*Projeto e Análise de Algoritmos: INF 2926*

Pontifícia Universidade Católica do Rio de Janeiro

Professor: Eduardo Sany Laber Trabalho de Implementação

Alunos: Leonardo Quatrin Campagnolo – 1312529 Data: 12 de Junho de 2013

Leonardo de Paula Batista Benevides – 1312379

# Objetivo:

Implementar e analisar os algoritmos de Kruskal e Prim, analisar os resultados obtidos com cada um dos grafos de entrada e gerar uma análise a partir das implementações realizadas.

# Implementação e técnicas escolhidas:

Para fins de melhorar a estrutura das tabelas contidas em cada anexo e das análises, cada um dos algoritmos foi enumerado para serem indicados nas tabelas, como segue:

* Algoritmo 1: Algoritmo Kruskal utilizando o heap sort e a estrutura union-find (utilizando as heurísticas de union by rank e path compression);
* Algoritmo 2: Algoritmo Kruskal utilizando o counting sort e a estrutura union-find (utilizando as heurísticas de union by rank e path compression);
* Algoritmo 3: Algoritmo Prim utilizando a fila de prioridade sobre as arestas;
* Algoritmo 4: Algoritmo Prim utilizando a fila de prioridade com a operação change-key sobre os vértices.

O desenvolvimento dos algoritmos foi realizado em C++, utilizando a IDE Microsoft Visual Studio 2012 e 2008. Para cada um deles, foram utilizadas as seguintes estratégias:

## Decisões de implementação para os algoritmos de Kruskal:

Para os algoritmos de kruskal, foi implementada a estrutura union-find, utilizando heurísticas de union by rank e path compression. A estrutura implementada consiste em representar cada nó da estrutura union-find, sendo cada um deles um dos vértices do grafo de entrada. Para cada nó, é guardado o vértice que o nó representa, um ponteiro para um nó pai e um valor de altura, representando a altura de sua sub árvore.

Esta estrutura de ponteiros foi utilizada para minimizar a complexidade de encontrar um nó dentro da estrutura union-find, visto que o acesso para um nó dentro da estrutura pode ser dado em tempo constante, ao invés de ter uma estrutura de índices em um vetor.

O gasto maior de tal estrutura é relacionada à operação **Find\_set()**, porém o custo é amenizado quando utilizada a estratégia de path compression, que atualiza o ponteiro de cada nó para o representante do seu respectivo conjunto. Dessa forma, o representante é acessado em tempo constante.

Para fins de avaliar a diferença entre implementações, a estrutura Union-Find foi desenvolvida de duas maneiras: em uma delas cada vértice é representado por um nó que possui um ponteiro para um nó pai, na outra implementação, cria-se um vetor *sets[]* e, para cada vértice i procurado, o valor de *sets[]* será positivo se este não for a cabeça de um conjunto, portanto irá dizer em qual posição está o pai do vértice i, ou terá um valor negativo, correspondente à altura do conjunto.

O Anexo J mostra a diferença de tempo contabilizada entre a implementação da estrutura Union-Find utilizando vetores e a mesma estrutura utilizando ponteiros. Uma das hipóteses pelo aumento da constante de tempo na estrutura de ponteiros foi a necessidade de instanciar cada nó da estrutura Union-Find utilizando ponteiros, o que aumenta o custo total se for comparado com a inicialização de um vetor. Para realizar as comparações entre as duas estruturas foi necessário aumentar a quantidade de vezes que cada algoritmo é feito, para que fosse obtido um valor de tempo relevante. Para cada estrutura, o algoritmo kruskal foi repetido 30 vezes, descartando os tempos gastos com ordenação, focando apenas na parte funcional do algoritmo kruskal onde cada estrutura atua.

Algoritmo 1: Kruskal utilizando o heapsort e a estrutura union-find (utilizando as heurísticas de union by rank e path compression).

Complexidade prevista: . Segue abaixo o pseudo código implementado:

**Kruskal\_HeapSort(G())**

* Ordenação pelo HeapSort:
* Para cada aresta do grafo:
  + 2 operações **Find\_set()**, uma para extremidade da aresta:
  + Operação Union:

Para realizar o Heapsort foram utilizadas as funções **make\_heap()** e **sort\_heap()** da biblioteca STL. Para verificar a eficiência do algoritmo, foi realizada uma série de medições para verificar sua complexidade. Os resultados podem ser visualizados no gráfico abaixo:

Algoritmo 2: Kruskal utilizando o counting sort e a estrutura union-find (utilizando as heurísticas de union by rank e path compression).

Complexidade prevista: . Segue abaixo o pseudo código implementado:

**Kruskal\_CountingSort(G())**

* Ordenação pelo Counting Sort:
* Para cada aresta do grafo:
  + 2 operações **Find\_set()**, uma para extremidade da aresta:
  + Operação Union:

Para realizar a ordenação das arestas foi implementada a função **CoutingSort()**, realizandos as ordenações em onde é definido como o maior peso encontrado dentre as arestas de um grafo.

O Gráfico abaixo mostra a complexidade medida do algoritmo Counting Sort implementado, tendo como valor máximo (, por exemplo) sendo igual a 1000.

## Decisões de implementação para os algoritmos de Prim:

Para os algoritmos de prim, foi implementada a estrutura **Heap\_min**, e também foi utilizada a estrutura priority queue contida na biblioteca STL de C++. Foi decidido implementar uma própria estrutura de heap visto a necessidade de realizar mais testes relacionados ao processamento interno da estrutura, o que não se tem acesso quando se usa uma estrutura pronta. O **Heap\_min** foi implementado *in-place*, ou seja, utilizando vetores. Para não ocorrer problemas de alocação de memória, foi utilizada a estrutura **vector** da biblioteca STL, que consiste em um vetor que aloca memória dinamicamente, a partir da inserção de novos elementos no vetor feitas pelo programa.

Outra decisão tomada na hora da implementação foi a não utilização de recursão nas funções de *heapfy*, visto que isto geraria um delay adicional por causa da troca de contexto realizada entre chamadas de função, além da necessidade de empilhar e desempilhar funções.

Algoritmo 3: Prim utilizando a fila de prioridade sobre as arestas.

Complexidade prevista: . Segue abaixo o pseudo código implementado:

**Prim\_Edges(G())**

* Ordenar:
* Para cada aresta do grafo:
  + Pegar a menor aresta:
  + Adicionar na MST caso a propriedade de corte:

Para este algoritmo foi utilizado inicialmente a estrutura *Priority\_queue* da biblioteca STL disponível em C++, que possui todos os comportamentos de uma fila de prioridade. Para cada nó do heap, foram armazenados 3 valores: as duas extremidades de uma aresta e o peso da mesma. Dessa forma, era utilizado o valor do peso de cada aresta como chave para realizar as comparações.

Para cada nova aresta adicionada, a estrutura utilizada mantém a propriedade do heap, posicionando a nova aresta ono lugar correto.

Para verificar quais vértices já haviam sido adicionados na árvore geradora mínima resultante, foi utilizado um vetor booleano ‘S’ com tamanho igual ao número de vértices do grafo de entrada, onde para cada valor S[v], significa se o vértice v já foi adicionado ou não na árvore geradora mínima. Dessa forma, a consulta de cada vértice em ‘S’ é feita em tempo constante.

Visto o interesse em contabilizar a quantidade de operações de heapfy que o algoritmo executaria para cada entrada, foi utilizada a classe **Heap\_min** com uma pequena modificação para mapear cada nó do heap como uma aresta de um grafo. Esta outra abordagem se mostrou levemente mais custosa em relação às constantes, porém possibilitou testes mais internos em relação ao comportamento do algoritmo para cada entrada fornecida. O Anexo F ilustra as medições realizadas com o algoritmo Prim utilizando a estrutura *Priority\_queue* da biblioteca STL e a classe Heap\_min que foi implementada. Para cada estrutura, o algoritmo foi rodado 10 vezes e foi tirada a média dos tempos.

Desta forma, a classe **Heap\_min** implementada foi utilizada para realizar testes internos (quantidade de operações heapfyup e heapfydown, por exemplo), mas para calcular os tempos deste algoritmo, foi utilizada a estrutura *Priority\_queue* disponível pela biblioteca STL, pois gerou resultados mais otimizados.

Algoritmo 4: Prim utilizando a fila de prioridade com a operação change-key sobre os vértices.

