**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS**

**CAMINHO MÍNIMO APLICADO À CLUSTERS B-WOULF**

**HIGOR FERREIRA ALVES SANTOS**

**Goiânia**

**2019**

Sumário

[Prólogo 4](#_Toc12383004)

[Condições iniciais 4](#_Toc12383005)

[Representação 4](#_Toc12383006)

[Matriz de adjacência 5](#_Toc12383007)

[Desvantagem da matriz de adjacência 5](#_Toc12383008)

[Lista de adjacência 5](#_Toc12383009)

[Espaço ocupado pela lista de adjacência 6](#_Toc12383010)

[Estimando o tamanho do grafo com base em lista de adjacência 6](#_Toc12383011)

[Modelagem do grafo 6](#_Toc12383012)

[Vertex 7](#_Toc12383013)

[O Grafo 8](#_Toc12383014)

[Opções de construção do grafo 9](#_Toc12383015)

[Espaço ocupado pelo grafo 9](#_Toc12383016)

[Conectividade 9](#_Toc12383017)

[Bellman Ford 12](#_Toc12383018)

[Processamento em paralelo 12](#_Toc12383019)

Lista de Figuras

# Prólogo

Há diversas aplicações em muitas áreas que podem ser resolvidas com uma ferramenta matemática denominada Grafo.

Segundo Rosen “Um grafo G=(V, E) consiste em V, um conjunto não vazio de vértices (ou nós), e E, um conjunto de arestas. Cada aresta tem um ou dois vértices associados a ela, chamados de suas extremidades. Dizemos que uma aresta liga ou conecta suas extremidades.”, (ROSEN KENNETH H., Matemática Discreta e Suas Aplicações, 6ª ed).

Neste artigo, abordaremos um uso abstrato e genérico de grafos aplicados a dois ou mais nós de processamento, em que determinado nó requisita o menor caminho para um grafo . Esta requisição deverá ser processada por um nó servo disponível em nossa rede de Clusters (Figura 1). Abordaremos também o problema de garantir a integridade do grafo nos diversos nós, e isto será feito através de uma cópia do grafo para cada nó, sendo que, uma mudança no grafo causada pelo nó , refletirá nos nós , com .

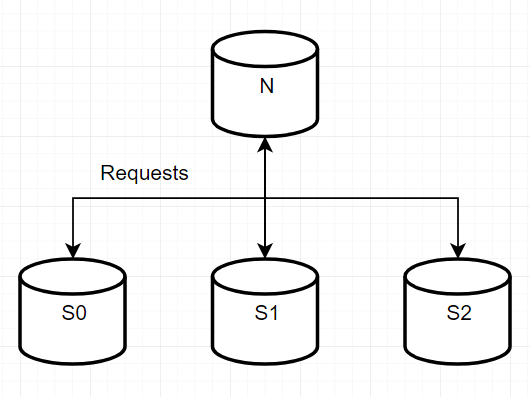


Figura 1: Exemplo de Clusters com n = 3 servos

Basicamente, haverá um nó comandante que se comunicará com seus servos, usaremos o MPI em nossa modelagem, de forma que a comunicação entre os Clusters por meio de mensagens possa ser facilitada.

# Condições iniciais

O grafo escolhido para este trabalho, será um grafo que poderá conter até 65 mil vértices diferentes, sendo que 5 mil arestas serão criadas aletoriamente, resultando em um número não maior que 10 mil vértices úteis. Será um grafo valorado, (com pesos nas arestas), com todas as arestas iniciando com valor 100. Aplicaremos o algoritmo de Bellman Ford, responsável por encontrar o caminho mínimo, que decrementará os pesos das arestas onde ele passar por 1 a cada iteração.

# Representação

Há, basicamente, duas maneiras mais utilizadas para se representar um grafo computacionalmente. Uma forma é através de uma matriz de adjacência, e a outra é a lista de adjacência.

