Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Curso de Graduação em Ciência da Computação

Trabalho Prático 2 - Estrutura de Dados
Escapando da Floresta da Neblina

Lucas Affonso Pires

Matrícula: 2023028420

Belo Horizonte, Julho de 2024

1 Introdução

Como proposta do trabalho, foi selecionado o seguinte problema: Ajudar o herói Linque a escapar de uma floresta neblinosa, utilizando os algoritmos de Djikstra e A* para informar se o caminho mínimo necessário para escapar é condizente com a energia restante que o herói possui. Entre os elementos principais do problema podemos citar: as clareiras, numeradas de 0 a $n-1$ e que indicam cada vértice do grafo que representa a floresta; as trilhas que conectam as clareiras umas as outras e possuem um número que define a distância; e os portais, que conectam uma clareira a outra porém possuem distância 0, ou seja, não utilizam a energia do herói. O objetivo deste documento é bem simples, explicar as ideias e escolhas realizadas para a resolução do problema, além de demonstrar por meio de análises experimentais a complexidade dos algoritmos utilizados.

2 Método

O programa foi implementado visando unicamente resolver o problema acima, e, para tal, foram utilizadas poucas bibliotecas do $C++$ fora da padrão $< iostream >$, como a biblioteca $< cmath >$, utilizada apenas para auxiliar no cálculo da distância euclidiana do caminho, a biblioteca $< limits >$, utilizada apenas para inicializar locais não visitados com valor infinito, facilitando a identificação do que já foi computado e o que ainda não foi e a biblioteca $< stdexcept >$, usada para o tratamento de exceções. Como a base para a solução do problema são os algoritmos de Djikstra e $A*$, eles serão resumidamente explicados separadamente do resto do código, destacando sua importância.

- Algoritmo de Djikstra:

  O algoritmo de Djikstra é um algoritmo utilizado para calcular a menor distância entre um vértice de origem e algum outro vértice selecionado, desde que as arestas que os conectem não possuam valor negativo. Seu funcionamento segue os seguintes passos: Inicializa o vértice de origem como 0 e todos os outros com valor infinito; utiliza uma fila de prioridade (será devidamente explicada abaixo nos TADs) que remove sempre o vértice com menor distância até ele; calcula a nova distância para cada vizinho do vértice atual; atualiza a distância e insere o vizinho na fila de prioridade, caso a nova distância seja menor que a conhecida; por fim, finaliza o funcionamento quando a fila de prioridade está vazia ou o vértice de destino foi removido da fila.

- Algoritmo $A*$:

  O algoritmo A* tem funcionamento parecido com o Djikstra já que é uma extensão dele, a principal diferença é a utilização de uma heurística (nesse caso a distância euclidiana calculada) que estima a distância, tornando a busca pelo caminho mais eficiente e direcionada. Os passos para seu funcionamente são, também, similares ao do Djikstra: Inicializa o vértice de origem como 0 e todos os outros com valor

infinito; calcula a heurística para estimar o custo do vértice de origem ao de destino; insere o vértice atual na fila de prioridade com o valor da heurística; remove sempre o vértice de menor prioridade (heurística + distância) da fila; calcula a nova distância para cada vizinho do vértice atual; atualiza a distância e insere o vizinho na fila de prioridade, caso a nova distância seja menor que a conhecida; por fim, finaliza o funcionamento quando a fila de prioridade está vazia ou o vértice de destino foi removido da fila.

Como método de implementação do restante do código, foram criados estruturas e funções simples, que obedecem o enunciado proposto para o trabalho. Abaixo, será comentados cada TAD (Tipo abstrato de dados) utilizado e cada uma das funções nele contido, de maneira breve, já que o próprio código possui comentários que auxiliam o entendimento. Entre os TADs utilizados estão:

- Fila de prioridade:

  O TAD de fila de prioridade, como o próprio nome explicita, é responsável apenas por gerir uma fila de prioridade. Apesar de realizar apenas isso, sua função é imprescindível para o funcionamento correto do programa, já que os algoritmos de Djikstra e $A*$ necessitam dessa fila para processar qual a aresta de menor distância que será utilizada para o cálculo do caminho. Entre as funções do TAD estão:

  - FilaPrioridade(): Inicializa os valores da fila de prioridade.
  - FilaPrioridade(): Limpa a memória utilizada na construção.
  - void push(int vertice, double prioridade, int contadorPortal): Insere um vértice na fila de prioridade com uma prioridade e um contador dos portais usados.
  - void pop(): Remove o elemento de maior prioridade da fila
  - bool empty(): Vê se a fila está vazia.
  - NodeFilaPrioridade* top(): Retorna o elemento de maior prioridade (no caso o vértice com menor distância total).

