Trabalho Prático 2 – Escapando da floresta da neblina

Daniel Augusto Castro de Paula – 2022060550

Universidade Federal de Minas Gerais - UFMG

1. Introdução

O trabalho proposto consiste em determinar se o heroi lendário Linque irá conseguir escapar da floresta da neblina de acordo com a quantidade de energia e portais máximos que ele possui. Para cumprir esse requisito iremos modelar a floresta como um grafo direcionado no qual os pesos entre os vértices são calculados pela distância euclidiana de acordo com as coordenadas dos vértices, considerando também que os portais possuem peso 0. Para a representação do grafo utilizaremos a representação por lista de adjacência e matriz de adjacência.

Para determinar se será possível escapar ou não, implementamos dois algoritmos, o dijkstra e o a-estrela. Ambos os algoritmos buscam o menor caminho considerando que sempre iremos partir do vértice 0 ao n-1. Para isso, ambos utilizam uma estrutura de dados de minHeap que foi modelada com especificações para resolver o problema proposto. A principal diferença do a-estrela é que ele utiliza a distância euclidiana para o vértice final como heurística.

2. Método

O programa foi desenvolvido na linguagem C++ compilada pelo compilador G++ da GNU Compiler Collection.

Para modelar os grafos direcionados utilizamos dois tipos de representações, a lista de adjacência e a matriz de adjacência. Para os algoritmos de busca utilizou-se a implementação canônica deles como referência, foi necessário realizar algumas alterações na fila de prioridade utilizada por eles e adicionar verificações para controlar a quantidade de portais. Essas implementações serão abordadas com mais detalhes no tópico abaixo.

2.1 Classes, TADS e métodos

Grafo lista de adjacência: no arquivo listaAdjGrafo implementou-se a classe GraphList que é uma estrutura que representa o grafo por lista de adjacência, seguindo a ideia de implementação vista em sala de aula, utilizando uma lista de listas. Para auxiliar na sua implementação considerando o peso das arestas, criou-se a struct “aresta” que possui um valor que faz referência ao vértice, um valor que armazena o peso para chegar até ele e um ponteiro para a próxima aresta. Nessa classe temos métodos que criam arestas e portais, retornam a lista de adjacência e o número de vértices.

Grafo matriz de adjacência: no arquivo matrizAdjGrafo implementou-se a classe GraphMatriz que representa o grafo por uma matriz de tamanho n por n, na qual n é o número de vértices do grafo. Esse modelo também segue a implementação vista em sala de aula, a única diferença é que na matriz adicionamos o peso da distância que foi calculado das trilhas. Nessa classe temos métodos que criam arestas e portais, retornam a matriz de adjacência e o número de vértices.

MinHeap: no arquivo minHeap implementou-se uma estrutura de minHeap binário onde a estrutura ordenada é uma struct criada chamada “State” que possui a informação do

vértice, qual foi o custo para chegar até ele e quantos portais ainda restam para ser utilizado. Nesse caso a implementação também é bastante similar com o exemplo visto em sala, os principais atributos da classe são um vetor que armazena os states, o valor do tamanho do heap e o tamanho da capacidade. Nesse caso o tamanho serve para controlar até que ponto do vetor realmente temos dados e a capacidade serve para ajustar na alocação dinâmica do tamanho do vetor, se inserimos um novo elemento mas o tamanho for igual a capacidade aumentamos a capacidade desse vetor. Dessa forma, nessa classe temos métodos que fazem um resize na fila caso atinja a capacidade máxima, métodos para inserir um novo elemento, remover elemento, retornar o elemento do topo e funções auxiliares para percorrer o heap de forma que mantenha a ordenação dele da forma correta, sempre mantendo o mínimo no topo.

O arquivo “auxStructs” armazena o conjunto da struct “Coords” que armazena o valor da coordenada x e y de um vértice e da struct “Portal” que possui a informação de dois inteiros que representam os vértices u e v, no qual o portal sempre será direcionado do vértice u para v.

Para a execução final, juntando as estruturas mencionadas acima, temos dois arquivos, o tpLista e o tpMatriz que englobam os algoritmos de busca. Vale comentar também que temos a função distanciaEuclidiana que recebe duas structs de coords e retorna o peso da aresta e a função main que é responsável por gerenciar a entrada do programa da maneira que foi especificada no enunciado.

