Trabalho Prático 1 - Busca em Labirinto

SCC0230 - Inteligência Artificial

Marcelo Duchêne, 8596351
Matheus Carvalho Raimundo, 10369014
Michelle Wingter da Silva, 10783243

Orientadora: Prof. Alneu de Andrade Lopes

USP - São Carlos

Outubro/2020

Índice Analítico

1. Introdução Teórica

O objetivo desse documento é descrever o projeto que foi implementado como trabalho prático 1 da disciplina de Inteligência artificial. O projeto implementado coloca em prática diversos algoritmos de busca em um labirinto.

O labirinto consiste em uma matriz de pontos L de dimensão $m \times n$. Cada ponto dessa matriz pode adotar um dos seguintes caracteres como valor:

- '-': indica um ponto de parede, no qual não pode haver locomoção na busca;
- '*': indica um ponto de caminho, no qual pode haver locomoção na busca;
- '#': indica um ponto de entrada, no qual a busca se inicia;
- '$': indica um ponto de saída, ao qual é o destino da busca.

A busca, então, começa no ponto de valor '#' e, percorrendo apenas os pontos de valor '-', tenta encontrar o ponto final de valor '$'. A entrada do programa consiste de uma linha contendo dois inteiros M e N (a dimensão da matriz L), seguida de M linhas contendo N caracteres dos descritos acima.

Cada algoritmo que realiza essa busca pode adotar diferentes estratégias ou caminhos para tentar chegar ao destino. Dentre todos os algoritmos existentes, foram implementados os seguintes para este projeto:

- Busca em Profundidade (DFS);
- Busca em Largura (BFS);
- Busca Best-First;
- Busca A*;
- Busca Hill Climbing.

Alguns destes algoritmos realizam busca cega - DFS e BFS. Outros realizam busca informada, e para estes em específico, são necessárias funções (h, g e/ou f) que determinam o próximo passo do algoritmo. Uma dessas funções é a Distância de Manhattan em relação ao ponto de destino. A Distância de Manhattan foi optada porque é a mais recomendada para labirintos em que só é possível se movimentar em quatro direções, que é o caso deste projeto.

Neste relatório vamos descrever profundamente e fazer uma análise do funcionamento de cada um desses algoritmos para busca em labirinto.

2. Detalhes de implementação

O projeto consiste na implementação de cinco algoritmos de busca, dois de busca cega e três de busca informada e, para tal, as tecnologias utilizadas foram:

- Linguagem de programação **Python 3**;
- Bibliotecas de linguagem: numpy (manipulação de matrizes), heapq (fila de prioridades).
- Sistema de versionamento Git.

Algumas características de implementação podem influenciar os resultados de execução em complexidade de tempo, como o fato de que Python não é uma linguagem de programação compilada para código de máquina nativo, ou fatores como outros programas em execução na máquina. Contudo, para fins deste relatório, tais foram desconsiderados pelo motivo de suas magnitudes não afetarem a conclusão final, sendo portanto considerados insignificantes.

Por fim, também foi implementado um programa que gera labirintos, que será usado para gerar alguns casos de teste para nosso projeto. Para gerar labirintos, tal programa se baseou no conceito de peso aleatório em arestas a fim de obter a árvore geradora mínima (Algoritmo de Kruskal).

3. Guia de Execução

Para ser executado, é necessário ter Python 3 instalado em sua máquina. Para instalar, faça download da última versão de Python 3 (disponível em https://www.python.org/downloads/) e instale-o. Feito isto, o trabalho pode ser executado a partir do comando ***python3 trab1.py < labirinto.txt*** em seu terminal. Ao executar, será imprimido na tela os caminhos encontrados utilizando cada um dos algoritmos. O arquivo de entrada (no exemplo, *labirinto.txt*) pode ser alterado para qualquer um dos disponíveis - desde que esteja no mesmo formato especificado.

4. Algoritmos

Nesta seção será detalhado o funcionamento de cada um dos algoritmos implementados no projeto.

4.1 Busca em Profundidade (DFS)

O DFS consiste em percorrer a árvore em profundidade. Mais especificamente, o algoritmo dá maior prioridade sempre aos vértices encontrados por último. No caso do labirinto, isso significa que o algoritmo sempre vai seguir em frente, e sempre que encontrar um novo vértice, vai preferir seguir por esse novo vértice encontrado do que seguir pelos vértices que encontrou anteriormente.

