

Trabalho Final

Deyvid W. S. Medeiros¹, Guilherme E. D. Silva², Nathan Vitor de Lima³

¹ Instituto Metrópole Digital – Universidade Federal do Rio Grande do Norte (UFRN)
Caixa Postal 15.24 – 59.078-900 – Natal – RN – Brazil

nathan.lima.132@ufrn.edu.br, guilherme.euller.dantas.117@ufrn.edu.br,
deyvid.willian.017@ufrn.edu.br

Abstract. *This report aims to present the final work of the disciplines Estrutura de Dados Básicos II and Linguagem de Programação II. Basically, the objective of this project is to make the internet connection between houses in a certain neighborhood of a city in a way that generates the lowest cost. For this, we used the Kruskal and K-best algorithms as a basis for solving the problem.*

Resumo. *Este relatório visa apresentar o trabalho final das disciplinas de Estrutura de Dados Básicos II e Linguagem de Programação II. Basicamente, o objetivo do projeto é fazer a conexão de internet entre casas de um determinado bairro de uma cidade de forma que seja gerado o menor custo. Para isto, utilizamos como base de resolução do problema os algoritmos de Kruskal e K-best.*

1. Introdução

Como problema tem-se que “dada uma matriz $n \times n$ contendo o custo de se criar uma conexão entre duas casas quaisquer em Parnatal e um valor d , encontrar essa estrutura de conexão de menor custo (em forma de árvore) entre as n residências de Parnatal, de maneira que nenhuma residência tenha mais que d conexões diretas com outras residências”.

Para resolução deste, foram utilizados como base os algoritmos de Kruskal e K-best, e a estrutura de dados Conjunto Disjunto.

2. Descrição da abordagem de solução do problema

Para resolução do problema, tem-se duas abordagens, as quais foram chamadas de “Processador Simples” e “Processador Complexo”. A qual o “Processador Simples” verifica todas as combinações de ligações possíveis, e em seguida sai verificando se o resultado é uma árvore válida, ou seja, com $n-1$ ligações, onde n é o número de casas, que não possui ciclos e que não haja nenhuma casa isolada da árvore final, lembrando que cada casa deve conter d ligações. Referente ao “Processador Complexo” tem-se que são pegadas todas as arestas e aplicado o Kruskal, gerando a “árvore geradora mínima” (MST) e depois, então, se verifica se atende aos requisitos, por exemplo, o valor “ d ”. Caso não seja a árvore que queremos, então essa é particionada e é aplicado novamente o método de Kruskal, de forma que isso ocorre até conseguirmos nossa MST, esse é o processo de K-best (particionar e aplicar o Kruskal).

2.1. Processador Simples

Verifica todas as combinações de arestas possíveis, então cada combinação se é válida (possui $n-1$ ligações, com n sendo o número de casas, não tem ciclos e não tem casas isoladas no resultado), e então seleciona a combinação com menor custo.

Para checar se uma determinada combinação é válida, primeiramente o programa gera um conjunto unitário para cada vértice da combinação, após isso, para cada aresta ele checa se os vértices já apresentam a quantidade máxima de arestas por vértice (d), checa se os representantes dos vértices no conjunto disjunto são os mesmos e caso não ocorram as condições apontadas anteriormente são unidos os conjuntos disjuntos, caso ocorram as condições a função retorna nulo. Ao final da união de todos os vértices ele checa se o representante é o mesmo para todos e retorna o custo total da combinação das arestas.

Sua complexidade é $O(2^n)$, já que checa uma a uma as combinações de $n-1$ arestas para n vértices.

2.2. Processador Complexo

Pega todas as arestas e aplica o algoritmo Kruskal, ignorando o máximo de arestas dos vértices, vulgo d , se essa melhor árvore gerada obedece a quantidade máxima de arestas " d ", então esse é o resultado final, caso contrário, ele irá particionar essa árvore gerando e aplicando o kruskal para cada partição. Após isso, ele irá pegar a árvore gerada com menor custo, verificando se ela é válida (se segue o máximo de ligações) se for válido esse é o resultado. Caso contrário, irá particionar essa nova árvore e irá realizar o mesmo procedimento repetidamente com as novas partições e as já encontradas até encontrar uma árvore válida, esse é o K-best. Nesse caso é retornada apenas uma árvore, mas o K-best poderia prosseguir gerando diversas novas árvores.