Complexidade prevista: , para um grafo . A implementação foi feita da seguinte maneira:

**Prim\_Vertex(G())**

* Ordenar heap com vértices e seus respectivos pesos:
* Para cada vértice com menor grau:
  + Adicionar na MST caso a propriedade seja válida (componente de um vértice ser diferente da componente do outro):
  + Atualizar os custos de cada um dos vértices vizinhos:

Para este algoritmo, foi implementada uma variação da estrutura **Heap\_min**, manipulando cada posição do heap como um nó do heap. Para manipular os custos de cada nó do heap foi implementada a função **decrease\_key()**, que busca um nó no heap e diminui o custo dele para um valor passado como parâmetro.

Para minimizar o custo da busca de um nó em um heap, foi adicionado à estrutura um vetor auxiliar para guardar as posições de cada vértice do grafo dentro do heap. Dessa forma, o acesso a cada vértice dentro do heap custa O(1), pois para cada vértice do grafo, sua posição no heap passa a ser acessada semelhante a uma função hash, onde para cada vértice i, há uma referência a sua posição no heap dentro do vetor auxiliar de posições na posição i.

# Resultados obtidos e Análise:

Os testes foram realizados em um computador com processador Intel Core 2 Quad, 2,66 GHz e 4 GB de memória ram. As especificações dos grafos em relação à densidade, número de vértices e range de custo encontram-se no Anexo E.

Foram inicialmente contabilizados os tempos de cada algoritmo. O tempo gasto com a leitura do arquivo de entrada foi descartado, focando no gasto em instanciar as estruturas necessárias (heap, vetor), operações de ordenação e construção da árvore geradora mínima. Inicialmente havia se decidido descartar o gasto em instanciar os objetos, porém depois foi decidido não descartá-lo, visto que os resultados de cada algoritmo implementado também dependem do tempo que é gasto alocando e desalocando objetos e, em termos práticos, o tempo de um algoritmo para instanciar os objetos que serão manipulados também deve ser levado em conta. As tabelas presentes no Anexo A ilustram os tempos (em milissegundos) dos algoritmos contabilizados para cada entrada. Cada algoritmo foi computado 10 vezes e foi retirada a média dos tempos.

Foi constatado que, para os grafos com uma grande quantidade de arestas, os tempos computados nos algoritmos 1 e 3 foram superiores aos algoritmos 2 e 4, devido às ordenações realizadas a partir da quantidade de arestas. No caso do algoritmo 2, esse tempo de processamento é diminuído através da utilização do Counting Sort e, no caso do algoritmo 4, a ordenação é realizada tendo como base um heap dos vértices, diminuindo a quantidade de nós da estrutura e, consequentemente, a altura do heap.

Além disso, quanto mais denso o grafo, mais lento os algoritmos 1 e 3 são em relação aos algoritmos 2 e 4. Isso ocorre porque, quanto mais denso o grafo, mais arestas o mesmo tem se for comparado com o número de vértices. No caso do algoritmo 2, a otimização ocorre pelo uso do Counting Sort para ordenar as arestas.

Os bons resultados do algoritmo 4 estão diretamente ligados ao caminho máximo que um nó dentro do heap irá percorrer em cada operação de heapfy. Enquanto os algoritmos 1 e 3 realizam no máximo trocas e no mínimo trocas, o algoritmo 4 realiza no máximo trocas em uma operação de heapfy. No caso do algoritmo 2, há uma dependência em relação ao risco máximo entre as arestas para cada entrada e para grafos densos, onde chega perto de , a complexidade de passa a ser maior que .

Também foram computados o risco total e o risco médio das árvores geradoras mínimas para cada uma das entradas, apresentados nas tabelas do Anexo B. Todos os algoritmos geram as mesmas árvores geradoras mínimas, logo, os valores de risco total e risco médio para os quatro algoritmos são os mesmos.

## Testes do algoritmo Kruskal e path compression:

Voltando-se para o algoritmo kruskal, foi testada a eficiência da estratégia path compression, visto que a atualização do sub-caminho de um vértice para o seu conjunto é algo barato durante uma operação de **Find\_set()** e, para verificações futuras, diminui o tamanho do caminho de um nó até seu representante para 1. As tabelas do Anexo C ilustram as medições realizadas e a melhoria de tempo alcançada utilizando a estratégia path compression.

Além da medição dos tempos, foi calculado o caminho médio para um nó chegar até seu representante. O resultado ótimo segue quando a média possui valor 1, ou seja, cada nó da estrutura aponta diretamente para o representante, porém, para cada operação de Union entre dois conjuntos, o tamanho desse caminho aumenta até ser atualizado novamente. Baseado Algoritmo 1, foi calculado o caminho médio para um nó chegar até seu representante utilizando e não utilizando a estratégia path compression, através da quantidade de operações **Find\_set()** realizadas. As tabelas do Anexo D ilustram os resultados obtidos.

Como foi visto nas tabelas do Anexo D, o valor do caminho médio utilizando a estratégia de path compression diminui e geralmente tem o valor médio de 1, aumentando a eficiência da estrutura a partir de uma simples atualização referências. Outro ponto que pode-se notar é que a quantidade de operações **Find\_set()** feitas deve ser um pouco maior que a quantidade de arestas que o grafo possui (visto que para cada aresta testada é realizado um **Find\_set()** para cada extremidade, além de alguns adicionais quando se é perguntada a altura do conjunto em que um valor em questão se encontra).

## Testes do algoritmo Prim e priority queue:

Visto que o algoritmo 4 gerou melhores resultados, decidiu-se verificar o quão mais lento o algoritmo 3 se deu em relação ao algoritmo 4. O Anexo G aprensenta as proporções de tempo calculadas entre os dois algoritmos.

Verificou-se que, mesmo utilizando uma estratégia para melhorar o algoritmo näive, o algoritmo 3 gerou resultados piores que o algoritmo 4. Isso acontece pelo fato do heap construído pelo algoritmo 3 ser muito maior que o heap construído pelo algoritmo 4, visto que um constrói um heap a partir das arestas e outro constrói o heap a partir dos vértices. Dessa forma, o algoritmo 3 acaba executando uma quantidade maior de operações de troca se comparado com o algoritmo 4, através das execuções das operações heapfyDown e heapfyUp. O Anexo H apresenta o número de trocas realizadas entre operações heapfyDown e heapfyUp para os algoritmos 3 e 4, mostrando que o algoritmo 3 gasta mais em trocas de nós dentro do heap se comparado com o algoritmo 4.

Essa maior quantidade de trocas entre operações gerais de Heapfy são diretamente relacionadas ao fato do algoritmo 3 lidas com um heap das arestas, enquanto o algoritmo 4 lida com um heap de vértices. Sendo assim, para manter o heap correto, a quantidade de trocas do algoritmo 4 é, no geral, menor que a quantidade de trocas do algoritmo 3. O Anexo I apresenta a quantidade máxima de trocas feitas em uma operação de HeapfyDown e heapfyUp para os algoritmos 3 e 4.

# Conclusões:

Foram realizados experimentos com 7 instâncias diferentes, cada uma com 8 grafos, entre ele 4 esparsos e 4 densos. Para cada um destes experimentos, foram realizadas algumas medições adicionais com o objetivo de entender o porquê dos tempos que foram inicialmente obtidos.

O algoritmo 4 gerou os melhores resultados devido a construção do heap utilizando o conjunto de vértices, o que diminuiu o tamanho do heap e a quantidade de trocas entre operações de heapfy. Além disso, a estratégia de utilizar um vetor auxiliar para guardar a posição de cada vértice no heap melhorou a performance do algoritmo, visto que não foi necessário percorrer o heap para procurar um vértice.

Mesmo para entradas em que o risco máximo de uma aresta é maior que o número de arestas do grafo, a ordenação pelo Couting Sort se demonstrou mais eficiente do que ordenações com heapsort, devido à complexidade linear de tal algoritmo. Porém não chegou a ser tão eficiente quanto o algoritmo 4, pois em alguns casos o risco máximo chegou a 1000000, sendo este valor, para certos grafos, o quadrado do número de vértices.

Inicialmente, esperava-se que o algoritmo 2, por utilizar um algoritmo de ordenação em tempo linear, seria o mais rápido dentre os algoritmos implementados. Entretanto este algoritmo se demonstrou menos eficiente do que o algoritmo 4, sendo este um algoritmo com uma complexidade esperada maior que o algoritmo 2.