## Matriz de adjacência

A representação na forma de matriz de adjacência, basicamente consiste em uma matriz , sendo o número de vértices do grafo. Esta matriz é formada por 0s e 1s, em que no caso de um grafo não valorado uma aresta qualquer é marcada com 1 na matriz, sendo , e , o 0 indica a ausência de arestas. Também em um grafo valorado, pode-se representar o peso da aresta colocando-a como valor do par ordenado , a ausência de arestas pode ainda ser o próprio 0 ou qualquer outro valor fora do intervalo de interesse.

### Desvantagem da matriz de adjacência

Embora muito intuitiva para visualizar o Grafo, a matriz de adjacência (figura 2) tem a desvantagem de ocupar de espaço. No nosso caso, dada uma máquina com inteiros de 4 bytes, nosso grafo ocuparia   
Vide: <<https://convertlive.com/u/convert/bytes/to/gigabytes#67600000000>>

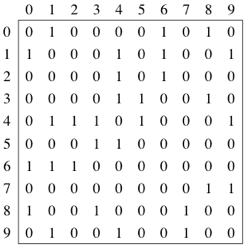
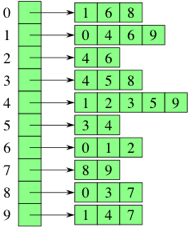


Figura 2: Matriz de adjacência de um grafo com dez vértices

## Lista de adjacência

A representação de um grafo por lista de adjacência pode ser entendida como um grande vetor, em que cada índice é um determinado vértice, que contém um ponteiro para uma lista de vértices adjacentes a . A figura 3 mostra uma representação básica de um grafo por lista de adjacência. Podemos ver claramente que cada vértice do grafo, contém uma lista com seus vértices adjacentes, em outras palavras, temos uma lista de vértices. Por exemplo: é uma aresta no grafo da figura 3, aresta essa que parte de 0 para 1. Se queremos um grafo não direcional, basta inserir os dados como . É claro que, num nível abstrato, vamos lidar com ponteiros e nós.

Figura 3: Grafo representado em lista de adjacência



### Espaço ocupado pela lista de adjacência

Na representação por lista de adjacência, teremos um total de ponteiros para conjuntos de até  
 vértices, estas são as representações de nossas arestas. Com respeito ao espaço ocupado, para ser mais preciso, nossa lista ocupará elementos, pois, devido ao fato de nosso grafo não ser direcionado, a aresta tem de ser representada com , e . Devemos levar também em consideração, que os ponteiros possuem um tamanho fixo dada a arquitetura da máquina.

### Estimando o tamanho do grafo com base em lista de adjacência

Dada uma arquitetura de 64bits, sabemos que o endereçamento desta máquina necessita de 8bytes. Sendo assim, nosso grafo inicial consistiria em um buffer de ponteiros com  
 (PB = Pointer Buffer). Esses ponteiros deverão apontar para outros buffers que são os vértices adjacentes a cada vértice do nosso grafo, assumindo que estes buffers serão buffers simples de inteiros, e que os inteiros ocupam 4bytes, dizemos que os buffers de adjacência irão nos custar (AB = Adjacency Buffer).

Sendo assim, o tamanho total de nosso grafo representado como lista de adjacência, seria, num primeiro momento , que é centenas de vezes menor que a representação anterior.

É claro que nosso grafo ocupará um espaço muito maior, pois não usaremos simples valores inteiros, mas uma lista de um tipo abstrato de dados chamado Vertex, que conterá o valor do nosso vértice, bem como um campo para cor, outro para o peso da aresta, e também um ponteiro para o próximo elemento da lista. Isto veremos mais adiante.

# Modelagem do grafo

Para obter o nosso grafo desejado e fazermos todas as operações desejadas sobre o mesmo, necessitaremos de alguns algoritmos que dependem que algumas informações extras que armazenaremos em nosso próprio grafo, informações como o peso da aresta, (que não é representado no grafo da figura 3), cor do vértice e etc. Para que possamos fazer uso destas informações com facilidade, utilizaremos um tipo abstrato de dados chamado Vertex.

## Vertex



(a)

(b)

Arquivo de código de Vertex com suas diversas opções de construção, sendo que um vértice genérico é construído com chave -1, peso de aresta 100 e nenhum vizinho.