2.2 Algoritmos

Por fim, pode-se dizer que as funções de implementação do algoritmo de dijkstra e do a-estrela foram baseadas nos exemplos disponibilizados, as principais diferenças são em relação à fila de prioridade, já que nesse caso utilizou-se a estrutura do minHeap descrita previamente. A parte do “relaxamento” do algoritmo também teve verificações adicionais para adequar a lógica de portais, o principal objetivo é garantir que a busca prossiga de acordo com a quantidade de portais que o Linque pode utilizar.

3. Análise de complexidade

Para compreender a análise de complexidade do programa como um todo é importante destacar e analisar as principais estruturas utilizadas no programa.

Grafo lista de adjacência:

Para esse modelo em questão iremos evidenciar o custo das principais funções utilizadas no programa. Para criação do grafo inicializamos o array de listas de adjacência com v elementos no qual cada elemento do array é inicialmente apontado para um valor nulo possuindo então uma complexidade temporal O(V) onde v é o numero de vertices. Para destruir o grafo temos que percorrer totalmente todas as listas de adjacência desalocando todas as arestas, nesse caso a complexidade é O(V+E) no qual E é o número de arestas.

Em relação a outras operações como por exemplo a criação de uma aresta e adição dela, evidenciamos que o custo dessas operações é constante e portanto O(1) tanto para a análise temporal quanto espacial já que realiza sempre os mesmos procedimentos. Uma desvantagem que poderia existir nessa representação mas não é um problema para nosso programa pois não precisamos dessa função, seria a verificação se uma aresta existe que

nesse caso seria O(n) por ter que percorrer toda a lista de adjacência.

Em relação à complexidade espacial conseguimos obter ela pela soma do espaço ocupado pelo array de listas de adjacência e pelo espaço ocupado pelos nós das listas de adjacência ficando com a complexidade O(V+E). Isso é uma vantagem desse modelo se comparado com a matriz de adjacência, principalmente para grafos esparsos.

Grafo matriz de adjacência:

Para esse modelo, a complexidade temporal para inicialização do grafo é $O(V^2)$ pois precisamos inicializar toda a matriz. As operações implementadas como por exemplo adição de uma aresta, ou obtenção de uma aresta, da matriz de adjacência e do número de vértices são feitas com custos constantes e portanto apresentam complexidade assintótica O(1). Para consultar a vizinhança de um vértice a complexidade seria O(V) pois seria necessário percorrer toda a linha da matriz, porém como não precisamos dessa operação ela se torna irrelevante para nosso exemplo.

Em relação a complexidade espacial ela é sempre $O(V^2)$ evidenciando uma possível desvantagem desse modelo, já que para certos modelos de grafos grande parte da matriz irá armazenar uma informação desnecessária.

MinHeap:

Para o TAD fila de prioridade, optou-se por implementar um minHeap que basicamente representa uma árvore binária na qual os elementos de maior prioridade são removidos primeiro, no caso do minHeap quanto menor o valor de determinada condição maior a prioridade. Para resolução do problema em questão criamos a estrutura “State” que possui o valor do vértice, o custo para chegar nele e o número de portais restantes de acordo com o que foi utilizado até esse estado, a condição que verificamos nesse caso para determinar a prioridade é o custo.

Para construção do heap temos que fazer uma alocação inicial na qual possui um custo proporcional à capacidade do heap e portanto dizemos que a complexidade espacial é O(capacidade) e será feita em um tempo constante. Para inserção pelo método push adicionamos um elemento no final do array e depois realizamos o heapifyUp que pode ter que subir até a raiz para corrigir o heap e portanto ter que percorrer log(n) que seria o tamanho da árvore onde n é o tamanho da fila. Por esse motivo, concluímos que a complexidade temporal de inserção é O(log n). Importante destacar que se o heap estivesse cheio deveríamos chamar a função resize para redimensionar ele, essa função possui complexidade O(n) para copiar todos elementos para um novo array.

A remoção consiste em substituir a raiz pelo último elemento e realizar o heapifyDown que de maneira similar a inserção pode descer até a folha possuindo complexidade temporal O(log n).Além disso, é importante destacar que precisamos decrementar a variável de classe que controla o tamanho do heap. Essas operações possuem complexidade espacial O(1) pois não demandam espaço adicional. Outras operações como acesso ao menor elemento e verificar se o heap está vazio possuem complexidade constante O(1).