Anexos

**Anexo A: Medição dos tempos de cada algoritmo.**

**Anexo B: Risco médio e risco total para cada árvore geradora mínima construída.**

**Anexo C: Medição dos tempos utilizando e não utilizando a estratégia Path Compression.**

**Anexo D: Caminhos médios de cada operação find\_set().**

**Anexo E: Relação entre densidade e quantidade de vértices.**

**Anexo F: Comparação do algoritmo Prim utilizando Priority Queue da STL e a estrutura Heap.**

**Anexo G: Proporção de tempo entre os dois algoritmos Prim.**

**Anexo H: Quantidade de trocas realizadas entre operações Heapfyup e Heapfydown no algoritmo Prim.**

**Anexo I: Quantidade máxima de operações Heapfyup e Heapfydown em um balanceamento.**

**Anexo J: Medição dos tempos da estrutura Union Find do algoritmo Kruskal implementada utilizando ponteiros e vetores.**

## **Anexo A: Medição dos tempos de cada algoritmo**

# Instâncias 01:

**Tabela A.1:** Tempo contabilizado para cada algoritmo para as entradas da instância 01.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 3 (ms) | Algoritmo 4 (ms) |
| graph\_a1.in | 18,5 | 10,8 | 23,8 | 3,7 |
| graph\_a2.in | 359,8 | 100 | 386,3 | 33,9 |
| graph\_a3.in | 19,9 | 37,6 | 24,3 | 4,4 |
| graph\_a4.in | 378,7 | 242,7 | 410,3 | 33,7 |
| graph\_b1.in | 129,7 | 44,9 | 131,9 | 14,1 |
| graph\_b2.in | 2139,8 | 431,1 | 2116,2 | 104,2 |
| graph\_b3.in | 131,2 | 105,8 | 134 | 13,9 |
| graph\_b4.in | 2195 | 890,8 | 2269,7 | 109,7 |

# Instâncias 02:

**Tabela A.2:** Tempo contabilizado para cada algoritmo para as entradas da instância 02.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 3 (ms) | Algoritmo 4 (ms) |
| graph\_a1.in | 18,8 | 10,8 | 22,7 | 3,5 |
| graph\_a2.in | 358,2 | 98,4 | 393,3 | 32,9 |
| graph\_a3.in | 18,9 | 36,5 | 26 | 3,6 |
| graph\_a4.in | 375,2 | 224 | 406,9 | 33 |
| graph\_b1.in | 125,4 | 43,4 | 125,1 | 13,3 |
| graph\_b2.in | 2011,5 | 446,6 | 2042,2 | 106 |
| graph\_b3.in | 132 | 105,8 | 136,1 | 13,6 |
| graph\_b4.in | 2165,9 | 884,1 | 2290,6 | 109,6 |

# Instâncias 03:

**Tabela A.3:** Tempo contabilizado para cada algoritmo para as entradas da instância 03.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 3 (ms) | Algoritmo 4 (ms) |
| graph\_a1.in | 18,8 | 10,6 | 23,4 | 3,5 |
| graph\_a2.in | 355 | 99,1 | 392,2 | 33 |
| graph\_a3.in | 19 | 36,6 | 23,2 | 3,8 |
| graph\_a4.in | 372 | 223,2 | 403,4 | 33,1 |
| graph\_b1.in | 123,6 | 43,8 | 128,5 | 13,6 |
| graph\_b2.in | 1929,6 | 418,7 | 2040,6 | 103,9 |
| graph\_b3.in | 131,6 | 107 | 135,9 | 13,5 |
| graph\_b4.in | 2306,3 | 907,5 | 2314,6 | 110 |

# Instâncias 04:

**Tabela A.4:** Tempo contabilizado para cada algoritmo para as entradas da instância 04.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 3 (ms) | Algoritmo 4 (ms) |
| graph\_a1.in | 18,8 | 10,9 | 22,6 | 3,6 |
| graph\_a2.in | 362 | 99,3 | 392,3 | 33 |
| graph\_a3.in | 19,1 | 36,5 | 26,3 | 3,7 |
| graph\_a4.in | 389,6 | 222,7 | 408,8 | 33,4 |
| graph\_b1.in | 123,6 | 43,5 | 126,2 | 13,2 |
| graph\_b2.in | 1938,7 | 421,1 | 2057,2 | 106,2 |
| graph\_b3.in | 132 | 106,4 | 136,9 | 13,6 |
| graph\_b4.in | 2164,5 | 893,9 | 2361,7 | 107,2 |

# Instâncias 05:

**Tabela A.5:** Tempo contabilizado para cada algoritmo para as entradas da instância 05.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 3 (ms) | Algoritmo 4 (ms) |
| graph\_a1.in | 18,3 | 10,4 | 22,5 | 3,6 |
| graph\_a2.in | 345,5 | 96,3 | 367,3 | 32,6 |
| graph\_a3.in | 18,6 | 36,3 | 26 | 3,8 |
| graph\_a4.in | 360,8 | 216 | 387,8 | 32,7 |
| graph\_b1.in | 122,9 | 42,7 | 122,8 | 13,2 |
| graph\_b2.in | 1856,2 | 406,2 | 2280,3 | 114,8 |
| graph\_b3.in | 170,2 | 109,3 | 138,6 | 13,6 |
| graph\_b4.in | 2420,7 | 947,4 | 2561,6 | 113,2 |

# Instâncias 06:

**Tabela A.6:** Tempo contabilizado para cada algoritmo para as entradas da instância 06.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 3 (ms) | Algoritmo 4 (ms) |
| graph\_a1.in | 20,1 | 12,2 | 24 | 3,9 |
| graph\_a2.in | 384,2 | 104,9 | 403,1 | 36,1 |
| graph\_a3.in | 20 | 42,3 | 27 | 4,1 |
| graph\_a4.in | 384,4 | 225,7 | 398,4 | 33,6 |
| graph\_b1.in | 124,3 | 43,6 | 125,7 | 13,3 |
| graph\_b2.in | 2008,6 | 446,1 | 2037,9 | 105,8 |
| graph\_b3.in | 134,3 | 107,6 | 136,4 | 13,5 |
| graph\_b4.in | 2145,3 | 882,5 | 2207,5 | 107,3 |

# Instâncias 07:

**Tabela A.7:** Tempo contabilizado para cada algoritmo para as entradas da instância 07.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 3 (ms) | Algoritmo 4 (ms) |
| graph\_a1.in | 18,5 | 10,5 | 23,7 | 3,6 |
| graph\_a2.in | 346,7 | 97,3 | 373,2 | 32,7 |
| graph\_a3.in | 18,8 | 35,7 | 24,2 | 3,6 |
| graph\_a4.in | 363,4 | 216,8 | 390 | 32,7 |
| graph\_b1.in | 120,6 | 42,8 | 123,5 | 13 |
| graph\_b2.in | 1865,9 | 408,5 | 2115,9 | 104,1 |
| graph\_b3.in | 168,5 | 111,5 | 142,1 | 14,5 |
| graph\_b4.in | 2396,4 | 911,4 | 2308,8 | 110,6 |

## **Anexo B: Risco médio e risco total para cada árvore geradora mínima construída**

**Obs: O risco médio foi calculado a partir do risco total de cada árvore geradora mínima dividido pelo seu respectivo número de arestas.**

# Instâncias 01:

**Tabela B.1:** Risco médio e risco total para cada árvore geradora mínima construída.

|  |  |  |
| --- | --- | --- |
| Entrada | Risco total | Risco médio |
| graph\_a1.in | 6528 | 6,53 |
| graph\_a2.in | 7558 | 2,52 |
| graph\_a3.in | 5904397 | 5904,40 |
| graph\_a4.in | 6028080 | 2009,36 |
| graph\_b1.in | 2158 | 2,16 |
| graph\_b2.in | 3350 | 1,12 |
| graph\_b3.in | 1555533 | 1555,53 |
| graph\_b4.in | 1490970 | 496,99 |

# Instâncias 02:

**Tabela B.2:** Risco médio e risco total para cada árvore geradora mínima construída.

|  |  |  |
| --- | --- | --- |
| Entrada | Risco total | Risco médio |
| graph\_a1.in | 6788 | 6,788 |
| graph\_a2.in | 7506 | 2,469 |
| graph\_a3.in | 5977056 | 5977,06 |
| graph\_a4.in | 6158245 | 2052,75 |
| graph\_b1.in | 2112 | 2,11 |
| graph\_b2.in | 3311 | 1,10 |
| graph\_b3.in | 1549728 | 1549,73 |
| graph\_b4.in | 1567742 | 522,58 |

# Instâncias 03:

**Tabela B.3:** Risco médio e risco total para cada árvore geradora mínima construída.

|  |  |  |
| --- | --- | --- |
| Entrada | Risco total | Risco médio |
| graph\_a1.in | 6441 | 6,44 |
| graph\_a2.in | 7626 | 2,54 |
| graph\_a3.in | 6008695 | 6008,69 |
| graph\_a4.in | 6059565 | 2019,85 |
| graph\_b1.in | 2062 | 2,06 |
| graph\_b2.in | 3300 | 1,10 |
| graph\_b3.in | 1615616 | 1615,62 |
| graph\_b4.in | 1542920 | 514,31 |

