UNIVERSIDADE FEDERAL DE ITAJUBÁ
Campus Itabira

Engenharia da Computação
ECO027 – Projeto e Análise de Algoritmos

Projeto 4 - YODA

25037 – André Viana Sena de Souza

Prof. M.e Walter Aoiama Nagai
walternagai@unifei.edu.br

Monitor: Emmanuell Miqueleti
emmanuellnogmiq@hotmail.com

Itabira – MG

3 de dezembro de 2014

Sumário

1 Introdução **1**

1.1 Módulos . . . 1

1.1.1 GRAFO.H . . . 1

1.1.2 CRONOMETRO.H . . . 1

1.1.3 KRUSKAL.H . . . 2

1.1.4 CENTRO.H . . . 2

1.1.5 HEAP.H . . . 2

1.1.6 DIJKSTRA.H . . . 2

1.1.7 INTERFACE . . . 2

2 Estruturas de dados **3**

2.1 Grafo . . . 3

2.1.1 Variáveis . . . 3

2.1.2 Métodos . . . 4

2.1.3 Grafo::Aresta . . . 4

2.1.4 Grafo::Vertice . . . 5

2.2 Centro . . . 6

2.3 Heap . . . 6

2.3.1 Heap::Vertice . . . 7

2.4 Enumerações . . . 7

2.4.1 GALAXIA . . . 7

2.4.2 COMANDO_MENU . . . 8

3 Métodos **8**

3.1 GRAFO.H . . . 8

3.1.1 Grafo* CarregarArquivo(const char*); . . . 8

3.1.2 void ImprimirGrafo(Grafo*); . . . 9

3.2 CRONOMETRO.H . . . 9

3.2.1 void IniciarCronometro(); . . . 9

3.2.2 float TerminarCronometro(); . . . 9

3.3 CENTRO.H . . . 9

3.3.1 void CriarMatrizAdj(Grafo* grafo, int* grau, int** matriz_adj); . . . 9

3.3.2 int VerticeFolha(int** matriz_adj, int n); . . . 9

3.4 INTERFACE 10
3.4.1 COMANDO_MENU MenuInicial(); 10
3.4.2 COMANDO_MENU MenuCarregar(); 10
3.4.3 COMANDO_MENU MenuPrincipal(); 11
3.4.4 COMANDO_MENU MenuExecutar(const char* tarefa, const char* verbo); 11
3.4.5 void ExecutarAlgoritmoKruskal(Grafo**&, COMANDO_MENU); 11
3.4.6 void ExecutarAlgoritmoDijkstra(Grafo**&, COMANDO_MENU); . . . 12
3.4.7 void ExecutarAlgoritmoCentro(Grafo**, COMANDO_MENU); 12
3.4.8 void DescarregarGrafo(Grafo**&, COMANDO_MENU); 12
3.4.9 void CarregarGrafo(Grafo**&, COMANDO_MENU); 12
3.4.10 void ImprimirGalaxias(Grafo**, COMANDO_MENU); 12
3.4.11 int main(); 12

4 Algoritmos **13**
4.1 Algoritmo de Kruskal 13
4.2 Centro da árvore geradora mínima 14
4.3 Algoritmo de Dijkstra 15

5 Apresentação e discussão dos resultados obtidos pelos experimentos **16**
5.1 Algoritmo de Dijkstra 16
5.2 Algoritmo de Kruskal 16
5.3 Cálculo do centro 17

1 Introdução

O quarto projeto da disciplina de Projeto e Análise de Algoritmos utiliza-se de uma implementação similar à do terceiro, o C3PO. Neste projeto deve-se manter até 5 grafos de tamanhos diferentes, executar os algoritmos de Kruskal (ou Prim), Dijkstra, e determinar o centro das árvores geradoras mínimas criadas pelos mesmos. Para tal foram implementados métodos para contagem do tempo da execução dos algoritmos, leitura de arquivos, e outros demais auxiliares.

O projeto foi implementado na linguagem C++, utilizando-se da IDE CodeBlocks para compilação, e também do seu plugin DoxyBlocks, para a documentação do projeto a partir das anotações do código.

A documentação pode (e deve) ser acessada em: `ENDEREÇO_DO_PROJETO/doxygen/html/index.html`.