Dijkstra:

Para compreender a complexidade do algoritmo de dijkstra com as modificações necessárias para solucionar o problema do heroi lendário Linque temos que mencionar sobre as principais partes do código. A fim de simplificar a análise de complexidade desse método é importante ter o conhecimento das seguintes nomenclaturas:

V= número de vértices do grafo
E= número de arestas do grafo
P= número de portais permitidos

Para armazenar as distâncias mínimas em relação aos vértices dos grafos decidiu-se utilizar um vetor de valores do tipo double onde inicializamos essa estrutura com tamanho igual ao tamanho do número de vértices inserindo o maior valor possível em cada uma de suas posições. Essa inicialização tem complexidade temporal e espacial O(V).

O próximo procedimento relevante seria a criação da estrutura de fila de prioridade minHeap e inserção do primeiro elemento. Para procedimentos realizados no loop enquanto a fila de prioridade não está vazia, como por exemplo, inserir um elemento na fila (“push”) e remover o mínimo da fila (“pop”) pelas justificativas que já foram mencionadas anteriormente na análise do heap temos complexidade O(log V).

Iremos considerar que a fila de prioridades vai conter algo próximo a V elementos, já que apesar dela possuir a função de resize ,o número de portais permitidos P sempre vai ser relativamente menor que a quantidade de vértices e por isso não impacta significativamente a quantidade de elementos na fila. Isso significa que faremos um total de V operações de “pop” na fila, cada uma delas com complexidade O(log V) e portanto ficaremos com complexidade de O(V log V). No loop que percorremos cada aresta referente a um nó, no pior dos casos teremos que “relaxar” a quantidade E total de arestas do grafo, esse procedimento engloba a operação de “push” que também tem complexidade O(log V), como fazemos isso E vezes no máximo a complexidade fica O(E log V). Portanto, somando todos os custos abordados acima, a complexidade temporal final com a lista de adjacência seria algo próximo de:

$$O(V)+O(V \log V)+O(E \log V)$$

como O(V log V) e O(E log V) são dominantes podemos simplificar por O((V+E)log V).

Para complexidade de espaço podemos concluir que para esse modelo de implementação do dijkstra temos um custo adicional para o armazenamento das distâncias O(V).

Para o grafo com matriz de adjacência a análise é bem similar. Em relação a complexidade de tempo temos O(V) para inicializar o vetor de distâncias e complexidade O(log V) para operações de push e pop no minHeap, no qual também aproximamos para V a quantidade máxima de elementos. A principal diferença se dá na exploração dos vizinhos de cada nó, já que exploraremos todos vizinhos e como estamos utilizando a matriz de adjacência, para percorrer todos os nós isso resultará em $O(v^2)$ operações. Ao combinar todas essas complexidades obtemos a complexidade final que pode ser obtida por:
$O(V)+O((V+V^2)\log V)$ podemos simplificar a parte dominante ficando com $O(V^2 \log V)$.

Para complexidade espacial teremos a mesma análise do custo adicional linear já que precisamos do vetor de distâncias.

A-estrela:

A implementação deste algoritmo se deu de maneira muito similar a implementação do dijkstra, a principal diferença é que nesse caso temos que utilizar uma heurística, que para o problema em questão será a distância euclidiana até o vértice final, ao inserir o custo na fila de prioridade, influenciando assim na prioridade no minHeap. Optou-se por apenas chamar a função de calcular a distância euclidiana quando fosse inserir o novo elemento no heap, isso traz um custo adicional constante. Seria possível implementar um vetor para não

ter que ficar calculando a distância a cada vez que for inserir no heap, porém optou-se por não fazer isso considerando que o custo adicional temporal seria pequeno e esse vetor auxiliar poderia ocupar um grande espaço na memória caso o grafo tenha muitos vértices.

É importante destacar que para o algoritmo do a-estrela sua complexidade temporal no geral depende diretamente da qualidade da heurística utilizada, se ela for “perfeita” encontraremos o custo até o final rapidamente em um tempo que não irá variar muito. No pior caso, a implementação da forma canônica do a-estrela seria O(b^d) onde b é o número médio de vizinhos por nó e d seria a distância entre o vértice inicial e final.