Em termos de estrutura de dados, estamos falando aqui de uma recursão ou de uma pilha (o último a entrar é o primeiro a sair). Se existe um caminho possível até o destino, é garantido que o algoritmo de DFS vai encontrar. Não é garantido, porém, que este seja o melhor caminho (ou seja, aquele em que são percorridos menos pontos). Nas seções abaixo isso será discutido mais profundamente.

Abaixo, segue pseudocódigo **simplificado** do algoritmo de DFS:

- alocar pilha P
- incluir PontoInicial (#) em P
- enquanto P não está vazio:
    - V = retirar elemento de P
    - se V for Destino ($):
        - encerrar porque encontrou caminho
    - para cada vértice A adjacente de V:
        - se A não foi visitado, incluir A em P
- encerrar porque não encontrou caminho

O código-fonte de nosso programa principal pode ser acessado para detalhamento de implementação do algoritmo.

4.1 Busca em Largura (BFS)

O BFS consiste em percorrer a árvore em largura. Mais especificamente, o algoritmo dá maior prioridade sempre aos vértices encontrados primeiro. No caso do labirinto, isso significa que o algoritmo nunca vai seguir imediatamente em frente, ou seja, sempre que encontrar um novo vértice, vai preferir seguir pelos vértices que ele já encontrou anteriormente do que por esse novo vértice encontrado agora.

Em termos de estrutura de dados, estamos falando aqui de uma fila (o primeiro a entrar é o primeiro a sair). Se existe um caminho possível até o destino, é garantido que o algoritmo de BFS vai encontrar. E mais ainda: é garantido que este seja o melhor caminho (ou seja, aquele em que são percorridos menos pontos), pois não há diferença de pesos entre as arestas de um labirinto. Nas seções abaixo isso será discutido mais profundamente.

Abaixo, segue pseudocódigo **simplificado** do algoritmo de BFS:

- alocar fila F
- incluir PontoInicial (#) em F
- enquanto F não está vazio:
    - V = retirar elemento de F
    - se V for Destino ($):
        - encerrar porque encontrou caminho
    - para cada vértice A adjacente de V:
        - se A não foi visitado, incluir A em F
- encerrar porque não encontrou caminho

O código-fonte de nosso programa principal pode ser acessado para detalhamento de implementação do algoritmo.

4.1 Busca Best-First

A Busca Best-First consiste em percorrer a árvore usando uma heurística. Mais especificamente, o algoritmo dá maior prioridade sempre aos vértices cuja função *h(v)* calculada no vértice em questão assume menores valores. No caso do labirinto, essa função *h(v)* será a função de Distância de Manhattan em relação ao ponto de destino. Por exemplo, dado o vértice V e o ponto de destino é o D, a equação seria:

$$h(V) = |V_x - D_x| + |V_y - D_y|$$

Em termos de estrutura de dados, estamos falando aqui de uma fila de prioridades ou min-heap (o primeiro a sair é o de menor valor numérico de prioridade). Se existe um caminho possível até o destino, é garantido que o algoritmo Best-First vai encontrar. Não é garantido, porém, que este seja o melhor caminho (ou seja, aquele em que são percorridos menos pontos), apesar de que o algoritmo usa uma heurística que tenta encontrar o melhor caminho. Nas seções abaixo isso será discutido mais profundamente.

Abaixo, segue pseudocódigo **simplificado** do algoritmo do algoritmo Best-First:

- alocar fila de prioridades F
- incluir PontoInicial (#) em F com prioridade 0
- enquanto F não está vazio:
    - V = retirar elemento de F
    - se V for Destino ($):
        - encerrar porque encontrou caminho
    - para cada vértice A adjacente de V:
        - se A não foi visitado, incluir A em F com prioridade h(A)
- encerrar porque não encontrou caminho

O código-fonte de nosso programa principal pode ser acessado para detalhamento de implementação do algoritmo.

4.1 Busca A*

A Busca A* consiste em percorrer a árvore usando uma heurística, sendo muito semelhante ao algoritmo Best-First. Mais especificamente, o A* dá maior prioridade sempre aos vértices cuja soma da função *h(v)* com a função *g(v)* - calculadas no vértice em questão - assume menor valor. No caso do labirinto, a função *g(x)* representaria simplesmente quantos passos foram dados no caminho até o momento - um inteiro que representa por quantas arestas passou. Já a função *h(v)* será a função de Distância de Manhattan em relação ao ponto de destino. Por exemplo, dado o vértice V e o ponto de destino é o D, a equação de *h(V)* seria:

$$h(V) = |V_x - D_x| + |V_y - D_y|$$

Em termos de estrutura de dados, estamos falando aqui de uma fila de prioridades ou min-heap (o primeiro a sair é o de menor valor numérico de prioridade). Se existe um caminho possível até o destino, é garantido que o algoritmo A* vai encontrar. Não é garantido, porém, que este seja o melhor caminho (ou seja, aquele em que são percorridos menos pontos), apesar de que o algoritmo usa uma heurística que tenta encontrar o melhor caminho. Nas seções abaixo isso será discutido mais profundamente.