1.1 Módulos

Os módulos do projeto foram divididos em seis cabeçalhos e a implementação da interface, que está contida no arquivo main.cpp.

1.1.1 GRAFO.H

Responsável pelo tipo de dados grafo e todas operações básicas relacionadas à ele, como manipulação de vértices, arestas, leitura de arquivos, e impressão dos seus componentes. Em seu escopo estão definidos:

- `struct Grafo{};`
  - `struct Grafo::Aresta{};`
  - `struct Grafo::Vertice{};`
- `Grafo* CarregarArquivo(const char*);`
- `void ImprimirGrafo(Grafo*);`

1.1.2 CRONOMETRO.H

Define métodos para contagem de tempo:

- `void IniciarCronometro();`
- `float TerminarCronometro();`

1.1.3 KRUSKAL.H

A implementação do algoritmo de Kruskal para criação de árvores geradoras mínimas.

- `void Kruskal(Grafo* &grafo, bool verbose);`

1.1.4 CENTRO.H

Métodos utilizados para calcular o centro de uma árvore geradora mínima:

- `struct Centro{};`
- `void CriarMatrizAdj(Grafo* grafo, int* grau, int** matriz_adj);`
- `int VerticeFolha(int** matriz_adj, int n);`
- `Centro* DeterminarCentro(Grafo* grafo, bool verbose);`

1.1.5 HEAP.H

Heap mínimo para a execução do algoritmo de Dijkstra:

- `class Heap{};`

1.1.6 DIJKSTRA.H

Implementação do algoritmo de Dijkstra:

- `void Dijkstra(Grafo* &grafo, bool verbose);`

1.1.7 INTERFACE

Componentes da interface:

- `enum GALAXIA{};`
- `enum COMANDO_MENU{};`
- `COMANDO_MENU MenuInicial();`
- `COMANDO_MENU MenuCarregar();`
- `COMANDO_MENU MenuPrincipal();`
- `COMANDO_MENU MenuExecutar(const char*, const char*);`

- `void ExecutarAlgoritmoKruskal(Grafo**&, COMANDO_MENU);`
- `void ExecutarAlgoritmoDijkstra(Grafo**&, COMANDO_MENU);`
- `void ExecutarAlgoritmoCentro(Grafo**, COMANDO_MENU);`
- `void DescarregarGrafo(Grafo**&, COMANDO_MENU);`
- `void CarregarGrafo(Grafo**&, COMANDO_MENU);`
- `void ImprimirGalaxias(Grafo**, COMANDO_MENU);`
- `int main();`

2 Estruturas de dados

2.1 Grafo

O tipo grafo define a estrutura base utilizada no programa. Nele estão contidas as arestas e vértices do grafo, e também métodos para acesso às arestas da AGM.

2.1.1 Variáveis

- `Vertice* v`

  Vetor de vértices do grafo.

- `Aresta* a`

  Vetor de arestas do grafo.

- `std::vector<Aresta>* a_agm`

  Conteiner dinâmico para arestas da AGM do grafo. Recomenda-se utilizar os métodos auxiliares para adicionar e iterar pelas arestas.

- `int nv`

  Número de vértices do grafo.

- `int na`

  Número de arestas do grafo. (para o número de arestas da agm, acesse o campo size() de a_agm)

2.1.2 Métodos

- CONSTRUTOR - Faz todos os ponteiros internos apontarem para NULL e zera os seus respectivos contadores.
- DESTRUTOR - Deleta os componentes dinâmicos caso eles estejam instanciados.
- `void AdicionarArestaAGM(Grafo::Aresta)`

  Adiciona uma aresta ao conteiner dinâmico de arestas da AGM a_agm;
- `bool ProximaArestaAGM(Grafo::Aresta* a)`

  Através de sucessivas chamadas a essa função, é possível iterar pelos membros da estrutura interna de arestas da AGM. Antes de tudo, é necessário fazer uma chamada da função com o parâmetro nulo (**ProximaArestaAGM(NULL)**) para inicializar o contador estático da função. Após isso basta seguir a linha 5 do código de exemplo e proceder com o processamento da variável buff:

```
1  Grafo g;
2  Grafo::Aresta buff;
3
4  g.ProximaArestaAGM(NULL);
5  while(g.ProximaArestaAGM(&buff)){
6      //PROCESSAR BUFF.
7  }
```

Listing 1: Exemplo de uso do método Grafo::ProximaArestaAGM(Grafo::Aresta*).

