**UNIVERSIDADE DE SÃO PAULO**

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Departamento de Ciências de Computação

# 1º Trabalho Prático

## SCC0230-2019 – Inteligência Artificial

Integrantes:

Clara Rosa – 9021070

Oliver Becker – 10284890

Rafael Roque – 10295412

**São Carlos**

**2º Semestre 2019**

# Sumário

# Introdução

O trabalho possui como objetivo a execução de diferentes algoritmos para a busca de um possível caminho em um labirinto. Dentre estes algoritmos, foram implementados duas buscas cegas e duas buscas informada. Os algoritmos de busca cega que devem ser implementados são a Busca em Profundidade e a Busca em Largura. Por sua vez, os algoritmos de busca informada são a Busca Best-First-Search e Busca A*.

Ambas as buscas possuem vantagens e desvantagens. Para analisar qual método é o mais adequado serão testados diversos tipos e tamanhos de labirintos e serão armazenados os tempos que cada algoritmo gastou em sua execução. Ao final do trabalho será feita uma média desse tempo gasto para que seja possível concluir qual é o melhor método e qual algoritmo é mais eficiente para cada situação.

# Descrição das implementações

O programa foi desenvolvido em ambiente Linux e a linguagem de programação escolhida foi C++. Para a implementação dos algoritmos de busca foi o utilizado o paradigma de programação orientada à objetos. Deste modo, o programa contém classes que são correspondentes as buscas e ao labirinto e todas classes são instanciadas na função principal.

Para a representação do labirinto foi escolhido uma matriz de char. Todas as classes possuem construtores e destrutores que são responsáveis por alocar e desalocar dinamicamente essa estrutura, a fim de não desperdiçar espaço de memória. A classe “Labyrinth” é responsável por receber os dados dos labirintos e armazenar informações como posição de início e de saída, dimensões da matriz, além de fazer a leitura dos dados de entrada.

A classe “DepthSearch” é responsável por realizar a busca em profundidade. Na instanciação desta classe, é realizada a cópia das informações cruciais do labirinto para que não seja necessária consulta direta ao labirinto original e, consequentemente, conservando os dados para as demais buscas. Para a realização das buscas, foi utilizada um método recursivo onde os filhos eram visitados em sentido horário começando de cima e incrementando o valor do custo

até o ponto, caso seja encontrada a saída, as coordenadas de cada ponto do labirinto são armazenadas em um stack que será utilizada para imprimir o caminho encontrado. Após o término da execução, o tempo gasto é retornado para que seja calculada a média de tempo e o objeto é destruído.

A classe “BreadthSearch” é responsável por realizar a busca em largura. Assim como na busca anterior, os dados do labirinto são armazenados para uso futuro. Para essa busca foi utilizada uma fila na qual eram adicionados os filhos do ponto atual do labirinto. Para que pudessemos definir qual é o caminho até a saída, foi utilizada uma matriz de predecessores onde a cada iteração da busca era calculado o custo até o ponto e a qual o seu predecessor. Assim, ao término da execução da busca será utilizada uma stack para acumular o caminho e o tempo gasto na operação será retornado para o cálculo da média.

A classe “BestSearch” é responsável por realizar a busca Best-First-Search. Para essa busca foi utilizada as mesmas estruturas da classe “BreadthSearch”. No entanto há o acréscimo de uma função de heurística, a qual será explica com mais detalhes na próxima sessão, e a fila utilizada na “BreadthSearch” será substituída por uma fila de prioridades.

A classe “A_StarSearch” é responsavel por realizar a busca A*. Neste algorítmo, a ideia é ir explorando os próximos nós com base em uma função de custo, chamada f neste programa. Esta função f é a soma de dois parâmetros: g e h. Este é a heurística utilizada, que tenta prever o quão boa aquela posição é para se tornar uma candidata ao caminho real, tendo sido utilizada a distância euclidiana nesta implementação, que mostra a distância entre o ponto em questão e a posição do objetivo. Já aquela, é a distância já percorrida até chegar na posição atual, ou seja, a soma dos pesos de todas as posições antecessores que levaram àquele ponto. Para fazer tudo isto, utiliza-se uma matriz de Nodes, que indicam qual a posição imediatamente anterior à atual, além de armazenar os valores de f, g e h de seu próprio nó. Além disso, há também um set de SetPos, uma estrutura que armazena a posição de um nó e o valor de sua função f. Isto fica em um set para manter ordenado em relação à f, pois a próxima posição a ser acessada é aquela com o menor custo. Por fim, o set garante que não haverá repetições.

A criação dos labirintos foi feita com base em 4 parâmetros: dimensão das linhas, dimensão das colunas, quantidade de matrizes a serem geradas e a granularidade, que indica a

“dificuldade” do labirinto. Para a geração destes, foram sorteados números aleatórios que representam a quantidade de paredes que o labirinto possuirá. A partir desta quantidade serão gerados três números aleatórios que representam a posição inicial na matriz e a quantidades de casas que serão marcadas e a direção que seguirão. Com estes números, a matriz começará a ser marcada, caso a parede se encontrar com outra, ela mudará de direção aleatoriamente.