- Grafo utilizando uma Lista de Adjacência:

  O TAD de grafo que utiliza uma Lista de adjacência é uma estrutura de dados onde cada vértice possui uma lista das arestas a ele conectado (e consequentemente os vértices conectados), e por isso, é eficiente no uso do espaço para grafos muito esparsos, já que facilita a iteração dos vértices com seus vizinhos. Entre as funções do TAD estão:

  - GrafoLista(int n): Inicializa o grafo com n vértices.
  - GrafoLista(): Limpa a memória utilizada na construção.
  - void novaAresta(int w, int z, double peso): Adiciona uma aresta de um vértice

w a z com uma distância peso.

- void novoPortal(int w, int z): Adiciona um novo portal de um vértice w a z.
- void setCoordenadas(int w, double x, double y): Define as coordenadas de um vértice.
- int getTamanho(): Retorna o número de vértices do grafo.
- double getCoordenadaX(int w): Retorna a coordenada X de um vértice.
- double getCoordenadaY(int w): Retorna a coordenada Y de um vértice.
- Aresta* getAresta(int w): Retorna as arestas que saem de um vértice.
- Portal* getPortais(int w): Retorna os portais que saem de um vértice..
- double heuristicaLista(double x0, double y0, double x1, double y1): Calcula a distância euclidiana da origem ao destino.
- int dijkstraL(GrafoLista& grafo, int comeco, int fim, double s, int q): Executa o Djikstra.
- int aEstrelaL(GrafoLista& grafo, int comeco, int fim, double s, int q): Executa o $A*$.
- void leitorLista(GrafoLista& grafo, int& n, int& m, int& k, double& s, int& q): Recebe os valores que serão utilizados para o programa.

- Grafo utilizando Matriz:

  O TAD de grafo que utiliza uma Matriz é uma estrutura de dado onde uma matriz é utilizada para armazenar a distância das arestas entre cada par de vértice, sendo eficiente no uso do tempo para grafos muito densos, onde a maioria dos vértices se conectam. Entre as funcções do TAD estão:

  - GrafoMatriz(int n): Inicializa o grafo com n vértices.
  - GrafoMatriz(): Limpa a memória utilizada na construção.
  - void novaAresta(int w, int z, double peso): Adiciona uma aresta de um vértice w a z com uma distância peso.
  - void novoPortal(int w, int z): Adiciona um novo portal de um vértice w a z.
  - void setCoordenadas(int w, double x, double y): Define as coordenadas de um vértice.
  - int getTamanho(): Retorna o número de vértices do grafo.
  - double getCoordenadaX(int w): Retorna a coordenada X de um vértice.
  - double getCoordenadaY(int w): Retorna a coordenada Y de um vértice.

- double getArestaPeso(int w, int z): Retorna o peso da aresta que sai do vértice w para z.
- int* getPortais(int w, int& count): Retorna os portais que saem de um vértice.
- double heuristicaMatriz(double x0, double y0, double x1, double y1): Calcula a distância euclidiana da origem ao destino.
- int dijkstraM(GrafoMatriz& grafo, int comeco, int fim, double s, int q): Executa o Djikstra.
- int aEstrelaM(GrafoMatriz& grafo, int comeco, int fim, double s, int q): Executa o $A*$.
- void leitorMatriz(GrafoMatriz& grafo, int& n, int& m, int& k, double& s, int& q): Recebe os valores que serão utilizados para o programa.

3 Análise de Complexidade

A análise de complexidade de cada algoritmo foi relizada baseando-se no tempo e espaço utilizado por cada algoritmo para cada tipo de implementação diferente. No caso, serão analizados a complexidade de tempo e espaço dos algoritmos tanto para a implementação que utiliza Lista de Adjacência quanto para a que utiliza Matriz.

- Algoritmo de Djikstra:

  Analisaremos primeiro a complexidade de espaço. Para a implementação da Lista de adjacência, percebemos que cada vértice armazena a lista das arestas que saem dele (já explicado anteriormente na parte da implementação dos algoritmos), logo a complexidade de espaço para a lista é o espaço para armazenar os vértices (V) somado ao espaço para armazenar o peso de cada aresta ligada a esse vértice (E), e portanto a complexidade espacial da Lista de Adjacência é $O(V + E)$. Para a Matriz, utilizamos uma matriz VxV onde cada termo da matriz armazena o peso da aresta que vai de i a j (caso não haja aresta que vai de i a j, ainda assim o espaço é utilizado, já que no algoritmo damos às arestas não utilizadas valor infinito), logo, para a complexidade espacial da Matriz, independe o número de arestas e seu valor é dado por $O(V^2)$.