O método retorna false quando não é possível iterar sobre uma nova aresta. Quando isso acontece, o valor do buffer não é alterado. Para reiterar pelas arestas, basta reinicializar o contador da função.

2.1.3 Grafo::Aresta

- `int origem`

  Vértice origem da aresta.
- `int destino`

  Vértice destino da aresta.

- `int peso`

  Peso da aresta, ou distância entre planetas.

- `static int Comparador(const void*, const void*)`

  Método estático utilizado como parâmetro no uso da função qsort no algoritmo de Kruskal. No algoritmo é necessário ordenar um vetor de arestas em ordem crescente. Esta função compara as arestas por peso para que o método qsort funcione corretamente.

2.1.4 Grafo::Vertice

- CONSTRUTOR - Inicializa as variáveis internas.
- DESTRUTOR - Destroi adj se estiver instanciada.
- `int peso;`

  Peso do vértice. Recebe valor durante o algoritmo de Dijkstra. Se seu valor é -1, significa que seu peso não foi calculado ainda.

- `int ant;`

  Vértice anterior à este, segundo a AGM. Se seu valor é -1, ele não tem vértice antecessor ou não foi calculado ainda.

- `std::vector<Aresta>* adj;`

  Conteiner dinâmico para arestas adjacentes.

- `void AdicionarArestaAdj(Aresta a\_adj);`

  Adiciona uma aresta adjacente à adj.

- `bool ProximaArestaAdj(Aresta* a\_adj);`

  Itera pelas arestas adjacentes do vértice, funciona da mesma maneira que Grafo::Proxi maArestaAGM(Grafo::Aresta*).

- `int tropas;`

  Armazena a quantidade de tropas que o planeta tem em sua guarnição.

2.2 Centro

- CONSTRUTOR - Inicializa o std::vector interno.
- DESTRUTOR - Destroi o conteiner caso ele esteja instanciado.
- `std::vector<int>* c;`

  Conteiner dinâmico, armazena indexadores de vértices.
- `void AdicionarVertice(int)`

  Use este método para adicionar vértices ao conteiner.
- `bool ProximoVertice(int*)`

  Método utilizado para iterar pelos vértices do conteiner. Esta função deve ser utilizada de maneira similar à Grafo::ProximaArestaAGM(Grafo::Aresta*) e Grafo::Vertice::Proxima ArestaAdj (Grafo::Aresta*).

2.3 Heap

- CONSTRUTOR Heap(Grafo::Vertice* vertices, int n) - Lê o vetor de vértices de um grafo qualquer e instancia um heap mínimo com seus valores. Para construir o heap foi utilizado o macro std::MAKE_HEAP em um std:vector de uma estrutura de dados interna personalizada, o Heap::Vertice.
- DESTRUTOR - Destroi os componentes internos.
- `std::vector<Vertice>* heap`

  Conteiner principal do heap, atualizado através dos macros std::MAKE_HEAP e std:: POP_HEAP. Mais detalhes sobre a estrutura vértice logo a seguir.
- `void AtualizarVertice(int v, int novo)`

  Este método troca o valor do peso do vértice v pelo especificado na variável novo. Após a operação, o heap é atualizado automaticamente.
- `bool Vazio()`

  Retorna se o heap está vazio.

- `int RetirarMinimo()`

  Remove o vértice mínimo do heap, com a ajuda do macro std::POP_HEAP, e retorna o seu indexador.

- `void ImprimirEstado()`

  Imprime o estado atual do heap, na ordem do vetor, em colunas Vértice e Peso.

2.3.1 Heap::Vertice

Estrutura de dados base do heap. Composta por dois inteiros, um armazena o vértice e outro o peso. O macro std::MAKE_HEAP foi projetado para criar um heap máximo, e não mínimo, mas para isso precisa que a estrutura de dados do vetor tenha um operador de <.