# Instâncias 04:

**Tabela B.4:** Risco médio e risco total para cada árvore geradora mínima construída.

|  |  |  |
| --- | --- | --- |
| Entrada | Risco total | Risco médio |
| graph\_a1.in | 6181 | 6,18 |
| graph\_a2.in | 7365 | 2,45 |
| graph\_a3.in | 6155618 | 6155,61 |
| graph\_a4.in | 6000039 | 2000,01 |
| graph\_b1.in | 2069 | 2,07 |
| graph\_b2.in | 3304 | 1,10 |
| graph\_b3.in | 1518582 | 1518,58 |
| graph\_b4.in | 1516448 | 505,48 |

# Instâncias 05:

**Tabela B.5:** Risco médio e risco total para cada árvore geradora mínima construída.

|  |  |  |
| --- | --- | --- |
| Entrada | Risco total | Risco médio |
| graph\_a1.in | 6765 | 6,76 |
| graph\_a2.in | 7594 | 2,53 |
| graph\_a3.in | 5943188 | 5943,19 |
| graph\_a4.in | 5862310 | 1954,10 |
| graph\_b1.in | 2006 | 2,01 |
| graph\_b2.in | 3340 | 1,11 |
| graph\_b3.in | 1514269 | 1514,27 |
| graph\_b4.in | 1456828 | 485,61 |

# Instâncias 06:

**Tabela B.6:** Risco médio e risco total para cada árvore geradora mínima construída.

|  |  |  |
| --- | --- | --- |
| Entrada | Risco total | Risco médio |
| graph\_a1.in | 6586 | 6,59 |
| graph\_a2.in | 7505 | 2,50 |
| graph\_a3.in | 6207504 | 6207,50 |
| graph\_a4.in | 5955817 | 1985,27 |
| graph\_b1.in | 2031 | 2,03 |
| graph\_b2.in | 3320 | 1,11 |
| graph\_b3.in | 1420604 | 1420,60 |
| graph\_b4.in | 1535820 | 511,94 |

# Instâncias 07:

**Tabela B.7:** Risco médio e risco total para cada árvore geradora mínima construída.

|  |  |  |
| --- | --- | --- |
| Entrada | Risco total | Risco médio |
| graph\_a1.in | 6849 | 6,85 |
| graph\_a2.in | 7593 | 2,531 |
| graph\_a3.in | 5623860 | 5623,86 |
| graph\_a4.in | 5959458 | 1986,49 |
| graph\_b1.in | 2036 | 2,04 |
| graph\_b2.in | 3349 | 1,12 |
| graph\_b3.in | 1585274 | 1585,27 |
| graph\_b4.in | 1491597 | 497,20 |

## **Anexo C: Medição dos tempos utilizando e não utilizando a estratégia Path Compression**

# Instâncias 01:

**Tabela C.1:** Tempo contabilizado dos algoritmos utilizando e não utilizando Path Compression.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sem Path Compression | | Com Path Compression | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 21,2 | 17 | 18,5 | 10,8 |
| graph\_a2.in | 418,1 | 122,7 | 359,8 | 100 |
| graph\_a3.in | 20,9 | 38,8 | 19,9 | 37,6 |
| graph\_a4.in | 399,8 | 269,5 | 378,7 | 242,7 |
| graph\_b1.in | 145,2 | 50 | 129,7 | 44,9 |
| graph\_b2.in | 2040,3 | 499,3 | 2139,8 | 431,1 |
| graph\_b3.in | 163 | 117,6 | 131,2 | 105,8 |
| graph\_b4.in | 2572,1 | 1144,8 | 2195 | 890,8 |

# Instâncias 02:

**Tabela C.2:** Tempo contabilizado dos algoritmos utilizando e não utilizando Path Compression.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sem Path Compression | | Com Path Compression | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 26,3 | 15,1 | 18,8 | 10,8 |
| graph\_a2.in | 444,5 | 118,9 | 358,2 | 98,4 |
| graph\_a3.in | 21,3 | 39,7 | 18,9 | 36,5 |
| graph\_a4.in | 507,8 | 306,4 | 375,2 | 224 |
| graph\_b1.in | 188,7 | 50,6 | 125,4 | 43,4 |
| graph\_b2.in | 2047,5 | 484,1 | 2011,5 | 446,6 |
| graph\_b3.in | 161,4 | 114,3 | 132 | 105,8 |
| graph\_b4.in | 2367,6 | 989,4 | 2165,9 | 884,1 |

# Instâncias 03:

**Tabela C.3:** Tempo contabilizado dos algoritmos utilizando e não utilizando Path Compression.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sem Path Compression | | Com Path Compression | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 22,6 | 12,9 | 18,8 | 10,6 |
| graph\_a2.in | 389,6 | 114,6 | 355 | 99,1 |
| graph\_a3.in | 20,6 | 41,8 | 19 | 36,6 |
| graph\_a4.in | 396 | 241,8 | 372 | 223,2 |
| graph\_b1.in | 130,6 | 49 | 123,6 | 43,8 |
| graph\_b2.in | 1996,2 | 467,7 | 1929,6 | 418,7 |
| graph\_b3.in | 140,1 | 113,4 | 131,6 | 107 |
| graph\_b4.in | 2305,2 | 949,1 | 2306,3 | 907,5 |

# Instâncias 04:

**Tabela C.4:** Tempo contabilizado dos algoritmos utilizando e não utilizando Path Compression.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sem Path Compression | | Com Path Compression | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 20,5 | 12,5 | 18,8 | 10,9 |
| graph\_a2.in | 381,5 | 111,9 | 362 | 99,3 |
| graph\_a3.in | 20,9 | 38,3 | 19,1 | 36,5 |
| graph\_a4.in | 425,2 | 239,1 | 389,6 | 222,7 |
| graph\_b1.in | 131,6 | 49,2 | 123,6 | 43,5 |
| graph\_b2.in | 2019,1 | 468 | 1938,7 | 421,1 |
| graph\_b3.in | 143,2 | 112,8 | 132 | 106,4 |
| graph\_b4.in | 2269,3 | 974,8 | 2164,5 | 893,9 |

# Instâncias 05:

**Tabela C.5:** Tempo contabilizado dos algoritmos utilizando e não utilizando Path Compression.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sem Path Compression | | Com Path Compression | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 20,5 | 12,2 | 18,3 | 10,4 |
| graph\_a2.in | 376,7 | 120,1 | 345,5 | 96,3 |
| graph\_a3.in | 22 | 51,1 | 18,6 | 36,3 |
| graph\_a4.in | 394,9 | 243,6 | 360,8 | 216 |
| graph\_b1.in | 134,6 | 51,2 | 122,9 | 42,7 |
| graph\_b2.in | 2266,5 | 533,4 | 1856,2 | 406,2 |
| graph\_b3.in | 148,9 | 127,2 | 170,2 | 109,3 |
| graph\_b4.in | 2747,5 | 1134,7 | 2420,7 | 947,4 |

# Instâncias 06:

**Tabela C.6:** Tempo contabilizado dos algoritmos utilizando e não utilizando Path Compression.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sem Path Compression | | Com Path Compression | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 24,9 | 13,5 | 20,1 | 12,2 |
| graph\_a2.in | 466,8 | 125 | 384,2 | 104,9 |
| graph\_a3.in | 23,8 | 46,6 | 20 | 42,3 |
| graph\_a4.in | 416,7 | 258,3 | 384,4 | 225,7 |
| graph\_b1.in | 143,1 | 59,3 | 124,3 | 43,6 |
| graph\_b2.in | 2267,3 | 540,2 | 2008,6 | 446,1 |
| graph\_b3.in | 166,6 | 141,4 | 134,3 | 107,6 |
| graph\_b4.in | 2435,6 | 1117,6 | 2145,3 | 882,5 |

# Instâncias 07:

**Tabela C.7:** Tempo contabilizado dos algoritmos utilizando e não utilizando Path Compression.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sem Path Compression | | Com Path Compression | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 31,6 | 18,6 | 18,5 | 10,5 |
| graph\_a2.in | 391,1 | 114,3 | 346,7 | 97,3 |
| graph\_a3.in | 26,5 | 44,8 | 18,8 | 35,7 |
| graph\_a4.in | 501,2 | 246,9 | 363,4 | 216,8 |
| graph\_b1.in | 156,1 | 56,4 | 120,6 | 42,8 |
| graph\_b2.in | 2337,9 | 523,5 | 1865,9 | 408,5 |
| graph\_b3.in | 157,2 | 116,3 | 168,5 | 111,5 |
| graph\_b4.in | 2504,6 | 1055,3 | 2396,4 | 911,4 |