  Analisando agora a complexidade de tempo. Para cada operação na Lista de adjacência, adicionar ou remover um elemento da fila de prioridade tem complexidade $O(logV)$ (isso pois a fila de prioridade é baseada em um heap e a complexidade de um heap é $O(logn)$), e, como removemos cada vértice uma única vez, a complexidade temporal se torna $O(VlogV)$, porém, como temos que analisar cada aresta ligada ao vértice removido, a complexidade temporal final da Lista de Adjacência é $O((V + E)logV)$. Para a Matriz, o custo é o mesmo até o momento da remoção,

$O(VlogV)$, mas, como ao remover um vértice da fila de prioridade na matriz, precisamos analisar todas as possíveis conexões V, a complexidade final da Matriz é $O(V^2logV)$.

- Algoritmo $A*$:

  Analisaremos primeiro a complexidade de espaço. Perceba que para o algoritmo $A*$, a complexidade espacial é idêntica á complexidade espacial do algoritmo de Djikstra, já que a utilização da heurística não afeta o espaço utilizado pelo algoritmo e sim o tempo. Desse modo, a complexidade espacial da Lista de Adjacência é $O(V + E)$, e a complexidade espacial da Matriz é $O(V^2)$.

  Analisando agora a complexidade de tempo. Perceba que o funcionamento do $A*$ é basicamente idêntico ao do Djikstra com exceção do uso da heurística, portanto, para a Lista de Adjacência seu custo é $O((V + E)logV)$ e para a Matriz seu custo é $O(V^2logV)$.

Partindo apenas da complexidade dada, seria possível acreditar então, num primeiro momento, que não faz sentido utilizar o algoritmo $A*$, já que supostamente seu gasto seria idêntico ao do Djikstra e seu resultado seria basicamente o mesmo, porém, a utilização da heurística, principalmente quando ela se aproxima do ideal, pode fazer com que o aproveitamente de tempo melhore bastante, já que a heurística como guia, limita a execução do $A*$ para um caminho que se aproxima do mais curto, enquanto o Djikstra irá explorar todos os caminhos.

4 Estratégias de Robutez

Entre algumas das estratégias de robustez utilizadas estão: aquelas que evitam o uso desnecessário de memória e o uso desnecessário de tempo na execução, como o uso de construtores e destrutores; o encapsulamento de estruturas específicas para a execução do algoritmo, fazendo, assim, com que a resolução de possíveis problemas se torne mais simples e localizada; a utilização de condições de parada na execução dos algoritmos, garantindo que o programa finalize quando a fila de prioridade estiver vazia; utilização da biblioteca de tratamento de exceções $< stdexcept >$, principalmente para a entrada de dados que não estão de acordo com o padrão fornecido no enunciado, utilizando as funções throw, try e catch. De modo geral, as estratégias de robustez que foram escolhidas tem como foco garantir o bom funcionamento do programa, tanto evitando casos de borda, como melhorando o gerenciamento de tempo e espaço.

5 Análise experimental

Para a análise experimental iremos comparar os tempos de execução do Djikstra e do $A*$, da Lista de Adjacência com a Matriz de Adjacência para grafos normais e grafos densos, e

analisaremos se os resultados produzidos pelo Djikstra e $A*$ são os mesmo.

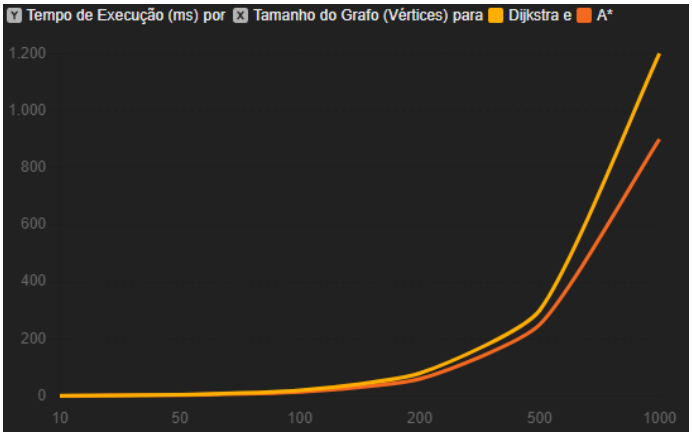

Figura 1: Comparativo Djikstra e $A*$.