Abaixo, segue pseudocódigo **simplificado** do algoritmo do algoritmo A*:

- alocar fila de prioridades F
- incluir PontoInicial (#) em F com prioridade 0
- enquanto F não está vazio:
    - V = retirar elemento de F
    - se V for Destino ($):
        - encerrar porque encontrou caminho
    - para cada vértice A adjacente de V:
        - se A não foi visitado, incluir A em F com prioridade h(A)+g(A)
        - se A foi visitado com prioridade de maior valor numérico, atualizar prioridade em F
- encerrar porque não encontrou caminho

Percebe-se claramente uma grande semelhança com o algoritmo Best-First, incluindo a função *g(A)*. O código-fonte de nosso programa principal pode ser acessado para detalhamento de implementação do algoritmo.

4.1 Busca Hill Climbing

A Busca Hill Climbing consiste em percorrer a árvore usando uma heurística, sendo o mais diferente de todos os algoritmos implementados no projeto. Mais especificamente, o Hill Climbing percorre o grafo apenas uma única vez utilizando uma função *h(x)* como guia e desistindo a partir do momento que acredita entrar em um caminho que não levará à solução. No caso do labirinto, a função *h(v)* será a função de Distância de Manhattan em relação ao ponto de destino. Por exemplo, dado o vértice V e o ponto de destino é o D, a equação de *h(V)* seria:

$$h(V) = |V_x - D_x| + |V_y - D_y|$$

Em termos de estrutura de dados, não é usada nenhuma especial (é um algoritmo iterativo que utiliza apenas variáveis da linguagem). Se existe um caminho possível até o destino, não é garantido que o algoritmo Hill Climbing vai encontrar, pois ele tenta uma única heurística e desiste se falhar. Contudo, se o algoritmo não falhar e conseguir encontrar efetivamente uma solução, é garantido que a solução encontrada é o melhor caminho até o destino - no caso específico de busca em labirintos. Nas seções abaixo isso será discutido mais profundamente.

Abaixo, segue pseudocódigo **simplificado** do algoritmo do algoritmo Hill Climbing:

- V <- PontoInicial (#)
- faça em repetição:
    - se V for Destino ($):
        - encerrar porque encontrou caminho
    - se V não possui vértices adjacentes:
        - encerrar porque não encontrou caminho
    - A <- adjacente de V cujo valor h(A) seja menor
    - se h(A) > h(V):
        - encerrar porque não encontrou caminho
    - V <- A

Percebe-se claramente a diferença dos outros algoritmos, principalmente pelo fato de que sequer é necessária uma estrutura de dados complexa. O código-fonte de nosso programa principal pode ser acessado para detalhamento de implementação do algoritmo.

5. Casos de Teste

Será feita uma análise comparativa de diferentes casos de teste para os algoritmos implementados. Alguns casos de teste serão usados para testar a **consistência**, ou seja, a habilidade de cada um dos algoritmos de encontrar o **melhor caminho**. Para esta análise, foram escolhidos labirintos menores, mas que possuíssem diversos caminhos disponíveis para se chegar ao destino. Dessa maneira, apenas um desses caminhos é considerado o **melhor** caminho. Estes são os seguintes casos de teste:

- "pac-man": Fornecido pelo professor, tal caso de teste relembra o mapa do jogo do ano de 1980;
- "labirinto": Labirinto 49x73 com diversas possibilidades de caminhos;
- "labirinto2": Labirinto 33x55 com diversas possibilidades de caminhos;
- "campo-vazio": Labirinto sem paredes (totalmente livre).