Para criar um heap mínimo foi necessário implementar o operador < desta maneira:

```
1 bool Heap::Vertice::operator<(Heap::Vertice outro){
2     return(this->p > outro.p)
3 }
```

Listing 2: Implementação do operador < no Heap::Vertice. Arquivo heap.cpp linha 14

Assim, foi possível criar um heap mínimo a partir do macro std::MAKE_HEAP, ordenando os vértices pelo seu peso.

2.4 Enumerações

2.4.1 GALAXIA

Esta enumeração é bastante simples e foi criada somente para aumentar a legibilidade do programa. No int main(), um vetor de ponteiros de grafo é instanciado com 5 endereços, um para cada grafo diferente. Essa enumeração é utilizada para acessar os endereços do vetor.

```
1 enum GALAXIA{
2     GALAXIA_5,
3     GALAXIA_10,
4     GALAXIA_20,
5     GALAXIA_50,
6     GALAXIA_100,
7 };
```

Listing 3: Itens da enumeração GALAXIA. Arquivo main.cpp linha 147

2.4.2 COMANDO_MENU

Enumeração criada para listar todos os possíveis retornos das funções de menu da interface, simplificando a leitura e modularização do código da interface.

Item	**Significado**
CM_DESCONHECIDO	Comando desconhecido.
CM_SAIR	Sair do menu.
CM_IMPRIMIR	Imprimir grafos.
CM_CARREGAR	Carregar grafos.
CM_DESCARREGAR	Descarregar grafos.
CM_GALAXIA_100	Referência ao grafo Galáxia 100.
CM_GALAXIA_10	Referência ao grafo Galáxia 10.
CM_GALAXIA_20	Referência ao grafo Galáxia 20.
CM_GALAXIA_50	Referência ao grafo Galáxia 50.
CM_GALAXIA_5	Referência ao grafo Galáxia 5.
CM_TODOS	Referência a todos os grafos.
CM_TODOS_SEM_VERBOSE	Referência a todos os grafos, mas sem verbose.
CM_AGM	Executar algoritmo de Kruskal
CM_DIJKSTRA	Executar algoritmo de Dijkstra
CM_CENTRO	Calcular o centro da Árvore Geradora Mínima.

Tabela 1: Itens da enumeração COMANDO_MENU. Arquivo main.cpp linha 156

```
1 COMANDO_MENU c=MenuInicial();
2 if(c==CM_CARREGAR){
3     //CARREGAR GRAFO
4 }
5 else if(c==CM_SAIR){
6     //SAIR
7 }
```

Listing 4: Código de exemplo do uso desta enumeração.

3 Métodos

3.1 GRAFO.H

3.1.1 Grafo* CarregarArquivo(const char*);

Retorna um grafo instanciado, com todos seus elementos internos contruídos, ou nulo.

```
1 Grafo* g = CarregarArquivo("foo.txt");
2 if(g==NULL)
3     //GRAFO NÃO PODE SER CARREGADO.
4 else
5     //GRAFO CARREGADO.
```

Listing 5: Exemplo de uso do método CarregarArquivo.

3.1.2 void ImprimirGrafo(Grafo*);

Imprime o vetor de vértices, arestas, e arestas AGM do grafo em colunas.

3.2 CRONOMETRO.H

3.2.1 void IniciarCronometro();

Inicializa o temporizador.

3.2.2 float TerminarCronometro();

Retorna quantos segundos se passaram desde a última inicialização.

```
1 printf("Executando algoritmo de Dijkstra no grafo Galaxia 5.\n");
2 IniciarCronometro();
3 Dijkstra(g[GALAXIA_5], verbose);
4 printf("Pronto.\nTempo de execucao: %.3fs.\n\n", TerminarCronometro());
```

Listing 6: Código de execução do algoritmo de Dijkstra no grafo Galáxia 5. Arquivo main.cpp linha 431

3.3 CENTRO.H

3.3.1 void CriarMatrizAdj(Grafo* grafo, int* grau, int** matriz_adj);

Atualiza os valores da matriz de adjacências e do vetor de grau dos vértices, através da leitura dos vértices da árvore geradora mínima do grafo. Os parâmetros grau e matriz_adj devem ser instanciados antes da chamada deste método.

