Universidade de São Paulo, Escola de Artes, Ciências e Humanidades, Programa de Pós-Graduação em Sistemas de Informação

Relatório EP 1 - Parte 1 - Análise de Algoritmos e Estrutura de Dados

Arthur P. Grava, Lucas F. Brunialti, Rafael S. Torres, Filipe E. S. P. Palma

10 de junho de 2014

O trabalho consiste na implementação do algoritmo de Dijkstra usando filas de prioridades implementadas como Heap, sendo os vértices do grafo cidades, representadas por pontos (x, y) no plano cartesiano, e as arestas pela distância euclidiana entre dois pontos. A implementação do trabalho foi disponibilizada em `github.com/arthurgrava/twinkle`

Para melhor dispor as idéias do trabalho, este relatório foi disposto nas seguintes seções: Seção 1, aborda o diagrama de classes da implementação; Seção 2, apresenta a análise dos algoritmos usados; e a Seção 3, que apresenta a visualização da saída do algoritmo.

1 Diagrama de Classes

A modelagem das classes foi construída visando simplificar o projeto, assim, foi construído quatro pacotes *model*, *controller*, *util* e *algorithm*. Os pacotes *model* e *controller* foram pensados com base no framework MVC, para facilitar a organização do código. O pacote *util* foi construído para reunir funções comuns ao todo, e o pacote *algorithm* contendo o algoritmo explicitamente. Além desses, o programa pode ser executado na classe *App*, que possui o método *main* necessário para tal.

Os possíveis caminhos foram representados em forma de um grafo, essa escolha foi feita para melhor abstração do problema, facilidade para a análise e implementa-

ção em java. Essa representação pode ser vista no pacote *model*, onde foi criada a classe *Graph* para a representação do grafo e pela classe *Vertex*, que são identificados pelo atributo *id*, contém coordenadas do plano cartesiano (x, y) e apresentam as variáveis *pi* e *distance* para auxílio no cálculo do algoritmo de *Dijkstra*.

O pacote *controller* possui as classes *GraphLoader* e *GraphOperations* que são responsáveis pelo controle do grafo, manipulações como construção, transformação e visualização.

O pacote *util* contém apenas o cálculo de distância euclidiana, que é usada por todo o programa. No pacote *algorithm* contidas as classes *Node* e *Dijkstra* que implementam todos os algoritmos usados: *Dijkstra, buildMinHeapify, minHeapify, retrieveMinimum* e *relax*.

No algoritmo de *Dijkstra*, foram utilizados dicionários chave-valor na implementação para auxiliar nas trocas de variáveis entre os elementos da fila de prioridade, para melhor aproveitar o uso da memória do *JAVA*, evitando-se de criar cópias das posições do array e, ao invés disse, aproveitando-se da memória já alocada previamente durante a construção do dicionário, otimizando, assim, o gasto de memória, e não afetando no desempenho do algoritmo.

O diagrama pode ser visualizado na Figura 1.1.

2 Análise dos Algoritmos

Para a análise dos algoritmos foi usado as notações: $|V|$ como o número de vértices e $|E|$ como o número de arestas. Também, assume-se que n, o tamanho do heap, é o número de vértices $|V|$. Os algoritmos (pseudocódigo), assim como as análises, são mostradas a seguir:

minHeapify(A, n, i)

1	$e = 2i$	$\| O(n)$
2	$e = 2i + 1$	$\| O(n)$
3	**if** $e \leq n$ & $A[e] < A[i]$	$\| O(n)$
4	$min = e$	$\| O(k)$
5	**else**	
6	$min = i$	$\| O(k)$
7	**if** $d \leq n$ & $A[d] < A[min]$	$\| O(k)$
8	$min = d$	$\| O(k)$
9	**if** $min \neq i$	$\| O(k)$
10	$A[i] \Leftrightarrow A[min]$	$\| O(k)$
11	minHeapify(A, n, min)	$\| kO(\log n)$