Arquivo de cabeçalhos do tipo abstrato Vertex.h, podemos ver também a definição de algumas constantes que serão usadas como valores para Vertex

Figura 4: TAD Vertex

O tipo abstrato de dados da figura 4 será a representação de nossos vértices. Observe que cada vértice será identificado no grafo pela sua *key*, que nada mais é do que um numeral inteiro que . A propriedade color nos ajudará com o algoritmo de busca, que mais a frente veremos como ele encontra os componentes conectados para que o grafo não fique desconexo. A propriedade edgeWeight, representará o peso de nossa aresta e o ponteiro neighbor, é o ponteiro para o próximo vértice adjacente.

vertexWeight, e \*predecessor, são propriedads auxiliares necessárias ao algoritmo de Bellman Ford.

Observe que além dos métodos construtores, todos os membros da classe Vertex são públicos, isso é feito para facilitar a atribuição e o acesso aos dados, uma vez que não necessitaremos de métodos intermediários para tal.

A classe também define um conjunto de constantes, para nos ajudar com os algoritmos. Todos os inteiros que utilizamos em nossa classe, tem um intervalo de valores para os 4bytes, mas observe que todas as constantes são de 8. Isto é feito para que possamos definir números especiais, como o Infinito ou o Undefined. Isso ajuda na clareza do código.

## O Grafo

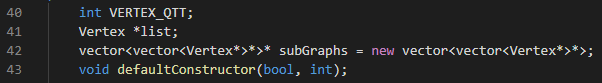
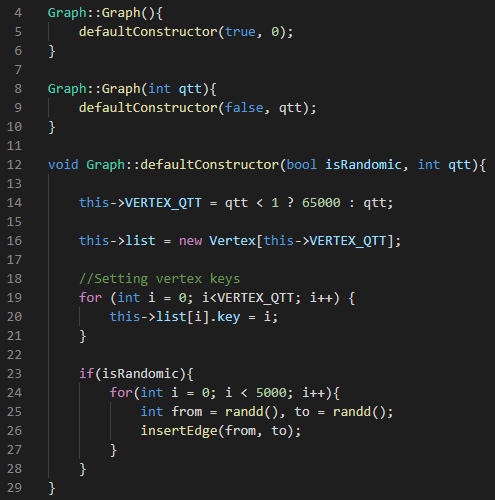


Figura 5: Arquivos de cabeçalho do grafo

Figura 6: Construtores do grafo



A estrutura do nosso grafo é bastante simples, consiste em um array, (Figura 5, linha 41), que conterá referencias para listas de vértices. O índice de nosso array, será também interpretado como a key de um vértice qualquer. Por exemplo, se quisermos verificar todos os vértices adjacentes ao vértice 8, basta olhar a lista de vértices atribuída ao índice 8 de nosso array. Temos também outras informações que nos serão úteis, como a quantidade de vértices na linha 40, (escrita em notação de constante), que será bastante útil nos iteradores. O subGraphs (linha 42), que guardará as componentes desconexas do grafo gerado aleatoriamente (Vide o tópico de conectividade), e o construtor padrão (linha 43), que será chamado pelas duas opções de construção do grafo.

## Opções de construção do grafo

Os construtores do grafo foram pensados de forma que o mesmo possa ser inicializado de maneiras diferentes, para que possamos, por exemplo, construir nosso próprio grafo ou gerar um grafo aleatório com os 65 mil vértices.

Observe na figura 6 como a ***randomicidade*** do grafo é decidida através da variável isRandomic, que é chamada com valores diferentes dependendo do construtor invocado. A quantidade de vértices é decidida através da variável *qtt*, observe na linha 14 como o operador ternário decide os 65 mil vértices caso , pois adotamos o grafo com como um grafo inválido.

# Espaço ocupado pelo grafo

Assim como visto na figura 4, o grafo será composto de vários nós denominados Vertex. Cada um destes tipos abstratos contém 3 inteiros, 2 ponteiros e o equivalente a 1 long. Somando estas quantidades com seus respectivos espaços ocupados na memória, temos que cada *TAD* ocupa de memória.

