Trabalho Prático 1 de Projeto e Análise de Algoritmos

Thiago Mateus de Almeida Dias Orsini

Vitor Ferreira França

Introdução

O trabalho prático 1 tem como objetivo elaborar um programa, na linguagem C, que seja capaz de encontrar K caminhos entre uma cidade A, chamada Mysthollow, e uma cidade B, chamada Luminae. E para representar essas cidades e seus caminhos utilizamos um grafo direcionado e arestas com pesos.

Primeira forma (baseado em Epstein):

Funções e estruturas de dados no programa

No arquivo func.h estão as estruturas de dados e as assinaturas das funções que serão utilizadas no programa

• node: struct responsável por representar os vértices do grafo. Ela possui um int vertex para representar o vértice adjacente, um int weight para guardar o peso e uma struct node* next que é para apontar para o próximo item da lista encadeada. Ela define

• Graph: struct responsável por representar o grafo. Ela possui um int numVertices para guardar a quantidade de vértices do grafo, possui um node** adjListas que guarda todos os vértices do grafo, possui também um int* visited para guardar os vértices já visitados durante a execução do programa.

• createNode: função que recebe dois parâmetros int, um chamado v e outro w. Essa função cria um nó com o vértice v e peso w e retorna um node*. Ela aloca o espaço de memória, define qual é o vértice, qual é o peso da aresta, coloca o próximo item da lista encadeada como NULL e retorna o nó criado.

• createGraph: função que recebe um parâmetro int vertice e inicia o grafo alocando os espaços para inserir os dados posteriormente e retorna um Graph*. Ela aloca o espaço de memória, define o número de vértices, aloca memória para a lista de adjacência e para o vetor de visitados, inicializa as listas de adjacência e o vetor de visitados e retorna o grafo.

• addEdge: função que recebe os parâmetros Graph* graph, int src, int dest, int weight para adicionar a aresta do nó src ao nó dest com o peso weight. Ela cria um nó com o destino e o peso especificados e o adiciona na lista de adjacência do vértice de origem, cria outro nó com a origem e o peso especificados e o adiciona na lista de adjacência do vértice de destino.

•freeGraph: função que recebe um Graph* graph como parâmetro para desalocar a memória usada para guardar o grafo. Ela libera a memória alocada para cada lista de adjacência, libera a memória alocada para a lista de adjacência, para o vetor de vértices visitados e para a estrutura do grafo.

•Heap_Priority createPriorityHeap:Função que recebe um parâmetro int capacity e cria uma nova heap de prioridade.

•swapNodes:Função para trocar dois nós na heap de prioridade. Ela recebe dois ponteiros para HeapNode e troca seus conteúdos.

•Heapify:Função para reorganizar a heap de prioridade após uma inserção ou remoção. Ela recebe um ponteiro para a heap de prioridade e um índice, e reorganiza a heap de forma apropriada.

•insertPriorityHeap:Função para inserir um nó na heap de prioridade. Ela recebe um ponteiro para a heap de prioridade, um vértice e uma prioridade, e insere o nó na heap.

•extractMin:Função para remover e retornar o nó de menor prioridade da heap de prioridade. Ela recebe um ponteiro para a heap de prioridade e retorna o nó de menor prioridade.

•DijkstraForKpas:Algoritmo de Dijkstra modificado para contar as visitas ao destino. Recebe um ponteiro para o grafo, um valor k e um ponteiro para o arquivo de saída. Retorna verdadeiro se o algoritmo foi executado com sucesso.

Análise de complexidade

A complexidade total do algoritmo é dominada pela complexidade do loop principal, que depende do número de iterações até que o destino seja visitado k vezes. Em geral, a complexidade do algoritmo é aproximadamente O((V + E) * log V), onde V é o número de vértices e E é o número de arestas no grafo

logo: O((V + E) * log V),

.

Segunda forma (baseado em dfs “força bruta):

Funções e estruturas de dados no programa

No arquivo func.h estão as estruturas de dados e as assinaturas das funções que serão utilizadas no programa.

• node: struct responsável por representar os vértices do grafo. Ela possui um int vertex para representar o vértice adjacente, um int weight para guardar o peso e uma struct node* next que é para apontar para o próximo item da lista encadeada. Ela define

• Graph: struct responsável por representar o grafo. Ela possui um int numVertices para guardar a quantidade de vértices do grafo, possui um node** adjListas que guarda todos os vértices do grafo, possui também um int* visited para guardar os vértices já visitados durante a execução do programa.

• DynamicArray: struct responsável pelo vetor dinâmico. Ela possui um int *array para armazenar os valores, possui dois size_t, um chamado size para guardar o tamanho atual do vetor e outro chamado capacity para guardar a capacidade atual do vetor.

• createNode: função que recebe dois parâmetros int, um chamado v e outro w. Essa função cria um nó com o vértice v e peso w e retorna um node*. Ela aloca o espaço de memória, define qual é o vértice, qual é o peso da aresta, coloca o próximo item da lista encadeada como NULL e retorna o nó criado.

• createGraph: função que recebe um parâmetro int vertice e inicia o grafo alocando os espaços para inserir os dados posteriormente e retorna um Graph*. Ela aloca o espaço de memória, define o número de vértices, aloca memória para a lista de adjacência e para o vetor de visitados, inicializa as listas de adjacência e o vetor de visitados e retorna o grafo.

