Relatório 2º projecto ASA 2022/2023

Grupo: AL059

Aluno(s): Tiago Deane (103811) e Artur Krystopchuk (104145)

Descrição do Problema e da Solução

Na nossa solução, decidimos usar uma versão alterada do algoritmo de Kruskal que percorre as arestas por ordem decrescente dos pesos e soma estes pesos ao resultado final, ao invés de adicionar estas arestas à uma *tree*. De forma resumida, é um algoritmo que determina o peso de uma *Maximum Spanning Tree*. Sendo os vértices regiões e os pesos o valor de trocas comerciais, este processo vai nos devolver o valor máximo de trocas comerciais entre regiões, minimizando os custos infraestrutura (número de arestas).

No nosso algoritmo, guardamos os vértices e as arestas em vetores e organizamos as arestas por ordem decrescente dos seus pesos. A seguir, analisamos as arestas do vetor, e se estas não formarem um loop, é acrescentado o seu peso ao resultado final. A forma como verificamos se uma aresta não forma um ciclo com outra é por guardar em cada vértice o seu pai e o seu rank; podemos determinar se uma aresta forma um loop com outra se, ao percorrer o caminho de pais dos vértices da aresta, os dois vértices têm o mesmo predecessor final. Para melhor performance, utilizamos a estrutura Union-Find. Para a análise teórica, nota-se que cada vértice é guardado no índice correspondente do vetor "vertices", isto é, o vértice "1" é guardado no índice 1, o vértice "2" no índice 2, e assim em diante.

Análise Teórica

1. Pseudocódigo:

Nota: não vamos incluir as funções do Union-Find, uma vez que elas foram apresentadas nas aulas teóricas.

```
WeightComparator(e1, e2)
   return e1.weight > e2.weight

AdaptedKruskal(G.E, G.V)
   sum = 0
   // Nota: já foi executado MakeSet em todos os vértices
   sortedEdges = sort(G.E, WeightComparator)
   for each (u, v) ∈ sortedEdges do
      if FindSet(u) ≠ FindSet(v) then
         sum += weight of (u, v)
         TreeUnion(u, v)
      end if
   end for
   return sum
```

2. Complexidade
   - Leitura dos dois primeiros inputs (O(1)), e *resize* dos vetores onde vamos guardar os vértices e arestas. Logo, O(V) + O(E).
   - Execução de *Make-Set* (O(1)) em todos os vértices: O(V)
   - Leitura dos valores de cada aresta, com ciclo que corre "E" vezes. Logo, O(E)

Relatório 2º projecto ASA 2022/2023

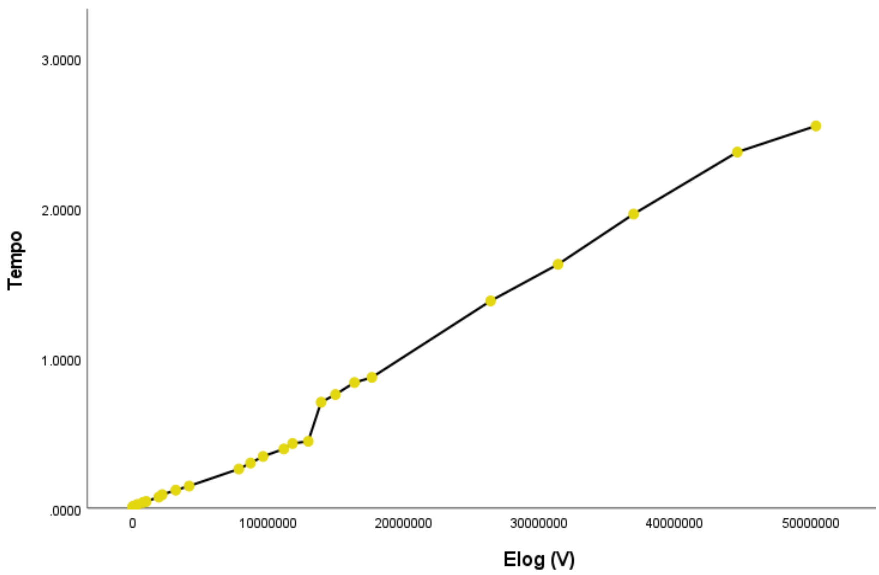
Grupo: AL059

Aluno(s): Tiago Deane (103811) e Artur Krystopchuk (104145)

- Aplicação do algoritmo AdaptedKruskal:
  - Usamos as funções *sort* (com o *weightComparator*) para ordenar as edges por ordem decrescente dos pesos, algo que leva O(Elog(E)) tempo.
  - Percorremos todas as edges, tendo assim O(E) operações de *FindSet* e *TreeUnion*. O *Find-Set* tem, no pior caso, complexidade de O(log(V)), e *TreeUnion* vai ser sempre O(1) (apesar de o *TreeUnion* conter as funções *FindSet*, estas terão complexidade O(1), uma vez que já foram corridas uma vez e o parent dos vértices já é diretamente a raiz das suas árvores). Logo, este ciclo é O(Elog(V)).
- Apresentação dos dados. O(1)

Complexidade global da solução: O(Elog(V)).

Avaliação Experimental dos Resultados



Quando o eixo do tempo varia 1 unidade de desvio padrão, o Elog(V) varia 0.995, ou seja, eles têm uma relação linear quase perfeita. Tal como podemos verificar no gráfico, o tempo cresce de forma linear em relação à Elog(V). Assim, podemos concluir que a nossa implementação está de acordo com a análise teórica de O(Elog(V)).

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.	Collinearity Statistics	
		B	Std. Error	Beta			Tolerance	VIF
1	(Constant)	-.056	.021		-2.695	.013		
	Elog (V)	5.266E-8	.000	.995	48.083	<.001	1.000	1.000

a. Dependent Variable: Tempo