**TODO:**

Ver quantos vértices é necessário percorrer para chegar no find-set (com e sem path compression).

Implementar union-find com vetor ao invés de utilizar ponteiros.

Falar sobre STL

**Implementação:**

O desenvolvimento dos algoritmos foi realizado em C++. Para cada um deles, foram utilizadas as seguintes estratégias:

**Algoritmos de Kruskal:**

Para os algoritmos de kruskal, foi utilizada 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 grupo. Dessa forma, o representante do grupo é acessado em tempo constante.

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

Complexidade prevista: O(mlgn) + O(m α(m,n))

- Ordenação: O(mlgm) = O(mlgn)

- Para cada aresta do grafo: O(m)

- 2 operações find\_set(x) para cada aresta: O(2\*α(m,n))

- 1 operação Union: O(1)

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

Complexidade prevista: O(w\_max) + O(m α(m,n))

- Ordenação: O(m + w\_max)

- Para cada aresta do grafo: O(m)

- 2 operações find\_set(x) para cada aresta: O(2\*α(m,n))

- 1 operação Union: O(1)

**Algoritmos de Prim:**

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

Complexidade prevista: O(mlgm)

- Ordenar: O(mlgm)

- Para cada aresta do grafo: O(m)

- pegar a menor aresta: O(lgm)

- adicionar na MST caso a propriedade de corte. O(1)

Para este algoritmo foi utilizada a estrutura *Priority\_queue*, 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 no 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.

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

Complexidade prevista: O(mlgn)

- Ordenar heap com vértices e seus respectivos pesos: O(nlgn)

- Para cada vértice com menor grau: O(n)

- Adicionar na MST caso a propriedade seja válida (componente de um ser diferente da componente do outro): O(1)

- Atualizar os custos de cada um dos vértices vizinhos: O(deg(v)\*lgn)

Para a implementação dos algoritmos foi utilizada a IDE Microsoft Visual Studio 2012 e 2008.

**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.

Para cada tabela, foi contabilizado apenas o tempo das operações de cada algoritmo. O tempo gasto com inicializações foi descartado. 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 | Algoritmo 2 | Algoritmo 3 | Algoritmo 4 |
| graph\_a1.in | 0,016 | 0,008 | 0,024 | 0,003 |
| graph\_a2.in | 0,327 | 0,074 | 0,373 | 0,034 |
| graph\_a3.in | 0,017 | 0,032 | 0,023 | 0,004 |
| graph\_a4.in | 0,341 | 0,192 | 0,395 | 0,034 |
| graph\_b1.in | 0,112 | 0,031 | 0,128 | 0,014 |
| graph\_b2.in | 1,813 | 0,352 | 1,954 | 0,110 |
| graph\_b3.in | 0,119 | 0,091 | 0,130 | 0,014 |
| graph\_b4.in | 2,043 | 0,766 | 2,219 | 0,116 |

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 o tamanho do heap.

Também foram computados o risco total e o risco médio das árvores geradoras mínimas para cada uma das entradas, apresentados na tabela 02.

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

|  |  |  |
| --- | --- | --- |
| Entrada | Risco total | Risco médio |
| graph\_a1.in | 6528 | 6,528 |
| graph\_a2.in | 7558 | 2 |
| graph\_a3.in | 5904397 | 5904 |
| graph\_a4.in | 6028080 | 2009 |
| graph\_b1.in | 2158 | 2 |
| graph\_b2.in | 3350 | 1 |
| graph\_b3.in | 1555533 | 1555 |
| graph\_b4.in | 1490970 | 496 |

O risco médio foi calculado a partir do risco total dividido pelo número de vértices do grafo.

Também foi testada a eficiência da estratégia path compression para o algoritmo de Kruskal, visto que a atualização do sub-caminho de um vértice para o seu conjunto é algo barato durante a operação Find\_set e, para verificações futuras, diminui o tamanho do caminho atual para 1, pois o vértice aponta diretamente para a cabeça do conjunto. 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 | Algoritmo 2 | Algoritmo 1 | Algoritmo 2 |
| graph\_a1.in | 0,094 | 0,084 | 0,016 | 0,008 |
| graph\_a2.in | 6,533 | 5,933 | 0,327 | 0,074 |
| graph\_a3.in | 0,096 | 0,111 | 0,017 | 0,032 |
| graph\_a4.in | 6,293 | 6,158 | 0,341 | 0,192 |
| graph\_b1.in | 0,442 | 0,346 | 0,112 | 0,031 |
| graph\_b2.in | 27,000 | 28,013 | 1,813 | 0,352 |
| graph\_b3.in | 0,496 | 0,438 | 0,119 | 0,091 |
| graph\_b4.in | 25,924 | 24,676 | 2,043 | 0,766 |

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.