Contabilizando os 65 mil vértices, sabemos que o buffer de ponteiros para Vetexes nos custará . Cada ponteiro, apontará para um Vertex alocado dinamicamente na memória, o que custará . Até agora, sem as listas de adjacência atribuídas à cada vértice existente no grafo, nosso “*Buffer List”* está custando .

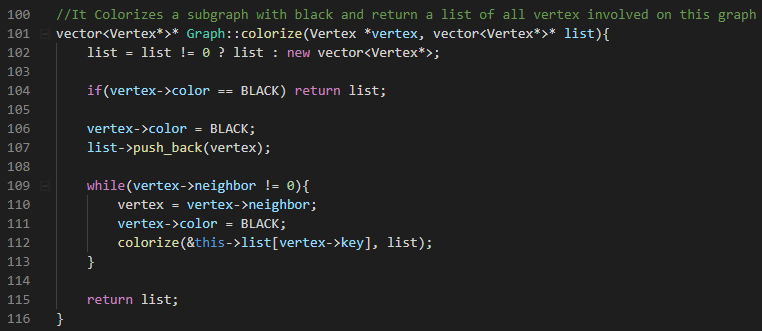
Para contabilizar o preço pelas listas de adjacência, devemos considerar a conda anterior de que na lista de adjacência, o espaço ocupado pelas listas é de , sendo que no pior caso, cada aresta var estar conectada à dois vértices, sendo então .

Sendo assim, podemos estimar no pior caso, que nosso grafo ocupará .

# Conectividade

Quando chamamos o construtor do grafo sem passar para ele a quantidade de vértices, fica subentendido que queremos um grafo com 65 mil vértices com 5 mil arestas geradas aleatoriamente (Vide figura 6, linha 23). O problema desse tipo de construção é que certamente serão gerados diversos sub-grafos desconexos, e como vimos na seção de Condições iniciais, queremos um grafo totalmente conexo. Este problema será *driblado* com um algoritmo de coloração de grafos, que basicamente, colore todos os vértices conectados a partir de um vértice qualquer. Observe a figura 7:

Figura 7: Algoritmo de coloração de grafos a partir de um vértice i.



Observe que o método de coloração recebe como parâmetros um vértice a partir do qual a coloração será feita e um ponteiro para uma lista de vértices. O propósito de tal lista é que ao final do algoritmo queremos uma lista de todos os vértices envolvidos no mesmo grafo, pois nosso objetivo, como veremos mais adiante, é obtermos uma lista de componentes desconexos, e esses componentes desconexos por sua vez, são listas de vértices. Deste modo, obtendo-se uma lista de listas de vértices, poderemos facilmente conectar os diversos componentes. Observe na figura 8 o método responsável por obter a lista das componentes desconexas:

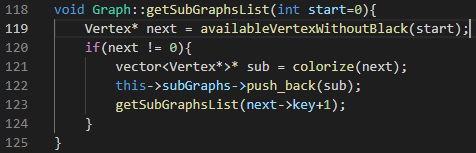
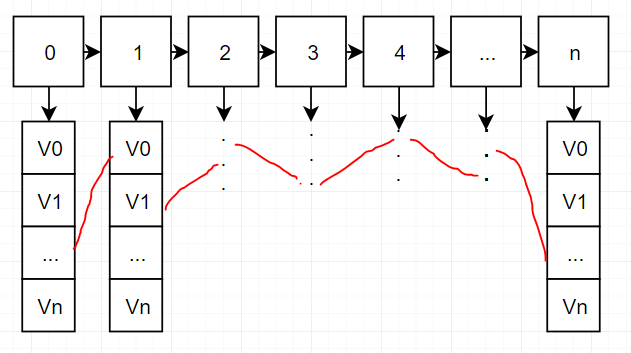


Figura 8: Método que obtém a lista de componentes desconexas

Observe na linha 121 que o método de coloração é chamado, obtendo uma lista como resultado. Esta lista, por sua vez, é inserida na lista de subgrafos como declaramos na linha 42, figura 5.

