UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO - UFES
CURSO CIÊNCIA DA COMPUTAÇÃO

MATHEUS DE OLIVEIRA LIMA
MATHEUS LOPES FERREIRA LIMA

Implementação de um algoritmo de agrupamento

VITÓRIA
2023

MATHEUS DE OLIVEIRA LIMA
MATHEUS LOPES FERREIRA LIMA

Trabalho de Técnicas de Busca e Ordenação: Implementação de um algoritmo de agrupamento

O objetivo deste trabalho é implementar um algoritmo de agrupamento, chamado de agrupamento de espaçamento máximo, utilizando uma MST (Minimum Spanning Tree), oferecida pela Universidade Federal do Espírito Santo para o Curso de Ciência da Computação.
Professor: Giovanni Ventorim Comarela.

Vitória
2023

1. INTRODUÇÃO

A problemática do trabalho é a necessidade de criar um algoritmo na linguagem C utilizando uma Árvore Geradora Mínima (MST) que cumpra os pré-requisitos da descrição do trabalho e seja eficiente ao ponto de executar em todos os casos de testes propostos pelo professor.

2. METODOLOGIA

Inicialmente, adotamos a leitura intermitente do arquivo de entrada até o final. Entretanto, encontramos dificuldades em determinar o tamanho inicial do vetor de double presente na estrutura "tPonto". Decidimos então ler a primeira linha do arquivo para definir a quantidade de dimensões do ponto e alocar uma estrutura "tPonto" enxuta em relação à memória e ao tempo. Dessa forma, evitamos o uso da função realloc, que tem alto custo em caso de necessidade de aumento do tamanho do vetor. Além disso, fizemos a contagem de linhas do arquivo de entrada para obter a quantidade exata de pontos, o que nos permitiu criar um vetor de pontos com tamanho preciso.

Com relação ao armazenamento dos vértices, inicialmente consideramos um vetor de pontos. Entretanto, percebemos que criar uma lista específica para as arestas seria muito custoso, pois exigiria a implementação de um algoritmo para ordenação. Decidimos então utilizar um vetor de arestas, que nos permitiu ordená-las facilmente com o uso da função "qsort()" do C.

O vetor de arestas foi inicialmente preenchido com todas as possíveis $((N^2 - N) / 2)$ e, em seguida, foi ordenado. Cada item do tipo "tAresta" contém a distância (dist), o ponto de origem (po) e o ponto de destino (pd). Inicialmente, "po" e "pd" eram do tipo "tPonto", mas decidimos substituí-los pelo tipo int para comparar mais rapidamente as arestas e seus pontos durante a seleção ou não das arestas que farão parte da Árvore Geradora Mínima (MST).

Após obtermos o vetor de arestas ordenado, executamos o algoritmo de Kruskal para formar a MST. Entretanto, em determinado momento, seria necessário retirar k(número de grupos)-1 arestas em ordem decrescente de distância para a obtenção dos grupos de espaçamento máximo. Percebemos que se o algoritmo de Kruskal for parado em n(número de vértices) - k(número de grupos) que corresponde ao número de arestas da MST(n-1) - o número de arestas que devem ser retiradas (k-1), a estrutura resultante já mostrará os grupos. Utilizamos o quick union com path compression para unir os pontos de forma eficiente e um vetor de inteiros representando cada ponto para que possamos mudar cada posição à medida que a união aconteça. Nesse algoritmo, a união entre dois elementos é representada pela troca de raízes de cada elemento. Dessa maneira, ao final teremos cada elemento

associado a uma das k raízes, que são identificadas quando um índice do vetor é igual ao inteiro daquela posição.