4. Estratégias de robustez

No contexto do desenvolvimento do programa diversas escolhas foram feitas para tornar a implementação mais simples, eficiente e segura, evitando possíveis erros. Dentre elas, destaca-se a modularização das classes utilizadas no programa por meio da separação do header em arquivos hpp e da implementação no cpp, além dos comentários nas funções dessas classes.

Outra estratégia importante é a liberação de memória e para isso as estruturas utilizadas foram implementadas com seus respectivos destrutores evitando vazamentos de memória. Para ajudar nessa verificação o valgrind foi utilizado. Ainda em relação a memória, no minHeap foi implementado um mecanismo de resize para caso o tamanho chegue no máximo da capacidade não tenhamos problema com acesso indevido de posições da fila.

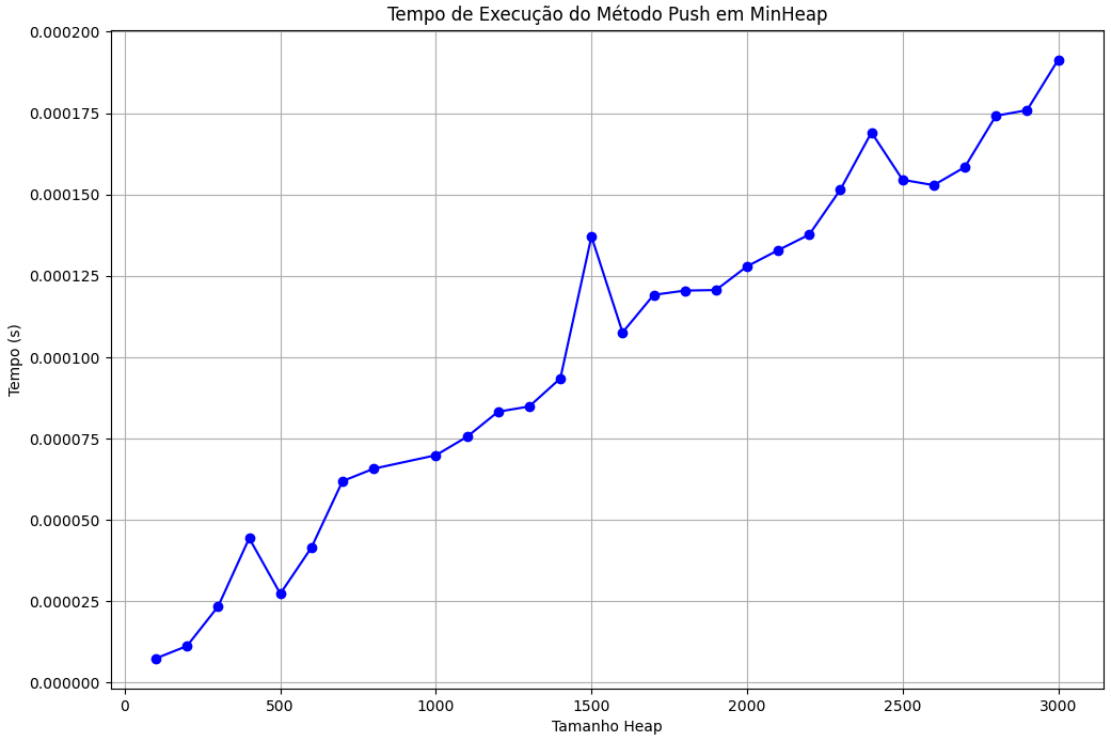
Por fim, algumas exceções foram implementadas no código para que caso ocorra algum acesso indevido na fila de prioridades o programa identifique e termine imediatamente a execução indicando o erro.

5. Análise experimental

Esta seção será dividida em subtópicos para analisar experimentalmente e aprofundar em diferentes tópicos que são relevantes para compreender o funcionamento da solução implementada.

Análise Heap

O minHeap binário implementado no programa influencia a complexidade dos algoritmos de busca principalmente por causa das operações de “push” e “pop” que são executadas diversas vezes. Para evidenciar a complexidade teórica deles que já foi mencionada, foi criado um caso de teste para medir o tempo dessas operações alterando o tamanho do heap. Nele variamos o tamanho do heap de 0 a 3000 de forma a permitir que seja possível dimensionar e correlacionar o aumento do tempo para inserção ou remoção de acordo com a quantidade de elementos na fila. Conforme a análise teórica é esperado que tenhamos um aumento temporal correlacionado com o logaritmo do tamanho desse minHeap.