Outros casos de teste serão usados para testar a **eficiência**, ou seja, a habilidade de encontrar algum caminho no **menor tempo** possível. Como descrito nas seções anteriores, alguns fatores serão desconsiderados do cálculo do tempo de execução, como a linguagem de programação em uso e o sistema operacional. Chamaremos estes fatores desconsiderados de **erro**. Desta maneira, casos de teste pequenos não servem para este propósito de cálculo do tempo de execução, pois muitas vezes o erro encontrado no cálculo do tempo é muito grande em relação à magnitude do real tempo de execução. Portanto, os casos de teste efetivamente usados são maiores, exigindo obviamente maior tempo para execução, porém reduzindo o erro final. Por fim, os casos de teste usados possuem uma única solução, pois o objetivo não é encontrar o melhor caminho, mas sim encontrar algum caminho rapidamente. Estes são:

- "labirinto1000": Tal caso de teste consiste de um labirinto 1000x1000;
- "labirinto5000": Tal caso de teste consiste de um labirinto 5000x5000;
- "labirinto10000": Tal caso de teste consiste de um labirinto 10000x10000.

6. Resultados

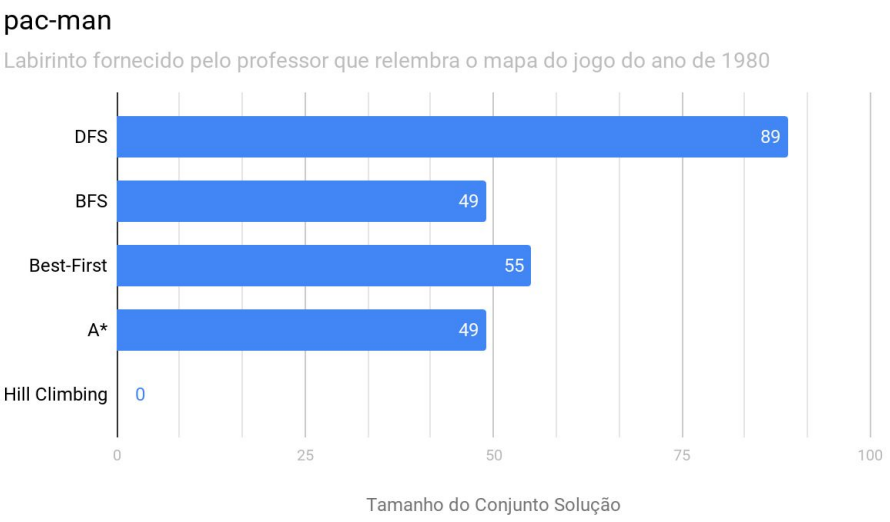
Para cada um dos casos de teste descritos na seção acima, vamos demonstrar os resultados em gráficos.

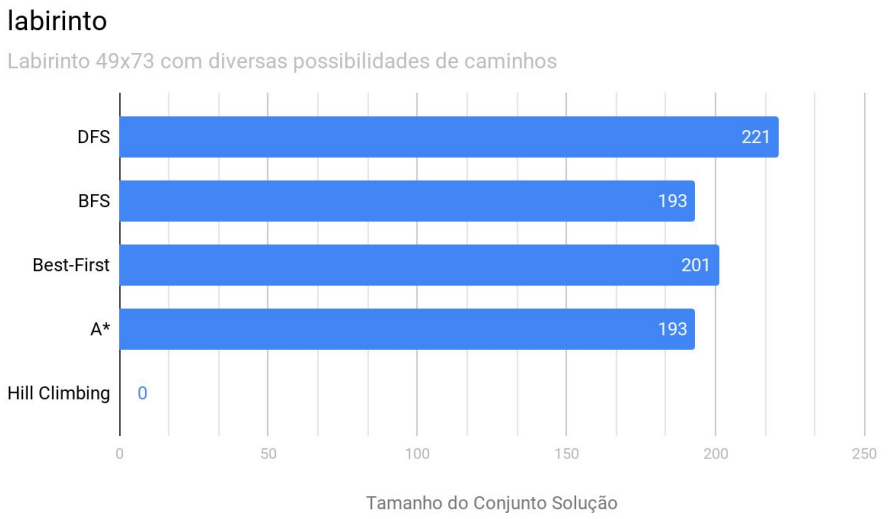
Nos casos de teste de busca de **melhor caminho**, o gráfico será “algoritmo” no eixo Y e “tamanho do caminho” no eixo X. Ou seja, quanto menor o valor, melhor, porque menor será o caminho percorrido até a solução.