Sua complexidade é $O(n \log n + n^2)$.

2.2.1. Kruskal

```
algorithm Kruskal(G) is
  F := ∅
  for each v ∈ G.V do
    MAKE-SET(v)
  for each (u, v) in G.E ordered by weight(u, v), increasing do
    if FIND-SET(u) ≠ FIND-SET(v) then
      F := F ∪ {(u, v)} ∪ {(v, u)}
      UNION(FIND-SET(u), FIND-SET(v))
  return F
```

Figura 1. Pseudocódigo do algoritmo de Kruskal

2.2.2 K-best

```
List = {A}
Calculate_MST (A)
while MST  $\neq \emptyset$  do
    Get partition  $P_s \in \text{List}$  that contains the smallest spanning tree
    Write MST of  $P_s$  to Output_File
    Remove  $P_s$  from List
    Partition( $P_s$ ).
```

Figura 2. Pseudocódigo do K-best

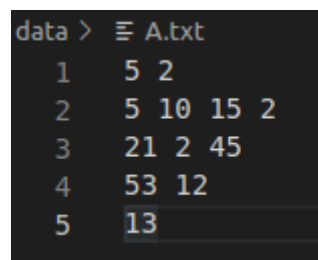
```
PROCEDURE PARTITION ( $P$ )
 $P_1 = P_2 = P$ ;
for each edge  $i$  in  $P$  do
    if  $i$  not included in  $P$  and not excluded from  $P$  then
        make  $i$  excluded from  $P_1$ ;
        make  $i$  included in  $P_2$ ;
        Calculate_MST ( $P_1$ );
        if Connected ( $P_1$ ) then
            add  $P_1$  to List;
 $P_1 = P_2$ ;
```

Figura 3. Pseudocódigo do particionamento

2.3 Entrada e saída de dados

2.3.1 Entrada de dados

O programa recebe como um dos parâmetros de entrada o endereço do arquivo contendo um grafo, neste arquivo a disposição das informações é como apresentada no exemplo abaixo:



```
data > ≡ A.txt
1    5 2
2    5 10 15 2
3    21 2 45
4    53 12
5    13
```

Figura 4. Exemplo de arquivo de entrada

O programa checa se é possível abrir o arquivo, se ele não está vazio, se todas as ligações entre as casas foram informadas, se a primeira linha do arquivo é válida, contendo o número de casas e a quantidade máxima de ligações por casa, e se o número de ligações informadas é válido.

2.3.2 Saída de dados

Na saída o resultado é escrito no arquivo presente em “./data/resultado.txt”, lá é possível ver um resultado geral não gráfico onde dá para se tirar as informações de forma mais enxuta como no exemplo abaixo.

```
#=====#  
|          COMBINAÇÃO COM MENOR CUSTO          |  
#=====#  
| Ligação entre as casas: 2 e 4; Custo: 1      |  
| Ligação entre as casas: 4 e 8; Custo: 4      |  
| Ligação entre as casas: 1 e 2; Custo: 6      |  
| Ligação entre as casas: 0 e 9; Custo: 12     |  
| Ligação entre as casas: 3 e 6; Custo: 12     |  
| Ligação entre as casas: 1 e 6; Custo: 13     |  
| Ligação entre as casas: 1 e 5; Custo: 15     |  
| Ligação entre as casas: 4 e 9; Custo: 22     |  
| Ligação entre as casas: 5 e 7; Custo: 31     |  
| O custo total é: 116                         |  
#=====#
```

Figura 5. Exemplo de arquivo de resultado

Além disso, durante a execução, o programa gera uma interface visual que apresenta o custo total do resultado e uma imagem contendo cada casa com sua numeração acima, cada ligação é representada por uma cor e acima dela acompanha o custo dela na mesma cor.