## **Anexo D: Caminhos médios de cada operação find\_set()**

# Instâncias 01:

**Tabela D.1:** Caminhos médios registrados para cada entrada.

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Quantidade de operações **Find\_Set()** | Caminho médio  utilizando Path Compression | Caminho médio não  utilizando Path Compression |
| graph\_a1.in | 199074 | 0,997 | 2,096 |
| graph\_a2.in | 1800338 | 0,999 | 1,972 |
| graph\_a3.in | 199794 | 0,997 | 1,927 |
| graph\_a4.in | 1797716 | 0,999 | 1,961 |
| graph\_b1.in | 799484 | 0,999 | 2,018 |
| graph\_b2.in | 7197222 | 0,999 | 1,930 |
| graph\_b3.in | 799560 | 0,998 | 1,968 |
| graph\_b4.in | 7198952 | 0,999 | 2,052 |

# Instâncias 02:

**Tabela D.2:** Caminhos médios registrados para cada entrada.

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Quantidade de operações **Find\_Set()** | Caminho médio  utilizando Path Compression | Caminho médio não  utilizando Path Compression |
| graph\_a1.in | 199406 | 0,997 | 1,848 |
| graph\_a2.in | 1798382 | 0,999 | 2,168 |
| graph\_a3.in | 199280 | 0,997 | 1,820 |
| graph\_a4.in | 1801238 | 0,999 | 1,962 |
| graph\_b1.in | 798652 | 0,998 | 1,863 |
| graph\_b2.in | 7199262 | 0,999 | 1,947 |
| graph\_b3.in | 799708 | 0,999 | 2,225 |
| graph\_b4.in | 7196960 | 0,999 | 2,048 |

# Instâncias 03:

**Tabela D.3:** Caminhos médios registrados para cada entrada.

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Quantidade de operações **Find\_Set()** | Caminho médio  utilizando Path Compression | Caminho médio não  utilizando Path Compression |
| graph\_a1.in | 200020 | 0,998 | 2,170 |
| graph\_a2.in | 1798596 | 0,999 | 2,194 |
| graph\_a3.in | 199530 | 0,998 | 2,253 |
| graph\_a4.in | 1798194 | 0,999 | 2,183 |
| graph\_b1.in | 798868 | 0,998 | 1,872 |
| graph\_b2.in | 7196930 | 0,999 | 1,991 |
| graph\_b3.in | 799842 | 0,999 | 2,091 |
| graph\_b4.in | 7195110 | 0,999 | 2,074 |

# Instâncias 04:

**Tabela D.4:** Caminhos médios registrados para cada entrada.

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Quantidade de operações **Find\_Set()** | Caminho médio  utilizando Path Compression | Caminho médio não  utilizando Path Compression |
| graph\_a1.in | 199612 | 0,997 | 1,971 |
| graph\_a2.in | 1799366 | 0,999 | 2,112 |
| graph\_a3.in | 199354 | 0,998 | 2,056 |
| graph\_a4.in | 1797598 | 0,999 | 1,988 |
| graph\_b1.in | 799302 | 0,998 | 1,980 |
| graph\_b2.in | 7201148 | 0,999 | 2,004 |
| graph\_b3.in | 800252 | 0,998 | 1,932 |
| graph\_b4.in | 7198448 | 1,000 | 2,307 |

# Instâncias 05:

**Tabela D.5:** Caminhos médios registrados para cada entrada.

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Quantidade de operações **Find\_Set()** | Caminho médio  utilizando Path Compression | Caminho médio não  utilizando Path Compression |
| graph\_a1.in | 199630 | 0,997 | 2,024 |
| graph\_a2.in | 1802030 | 0,999 | 2,041 |
| graph\_a3.in | 199784 | 0,997 | 2,035 |
| graph\_a4.in | 1799976 | 0,999 | 2,092 |
| graph\_b1.in | 799568 | 0,999 | 2,090 |
| graph\_b2.in | 7198692 | 1,000 | 2,207 |
| graph\_b3.in | 798274 | 0,998 | 1,790 |
| graph\_b4.in | 7199182 | 0,999 | 2,025 |

# Instâncias 06:

**Tabela D.6:** Caminhos médios registrados para cada entrada.

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Quantidade de operações **Find\_Set()** | Caminho médio  utilizando Path Compression | Caminho médio não  utilizando Path Compression |
| graph\_a1.in | 200816 | 0,997 | 2,101 |
| graph\_a2.in | 1797858 | 0,999 | 2,076 |
| graph\_a3.in | 198806 | 0,997 | 1,911 |
| graph\_a4.in | 1800534 | 0,999 | 2,092 |
| graph\_b1.in | 799476 | 0,998 | 1,880 |
| graph\_b2.in | 7196074 | 0,999 | 2,069 |
| graph\_b3.in | 799450 | 0,998 | 1,971 |
| graph\_b4.in | 7197834 | 0,999 | 2,004 |

# Instâncias 07:

**Tabela D.7:** Caminhos médios registrados para cada entrada.

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Quantidade de operações **Find\_Set()** | Caminho médio  utilizando Path Compression | Caminho médio não  utilizando Path Compression |
| graph\_a1.in | 199790 | 0,997 | 1,847 |
| graph\_a2.in | 1798862 | 0,999 | 1,993 |
| graph\_a3.in | 200078 | 0,997 | 1,892 |
| graph\_a4.in | 1799028 | 0,999 | 2,115 |
| graph\_b1.in | 799096 | 0,998 | 1,941 |
| graph\_b2.in | 7198176 | 0,999 | 2,001 |
| graph\_b3.in | 798992 | 0,999 | 2,174 |
| graph\_b4.in | 7199690 | 0,999 | 2,018 |

## **Anexo E: Relação entre densidade e quantidade de vértices**

# Instâncias 01 a 07:

**Tabela E:** Relação entre densidade e quantidade de vértices entre os grafos dados em cada instância.

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Densidade | Número de vértices | Risco |
| graph\_a1.in | 0,2 | 1000 | 1 - 1000 |
| graph\_a2.in | 0,2 | 3000 | 1 – 1000 |
| graph\_a3.in | 0,2 | 1000 | 1 - 1000000 |
| graph\_a4.in | 0,2 | 3000 | 1 – 1000000 |
| graph\_b1.in | 0,8 | 1000 | 1 – 1000 |
| graph\_b2.in | 0,8 | 3000 | 1 – 1000 |
| graph\_b3.in | 0,8 | 1000 | 1 - 1000000 |
| graph\_b4.in | 0,8 | 3000 | 1 - 1000000 |

## **Anexo F: Comparação do algoritmo Prim utilizando Priority Queue da STL e a estrutura Heap**

# Instâncias 01:

**Tabela F.1:** Tempos computados do algoritmo 03 utilizando a classe Priority\_Queue da STL e a estrutura de heap implementada.

|  |  |  |
| --- | --- | --- |
| Entrada | Heap\_min (ms) | Priority\_queue - STL (ms) |
| graph\_a1.in | 38,1 | 22,6 |
| graph\_a2.in | 533,4 | 369,4 |
| graph\_a3.in | 42,6 | 23,5 |
| graph\_a4.in | 563,5 | 389,8 |
| graph\_b1.in | 188,3 | 122,6 |
| graph\_b2.in | 2700,6 | 2018,3 |
| graph\_b3.in | 199,1 | 131,4 |
| graph\_b4.in | 3022,1 | 2204 |

# Instâncias 02:

**Tabela F.2:** Tempos computados do algoritmo 03 utilizando a classe Priority\_Queue da STL e a estrutura de heap implementada.

|  |  |  |
| --- | --- | --- |
| Entrada | Heap\_min (ms) | Priority\_queue - STL (ms) |
| graph\_a1.in | 37,6 | 22,2 |
| graph\_a2.in | 521,8 | 366,1 |
| graph\_a3.in | 39 | 22,9 |
| graph\_a4.in | 554,6 | 387,9 |
| graph\_b1.in | 189,2 | 122,6 |
| graph\_b2.in | 2693,1 | 1961,1 |
| graph\_b3.in | 199,5 | 132,2 |
| graph\_b4.in | 3009,1 | 2203,1 |

# Instâncias 03:

**Tabela F.3:** Tempos computados do algoritmo 03 utilizando a classe Priority\_Queue da STL e a estrutura de heap implementada.