A partir dele conseguimos perceber a tendência logarítmica da curva em relação ao tamanho do heap.

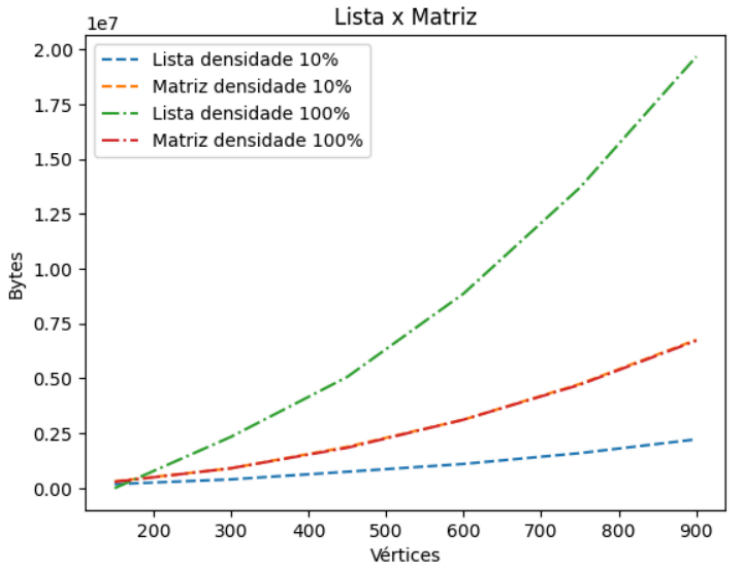
Comparação lista de adjacência x matriz de adjacência

Para modelar o grafo direcionado utilizamos a modelagem por lista de adjacência e matriz de adjacência. Como já abordado na análise teórica, a modelagem por matriz sempre vai ter um custo memória fixo quadrático enquanto na lista de vai ter um custo proporcional a quantidade de vértices mais as arestas desse grafo já que apenas armazenamos nas listas as estruturas que representam os vértices vizinhos de determinado nó.

Essa diferença nos permite concluir que o programa desenvolvido terá um resultado diferente referente a alocação de bytes para o mesmo grafo ao comparar os dois modelos. Para grafos de densidade bem pequena, ou seja aqueles possuem poucas arestas em relação a quantidade total para que todos os vértices tenham uma conexão entre si, espera-se que a lista de adjacência seja mais eficiente nesse sentido pois não irá alocar grande parte do espaço desnecessário da matriz. Por outro lado, para grafos com densidade próxima a cem por cento, espera-se que a matriz exija menos bytes alocados já que o custo para armazenar os objetos da struct criada para o problema vai ser maior considerando que as listas de adjacências formarão praticamente uma matriz de struct ao invés da matriz de double.

Outro impacto desses modelos que podemos verificar explicitamente no nosso programa é em relação a operação de percorrer todos os vizinhos. Os algoritmos de buscas usam essa operação para tentar identificar casos onde a distância até um vértice seria menor e portanto seria adicionado na fila de prioridade. De maneira similar ao raciocínio anterior, podemos concluir que para grafos pouco densos a matriz de adjacência irá ser menos efetiva porque teremos que percorrer toda a linha da matriz, já a lista de adjacência irá percorrer somente os vizinhos pois só armazenamos eles.

