Henrique Coutinho Layber

Agrupamento baseado em árvore geradoras mínimas

Vitória, ES
2020/2 EARTE

Henrique Coutinho Layber

Agrupamento baseado em árvore geradoras mínimas

Relatório apresentado como parte do primeiro trabalho prático da disciplina Estrutura de Dados II, ministrada por Dr. Giovanni Comarela pela Universidade Federal do Espírito Santo.

Universidade Federal do Espírito Santo

Centro Tecnológico

Departamento de Informática

Vitória, ES

2020/2 EARTE

Sumário

Sumário **2**

1 METODOLOGIA **4**
1.1 Estruturas **4**
1.2 Fluxo do Programa **5**
1.2.1 Leitura do arquivo de entrada 5
1.2.2 Cálculo das distâncias 5
1.2.3 Ordenação das distâncias 5
1.2.4 Cálculo da MST (Kruskal) 6
1.2.5 Identificação dos grupos 6
1.2.6 Escrita do arquivo de saída 6

2 ANÁLISE DE COMPLEXIDADE **7**
2.1 Cálculo das distâncias **7**
2.2 Ordenação das distâncias **7**
2.3 Cálculo da MST (Kruskal) **7**
2.4 Identificação dos grupos **7**

3 ANÁLISE EMPÍRICA **8**

Introdução

Este documento apresenta o relatório de resultados da implementação do algoritmo descrito na especificação do primeiro trabalho prático de Estrutura de Dados II (Trabalho), que visa solucionar o problema de agrupamento genérico de espaçamento máximo com base em árvores geradoras mínimas (*Minimum Spanning Tree* ou MST).

O conteúdos dos seguintes capítulos é como segue:

- O capítulo 1 contém a metodologia utilizada, incluindo descrições e justificativas das estruturas de dados e principais decisões durante a implementação;
- O capítulo 2 contém as análises de complexidades das principais funções para a solução do problema;
- O capítulo 3 contém as análise empírica das principais funções

1 Metodologia

A ideia foi implementar usando structs e ponteiros de uma maneira que fique intuitivo e fácil de acessar a informação necessária, mas mantendo a separação de dados de cada struct.

1.1 Estruturas

As principais estruturas usadas na implementação são, na ordem em que são instanciadas:

- `sample_t` contém os dados do arquivo, já tipados de acordo. Seu conteúdo é qual a linha (amostra) no arquivo em que o identificador aparece (`index`), o identificador em si (`id`) e um array com os valores atribuídos a ele (`features`).

- `dataSet_t` representa uma matriz de entrada, e é composto simplesmente de um array de `sample_t`, e informações de quantas amostras ele guarda e quantas *features* cada um tem (todos têm a mesma quantidade de *features*).
  Na implementação, ele é instanciado como um array assim dando acesso tanto aos valores das `features` quanto às informações da amostra.

- `distanceSample_t` contém informações relativa ao cálculo das distâncias. Ela armazena as duas amostras qual a distância relativa é calculada e o valor da distância.

- `distanceDataSet_t` está para `distanceSample_t` como o `dataSet_t` está para o `sample_t`. Ao invés de armazenar o número de *features*, armazena a quantidade de linhas (é uma matriz quadrada) e a quantidade de elementos (para evitar recálculo). Ele representa na realidade uma matriz, mas uma triangular $-I$ de distâncias entre cada duas amostras diferentes.

- `unionCell_t` é o vértice do grafo, e tem sua definição na TAD `unionFind`. O `unionFind` foi implementado com ponteiros, permitindo que não seja necessário fornecer o array de raízes juntamente com o elemento que quer verificar ou unir, e como o algoritmo independe do tamanho de vértices (não é uma informação que o TAD precisa saber para operar), foi possível remover o array separado completamente.
  Ela contém um ponteiro que aponta para o pai (`NULL` se não houver), o tamanho da árvore para balancear as uniões, e um ponteiro para uma `distanceSample_t` e uma `sample_t`, referente ao cálculo da `MST`.

Quanto a separação das structs, foi definida de modo que evite "acesso direto" para que o programa no final seja mais intuitivo de acessar valores iniciais. Por exemplo, a distância não carregaria o índice de cada amostra origem, mesmo que só use isto, pois no ciclo de desenvolvimento seria fácil que essa ideia seja perdida ou difusa, ao invés disso ele guarda uma referência para cada amostra origem, e então pode acessar os campos das amostras.

1.2 Fluxo do Programa

1.2.1 Leitura do arquivo de entrada

Sabendo o caminho do arquivo de entrada, abrindo ele, e usando as funções na biblioteca `file.h`, é feita uma chamada para contagem de linhas `countLines`. Sabendo o tamanho da entrada, pode-se iniciar o `dataSet_t*` de acordo, e, para cada índice desde, fazer uma chamada para `readLine`, que retorna a linha do arquivo de entrada tratada como um array de `char*`. Foi determinado que não era parte do utilitário tratara a conversão, então isso é feito na função de *parsing* `loadData`.

1.2.2 Cálculo das distâncias

Obtido o `dataSet_t`, pode-se supri-lo à função `calculateDistances`, que cria um `distanceDataSet_t` e o popula de acordo, calculando a distância em cada duas amostras, formando uma matriz triangular inferior esquerda sem a diagonal, e portanto contendo $\frac{i \cdot (i-1)}{2}$ elementos ao todo.
Aqui, a função de distância Euclidiana foi modificada para poupar tempo de processamento. Como o algoritimo de *Kruskal* não usa os valores de distância (só importa qual é menor ou maior), contanto que a função continue proporcional (o menor continue sendo o menor e o maior continue sendo o maior). Assim, pode-se remover o cálculo da raiz quadrada, e com isso a biblioteca `math.h` e a *flag* de compilação `-lm`. Após, pode-se remover o expoente à segunda ordem pelo mesmo motivo, os valores menores continuam sendo os menores e os maiores continuam sendo os maiores, mas somente em grandeza. Então a função de distância final ficou como simplesmente $|a - b|$, que pode eficientemente ser resolvido com um `if`.