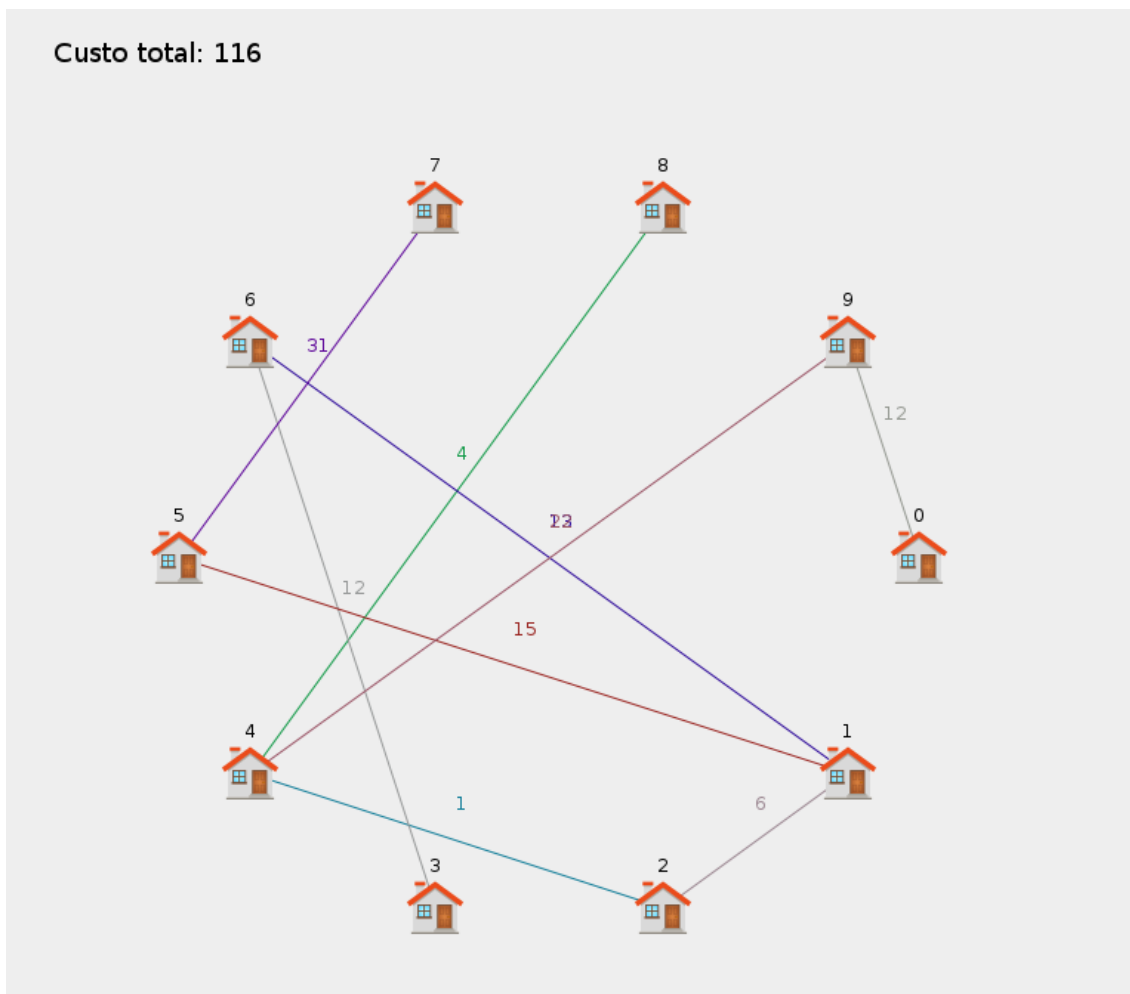


Figura 6. Exemplo de interface gráfica gerada pelo programa

3. Descrição geral das estruturas de dados utilizadas com explicação dos algoritmos utilizados

No geral, a estrutura de dados utilizada foi o Conjunto Disjunto. Isso se deu através do fato de que, graças a ele, era possível adicionar os valores (as casas, por assim dizer) de forma que não houvesse ligações circulares entre eles, podendo, assim, sanar a carência que era impedir que duas casas com a ligação para o mesmo pai se ligassem. Além disso, graças a este, foi possível impedir gerar mais de uma árvore no mesmo resultado, tendo em vista que o resultado poderia ser duas ou mais árvores separadas caso alguma casa ou várias ligações ficassem isoladas de forma que graças ao representante do conjunto disjunto nos garantiria a unicidade da árvore.