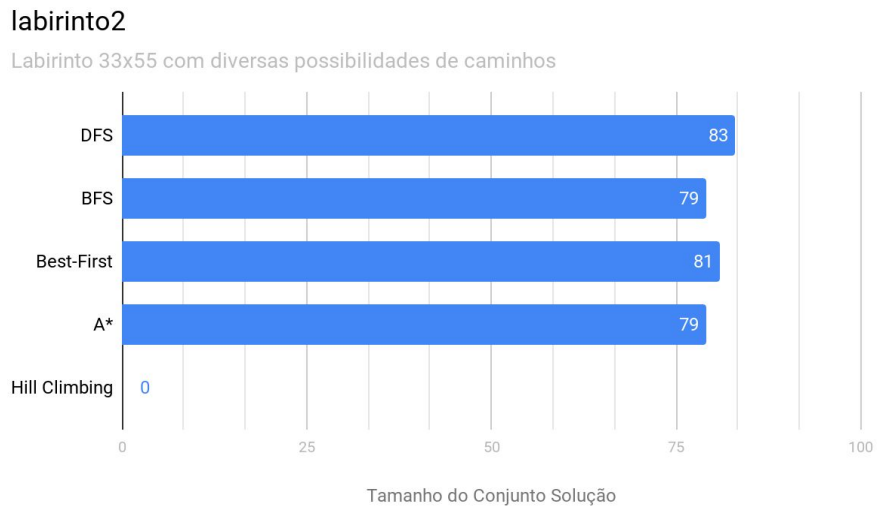
Já nos casos de teste de busca em **menor tempo**, o gráfico será “algoritmo” no eixo Y e “tempo de execução” no eixo X. Similarmente, quanto menor o valor, melhor, porque menor será o tempo de execução do algoritmo. Deve-se considerar aqui, porém, se um caminho ao menos é encontrado - a ser discutido posteriormente nas conclusões.

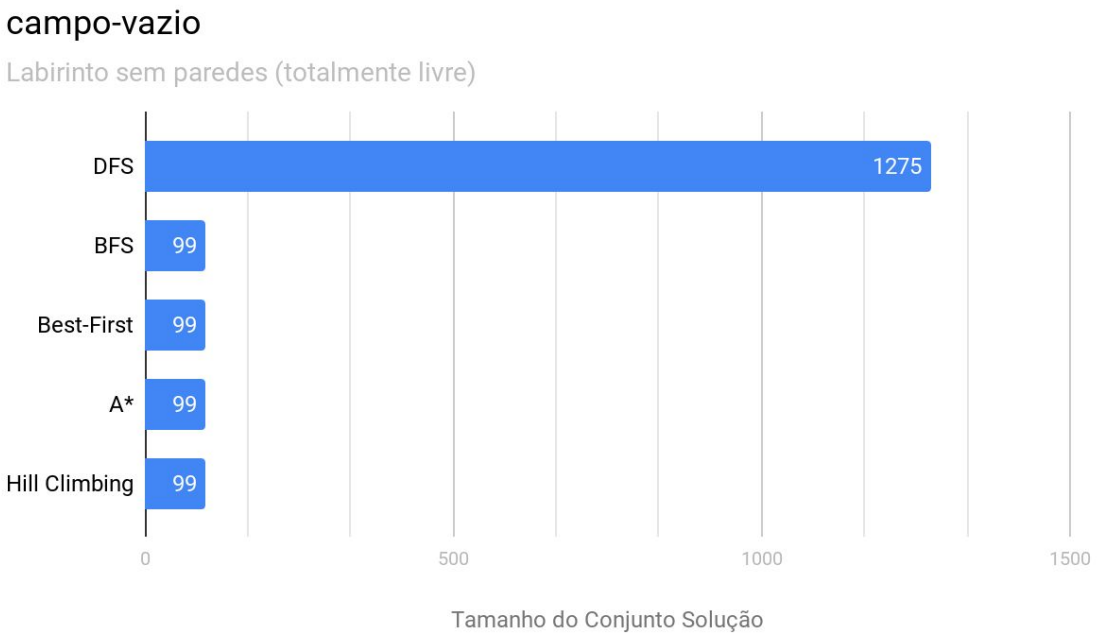
A variedade dos labirintos possibilita gerar conclusões após análise dos resultados.