|  |  |  |
| --- | --- | --- |
| Entrada | Heap\_min (ms) | Priority\_queue - STL (ms) |
| graph\_a1.in | 37,8 | 22 |
| graph\_a2.in | 524,2 | 366,3 |
| graph\_a3.in | 39,5 | 22,8 |
| graph\_a4.in | 556,2 | 388,7 |
| graph\_b1.in | 188,3 | 123,2 |
| graph\_b2.in | 2708 | 1968,9 |
| graph\_b3.in | 201,1 | 135,2 |
| graph\_b4.in | 3013,1 | 2188,3 |

# Instâncias 04:

**Tabela F.4:** Tempos computados do algoritmo 03 utilizando a classe Priority\_Queue da STL e a estrutura de heap implementada.

|  |  |  |
| --- | --- | --- |
| Entrada | Heap\_min (ms) | Priority\_queue - STL (ms) |
| graph\_a1.in | 37,8 | 22,4 |
| graph\_a2.in | 523,3 | 370,3 |
| graph\_a3.in | 38,6 | 23,2 |
| graph\_a4.in | 553,8 | 387,7 |
| graph\_b1.in | 188,8 | 123,2 |
| graph\_b2.in | 2696,8 | 1965,3 |
| graph\_b3.in | 200,3 | 130,8 |
| graph\_b4.in | 3024,1 | 2200 |

# Instâncias 05:

**Tabela F.5:** Tempos computados do algoritmo 03 utilizando a classe Priority\_Queue da STL e a estrutura de heap implementada.

|  |  |  |
| --- | --- | --- |
| Entrada | Heap\_min (ms) | Priority\_queue - STL (ms) |
| graph\_a1.in | 38,1 | 23,3 |
| graph\_a2.in | 524,3 | 365,9 |
| graph\_a3.in | 39 | 23 |
| graph\_a4.in | 556,5 | 387,8 |
| graph\_b1.in | 188,6 | 122,8 |
| graph\_b2.in | 2696,7 | 1965,1 |
| graph\_b3.in | 199 | 130,4 |
| graph\_b4.in | 3008,5 | 2193,3 |

# Instâncias 06:

**Tabela F.6:** Tempos computados do algoritmo 03 utilizando a classe Priority\_Queue da STL e a estrutura de heap implementada.

|  |  |  |
| --- | --- | --- |
| Entrada | Heap\_min (ms) | Priority\_queue - STL (ms) |
| graph\_a1.in | 38,1 | 23,2 |
| graph\_a2.in | 522,5 | 367,8 |
| graph\_a3.in | 38,9 | 23,2 |
| graph\_a4.in | 554,4 | 388,8 |
| graph\_b1.in | 188 | 123,6 |
| graph\_b2.in | 2684,5 | 1959,7 |
| graph\_b3.in | 198,6 | 131 |
| graph\_b4.in | 3016,4 | 2199,5 |

# Instâncias 07:

**Tabela F.7:** Tempos computados do algoritmo 03 utilizando a classe Priority\_Queue da STL e a estrutura de heap implementada.

|  |  |  |
| --- | --- | --- |
| Entrada | Heap\_min (ms) | Priority\_queue - STL (ms) |
| graph\_a1.in | 37,5 | 23,6 |
| graph\_a2.in | 530,5 | 370,7 |
| graph\_a3.in | 38,9 | 23,9 |
| graph\_a4.in | 557 | 393,1 |
| graph\_b1.in | 198 | 122,7 |
| graph\_b2.in | 2691,5 | 1961,2 |
| graph\_b3.in | 199,5 | 130,1 |
| graph\_b4.in | 3014,3 | 2191,4 |

## **Anexo G: Proporção de tempo entre os dois algoritmos Prim**

# Instâncias 01:

**Tabela G.1:** Proporção de tempo do algoritmo 3 pelo algoritmo 4 (quão mais lento foi o algoritmo 3 em relação ao algoritmo 4).

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Algoritmo 3 (ms) | Algoritmo 4 (ms) | Proporção de tempo |
| graph\_a1.in | 23,8 | 3,7 | 6,43 |
| graph\_a2.in | 386,3 | 33,9 | 11,40 |
| graph\_a3.in | 24,3 | 4,4 | 5,52 |
| graph\_a4.in | 410,3 | 33,7 | 12,18 |
| graph\_b1.in | 131,9 | 14,1 | 9,35 |
| graph\_b2.in | 2116,2 | 104,2 | 20,31 |
| graph\_b3.in | 134 | 13,9 | 9,64 |
| graph\_b4.in | 2269,7 | 109,7 | 20,69 |

# Instâncias 02:

**Tabela G.2:** Proporção de tempo do algoritmo 3 pelo algoritmo 4 (quão mais lento foi o algoritmo 3 em relação ao algoritmo 4).

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Algoritmo 3 (ms) | Algoritmo 4 (ms) | Proporção de tempo |
| graph\_a1.in | 22,7 | 3,5 | 6,49 |
| graph\_a2.in | 393,3 | 32,9 | 11,95 |
| graph\_a3.in | 26 | 3,6 | 7,22 |
| graph\_a4.in | 406,9 | 33 | 12,33 |
| graph\_b1.in | 125,1 | 13,3 | 9,40 |
| graph\_b2.in | 2042,2 | 106 | 19,27 |
| graph\_b3.in | 136,1 | 13,6 | 10,01 |
| graph\_b4.in | 2290,6 | 109,6 | 20,90 |

# Instâncias 03:

**Tabela G.3:** Proporção de tempo do algoritmo 3 pelo algoritmo 4 (quão mais lento foi o algoritmo 3 em relação ao algoritmo 4).

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Algoritmo 3 (ms) | Algoritmo 4 (ms) | Proporção de tempo |
| graph\_a1.in | 23,4 | 3,5 | 6,69 |
| graph\_a2.in | 392,2 | 33 | 11,88 |
| graph\_a3.in | 23,2 | 3,8 | 6,11 |
| graph\_a4.in | 403,4 | 33,1 | 12,19 |
| graph\_b1.in | 128,5 | 13,6 | 9,45 |
| graph\_b2.in | 2040,6 | 103,9 | 19,64 |
| graph\_b3.in | 135,9 | 13,5 | 10,07 |
| graph\_b4.in | 2314,6 | 110 | 21,04 |

# Instâncias 04:

**Tabela G.4:** Proporção de tempo do algoritmo 3 pelo algoritmo 4 (quão mais lento foi o algoritmo 3 em relação ao algoritmo 4).

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Algoritmo 3 (ms) | Algoritmo 4 (ms) | Proporção de tempo |
| graph\_a1.in | 22,6 | 3,6 | 6,28 |
| graph\_a2.in | 392,3 | 33 | 11,89 |
| graph\_a3.in | 26,3 | 3,7 | 7,11 |
| graph\_a4.in | 408,8 | 33,4 | 12,24 |
| graph\_b1.in | 126,2 | 13,2 | 9,56 |
| graph\_b2.in | 2057,2 | 106,2 | 19,37 |
| graph\_b3.in | 136,9 | 13,6 | 10,07 |
| graph\_b4.in | 2361,7 | 107,2 | 22,03 |

# Instâncias 05:

**Tabela G.5:** Proporção de tempo do algoritmo 3 pelo algoritmo 4 (quão mais lento foi o algoritmo 3 em relação ao algoritmo 4).

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Algoritmo 3 (ms) | Algoritmo 4 (ms) | Proporção de tempo |
| graph\_a1.in | 22,5 | 3,6 | 6,25 |
| graph\_a2.in | 367,3 | 32,6 | 11,27 |
| graph\_a3.in | 26 | 3,8 | 6,84 |
| graph\_a4.in | 387,8 | 32,7 | 11,86 |
| graph\_b1.in | 122,8 | 13,2 | 9,30 |
| graph\_b2.in | 2280,3 | 114,8 | 19,86 |
| graph\_b3.in | 138,6 | 13,6 | 10,19 |
| graph\_b4.in | 2561,6 | 113,2 | 22,63 |

# Instâncias 06:

**Tabela G.6:** Proporção de tempo do algoritmo 3 pelo algoritmo 4 (quão mais lento foi o algoritmo 3 em relação ao algoritmo 4).