No que diz respeito ao âmbito do código, foi necessário apenas fazer um conjunto disjunto simples, onde temos o item, o pai e o ranking/tamanho e as funções de get e set padrões para os atributos da classe e temos as funções da própria estrutura de dados, como o union (verifica se podem juntar e se puder ele junta), link (olha qual dos dois é o maior e junta), find (encontra o representante) e o areMerged (verifica se os itens estão no mesmo conjunto).

4. Diagrama de Classes

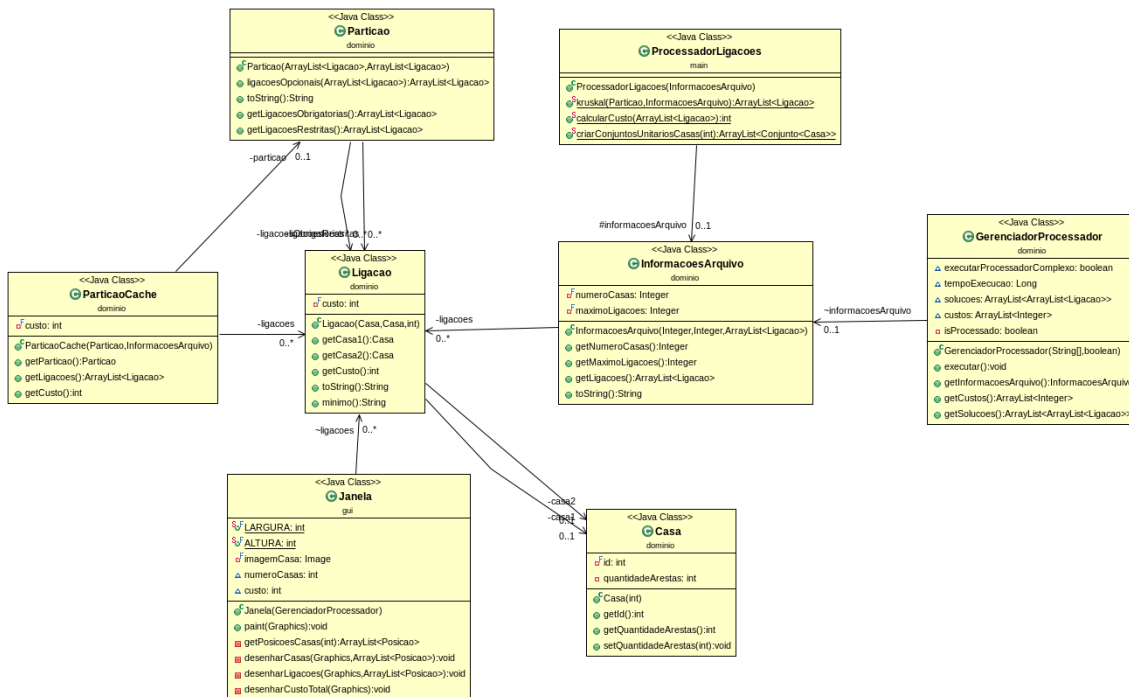


Figura 7. Diagrama de classes principais

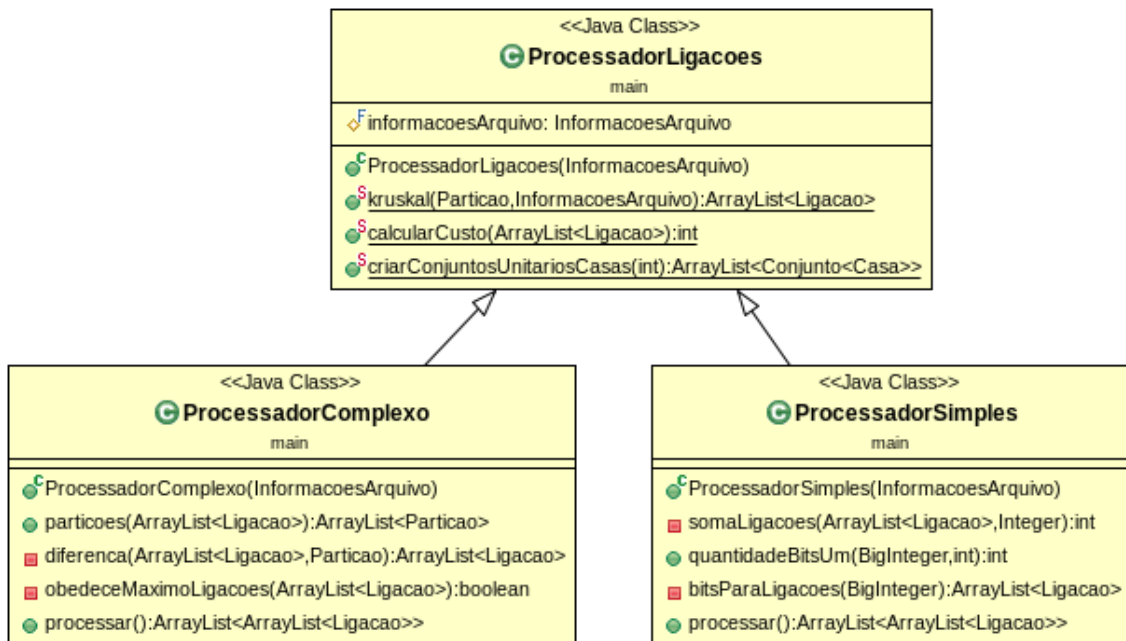


Figura 8. Diagrama das classes processadoras do problema

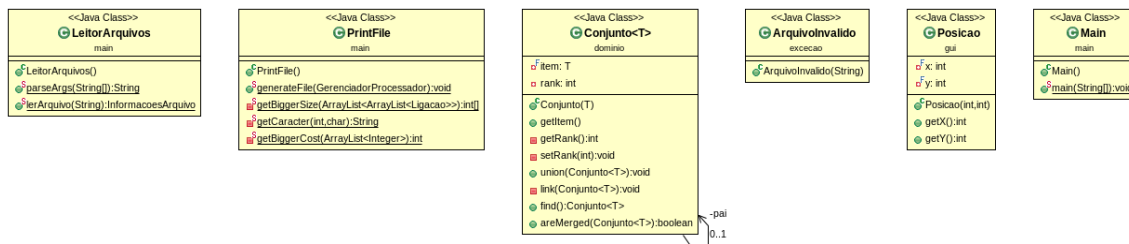


Figura 9. classes utilitárias

5. Conclusão

Pode-se concluir que por meio do K-best foi mais eficiente encontrar a árvore geradora mínima, MST. De fato, tendo em vista que por meio deste algoritmo não foi necessário gerar todas as combinações como na primeira forma “Processador Simples”. E com isso foi possível atingir o objetivo do projeto que era fazer a conexão de internet entre casas de um determinado bairro de uma cidade em um tempo bem menor do que seria utilizando todas as possibilidades de conexões.

6. Referências

- Sörensen, K. and Janssens, G. K. (2005) “AN ALGORITHM TO GENERATE ALL SPANNING TREES OF A GRAPH IN ORDER OF INCREASING COST”, <https://www.scielo.br/j/pope/a/XHswBwRwJyrfL88dmMwYNWp/?format=pdf&lang=en>
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 23.2: The algorithms of Kruskal and Prim, pp. 567–574.
- Michael T. Goodrich and Roberto Tamassia. Data Structures and Algorithms in Java, Fourth Edition. John Wiley & Sons, Inc., 2006. ISBN 0-471-73884-0. Section 13.7.1: Kruskal's Algorithm, pp. 632.
- (2022) “Kruskal's algorithm”, https://en.wikipedia.org/wiki/Kruskal%27s_algorithm