3.3.2 int VerticeFolha(int** matriz_adj, int n);

Retorna o primeiro vértice folha que for encontrado na matriz de adjacências. Caso não exista nenhum vértice folha, o retorno é -1.

```
1 int f=VerticeFolha(matriz_adj, grau, grafo->nv);
2 while(f>-1)
3 {
4     //REMOVER VÉRTICE FOLHA DA MATRIZ.
5     f=VerticeFolha(matriz_adj, grau, grafo->nv);
6 }
```

Listing 7: Exemplo de uso do método VerticeFolha. Algo similar a isto é feito durante a execução do algoritmo de cálculo do centro da AGM.

3.4 INTERFACE

3.4.1 COMANDO_MENU MenuInicial();

Pergunta ao usuário se ele deseja carregar um grafo ou sair do programa. Retorna **CM_CARREGAR** ou **CM_SAIR**.

3.4.2 COMANDO_MENU MenuCarregar();

Lista os arquivos e pede que o usuário escolha qual arquivo deve ser carregado. Retorna **CM_GALAXIA_100**, **CM_GALAXIA_10**, **CM_GALAXIA_20**, **CM_GALAXIA_50**, **CM_GALAXIA_5** ou **CM_TODOS**.

3.4.3 COMANDO_MENU MenuPrincipal();

Oferece todas as opções disponíveis ao usuário, tais como executar algoritmos, imprimir, carregar, ou descarregar grafos, e sair.

Retorna **CM_AGM, CM_CENTRO, CM_DIJKSTRA, CM_IMPRIMIR, CM_CARREGAR, CM_DESCARREGAR,** ou **CM_SAIR**

3.4.4 COMANDO_MENU MenuExecutar(const char* tarefa, const char* verbo);

Menu com texto personalizável. Completa os diálogos a partir das strings dadas como parâmetro. Por exemplo, fazer uma chamada a:

```
1  MenuExecutar("executar o algoritmo do espaguete voador em", "Calcular");
```

Resulta no seguinte output:

```
Deseja executar o algoritmo do espaguete voador em qual(is) grafo(s)?
1-Calcular todos os grafos.
2-Calcular todos os grafos SEM VERBOSE.
3-Calcular somente no grafo Galaxia 5.
4-Calcular somente no grafo Galaxia 10.
5-Calcular somente no grafo Galaxia 20.
6-Calcular somente no grafo Galaxia 50.
7-Calcular somente no grafo Galaxia 100.
8-Voltar.
>>
```

Este método é extremamente útil para seleção de targets na execução de todos os algoritmos, pois evita que sejam criadas diversas outras funções que mesmo texto diferente, teriam este mesmo intuito, selecionar um ou todos os grafos.

Retorna **CM_GALAXIA_100**, **CM_GALAXIA_10**, **CM_GALAXIA_20**, **CM_GALAXIA_50**, **CM_GALAXIA_5, CM_TODOS_SEM_VERBOSE,** ou **CM_TODOS**.

3.4.5 void ExecutarAlgoritmoKruskal(Grafo**&, COMANDO_MENU);

Executa o algoritmo de Kruskal no(s) grafo(s) especificado(s), se carregado(s). Por default, a variável verbose tem valor true. Esse valor só será alterado se o comando especificado for CM_TODOS_SEM_VERBOSE.

3.4.6 void ExecutarAlgoritmoDijkstra(Grafo**&, COMANDO_MENU);

Executa o algoritmo de Dijkstra no(s) grafo(s) especificado(s), se carregado(s). Por default, a variável verbose tem valor true. Esse valor só será alterado se o comando especificado for CM_TODOS_SEM_VERBOSE.

3.4.7 void ExecutarAlgoritmoCentro(Grafo**, COMANDO_MENU);

Calcula o centro da AGM no(s) grafo(s) especificado(s), se carregado(s). Por default, a variável verbose tem valor true. Esse valor só será alterado se o comando especificado for CM_TODOS_SEM_VERBOSE. No final da execução do algoritmo, o resultado é sempre impresso.

3.4.8 void DescarregarGrafo(Grafo**&, COMANDO_MENU);

Descarrega o(s) grafo(s) especificado(s). Lê CM_TODOS_SEM_VERBOSE como CM_TODOS.