Por meio da análise desse gráfico, percebemos como o $A*$ e o Djikstra tendem a ter a mesma complexidade temporal, conforme visto anteriormente, porém a utilização da heurística no algoritmo $A*$ direcionando a busca pelo menor caminho dá uma ligeira vantagem a ele. Porém, se utilizarmos apenas o gráfico de tempo para decidir qual o melhor estariamos ignorando um ponto de grande importância, o A* pode apresentar falhas no problema que estamos lidando por conta da existência dos portais. Usaremos como exemplo um dos testes fornecidos:

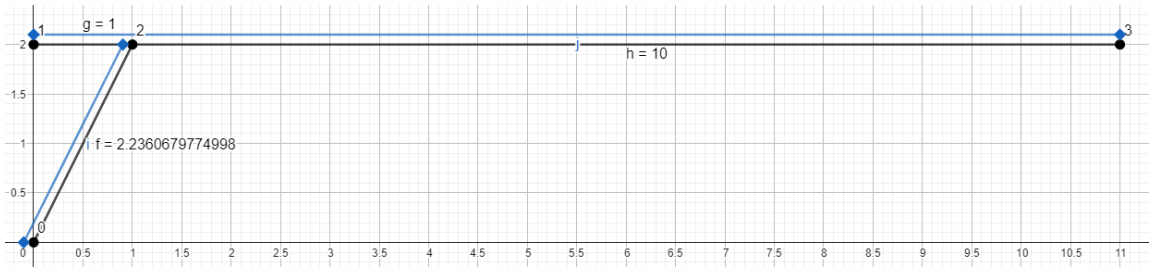

Figura 2: Grafo do teste.

Na imagem, os pontos representam as clareiras, as linhas em preto representam as trilhas e as linhas em azul representam os portais. No grafo dado, a energia que Linque possui é 1 e ele pode atravessar até 2 portais. Com isso, note que existe uma solução em que Linque consegue escapar da floresta, utilizando o portal da clareira 0 até a clareira 2, andando a trilha de peso 1 que leva da clareira 2 para a clareira 1 e por fim utilizando o

portal que leva da clareira 1 até 3. Sendo assim, para o algoritmo Djikstra, o resultado é 1 , já que Linque consegue escapar, porém, para o algoritmo $A*$ a heurística faz com que ele tenha resultado 0, dizendo que Linque não conseguiria escapar. Isso ocorre pois a heurística contabiliza a distância dos portais mesmo que ela não seja utilizada no fim, assim, o algoritmo enxerga que o menor caminho possível é da clareira 0 para a clareira 2, e então da 2 até a 3, necessitando de 12.23 de energia e considerando assim, que Linque não poderá escapar da floresta.

Dessa maneira, concluimos que apesar de o $A*$ ser mais rápido em relação ao tempo, sua utilização nem sempre nos dará o resultado desejado, já que os portais podem fazer com que ele considere um caminho possível de ser percorrido com certa energia como impossível.

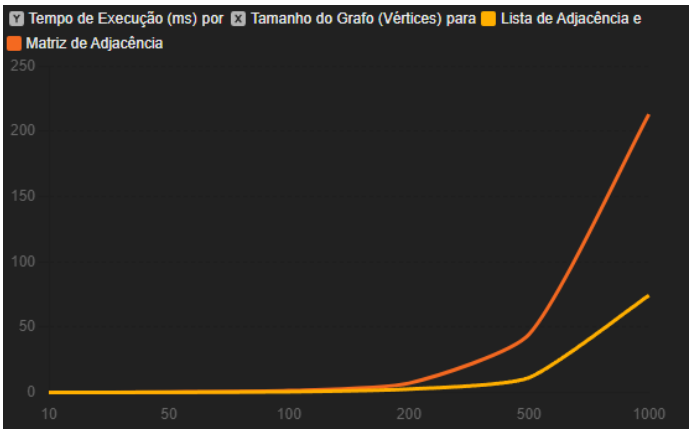

Figura 3: Comparativo Lista de Adjacência e Matriz.

Perceba, analisando os gráficos, que usam grafos aleatórios referentes ao tempo de execução, utilizando a Lista de Adjacência e a Matriz, que a Lista realmente tem uma execução mais rápida conforme foi definido anteriormente na complexidade temporal. Porém, vamos analisar casos em que os grafos são densos para poder chegar a uma conclusão.