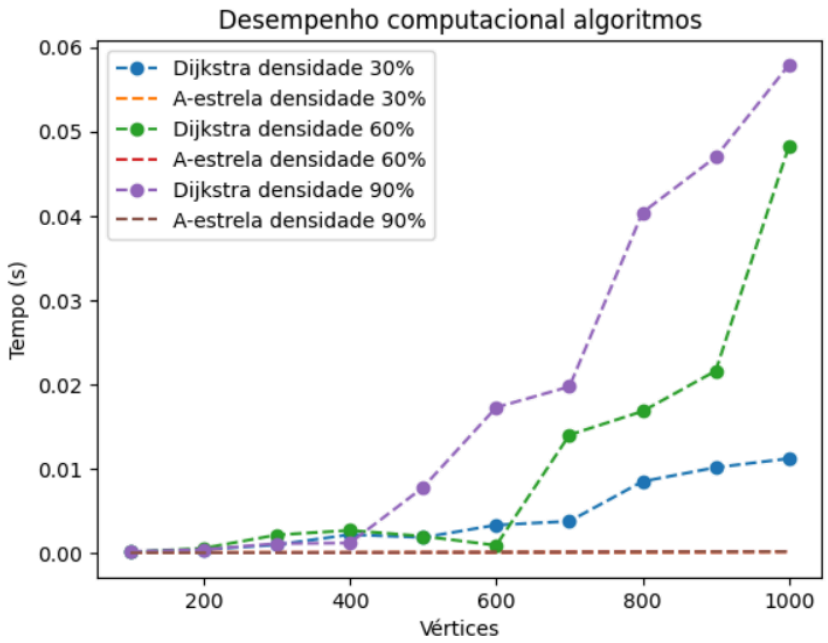
Foi feito um experimento utilizando o valgrind para identificar a quantidade de bytes alocados para grafos de diferentes tamanhos e densidades e comprovar as análises feitas. A partir dele podemos perceber que a alocação da matriz da não varia de acordo com a densidade e que para casos de grafos com densidade menor a lista se comprovou mais efetiva nesse quesito.



Desempenho dos algoritmos de busca

Para compreender o desempenho computacional acerca das implementações dos algoritmos de dijkstra e a-estrela é essencial relembrar a análise feita no tópico de análise de complexidade dessa documentação. De maneira resumida, espera-se que o dijkstra apresente um maior tempo de execução conforme o número de vértices e a densidade do grafo aumentem. O nosso a-estrela utiliza uma “boa” heurística que é a distância euclidiana até o ponto final,ou seja ele guia de maneira eficiente a busca até o vértice final.Dessa maneira, o comportamento esperado para determinados modelos de grafos seria um tempo de execução eficiente e com pouca variação.

Para testar essa análise geramos diferentes entradas de grafos que variam de 100 a 1000 vértices. Além dessa variação, para cada um desses grafos geramos 3 tipos de densidades diferentes, 30, 60 e 90 por cento. Essa variação foi feita para identificar o impacto da variação do número de arestas considerando que a quantidade de vértices é fixa, nesse caso quanto maior a densidade mais arestas temos. O teste foi feito com o grafo por lista de adjacência.



A partir desses resultados podemos comprovar nossa hipótese de que para determinados tipos de grafos e dependendo da heurística utilizada, o a-estrela pode convergir rapidamente para o ponto final do grafo e o dijkstra tem uma tendência de demorar mais para grafos com mais vértices e arestas.

Localidade de referência

Em relação às análises de localidade de referência precisamos destacar os conceitos da localidade temporal e espacial. A localidade temporal diz que se determinado item é acessado, é provável que ele seja acessado novamente em um período relativamente curto de tempo. A espacial diz que se um item é acessado, é provável que itens de memória próximos sejam acessados em breve. Essas características são importantes de serem levadas em consideração na implementação de algoritmos, pois podem reduzir o tempo de acesso aos dados.

Nesse sentido, para grafos muito densos a matriz de adjacência pode se mostrar mais eficiente em relação a localidade já que grande parte da matriz estará preenchida e consequentemente a maior parte será usada. Em grafos esparsos, pelo fato de possuir muitas posições vazias a localidade será afetada. Em contrapartida, a lista de adjacência pode ser mais eficiente em grafos esparsos pois apenas armazenaremos as arestas existentes e conseguiremos acessar todos os vizinhos rapidamente.

Para testar essas hipóteses utilizamos o módulo de cachegrind do valgrind para avaliar os parâmetros de “misses” no cache que englobam tanto a análise espacial quanto temporal. Utilizamos um grafo de 1000 vértices e variamos apenas a densidade. Assim, conseguimos comprovar as hipóteses feitas já que quanto menor o valor mais efetivo a localidade, ou seja, para densidade 10% a lista foi mais efetiva e para densidade 100% a matriz foi mais efetiva.