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Algoritmo 3 (ms) | Algoritmo 4 (ms) | Proporção de tempo |
| graph\_a1.in | 24 | 3,9 | 6,15 |
| graph\_a2.in | 403,1 | 36,1 | 11,17 |
| graph\_a3.in | 27 | 4,1 | 6,59 |
| graph\_a4.in | 398,4 | 33,6 | 11,86 |
| graph\_b1.in | 125,7 | 13,3 | 9,45 |
| graph\_b2.in | 2037,9 | 105,8 | 19,26 |
| graph\_b3.in | 136,4 | 13,5 | 10,10 |
| graph\_b4.in | 2207,5 | 107,3 | 20,57 |

# Instâncias 07:

**Tabela G.7:** Proporção de tempo do algoritmo 3 pelo algoritmo 4 (quão mais lento foi o algoritmo 3 em relação ao algoritmo 4).

|  |  |  |  |
| --- | --- | --- | --- |
| Entrada | Algoritmo 3 (ms) | Algoritmo 4 (ms) | Proporção de tempo |
| graph\_a1.in | 23,7 | 3,6 | 6,58 |
| graph\_a2.in | 373,2 | 32,7 | 11,41 |
| graph\_a3.in | 24,2 | 3,6 | 6,72 |
| graph\_a4.in | 390 | 32,7 | 11,93 |
| graph\_b1.in | 123,5 | 13 | 9,50 |
| graph\_b2.in | 2115,9 | 104,1 | 20,33 |
| graph\_b3.in | 142,1 | 14,5 | 9,80 |
| graph\_b4.in | 2308,8 | 110,6 | 20,88 |

## **Anexo H: Quantidade de trocas realizadas entre operações Heapfyup e Heapfydown no algoritmo Prim**

# Instâncias 01:

**Tabela H.1:** Quantidade de Heapfyup’s e de Heapfydown’s feitos nos algoritmos Prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HeapfyUp | | HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 132785 | 9086 | 1391893 | 7313 |
| graph\_a2.in | 1169745 | 32507 | 15441878 | 24555 |
| graph\_a3.in | 133988 | 9199 | 1398166 | 7373 |
| graph\_a4.in | 1170333 | 32601 | 15428957 | 26908 |
| graph\_b1.in | 516358 | 9565 | 6393277 | 6491 |
| graph\_b2.in | 4603535 | 27010 | 68920552 | 16642 |
| graph\_b3.in | 519447 | 10048 | 6398097 | 7378 |
| graph\_b4.in | 4637555 | 35708 | 68995899 | 26868 |

# Instâncias 02:

**Tabela H.2:** Quantidade de Heapfyup’s e de Heapfydown’s feitos nos algoritmos Prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HeapfyUp | | HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 132639 | 8882 | 1394420 | 7319 |
| graph\_a2.in | 1167631 | 32368 | 15423368 | 24264 |
| graph\_a3.in | 133855 | 9000 | 1394127 | 7380 |
| graph\_a4.in | 1177678 | 33097 | 15460748 | 26840 |
| graph\_b1.in | 515606 | 9280 | 6385241 | 6460 |
| graph\_b2.in | 4602870 | 26714 | 68942087 | 16338 |
| graph\_b3.in | 519669 | 9822 | 6398273 | 7384 |
| graph\_b4.in | 4636982 | 35400 | 68974562 | 26859 |

# Instâncias 03:

**Tabela H.3:** Quantidade de Heapfyup’s e de Heapfydown’s feitos nos algoritmos Prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HeapfyUp | | HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 133139 | 8985 | 1399187 | 7294 |
| graph\_a2.in | 1168139 | 33606 | 15424623 | 24342 |
| graph\_a3.in | 133501 | 9150 | 1396298 | 7387 |
| graph\_a4.in | 1173671 | 32836 | 15432821 | 26822 |
| graph\_b1.in | 514352 | 9434 | 6387226 | 6498 |
| graph\_b2.in | 4601760 | 26906 | 68920819 | 16627 |
| graph\_b3.in | 520000 | 9813 | 6400583 | 7392 |
| graph\_b4.in | 4633420 | 35563 | 68955984 | 26911 |

# Instâncias 04:

**Tabela H.3:** Quantidade de Heapfyup’s e de Heapfydown’s feitos nos algoritmos Prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HeapfyUp | | HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 133144 | 8670 | 1395400 | 7331 |
| graph\_a2.in | 1167143 | 32575 | 15431324 | 24605 |
| graph\_a3.in | 134379 | 9353 | 1395143 | 7402 |
| graph\_a4.in | 1177429 | 33231 | 15427956 | 26884 |
| graph\_b1.in | 516500 | 9480 | 6391492 | 6489 |
| graph\_b2.in | 4604904 | 26410 | 68962894 | 16348 |
| graph\_b3.in | 520202 | 10138 | 6404901 | 7402 |
| graph\_b4.in | 463570 | 35185 | 68987965 | 26898 |

# Instâncias 05:

**Tabela H.5:** Quantidade de Heapfyup’s e de Heapfydown’s feitos nos algoritmos Prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HeapfyUp | | HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 133294 | 8967 | 1395967 | 7336 |
| graph\_a2.in | 1168935 | 32441 | 15455906 | 24465 |
| graph\_a3.in | 134685 | 9291 | 1398184 | 7394 |
| graph\_a4.in | 1175821 | 33183 | 15448492 | 26862 |
| graph\_b1.in | 516017 | 9396 | 6393612 | 6447 |
| graph\_b2.in | 4601001 | 26662 | 68937444 | 16517 |
| graph\_b3.in | 519005 | 9840 | 6387046 | 7382 |
| graph\_b4.in | 4634454 | 35689 | 68998484 | 26904 |

# Instâncias 06:

**Tabela H.6:** Quantidade de Heapfyup’s e de Heapfydown’s feitos nos algoritmos Prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HeapfyUp | | HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 133846 | 9053 | 1405380 | 7310 |
| graph\_a2.in | 1167713 | 32612 | 15417760 | 24373 |
| graph\_a3.in | 133933 | 8964 | 1390984 | 7394 |
| graph\_a4.in | 1175767 | 33250 | 15453883 | 26855 |
| graph\_b1.in | 514627 | 9267 | 6392265 | 6538 |
| graph\_b2.in | 4601353 | 26875 | 68912999 | 16306 |
| graph\_b3.in | 520130 | 10145 | 6398135 | 7405 |
| graph\_b4.in | 4643332 | 35521 | 68985369 | 26923 |

# Instâncias 07:

**Tabela H.7:** Quantidade de Heapfyup’s e de Heapfydown’s feitos nos algoritmos Prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HeapfyUp | | HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 133286 | 8768 | 1397346 | 7345 |
| graph\_a2.in | 1169438 | 33125 | 15427187 | 24497 |
| graph\_a3.in | 135052 | 9155 | 1400240 | 7352 |
| graph\_a4.in | 1179443 | 32863 | 15440915 | 26823 |
| graph\_b1.in | 514602 | 9379 | 6389377 | 6529 |
| graph\_b2.in | 4608285 | 27130 | 68930730 | 16788 |
| graph\_b3.in | 520763 | 10006 | 6393465 | 7365 |
| graph\_b4.in | 4642744 | 34782 | 69004704 | 26900 |

## **Anexo I: Quantidade máxima de operações Heapfyup e Heapfydown em um balanceamento**

# Instâncias 01:

**Tabela I.1:** Quantidade máxima de operações Heapfyup ou de Heapfydown feitos nos algoritmos de prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Max HeapfyUp | | Max HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 16 | 9 | 16 | 9 |
| graph\_a2.in | 19 | 11 | 19 | 11 |
| graph\_a3.in | 16 | 9 | 16 | 9 |
| graph\_a4.in | 19 | 11 | 19 | 11 |
| graph\_b1.in | 18 | 9 | 18 | 9 |
| graph\_b2.in | 21 | 11 | 21 | 10 |
| graph\_b3.in | 18 | 9 | 18 | 9 |
| graph\_b4.in | 21 | 11 | 21 | 11 |

# Instâncias 02:

**Tabela I.2:** Quantidade máxima de operações Heapfyup ou de Heapfydown feitos nos algoritmos de prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Max HeapfyUp | | Max HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 16 | 9 | 16 | 9 |
| graph\_a2.in | 19 | 11 | 19 | 11 |
| graph\_a3.in | 16 | 9 | 16 | 9 |
| graph\_a4.in | 19 | 11 | 19 | 11 |
| graph\_b1.in | 18 | 9 | 18 | 9 |
| graph\_b2.in | 21 | 11 | 21 | 9 |
| graph\_b3.in | 18 | 9 | 18 | 9 |
| graph\_b4.in | 21 | 11 | 21 | 11 |

# Instâncias 03:

**Tabela I.3:** Quantidade máxima de operações Heapfyup ou de Heapfydown feitos nos algoritmos de prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Max HeapfyUp | | Max HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 16 | 9 | 16 | 9 |
| graph\_a2.in | 19 | 11 | 19 | 11 |
| graph\_a3.in | 16 | 9 | 16 | 9 |
| graph\_a4.in | 19 | 11 | 19 | 11 |
| graph\_b1.in | 18 | 9 | 18 | 9 |
| graph\_b2.in | 21 | 11 | 21 | 11 |
| graph\_b3.in | 18 | 9 | 18 | 9 |
| graph\_b4.in | 21 | 11 | 21 | 11 |