# Heurísticas usadas nos algoritmos de busca informada

## Busca Best-First-Search:

Para a busca Best-First-Search foi escolhida a heurística “Distância de Manhattan”, que consiste em calcular a distância em uma malha quadriculada entre duas coordenadas por meio da fórmula = $(x_2 - x_1 + y_2 - y_1)$.

Foi escolhida essa heurística pois é a que melhor se adequa ao problema do labirinto, visto que só é possível andar uma posição por vez e há possibilidades fixas de escolha, que no caso são os filhos do nó atual.

Para escolher o filho que possui a menor distância até a saída, é feito o cálculo da distância de Manhattan para todos os filhos do nó atual e adicionado em uma fila de prioridade. O filho com a menor distância se encontra no topo da lista, sendo apenas necessário torná-lo o nó atual e repetir o processo até que a saída seja encontrada.

## Busca A*:

Para a busca A* foi escolhida a heurística da “Distância Euclidiana”, que consiste em calcular a distância entre duas coordenadas por meio da fórmula = $\sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2)}$.

Foi escolhida essa heurística pois esta considera as movimentações diagonais, já que considera a reta com a menor distância entre os dois pontos, podendo esta inclinação assumir qualquer valor e não apenas múltiplos de 90°.

# Tempo despendido pelos algoritmos

Para realizar os testes e calcular a média do tempo dos algoritmos foram utilizadas as seguintes quantidades e tamanhos de labirinto:

- 20 labirintos 10x10
- 10 labirintos 30x30
- 10 labirintos 50x50
- 10 labirintos 100x100
- 10 labirintos 500x500

Para gerar os labirintos que foram utilizados no teste foi desenvolvido um código de geração aleatória de labirinto, que possui como parâmetros de entrada: as dimensões do labirinto, número de labirintos e granularidade. Sendo possível gerar diversos labirintos com diversas dificuldades, como foi descrito na seção acima.

## Resultados obtidos em segundos:

**Teste 10 labirintos 10x10 simples:**

Media Busca em Profundidade: 0.000080

Media Busca em Largura: 0.000089

Media Best First Search: 0.000094

Media Busca A*: 0.000063

**Teste 10 labirintos 10x10 complexos:**

Media Busca em Profundidade: 0.000087

Media Busca em Largura: 0.000091

Media Best First Search: 0.000100

Media Busca A*: 0.000053

**Teste 10 labirintos 30x30:**

Media Busca em Profundidade: 0.000397

Media Busca em Largura: 0.000389

Media Best First Search: 0.000429

Media Busca A*: 0.000491

**Teste 10 labirintos 50x50:**

Media Busca em Profundidade: 0.000992

Media Busca em Largura: 0.000961

Media Best First Search: 0.001258

Media Busca A*: 0.001095

**Teste 10 labirintos 100x100:**

Media Busca em Profundidade: 0.001545

Media Busca em Largura: 0.001638

Media Best First Search: 0.003277

Media Busca A*: 0.004517

**Teste 20 labirintos 500x500:**

Media Busca em Profundidade: 0.017233

Media Busca em Largura: 0.020546

Media Best First Search: 0.037246

Media Busca A*: 0.052532

# Resultados

Como visto a partir dos tempos médios de execução, a busca A* possui o melhor desempenho quando as matrizes são pequenas, uma vez que ela visita um pequena quantidade de pontos para determinar o caminho de maneira mais rápida e com um custo baixo. A busca em profundidade e a Best-first, como possuem características parecidas, possuem o desempenho mediano de tempo e ótimo desempenho de custo, sendo as que trazem os melhores resultados nas matrizes pequenas. A busca em profundidade, apesar de ser a mais simples de implementar, possui tempo alto de execução já que visita uma quantidade muito grande de casas para conseguir determinar um caminho, além de retornar um custo altíssimo.

Quando se trata de labirintos muito grandes, as buscas heurísticas tendem a demorar mais para achar um caminho, pois precisam realizar uma grande quantidade de cálculos de ponto flutuante para determinar o próximo passo. Como as buscas às cegas não realizam estes cálculos, possuem um desempenho em tempo muito maior, principalmente a busca em profundidade, uma vez que esta não faz acessos a estruturas auxiliares.

# Guia para compilação e execução do código

Os seguintes arquivos estão disponibilizados no arquivo zipado:

- search.cpp
- gerador_de_labirintos.cpp
- makefile
- 1.in, 2.in, 3.in, 4.in, 5.in, 6.in, 7.in, 8.in
- saida1.out, saida2.out, saida3.out, saida4.out, saida5.out, saida6.out, saida7.out, saida8.out

Para compilar o programa é preciso estar em ambiente Linux e possuir o compilador g++ instalado na máquina. Feito isso, descompacte o arquivo zipado e vá para seu diretório por meio do terminal e execute os seguinte comandos:

make

make run[1-8]

O programa geradorLabirinto.cpp foi utilizado para gerar os labirintos que estão nos arquivos de entrada .in, não sendo necessário executá-lo para a execução do programa. No entanto, caso queira executá-lo é preciso executar apenas o primeiro passo descrito acima, substituindo o nome do programa por geradorLabirinto.cpp. E inserir os valores X Y N G para que os labirintos sejam gerados, sendo X e Y a dimensão do labirinto, N a quantidade e G a granularidade, ou seja, a quantidade máxima de paredes que o labirinto possuirá.