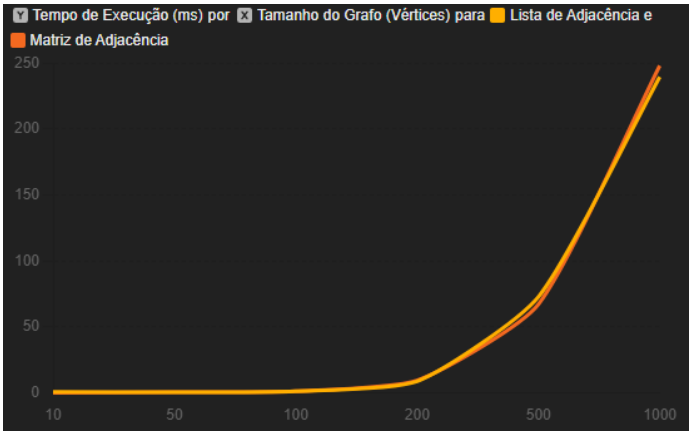

Figura 4: Comparativo Lista de Adjacência e Matriz, para grafos densos.

Analisando este segundo comparativo, percebemos que quando temos ao nosso dispor grafos mais densos, a utilização da Matriz de Adjacência se torna uma boa opção, já que seu custo se torna basicamente igual ao da Lista, visto que a maioria dos vértices estão conectados e, portanto, a maior parte dos elementos da matriz estão preenchidos.

6 Conclusões

Implementados os algoritmos Djikstra e $A*$ com uma fila de prioridade auxiliar, utilizando-se Lista de Adjacência e Matriz para calcular o menor caminho possível em um grafo, podemos fazer certas conclusões. Começaremos pela opção de escolha para utilizar ou Djikstra ou o $A*$. Apesar de terem complexidade bem similar, existem certos casos onde um se mostra melhor que o outro. O $A*$ é definitivamente mais eficiente quando a heurística utilizada é perto da ideal, já que assim, evita-se explorar caminhos que não serão úteis para o encontro do caminho de menor tamanho e a busca pelo caminho correto se torna mais direcionada e mais rápida. Apesar disso, o Djikstra pode se mostrar melhor quando não é possível extrair com eficiência uma heurística próxima à ideal, além de poder explorar todos os possíveis caminhos (no caso que estamos trabalhando isso acaba sendo útil em casos onde o melhor caminho não é guiado pela heurística, já que a existência de portais acaba enganando o $A*$, visto que a existência de uma aresta sem custo em um caminho não ideal, acaba não sendo computada pelo algoritmo). Além disso, a utilização da heurística quando temos a existência de elementos como portais pode acabar sendo uma opção ruim, já que ao limitar um caminho em favorecimento da heurística, pode ser que os resultados fornecidos pelo Djikstra e pelo $A*$ sejam diferentes, como visto na análise experimental.

Vamos agora revisar a utilização de uma Lista de Adjacência e de uma Matriz para a execução dos algoritmos. Com base nos custos definidos na análise de complexidade e do

exposto na análise experimental, vemos que no geral a Lista de Adjacência acaba sendo mais eficiente no quesito temporal e espacial que a Matriz, porém em casos específicos a Matriz se torna mais eficaz. Para grafos comuns e grafos mais esparsos, a Lista de Adjacência se mostra melhor, porém em grafos mais densos, onde o número de arestas se aproxima do número de vértices, a Matriz pode aparecer como uma opção mais interessante.

Concluindo, a utilização de um algoritmo ou de outro, a utilização de uma lista ou de uma matriz, acaba sendo definido pelo conhecimento prévio a partir do grafo que será utilizado. No caso trabalhado, por conta dos portais e dos grafos fornecidos como teste, a utilização do Djikstra por meio da Lista de Adjacência se mostrou como o caso mais seguro e eficaz para a produção de resultados.

7 Bibliografia

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados.

https://www.geogebra.org/calculator

https://www.youtube.com/watch?v=Kue4UUxstoU&ab_channel=MaratonaUFMG

https://pt.wikipedia.org/wiki/Algoritmo_A*

https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiytcL-s8CHAxW
url=https%3A%2F%2Fsol.sbc.org.br%2Findex.php%2Fetc%2Farticle%2Fdownload%2F9852%
2F9750%2F&usg=AOvVaw1hPuPjrAyoiNfb0MEv85st&opi=89978449