1.2.3 Ordenação das distâncias

Com as distâncias obtidas, pode-se ordenar elas com a ajuda do `qsort`, usando a distância para comparação de duas distâncias `compareDistanceSamples`. É necessário ordenar por distâncias ascendente para o algoritmo de Kruskal.

1.2.4 Cálculo da MST (Kruskal)

Usando o vetor de distâncias ordenados, é iniciado um grafo de tamanho do número de distâncias usando as primeiras N dessas distâncias e apontando para suas amostras correspondentes, e com a ajuda do TAD `unionFind.h`, é simples construir a árvore geradora mínima. Um truque para otimização é não contruir a MST inteira e depois tirar o $K-1$ maiores, mas construir a MST até ter K raízes (*clusters*), o resultado será o mesmo e não será necessário procurar a maior aresta três vezes.

1.2.5 Identificação dos grupos

Como foi contruído propositalmente na etapa de estruturas de dados, o `unionCell_t` aponta para uma amostra e pra uma raíz, então basta ordenar a MST pelo id da amostra da raíz, e em caso de empate, pelo id da amostra de si mesma. Então o vetor estará ordenado do jeito correto para impressão, basta apenas inserir as quebras de linhas quando o id da amostra da raíz mudar.
Importante notar que, como é usado ponteiros, quando ordenar o vetor, o endereço das raízes pode trocar, por estar alterando o mesmo vetor. Então, antes de ordenar copia-se todos as raízes para um array separado e faz-se todos os descendentes apontarem para este novo endereço. As raízes das raízes continuam apontando para `NULL`.

1.2.6 Escrita do arquivo de saída

Com a MST conforme descrito, e com as informações de árvore de cada *cluster*, pode-se usar o tamanho da raiz cada árvore, partindo do primeiro, e será escrito todos o de um único *cluster*. Após, basta inserir quebras de linhas e usar as informações da próxima raíz.

2 Análise de complexidade

2.1 Cálculo das distâncias

Seja $G(V, E)$ um grafo com N vértices, e sabendo que é necessário calcular a distância em todas as combinações de 2 vértices diferentes, nos daria $N \cdot (N-1)$, mas como sabemos que $\|N_i N_j\| \equiv \|N_j N_i\|$, não é necessário calcular metade das distâncias, o que nos dá o tamanho final de $fracN \cdot (N-1)2$ distâncias totais a serem calculadas. Sendo M a quantidade de *features* de um vértice, temos que será preciso calcular $frac(N \cdot M) \cdot ((N \cdot M) - 1)2$ distâncias.

2.2 Ordenação das distâncias

Consiste na implementação do *quickSort* no `qsort` disponível na `stdlib.h`, que tem custo $\sim N \log_2 N$ e $O(N \log_2 N)$ também, pois faz uma etapa para evitar a degeneração.

2.3 Cálculo da MST (Kruskal)

Sabendo que o Kruskal tem complexidade $O(E \log_2 V)$ÿ e que utilizamos somente $fracN \cdot (N-1)2$ distâncias, temos que o custo é $O(\frac{N^2-N}{2} \log_2 N) \Rightarrow O(\frac{N^2 \log_2 N}{2} - \frac{N \log_2 N}{2}) \Rightarrow O(N^2 \log_2 N)$.

2.4 Identificação dos grupos

Sendo necessário percorrer o MST duas vezes (uma para anotar as raízes, outra para atualizar os descendentes), e depois realizando um `qsort`, o custo é $O(2N + N \log_2 N) \Rightarrow O(N \log_2 N)$.

3 Análise empírica

O código foi executado em um computador equipado com um processador Intel®Core™i7 3770 e 16GB de memória RAM em dual-channel. Cada entrada foi executada 10 vezes e o valor exibido corresponde à média destas 10 vezes. O relógio para total é feito da contagem a partir do início do programa, e não da soma das demais e as porcentagens foram arredondadas para 2 casa decimais e podem não totalizar 100%.

	Leitura	Distâncias	Ordenação	MST	Agrupamento	Escrita	Total
Entrada 1	0.000195 2,25%	0.000049 0,56%	0.000368 4,25%	0.000029 0,33%	0.000017 0,19%	0.000175 2,02%	0.000864
Entrada 2	0.000342 13,89%	0.000178 7,23%	0.001627 66,11%	0.000114 5,85%	0.000034 1,38%	0.000113 4,59%	0.002461
Entrada 3	0.001759 0,91%	0.012047 6,27%	0.167831 87,47%	0.007767 4,04%	0.000354 0,18%	0.000203 0,10%	0.191856
Entrada 4	0.006742 0,05%	0.090215 6,77%	1.177708 88,41%	0.045467 3,41%	0.000863 0,06%	0.000329 0,02%	1.332072
Entrada 5	0.021688 0,37%	0.512253 0,87%	5.119178 87,36%	0.164390 2,80%	0.001938 0,03%	0.000680 0,01%	5.859441

Tabela 1 – Tempos de execução por etapa por entrada (em segundos)

Observando as entradas 3 ($N = 50, M = 2$), e 5 ($N = 1000, M = 2$), por terem o mesmo M:

- O cálculo das distânciae a ordenaçã