3.4.9 void CarregarGrafo(Grafo**&, COMANDO_MENU);

Carrega o(s) grafo(s) especificado(s). Lê CM_TODOS_SEM_VERBOSE como CM_TODOS.

3.4.10 void ImprimirGalaxias(Grafo**, COMANDO_MENU);

Chama a função ImprimirGrafo(Grafo*) para todos os grafos especificados, se carregados.

3.4.11 int main();

Instancia o vetor de ponteiros de Grafo e delega as operações selecionadas pelo usuário para os respectivos métodos.

4 Algoritmos

4.1 Algoritmo de Kruskal

O algoritmo funciona como está descrito no pseudo-código abaixo:

```
Ordenar arestas do grafo em ordem crescente de acordo com o peso;
enquanto(existir aresta)
{
    aresta_atual = proxima_aresta();
    se (a aresta atual nao formar ciclo)
        Adicionar aresta atual à agm;

    se(quantidade de arestas na agm == número de vértices do grafo-1)
        Finalizar o algoritmo;
}
```

Para ordenar as arestas do grafo, foi utilizado o método qsort da biblioteca stdlib.h, que acessa diretamente o vetor de arestas do grafo.

Dentro do método do algoritmo de Kruskal é mantido um vetor de variáveis bool que armazena quais vértices já foram descobertos pelas arestas da árvore geradora mínima. Um vértice é considerado descoberto quando uma aresta originado de si é adicionada à AGM. Assim, durante detecção dos ciclos entre arestas, é feita uma verificação nesse vetor da seguinte maneira:

```
se (o destino da aresta atual ainda não descoberto)
    A aresta atual não forma ciclo.
```

O algoritmo é incapaz de calcular o peso dos vértices, pois nenhum caminho entre os vértices é feito, somente as arestas acíclicas mais leves são ligadas. Apesar disso, os campos de antecessor de cada vértice do grafo, e a lista de arestas da agm do grafo são atualizados de acordo. Para visualizar o resultado do algoritmo, basta acessar esses campos diretamente, ou imprimir o grafo através do método auxiliar ImprimirGrafo(Grafo*).

4.2 Centro da árvore geradora mínima

O algoritmo funciona como está descrito no pseudo-código abaixo:

```
enquanto(existir vértice folha)
{
    vertice_atual = proximo_vertice_folha();
    Retirar vertice_atual da agm;

    se (algum vértice ficar isolado após a última operação)
        Adicionar o vértice ao conteiner de vértices centrais;
}
```

Para determinar os vértices centrais do grafo, é definido como vértice central todo vértice que se torna isolado após a remoção de um vértice folha. Assim, teoricamente, podem existir múltiplos vértices centrais em uma árvore geradora mínima, uma vez que os grafos utilizados neste projeto são direcionados. É seguro assumir tal, por que a maneira como as arestas são definidas nos arquivos dos grafos demonstra fortemente que este é o caso. Nem todos os vértices estão conectados por somente uma aresta e algumas vezes os pesos das mesmas tem valores diferentes. Em nenhum momento foi especificado o direcionamento (ou falta do) no dossiê do projeto, logo, esta implementação é válida.

O algoritmo cria internamente um vetor de int com tamanho N_VÉRTICES para armazenar o grau de cada vértice, e também uma matriz de adjacências N_VÉRTICESxN_VÉRTICES. Tanto a matriz quanto o vetor são inicializados a partir do método auxiliar CriarMatrizAdj(Grafo*, int*, int**). Para verificar se existem vértices folha, a matriz de adjacências é passada como parâmetro da função VérticeFolha(int**, int), que retorna o primeiro vértice folha encontrado na matriz ou -1 caso não exista nenhum.

Ao contrário dos outros dois algoritmos, este método retorna os resultados encontrados no lugar de armazená-los dentro do tipo Grafo. A estrutura de dados Centro é um conteiner dinâmico simples baseado em um std::vector com funções adicionais para inserção e iteração. O ponteiro de Centro que é retornado deve ser desalocado manualmente após uso.