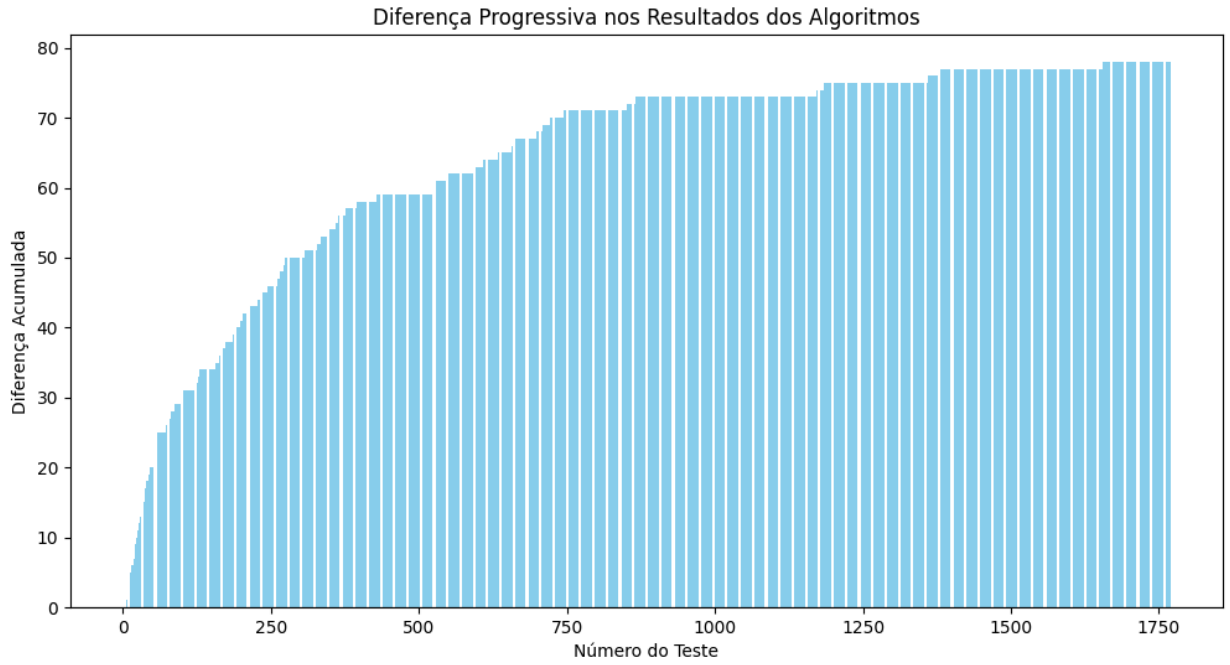
Tipo de modelagem	I1	LLi	D1	LLd	LL
Lista adj densidade 10%	2,579	2,358	246,326	60,089	62,447
Matriz adj densidade 10%	2,581	2,479	283,300	150,513	152,992
Lista adj densidade 100%	2,610	2,502	2,420,817	1,018,854	1,021,356
Matriz adj densidade 100%	2,581	2,481	1,597,799	394,290	396,771

Análise da qualidade e dos resultados do dijkstra e a-estrela

Para compreender os resultados dos algoritmos de busca do dijkstra e a-estrela é importante mencionar a principal diferença entre os dois. Para inserção na fila de prioridade o a-estrela utiliza uma heurística, que nesse caso é a distância euclidiana até o vértice final do grafo, que é adicionada no atributo “custo” dos elementos do heap. Essa soma impacta diretamente no resultado do algoritmo já que a ordem de prioridade da fila em alguns casos pode ser completamente alterada por conta da heurística. É importante mencionar também que esse valor não é inserido no vetor que armazena as distâncias mínimas até um vértice, portanto o custo mínimo para determinado vértice não seria alterado. Apesar disso, como nosso programa também utiliza a premissa de finalizar a busca caso o último vértice seja retirado do minHeap existem casos de grafos em que o a-estrela “erraria” o resultado em relação ao dijkstra.

