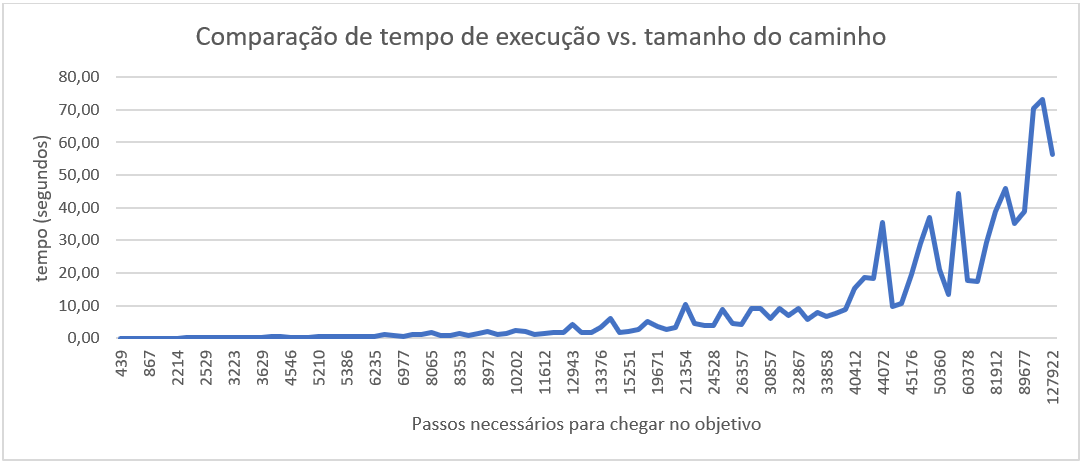


Figura 11 - comparação de tempo de execução e o tamanho do caminho no arquivo Mapa\_A\_\_V06.txt.

O teste da Figura 11 foi executado em 100 diferentes pontos de início e fim sobre o arquivo Mapa\_A\_\_V06.txt, de 1080 linhas e 540 colunas. Como podemos observar o gráfico, o algoritmo é bem performático até cerca de 33 mil passos distante do objetivo final, levando cerca de 10 segundos para encontrar o objetivo. Além disso, podemos observar que em cerca de 83 mil passos o algoritmo já levou cerca de 70 segundos para encontrar o objetivo. Claramente isso possui variáveis, como a disposição dos obstáculos sobre o labirinto, mas no geral, atende muito bem. Para trabalhos futuros seria interessante um gerador de labirintos, que possibilite gerar labirintos superpopulosos de obstáculos ou repleto de corredores que dificultem o trabalho do algoritmo.

# 5. Referencias

<https://homepages.dcc.ufmg.br/~chaimo/public/SBIA10.pdf>

https://github.com/brean/python-pathfinding