• addEdge: função que recebe os parâmetros Graph* graph, int src, int dest, int weight para adicionar a aresta do nó src ao nó dest com o peso weight. Ela cria um nó com o destino e o peso especificados e o adiciona na lista de adjacência do vértice de origem, cria outro nó com a origem e o peso especificados e o adiciona na lista de adjacência do vértice de destino.

•freeGraph: função que recebe um Graph* graph como parâmetro para desalocar a memória usada para guardar o grafo. Ela libera a memória alocada para cada lista de adjacência, libera a memória alocada para a lista de adjacência, para o vetor de vértices visitados e para a estrutura do grafo.

• insertionSort: função que recebe os parâmetros int arr[] e int n para ordenar um vetor de tamanho n.

• DFS: função que recebe os parâmetros Graph* graph, int vertex, int end, int path[], int path_index, int path_weight e DynamicArray *numbers. Ela explora o grafo para encontrar o caminho entre o vértice de número “vertex” e o vértice de número “end”. O parâmetro path[] é o caminho percorrido, o path_index é a posição no vetor de caminho, o path_weight guarda o peso do caminho e o numbers é um vetor dinâmico para guardar os pesos do caminho.

• initDynamicarray: função que recebe os parâmetros DynamicArray *dynArray e size_t initialCapacity para iniciar um vetor dinâmico com a capacidade inicial especificada em initialCapacity. Ela aloca memória para o vetor com a capacidade inicial especificada, verifica se a alocação foi bem-sucedida e inicializa o tamanho e a capacidade inicial do vetor.

• append: função que recebe os parâmetros DynamicArray *dynArray e int value para adicionar um valor no vetor. Se o valor não couber no vetor, função dobra o tamanho do vetor e adiciona o elemento. Ela verifica se o tamanho atual é maior ou igual à capacidade e caso seja ela dobra a capacidade, realoca memória para o vetor com a nova capacidade, verifica se a realocação foi bem-sucedida, atualiza o ponteiro do vetor e a capacidade e por fim, adiciona o novo elemento ao final do vetor.

• freeDynamicArray: função que recebe o parâmetro DynamicArray *dynArray para desalocar a memória usada por esse vetor. Ela libera a memória do vetor, define o ponteiro como NULL e o tamanho como 0.

Análise de complexidade

Função DFS= O(n!).

função de ordenação O(n^2)

então o algoritmo corresponde a O(n!)

Resultados dos testes

Primeira forma(baseado em Epstein):

Teste	n	m	k	Tempo CPU (seg)	Tempo relógio (seg)
1	10	20	3	0.000184	0.000264
2	10	12	3	0.000245	0.000272
3	100000	133332	5	0.063104	0.066029
4	100000	200000	5	0.147914	0.153391

5	100000	101969	5	0.045619	0.058512
6	100000	133332	10	0.134763	0.143645
7	100000	200000	10	0.256091	0.269432
8	100000	100202	10	0.170835	0.171951
9	100000	200000	7	0.192023	0.203433
10	4	5	3	0.000280	0.000319
11	4	5	3	0.000236	0.000285
12	60003	120000	10	0.034833	0.039376
13	60003	120000	10	0.046409	0.040555
14	20000	179955	10	0.365166	0.379466
15	20000	199945	10	0.419055	0.424806
16	2	3	3	0.000227	0.000223
17	4	6	4	0.000284	0.000231
18	4	6	10	0.000275	0.000267

Segunda forma(baseado em dfs “força bruta)

Teste	n	m	k	Tempo CPU (seg)	Tempo relógio (seg)
0	10	12	3	0.000490	0.010081
1	10	20	3	0.000329	0.005383
2	10	12	3	0.000527	0.036079
3	100000	133332	5	-	-
4	100000	200000	5		-
5	100000	101969	5	-	-
6	100000	133332	10	-	-
7	100000	200000	10	-	-
8	100000	100202	10	-	-
9	100000	200000	7	-	-
10	4	5	3	0.000255	0.000280
11	4	5	3	0.000292	0.000300
12	60003	120000	10	-	-
13	60003	120000	10	-	-
14	20000	179955	10	-	-

15	20000	199945	10	-	-
16	2	3	3	0.000287	0.035959

Como o programa encontra os caminhos por força bruta, quando testamos para valores muito elevados para n, m e k o algoritmo não consegue encontrar as respostas em tempo curto, no nosso teste ele estava demorando mais de 30 minutos quando nós forçamos a interrupção do programa.

Com os resultados nós percebemos que quando os valores de vértices e arestas não são elevados, o nosso programa demora menos para encontrar os caminhos quando não há muitos caminhos para o destino, como foi o caso do teste 5, que apesar de ter mais vértices, mas com poucos caminhos ele foi o 4º mais rápido nos nossos testes.

Conclusão

O problema de encontrar k caminhos é uma expansão do problema de Diikstra. Para resolução do problema foi estudado o algoritmo de yem, porém, foi percebido problemas na tentativa de aplicação, como resultados não verdadeiros, para tal ,após não conseguirmos resolver este problema, foi decidido focar na estratégia de “força bruta” por uma dfs (deep-first search) e um algoritmo de ordenação, após ser visto a ineficiência da “força bruta” para tal objetivo. Foram feitas pesquisas e foi encontrada o algoritmo de Epstein, no qual melhor se encaixou nos objetivos.

Referencias

https://codeforces.com/blog/entry/102085

https://www.youtube.com/watch?v=t5B064eGMMU&t=118s

-Materiais oferecido no curso de PAA