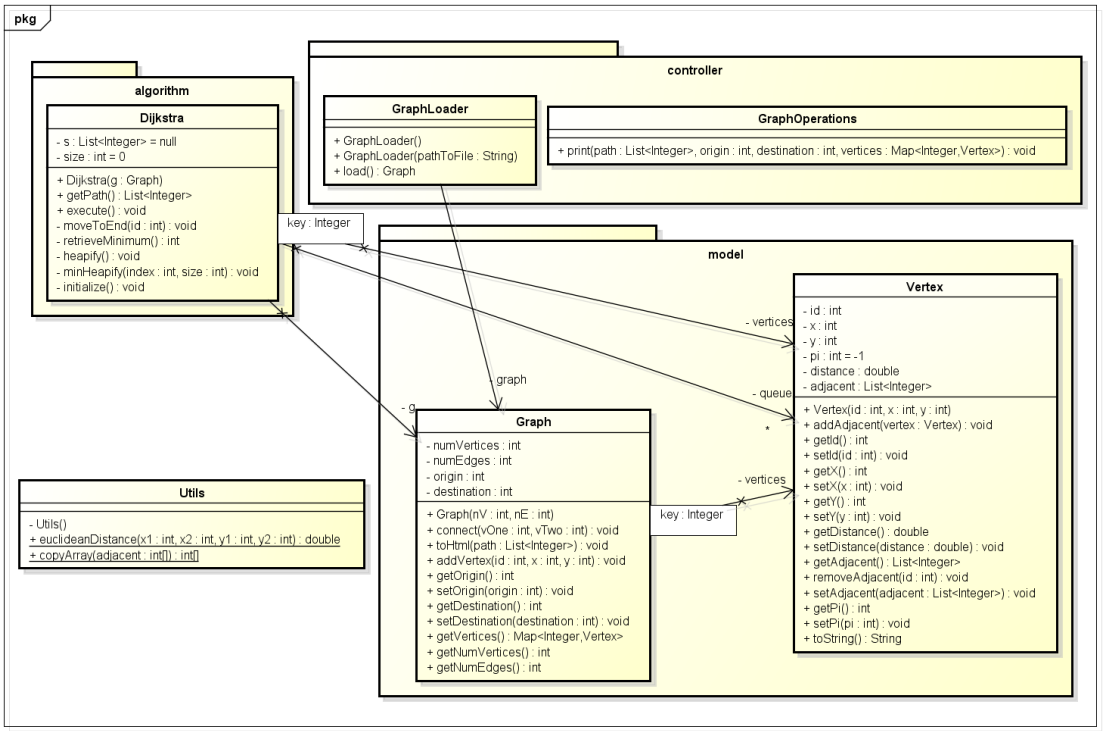

Figura 1.1: Diagarma de classes.

$$custo = O(\log|V|)$$

BUILDMINHEAPIFY(A, n)

1 **for** $i = \lfloor \frac{n}{2} \rfloor$ **to** 1 | $O(|V|)$
2 MINHEAPIFY(A, n, i) | $O(|V|\log|V|)$

SEJA h A ALTURA DO HEAP:

$$\sum_{h=0}^{\lfloor \log|V| \rfloor} \left\lceil \frac{|V|}{2^{h+1}} \right\rceil O(h) = O(|V| \sum_{h=0}^{\lfloor \log|V| \rfloor} \frac{h}{2^h}) \leq O(|V| \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(2|V|) = O(|V|)$$

RETRIEVEMINIMUM(A, n)

1 $min = A[1]$ | $O(1)$
2 $A[1] \Leftrightarrow A[n]$ | $O(1)$
3 MINHEAPIFY$(A, n-1, 1)$ | $O(\log|V|)$
4 **return** min | $O(1)$

$$custo = O(\log|V|)$$

EXECUTEDIJKSTRA(V, A, w, s)

1	INITIALIZE(V, A, s)	$\mid O(\|V\|)$
2	$S = \{\}$	$\mid O(1)$
3	$Q = V$	$\mid O(\|V\|)$
4	**while** Q is not Empty	$\mid O(\|V\|)$
5	u = RETRIEVEMINIMUM($Q, \|V\|$)	$\mid O(\|V\|\log\|V\|)$
6	$S = S \cup \{u\}$	$\mid O(\|V\|)$
7	**for** each vertex v in $Adj[u]$	$\mid O(\|V\|\|E\|)$
8	RELAX(u, v, w)	$\mid O(\|E\|\log\|V\|)$

A LINHA 8 DO ALGORITMO, ESTA IMPLÍCITO O USO DA FUNÇÃO DECREASEKEY, POR ISSO O CONSUMO DE $O(\log|V|)$.

$$custo = O((|V| + |E|)\log|V|)$$

NO ENTANTO, POR QUESTÕES DE ESTOURO DE *heap* DO JAVA (MEMÓRIA), A FUNÇÃO RELAX NÃO FAZ CHAMADA À FUNÇÃO DECREASEKEY, ENTÃO ENTRE AS LINHAS 4 E 5, É CHAMADA A FUNÇÃO BUILDMINHEAPIFY PARA REORDENAR Q, AUMENTANDO A COMPLEXIDADE DO ALGORITMO PARA $O(n\text{^}2)$.