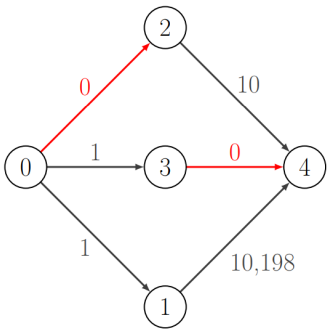
Considerando essa influência da heurística no heap, ao considerar um grafo com uma quantidade fixa de vértices com coordenadas e densidade também fixas podemos

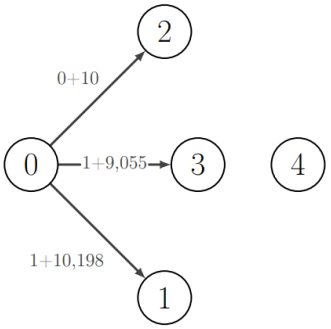
concluir que a quantidade de portais altera o resultado do a-estrela já que os portais alteram o peso das arestas para zero o que também influencia diretamente a prioridade no minHeap. Isso pode ser comprovado por meio de um teste que gera inputs para um grafo de número de sessenta vértices com coordenadas fixadas e densidade de cinquenta por cento mas variando o número de portais de 0 até 1770 que é o número máximo de arestas para o grafo nessas configurações.



A interpretação do gráfico pode ser feita da seguinte forma. Se a barra está subindo significa que o resultado do algoritmo 1 foi 1 e o do algoritmo 2 foi 0. Ou seja, o algoritmo 1 teve um resultado melhor ou mais frequente naquele teste específico.Se a barra está descendo significa que o resultado do algoritmo 1 foi 0 e o do algoritmo 2 foi 1. Ou seja, o algoritmo 2 teve um resultado melhor ou mais frequente naquele teste específico.Se a barra está estável (não subindo nem descendo) significa que ambos os algoritmos tiveram o mesmo resultado naquele teste específico, seja ambos 0 ou ambos 1. Esse resultado era esperado pois quanto mais portais menor será o impacto da prioridade no heap já que ao chegar no vértice final a energia será suficiente, portanto a curva deve se estabilizar.

Outra maneira simples de provar a influência no heap seria analisando o exemplo de referência do enunciado. A heurística faz o vértice prioritário ser o 2, portanto chegaremos ao vértice final pelo caminho do 2. Se Linque tivesse no mínimo 10 de energia o resultado seria o mesmo para os dois algoritmos.





Impacto das escolhas

Após as análises realizadas nas seções anteriores conseguimos comprovar experimentalmente que o programa desenvolvido traz diferentes vantagens e desvantagens em relação à utilização das estruturas de matriz ou lista de adjacência. Em relação ao modelo de fila de prioridade implementado, a estrutura utilizada se mostrou eficiente para encontrar a solução final dos algoritmos de busca, levando em consideração que ela também foi efetiva para lidar com o requisito dos portais. Outros modelos poderiam ter sido utilizados para implementar a solução, ao invés do heap binário poderia ser utilizado por exemplo o heap de fibonacci. Além disso, a struct que armazena os “estados” que estamos percorrendo no heap pode sofrer algumas alterações que não alteram o resultado final, por exemplo, ao invés de armazenar a quantidade de portais restantes poderia armazenar a quantidade de portais usados até aquele estado.

6. Conclusões

O trabalho em questão englobou o desenvolvimento de soluções para problemas de busca do caminho mínimo de acordo com as especificações propostas no enunciado. A escolha de grafos direcionados para modelar os caminhos colaborou para um estudo e compreensão maior sobre essas estruturas.

As demais estruturas escolhidas para implementação foram responsáveis por garantir uma boa modelagem do problema facilitando o desenvolvimento da lógica dos algoritmos de dijkstra e do a-estrela que de certa forma ficaram similares com a as suas respectivas implementações na forma canônica, de tal maneira que foi necessário apenas pequenas alterações para adequar as lógicas de portais. Além disso, é possível também concluir que apesar dessas adaptações que foram feitas não tivemos uma piora ou melhora significativa na complexidade se comparado em relação à forma canônica deles.

Com as realizações dos testes ficou evidente que não existe um modelo de representação do grafo que se sobressaia em todos os casos de testes, mostrando novamente a importância da análise dos requisitos do problema para identificar a melhor solução possível respeitando suas falhas. Uma conclusão similar pode ser obtida se comparado os resultados do dijkstra com o a-estrela, como foi mencionado na documentação, apesar do a-estrela convergir mais rapidamente para resposta seu resultado pode estar “errado” para alguns casos de teste.

Sendo assim, novamente o trabalho prático evidencia o fato de que para certos problemas não existe uma resposta 100% efetiva para todos os casos de teste.

7. Bibliografia

Algoritmos - Teoria e Prática Thomas H. Cormen, Charles E. Leiserson. 3ed

Wikipedia - Algoritmo A*