6.1 Busca do Melhor Caminho



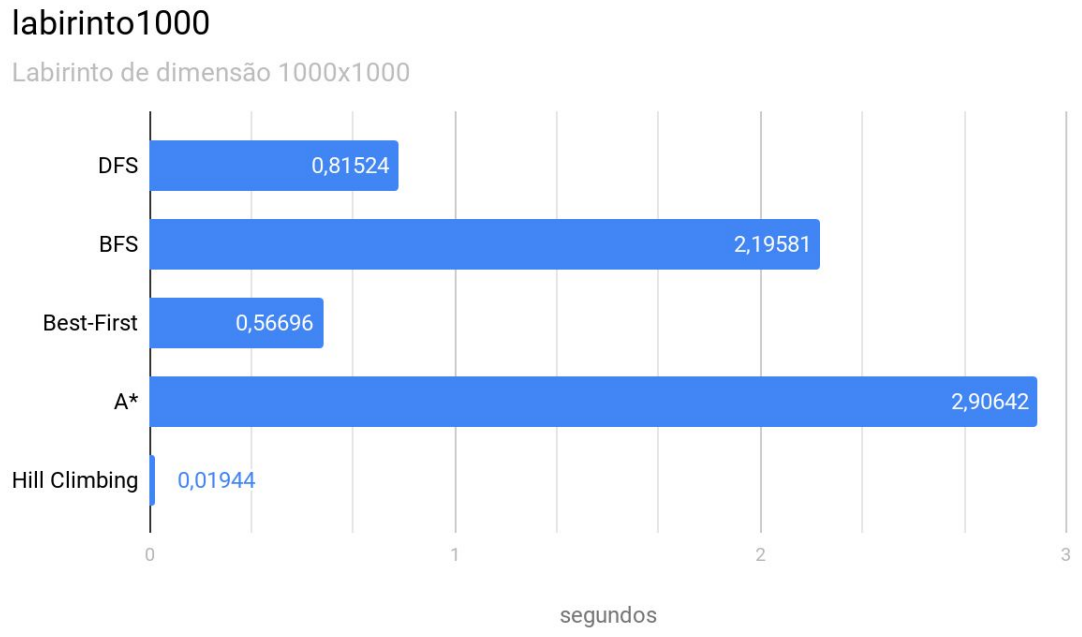






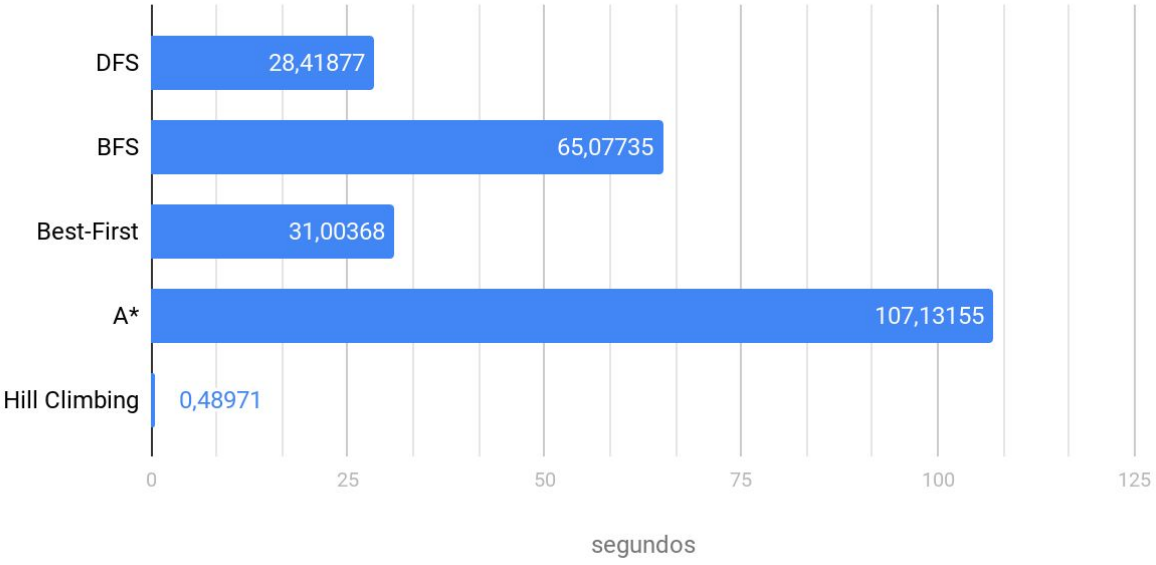
6.1 Busca em Menor Tempo

Atenção: tome cuidado ao executar estes casos de teste! Há alto consumo de memória RAM, o que pode provocar uso de armazenamento secundário (*swapping*) em máquinas com baixa quantidade de memória RAM disponível. Para fins do projeto, foram utilizadas máquinas com memória RAM suficiente (16 GB).