3 VISUALIZAÇÃO

PARA A VISUALIZAÇÃO DAS DUAS ENTRADAS, QUE FORAM PROVIDENCIADAS, FOI UTILIZADO O PACOTE *D3.js*. ESSE PACOTE UTILIZA A LINGUAGEM *javascript* PARA GERAR OS ELEMENTOS DE UM GRAFO EM UMA PÁGINA *web*. A GRANDE VANTAGEM DO USO DESSE PACOTE É QUE ELE UTILIZA UM ARQUIVO NO FORMATO *.json* APENAS PARA MONTAR O GRAFO. ASSIM, TODAS AS INFORMAÇÕES SOBRE VÉRTICES E ARESTAS, ASSIM COMO AS INFORMAÇÕES SOBRE FORMATAÇÃO (COR, POSICIONAMENTO, VALOR DOS ELEMENTOS) ESTÃO ARMAZENADAS EM UM ÚNICO ARQUIVO SEPARADO DA PÁGINA *web*, O QUE PERMITE QUE A APLICAÇÃO GERE APENAS ESSE ARQUIVO PARA CADA SAÍDA, SEM PRECISAR ALTERAR A PÁGINA *web*. AINDA, A ESTA PÁGINA CABE APENAS A FUNÇÃO DE INTERPRETAR O ARQUIVO *.json* E POSICIONAR OS ELEMENTOS NA TELA. AO FINAL DE CADA EXECUÇÃO DA APLICAÇÃO, A PÁGINA *web* É CHAMADA AUTOMATICAMENTE POR UM COMANDO QUE ABRE ELA NO NAVEGADOR PADRÃO, MOSTRANDO O RESULTADO VISUAL DA IMPLEMENTAÇÃO.

Figura 3.1: Grafo gerado para a primeira saída.

Para cada entrada fornecida como teste, foram gerados os grafos representando a estrutura. As informações sobre o conteúdo das arestas foram suprimidas para evitar poluição visual, principalmente no caso do segundo exemplo. Entretanto, essas informações podem ser obtidas por alterar a página *web*, que já possui os comandos para exibí-las. Para a primeira saída, os vértices foram posicionados de forma a manterem uma proporção entre as arestas e ficarem no centro da página. como mostram as Figuras 3.1 e 3.2.

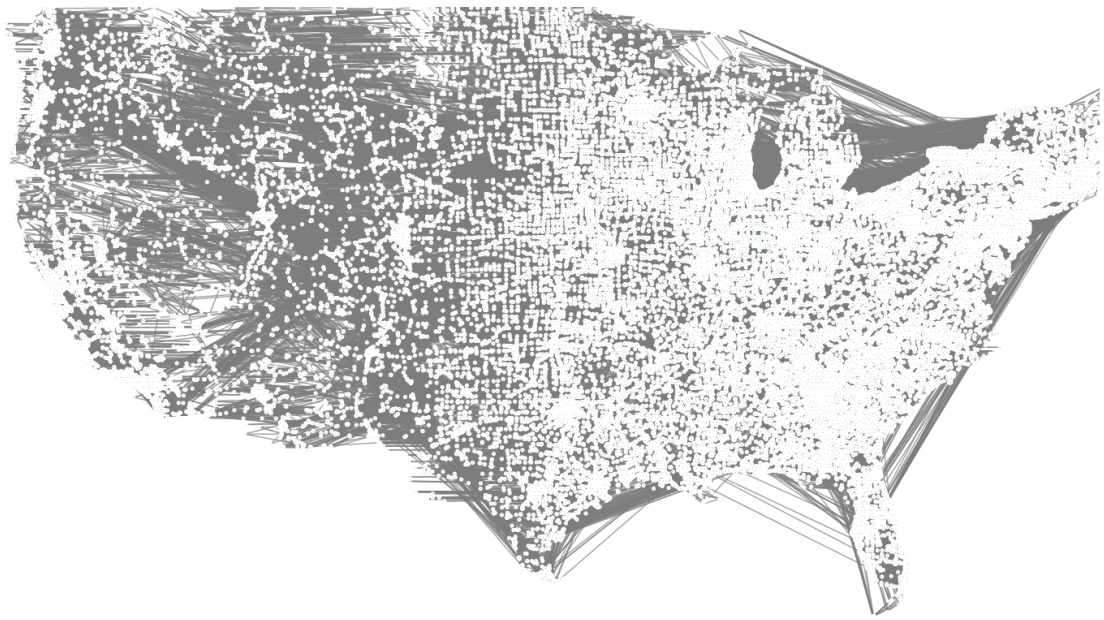

Figura 3.2: Grafo gerado para a segunda saída.