Para a impressão dos grupos, adotamos a estratégia de agrupar os elementos em listas encadeadas, onde cada lista corresponde a um grupo. Como o vetor de pontos já foi ordenado no início do programa, a lista correspondente a cada grupo já estará ordenada também. Em seguida, inserimos cada lista em uma posição específica de um vetor de listas, que tem o tamanho da quantidade total de pontos. Essa abordagem nos permite acessar cada lista de forma eficiente com custo O(1), de acordo com a posição no vetor correspondente. Então, com os IDs dos pontos ordenados nas listas, ainda é necessário ordenar o vetor de listas de acordo com o primeiro elemento de cada uma, garantindo que a forma de impressão requerida seja atendida. Finalmente, os grupos são impressos no arquivo de saída e o programa é encerrado.

3. ANÁLISE DE COMPLEXIDADE

As primeiras 40 linhas do arquivo "trab1.c" realizam a leitura com complexidade O(N), onde N é a quantidade de linhas do arquivo de entrada.

Em seguida, a função "OrdenaVetorPontos" chama a função "qsort()" para ordenar um vetor de pontos, cuja complexidade é $O(N^2)$ no pior caso e O(N*log(N)) no melhor e médio caso, onde N é a quantidade de pontos no vetor.

Posteriormente, a função "PreencheVetArestas" tem complexidade $O(N^2)$ e logo em seguida, o vetor de arestas é ordenado pela função "OrdenaVetArestas", que utiliza internamente a função "qsort()" com complexidade $O(N^2)$ no pior caso e O(N*log(N)) no melhor e médio caso, sendo N a quantidade de arestas no vetor.

A função "AlgoritmoKruskal" tem complexidade O(N+M*log(N)) no pior caso, sendo M o número de operações de "union-find" e N a quantidade de acessos no array. A complexidade deste algoritmo depende principalmente da técnica de "Quick-Union + path compression".

Por fim, a função "MontaGrupos" tem complexidade O(MN), onde M é a quantidade de pontos e N é a complexidade da função "UF_find". No entanto, a função "OrdenaGrupos" é a que tem maior complexidade, já que utiliza a função "qsort()" para ordenação, cuja complexidade é $O(N^2)$ no pior caso e O(Nlog(N)) no melhor e médio caso, sendo N a quantidade de elementos a serem ordenados.

4. ANÁLISE EMPÍRICA

Análise do in-exemplos/4.txt:

Leitura dos dados:	0.004828 segundos.
Cálculo das distâncias:	0.515409 segundos.
Ordenação das distâncias:	0.944671 segundos.
Obtenção da MST:	0.004469 segundos.
Identificação dos grupos:	0.000300 segundos.
Escrita do arquivo de saída:	0.000301 segundos.
Total tempo:	1.469978 segundos.

Análise do in-exemplos/5.txt:

Leitura dos dados:	0.018933 segundos.
Cálculo das distâncias:	3.844428 segundos.
Ordenação das distâncias:	4.073367 segundos.
Obtenção da MST:	0.009282 segundos.
Identificação dos grupos:	0.000511 segundos.
Escrita do arquivo de saída:	0.000583 segundos.
Total tempo:	7.947104 segundos.

PORCENTAGEM	Análise do in-exemplos/4.txt	Análise do in-exemplos/5.txt
Leitura dos dados	0.32%	0.23%
Cálculo das distâncias	35.06%	48.37%
Ordenação das distâncias	64.26%	51.25%
Identificação dos grupos	0.02%	0.006%
Escrita do arquivo de saída	0.02%	0.007%

Os valores obtidos nessa tabela estão de acordo com a análise de complexidade.

5. CONCLUSÃO

Ao finalizar o trabalho, é perceptível que as escolhas dos algoritmos são de suma importância para a eficiência de um programa. Principalmente os de complexidade inferior a O(N*logN) para casos maiores são essenciais para um código mais rápido. Além disso, notamos que a escolha correta das estruturas de dados gera um impacto significativo no consumo de memória e o raciocínio, análise e planejamento da abordagem de resolução é importante para codificar uma solução eficiente.

6. REFERÊNCIAS BIBLIOGRÁFICAS

https://ava.ufes.br/course/view.php?id=16990