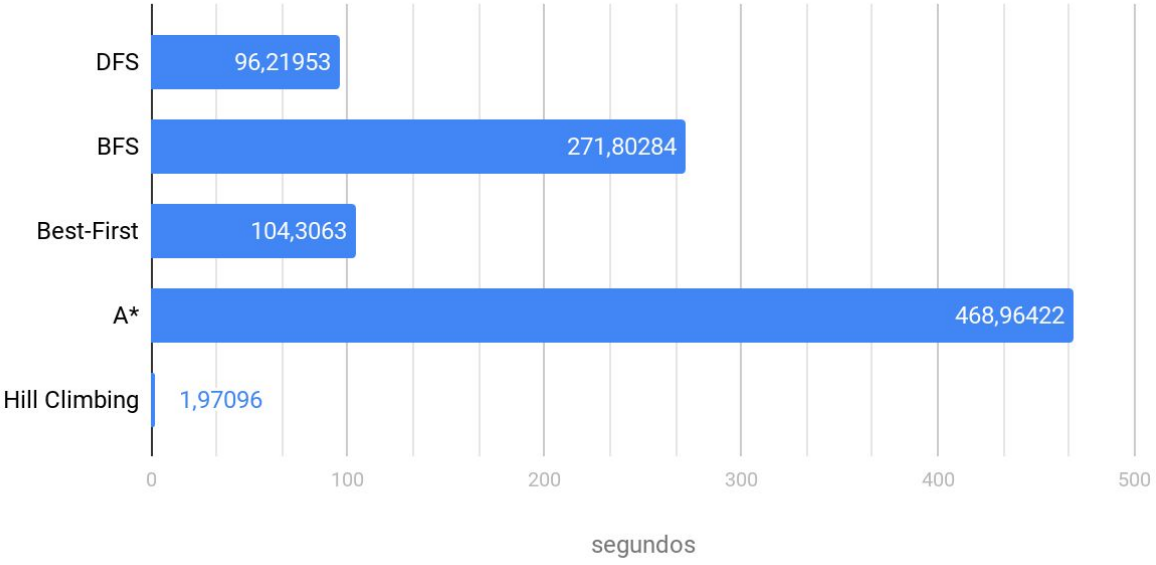
labirinto5000

Labirinto de dimensão 5000x5000



labirinto10000

Labirinto de dimensão 10000x10000



7. Conclusões

Analisando os resultados de cada um dos algoritmos, podemos concluir o seguinte sobre a busca pelo **melhor caminho**:

- Na **Busca em Profundidade (DFS)**, o caminho encontrado na grande maioria dos casos de teste foi o pior (o mais longo). Isso ocorreu porque o DFS não utiliza nenhum tipo de heurística, e ele prioriza sempre o caminho dado pelas arestas adjacentes, o que pode levar ele muitas vezes a seguir por um longo caminho até chegar no destino. Há um destaque maior para o caso de teste "campo-vazio", em que o algoritmo encontrou um caminho absurdamente péssimo.
- Na **Busca em Largura (BFS)**, o caminho encontrado em todos os casos de teste foi o melhor (o mais curto). Isso já é esperado do BFS, pois o algoritmo leva em consideração sempre as arestas descobertas inicialmente, o que faz que ao encontrar uma solução, esta seja automaticamente a menor possível. Em contrapartida, o algoritmo pode acabar executando em um tempo maior, pois muitas dessas arestas analisadas pelo BFS não levam a uma solução, mas acabam sendo analisadas de qualquer maneira.
- Na **Busca Best-First**, há um meio-termo. O algoritmo apresentou um desempenho muito bom porque encontrou caminhos menores que os caminhos encontrados pelo DFS. Mas ainda assim, os caminhos que o Best-First encontrou não são os melhores.
- Na **Busca A***, assim como no BFS, os resultados foram positivos. Em todos os casos de teste de busca por melhor caminho ele conseguiu encontrar o melhor caminho. A diferença do BFS é que ele fez isso mais rapidamente, o que dá uma vantagem para ele.
- Na **Busca Hill Climbing**, o algoritmo apresentou o pior desempenho. Em quase todos os casos de teste ele sequer encontrou um caminho - nem que seja um caminho péssimo. Todavia, percebe-se que o algoritmo é extremamente rápido, e que quando encontra algum caminho ele consegue ser mais rápido que os outros em uma expressiva magnitude.