4.3 Algoritmo de Dijkstra

Este algoritmo foi implementado exatamente como descrito neste pseudo-código[1]:

```
for(int v = 0; v < grafo.numVertices(); v++)
   p[v] = INFINITO;
   antecessor[v] = -1;


p[raiz] = 0;
Constroi heap sobre vértices do grafo;
S = VAZIO;
while (!heap.vazio())
   u = heap.retiraMin();
   S = S + u;
   for (v PERTENCE grafo.listaAdjacentes(u))
      if(p[v] > p[u] + peso da aresta (u, v))
         p[v] = p[u] + peso da aresta (u, v);
         antecessor[v] = u;
```

Após inicializar a raiz com peso 0 e todos os outros vértices com peso no valor de infinito, o algoritmo constroi um heap mínimo a partir dos vértices do grafo e seus respectivos pesos. Para tal, o algoritmo cria uma instância da classe Heap definida em heap.h.

Enquanto o heap não está vazio, seu vértice mínimo é adicionado ao conjunto de vértices solução e retirado do heap. O algoritmo, então, atualiza os pesos dos seus vértices adjacentes, sempre preferindo o valor mínimo possível.

Este método não retorna nenhum valor como resultado, mas atualiza a AGM do grafo, o peso dos vértices, e também os vértices antecessores.

5 Apresentação e discussão dos resultados obtidos pelos experimentos

5.1 Algoritmo de Dijkstra

	Algoritmo de Dijkstra	Razão aresta/vértice	Número de arestas
Galáxia 5	0,000 s	3	15
Galáxia 10	0,000 s	5,3	53
Galáxia 20	0,000 s	9,7	194
Galáxia 50	0,001 s	24,94	1247
Galáxia 100	0,005 s	54,7	5470

Tabela 2: Tempo de execução do algoritmo de Dijkstra.

A partir dos valores apresentados na tabela é fácil observar que a complexidade desse algoritmo não depende somente da quantidade de vértices, o processamento de arestas a cada iteração realmente influencia no tempo final de execução. Através da razão de arestas por vértice, fica evidente como esse fator influencia de maneira drástica o algoritmo de Dijkstra, que com o aumento da razão aresta/vértice mais que dobrado entre os grafos 50 e 100, o tempo de execução é multiplicado por 5.

5.2 Algoritmo de Kruskal

	Algoritmo de Kruskal	Razão aresta/vértice	Número de arestas
Galáxia 5	0,000 s	3	15
Galáxia 10	0,000 s	5,3	53
Galáxia 20	0,000 s	9,7	194
Galáxia 50	0,000 s	24,94	1247
Galáxia 100	0,000 s	54,7	5470

Tabela 3: Tempo de execução do algoritmo de Kruskal.

No algoritmo de Kruskal, a razão aresta/vértice teria mais influência no tempo de execução caso o vetor de arestas não fosse ordenado antes do uso, pois assim seria realmente necessário processar todas as arestas do grafo para chegar a um resultado.

Neste algoritmo acontece exatamente o contrário. Assim que o número de arestas da AGM chega a (N_Vértices -1), o algoritmo é interrompido pois este é o momento exato em que se pode ter certeza que a árvore geradora mínima está completa.

5.3 Cálculo do centro

	Cálculo do centro	Número de arestas
Galáxia 5	0,000 s	4
Galáxia 10	0,000 s	9
Galáxia 20	0,000 s	19
Galáxia 50	0,000 s	49
Galáxia 100	0,003 s	99

Tabela 4: Tempo de execução do algoritmo de cálculo do centro da AGM.

O algoritmo de cálculo do centro é executado a partir dos vértices da árvore geradora mínima, assim, o número de arestas processadas é reduzido para (N_Vértices -1).

Apesar do processamento minimizado, o algoritmo tem uma complexidade de espaço grande, na ordem de $O(n + n^2)|n = n\_vertices$.

O processo de reconhecimento de folhas se dá através da leitura da matriz de adjacências, procurando-se por vértices sem grau de saída. Sabendo-se disso, é seguro inferir que o tempo de execução desse algoritmo é muito mais dependente da quantidade de vértices do grafo do que o número de arestas.

Referências

[1] Nívio Ziviani, *Projeto de Algoritmos - Com implementações em Java e C++*, página 308, Programa 7.16 Primeiro refinamento do algoritmo de Dijkstra. 1ª edição, 2007