# Instâncias 04:

**Tabela I.4:** Quantidade máxima de operações Heapfyup ou de Heapfydown feitos nos algoritmos de prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Max HeapfyUp | | Max HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 16 | 9 | 16 | 9 |
| graph\_a2.in | 19 | 11 | 19 | 11 |
| graph\_a3.in | 16 | 9 | 16 | 9 |
| graph\_a4.in | 19 | 11 | 19 | 11 |
| graph\_b1.in | 18 | 9 | 18 | 9 |
| graph\_b2.in | 21 | 11 | 21 | 10 |
| graph\_b3.in | 18 | 9 | 18 | 9 |
| graph\_b4.in | 21 | 11 | 21 | 11 |

# Instâncias 05:

**Tabela I.5:** Quantidade máxima de operações Heapfyup ou de Heapfydown feitos nos algoritmos de prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Max HeapfyUp | | Max HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 16 | 9 | 16 | 9 |
| graph\_a2.in | 19 | 11 | 19 | 11 |
| graph\_a3.in | 16 | 9 | 16 | 9 |
| graph\_a4.in | 19 | 11 | 19 | 11 |
| graph\_b1.in | 18 | 9 | 18 | 9 |
| graph\_b2.in | 21 | 11 | 21 | 10 |
| graph\_b3.in | 18 | 9 | 18 | 9 |
| graph\_b4.in | 21 | 11 | 21 | 11 |

# Instâncias 06:

**Tabela I.6:** Quantidade máxima de operações Heapfyup ou de Heapfydown feitos nos algoritmos de prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Max HeapfyUp | | Max HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 16 | 9 | 16 | 9 |
| graph\_a2.in | 19 | 11 | 19 | 11 |
| graph\_a3.in | 16 | 9 | 16 | 9 |
| graph\_a4.in | 19 | 11 | 19 | 11 |
| graph\_b1.in | 18 | 9 | 18 | 9 |
| graph\_b2.in | 21 | 11 | 21 | 10 |
| graph\_b3.in | 18 | 9 | 18 | 9 |
| graph\_b4.in | 21 | 11 | 21 | 11 |

# Instâncias 07:

**Tabela I.7:** Quantidade máxima de operações Heapfyup ou de Heapfydown feitos nos algoritmos de prim.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Max HeapfyUp | | Max HeapfyDown | |
| Entrada | Algoritmo 3 | Algoritmo 4 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 16 | 9 | 16 | 9 |
| graph\_a2.in | 19 | 11 | 19 | 11 |
| graph\_a3.in | 16 | 9 | 16 | 9 |
| graph\_a4.in | 19 | 11 | 19 | 11 |
| graph\_b1.in | 18 | 9 | 18 | 9 |
| graph\_b2.in | 21 | 11 | 21 | 10 |
| graph\_b3.in | 18 | 9 | 18 | 9 |
| graph\_b4.in | 21 | 11 | 21 | 11 |

## **Anexo J: Medição dos tempos da estrutura Union Find do algoritmo Kruskal implementada utilizando ponteiros e vetores**

# Instâncias 01:

**Tabela J.1:** Medição dos tempos do algoritmo Kruskal via implementação da estrutura Union Find com ponteiros e com vetores.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Union Find utilizando vetores | | Union Find utilizando ponteiros | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 18 | 16 | 26 | 28 |
| graph\_a2.in | 161 | 150 | 279 | 263 |
| graph\_a3.in | 24 | 20 | 25 | 25 |
| graph\_a4.in | 160 | 146 | 293 | 280 |
| graph\_b1.in | 76 | 71 | 99 | 94 |
| graph\_b2.in | 616 | 578 | 1020 | 919 |
| graph\_b3.in | 76 | 68 | 94 | 99 |
| graph\_b4.in | 608 | 571 | 1109 | 1078 |

# Instâncias 02:

**Tabela J.2:** Medição dos tempos do algoritmo Kruskal via implementação da estrutura Union Find com ponteiros e com vetores.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Union Find utilizando vetores | | Union Find utilizando ponteiros | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 21 | 17 | 25 | 28 |
| graph\_a2.in | 159 | 149 | 280 | 265 |
| graph\_a3.in | 16 | 17 | 25 | 29 |
| graph\_a4.in | 154 | 150 | 289 | 285 |
| graph\_b1.in | 73 | 70 | 105 | 91 |
| graph\_b2.in | 609 | 557 | 1028 | 914 |
| graph\_b3.in | 69 | 63 | 93 | 95 |
| graph\_b4.in | 609 | 559 | 1102 | 1070 |

# Instâncias 03:

**Tabela J.3:** Medição dos tempos do algoritmo Kruskal via implementação da estrutura Union Find com ponteiros e com vetores.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Union Find utilizando vetores | | Union Find utilizando ponteiros | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 19 | 17 | 18 | 28 |
| graph\_a2.in | 154 | 145 | 284 | 267 |
| graph\_a3.in | 18 | 22 | 27 | 25 |
| graph\_a4.in | 151 | 145 | 291 | 286 |
| graph\_b1.in | 71 | 65 | 98 | 93 |
| graph\_b2.in | 607 | 552 | 1022 | 919 |
| graph\_b3.in | 72 | 63 | 94 | 91 |
| graph\_b4.in | 606 | 559 | 1096 | 1065 |

# Instâncias 04:

**Tabela J.4:** Medição dos tempos do algoritmo Kruskal via implementação da estrutura Union Find com ponteiros e com vetores.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Union Find utilizando vetores | | Union Find utilizando ponteiros | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 18 | 18 | 29 | 26 |
| graph\_a2.in | 153 | 141 | 271 | 260 |
| graph\_a3.in | 18 | 16 | 25 | 27 |
| graph\_a4.in | 151 | 143 | 281 | 273 |
| graph\_b1.in | 71 | 66 | 94 | 89 |
| graph\_b2.in | 596 | 551 | 1036 | 918 |
| graph\_b3.in | 71 | 65 | 90 | 92 |
| graph\_b4.in | 602 | 562 | 1094 | 1070 |

# Instâncias 05:

**Tabela J.5:** Medição dos tempos do algoritmo Kruskal via implementação da estrutura Union Find com ponteiros e com vetores.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Union Find utilizando vetores | | Union Find utilizando ponteiros | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 18 | 12 | 29 | 26 |
| graph\_a2.in | 155 | 145 | 273 | 266 |
| graph\_a3.in | 16 | 17 | 24 | 29 |
| graph\_a4.in | 156 | 149 | 284 | 272 |
| graph\_b1.in | 68 | 63 | 92 | 96 |
| graph\_b2.in | 611 | 576 | 1019 | 986 |
| graph\_b3.in | 75 | 73 | 93 | 93 |
| graph\_b4.in | 626 | 571 | 1122 | 1148 |

# Instâncias 06:

**Tabela J.6:** Medição dos tempos do algoritmo Kruskal via implementação da estrutura Union Find com ponteiros e com vetores.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Union Find utilizando vetores | | Union Find utilizando ponteiros | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 21 | 24 | 25 | 24 |
| graph\_a2.in | 170 | 149 | 285 | 265 |
| graph\_a3.in | 19 | 18 | 24 | 25 |
| graph\_a4.in | 166 | 151 | 292 | 285 |
| graph\_b1.in | 64 | 67 | 98 | 90 |
| graph\_b2.in | 617 | 576 | 1016 | 924 |
| graph\_b3.in | 72 | 72 | 98 | 96 |
| graph\_b4.in | 611 | 578 | 1110 | 1079 |

# Instâncias 07:

**Tabela J.7:** Medição dos tempos do algoritmo Kruskal via implementação da estrutura Union Find com ponteiros e com vetores.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Union Find utilizando vetores | | Union Find utilizando ponteiros | |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 1 (ms) | Algoritmo 2 (ms) |
| graph\_a1.in | 23 | 18 | 25 | 25 |
| graph\_a2.in | 158 | 162 | 286 | 278 |
| graph\_a3.in | 19 | 20 | 25 | 29 |
| graph\_a4.in | 171 | 153 | 288 | 280 |
| graph\_b1.in | 69 | 61 | 94 | 96 |
| graph\_b2.in | 625 | 560 | 1025 | 940 |
| graph\_b3.in | 70 | 67 | 95 | 101 |
| graph\_b4.in | 625 | 580 | 1140 | 1114 |