Sendo assim, podemos classificar os algoritmos no quesito busca pelo melhor caminho da seguinte maneira:

1. Busca em Largura (BFS);
2. Busca A*;
3. Busca Best-First;
4. Busca em Profundidade (DFS);
5. Busca Hill Climbing.

Em **relação ao tempo de execução**, percebe-se que os algoritmos são classificados na seguinte ordem de eficiência, desconsiderando o tamanho do caminho encontrado:

1. Busca em Profundidade (DFS);
2. Busca Best-First;
3. Busca em Largura (BFS);
4. Busca A*.

O algoritmo de busca Hill Climbing foi **desclassificado** porque em nenhum dos casos de teste foi capaz de encontrar um caminho como conjunto-solução. Contudo, ele com certeza é o mais rápido de todos, e pode funcionar muito bem, mas apenas para casos muito específicos - que não estão no escopo deste projeto. A conclusão final é de que o algoritmo tem um tempo de execução tão baixo, que ele pode ser usado como um "*overhead*" para qualquer programa de busca em labirinto: tenta o Hill Climbing primeiro, e só transfere o labirinto para outro algoritmo se ele não encontrar uma solução.

Apesar de que no primeiro gráfico observamos o Best-First mais rápido que o DFS, a complexidade de tempo deve ser considerada no infinito, e quanto maior possível for a dimensão da matriz. Nossa infraestrutura de testes nos limitou a uma matriz de 10000x10000 (não temos memória RAM o suficiente). Observa-se então, dado o que temos, que no infinito e considerando o pior cenário, o Best-First tem comportamento ligeiramente inferior ao DFS. Mas note que isso não quer dizer que o Best-First é pior que o DFS. Na verdade o Best-First utiliza de heurísticas que visam encontrar o melhor caminho quando possível, sem aumentar o grau do tempo de processamento. Sendo assim, na maioria dos cenários da vida real, é muito mais válido aplicar o Best-First que o DFS, visto que os pequenos segundos de diferença não justificam o fato de que o DFS encontra o pior caminho enquanto que o Best-First encontra um caminho melhor.

Além disso, deve-se atentar ao algoritmo A*. Apesar de que ficou em último lugar, o A* funciona melhor na grande maioria dos tipos de labirintos existentes em cenários reais. Acontece que o A* utiliza uma **heurística**, e heurísticas nunca são soluções gerais. Todos os

labirintos utilizados como caso de teste de tempo de execução foram gerados aleatoriamente, porém utilizando o mesmo método: *Kruskal Minimum Spanning Tree* - este é o método mais eficiente e comumente utilizado para geração de labirintos. Sendo assim, não pode-se concluir que o A* seja ineficiente em todos os labirintos, mas apenas para labirintos gerados usando o algoritmo de Kruskal.

Como conclusão final com base em toda nossa análise acima, temos:

- **Busca em Profundidade (DFS)**: em geral rápido, mas acha caminhos péssimos.
- **Busca Best-First**: um pouco mais lento, mas acha caminhos mais regulares.
- **Busca A*:** um pouco mais lento ainda, mas acha caminhos muito próximos do melhor, ou o melhor.
- **Busca em Largura (BFS)**: lento, mas acha o melhor caminho.
- **Busca Hill Climbing**: absurdamente rápido, mas quase nunca acha um caminho; pode ser usado em conjunto com os outros algoritmos.

8. Referências Bibliográficas

- Conteúdo e anotações de sala de aula foram utilizados no desenvolvimento deste projeto.

- Website “Iterative Depth First Traversal of Graph” <https://www.geeksforgeeks.org/iterative-depth-first-traversal/>. Disponível em 14 de outubro de 2020.

- Website “Breadth First Search or BFS for a Graph” <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>. Disponível em 14 de outubro de 2020.

- Website “Wikipedia - Best-first search” <https://en.wikipedia.org/wiki/Best-first_search>. Disponível em 14 de outubro de 2020.

- Website “Wikipedia - Best-first Search” <https://pt.wikipedia.org/wiki/Best-first_Search>. Disponível em 14 de outubro de 2020.

- Website “Wikipedia - A* search algorithm” <https://en.wikipedia.org/wiki/A*_search_algorithm>. Disponível em 14 de outubro de 2020.

- Website “A* Search Algorithm” <https://www.geeksforgeeks.org/a-search-algorithm/>. Disponível em 14 de outubro de 2020.

- Website “Wikipedia - Hill climbing” <https://en.wikipedia.org/wiki/Hill_climbing>. Disponível em 14 de outubro de 2020.

- Website “Wikipedia - Taxicab geometry” <https://en.wikipedia.org/wiki/Taxicab_geometry>. Disponível em 14 de outubro de 2020.