*Projeto e Análise de Algoritmos: INF 2926*

Pontifícia Universidade Católica do Rio de Janeiro

Professor: Eduardo Sany Laber

Alunos: Leonardo - Trabalho de Implementação

Leonardo Quatrin Campagnolo - 1312529 Data: 12 de Junho de 2013

# 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:

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:

## 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.

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 O(w\_max) onde ‘w\_max’ 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 (w\_max, por exemplo) sendo igual a 1000.

## 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 manipular cada posição do heap como uma aresta. Esta outra abordagem se mostrou levemente mais custosa em relação ás constantes, porém possibilitou testes mais internos em relação ao comportamento do algoritmo para cada entrada fornecida.

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.

[EXPLICAR SOBRE O VETOR DE POSIÇÕES AUXILIARES QUE FOI ADICIONADO PARA CONTROLAR O ACESSO A CADA NÓ EM O(1)]

# Resultados obtidos e Análise:

Para fins de melhorar a estrutura das tabelas e das análises, enumeramos cada um dos algoritmos para serem indicados nas tabelas, listados abaixo:

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

Os testes foram realizados em um computador com processador Intel Core 2 Quad, 2,66 GHz e 4 GB de memória ram.

Foram inicialmente contabilizados os tempos de cada algoritmo. O tempo gasto com inicializações, alocação e desalocação de memória foi descartado, focando apenas no gasto em operações de ordenação e construção da árvore geradora mínima. A tabela 01 ilustra os tempos (em segundos) contabilizados para cada algoritmo.

**Tabela 01:** Tempo contabilizado, em segundos, de cada algoritmo, para cada uma das entradas.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada | Algoritmo 1 (ms) | Algoritmo 2 (ms) | Algoritmo 3 (ms) | Algoritmo 4 (ms) |
| graph\_a1.in | 16 | 8 | 24 | 3 |
| graph\_a2.in | 327 | 74 | 373 | 34 |
| graph\_a3.in | 17 | 32 | 23 | 4 |
| graph\_a4.in | 341 | 192 | 395 | 34 |
| graph\_b1.in | 112 | 31 | 128 | 14 |
| graph\_b2.in | 1813 | 352 | 1954 | 110 |
| graph\_b3.in | 119 | 91 | 130 | 14 |
| graph\_b4.in | 2043 | 766 | 2219 | 116 |

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 ordenado pelos custos de cada vértice, diminuindo a quantidade de nós da estrutura.

Também foram computados o risco total e o risco médio das árvores geradoras mínimas para cada uma das entradas, apresentados na tabela 02. Todos os algoritmos geram as mesmas árvores geradoras mínimas. Logo, os valores de risco total e risco médio são os mesmos.

**Tabela 02:** Risco médio e risco total para cada árvore geradora mínima resultante a partir de cada uma das entradas.

|  |  |  |
| --- | --- | --- |
| 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 |

obs: O risco médio foi calculado a partir do risco total dividido pelo número de arestas da árvore geradora mínima.

## 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. A tabela 03 ilustra as medições realizadas e a melhoria de tempo alcançada utilizando a estratégia path compression.

**Tabela 03:** Tempo contabilizado em segundos de cada algoritmo Kruskal, 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 | 94 | 84 | 16 | 8 |
| graph\_a2.in | 6533 | 5933 | 327 | 74 |
| graph\_a3.in | 96 | 111 | 17 | 32 |
| graph\_a4.in | 6293 | 6158 | 341 | 192 |
| graph\_b1.in | 442 | 346 | 112 | 31 |
| graph\_b2.in | 27000 | 28013 | 1813 | 352 |
| graph\_b3.in | 496 | 438 | 119 | 91 |
| graph\_b4.in | 25924 | 24676 | 2043 | 766 |

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. A tabela 04 ilustra os resultados obtidos.

**Tabela 04:** 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 | 2,17 | 245,07 |
| graph\_a2.in | 1800338 | 2,10 | 777,02 |
| graph\_a3.in | 199794 | 2,17 | 249,68 |
| graph\_a4.in | 1797716 | 2,10 | 753,90 |
| graph\_b1.in | 799484 | 2,04 | 241,57 |
| graph\_b2.in | 7197222 | 2,03 | 173,33 |
| graph\_b3.in | 799560 | 2,04 | 262,01 |
| graph\_b4.in | 7198952 | 2,03 | 148,33 |

Como foi visto na tabela, o valor do caminho médio utilizando a estratégia de path compression diminui bruscamente, aumentando a eficiência da estrutura a partir de uma simples atualização de ponteiros.

## Testes do algoritmo Prim e priority queue:

**Tabela 05:** Algoritmo 03 comparado utilizando Priority\_Queue com STL e a estrutura heap utilizada.

|  |  |  |
| --- | --- | --- |
| Entrada | Algoritmo 03 com priority\_queue da STL (ms) | Algoritmo 03 com Heap\_min (ms) |
| graph\_a1.in | 27 | 37 |
| graph\_a2.in | 353 | 492 |
| graph\_a3.in | 21 | 36 |
| graph\_a4.in | 376 | 528 |
| graph\_b1.in | 116 | 177 |
| graph\_b2.in | 1921 | 2627 |
| graph\_b3.in | 186 | 122 |
| graph\_b4.in | 2123 | 3199 |

**Tabela 06:** Relação entre densidade e quantidade de vértices entre os grafos dados para cada arquivo de entrada.

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

**Tabela 07:** 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 | 8 |
| graph\_a2.in | 373 | 34 | 10,97 |
| graph\_a3.in | 23 | 4 | 5,75 |
| graph\_a4.in | 395 | 34 | 11,62 |
| graph\_b1.in | 128 | 14 | 9,15 |
| graph\_b2.in | 1954 | 110 | 17,76 |
| graph\_b3.in | 130 | 14 | 9,29 |
| graph\_b4.in | 2219 | 116 | 19,13 |

**Tabela 08:** Quantidade de Heapfyup’s e de Heapfydown’s feitos nos algoritmos de 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 |

**Tabela 09:** Quantidade máxima de recursões em um 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 |

# Conclusões:

O algoritmo 3 é mais devagar que o algoritmo 4 e, quando o grafo é denso, a proporção em que o algoritmo 3 é mais devagar aumenta.