Uma vez obtida a lista de componentes desconexas, a conexão entre elas será feita também de forma aleatória, em que dado um conjunto de listas, um iterador que itera sobre , utilizará uma função que conectará um vértice aleatório a outro vértice aleatório . Observe a ilustração desta operação na figura 9:

Figura 9: Conectando as componentes desconexas



Em que as linhas vermelhas simbolizam a conexão entre dois vértices aleatórios. A figura 10 nos traz o algoritmo responsável por tal conexão

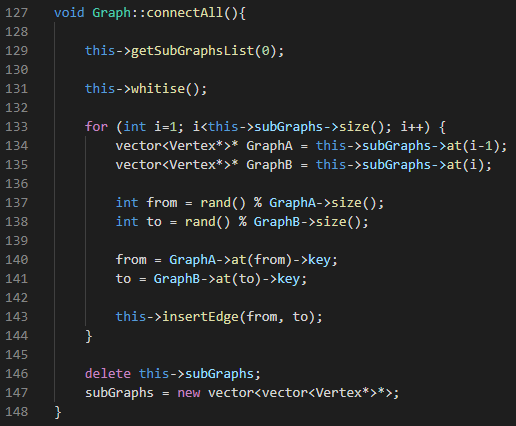


Figura 10: Algoritmo de conexão de componentes desconexas

# Bellman Ford

O algoritmo de Bellman Ford, é um algoritmo bastante conhecido e implementado e grafos para se obter o caminho mínimo de um dado vértice para um vértice . Sua principal diferença para o algoritmo de Dijkstra, é que o tal é capaz de tratar ciclos negativos, o que o Dijkstra não é capaz de fazer. Esta característica é essencial a nosso projeto, pois como o decremento dos pesos das arestas é um requisito, num determinado momento haverá arestas com pesos negativos, o que provocará ciclos negativos.

## Funcionamento

Basicamente, o algoritmo de Bellman Ford funciona percorrendo todos os vértices a partir de um dado vértice , e atribuindo pesos às arestas do grafo dependendo do caminho formado, que é conseguido através do algoritmo de relaxamento (Figura 12).

### Inicialização

Inicialmente, atribui-se a todos os vértices um peso infinito (Figura 11), e também um valor nulo ao vértice . Esse vértice , é o vértice de onde o algoritmo veio percorrendo, observe que em nosso TAD   
(Figura 4a), chamamos a propriedade de *vertexWeight,* e a chamamos predecessor.

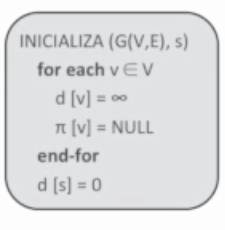


Figura 11: Inicialização do algoritmo de Bellman Ford

Fonte: (CORMEN THOMAS H., Algoritmos Teoria e Pratica, 3ª ed).

### Percurso

Como podemos observar na Figura 13, todos os vértices do grafo serão percorridos a partir de um vértice , o *Relaxamento* (Figura 12) é o responsável por tomar uma dada aresta , e atribuir a o peso do menor caminho entre e . Ao final do algoritmo, todos os vértices conterão um determinado peso mínimo, e o ponteiro par o vértice do qual este peso mínimo veio.

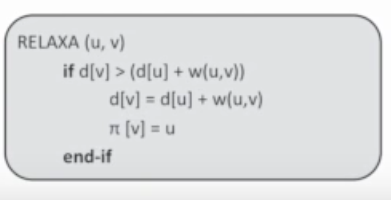


Figura 12: Bellman Ford – Relaxamento

Fonte: (CORMEN THOMAS H., Algoritmos Teoria e Pratica, 3ª ed).



Figura 13: Bellman Ford

Fonte: (CORMEN THOMAS H., Algoritmos Teoria e Pratica, 3ª ed).

## A formação do caminho mínimo

Apenas rodar o algoritmo de Bellman Ford não é o suficiente para se obter o caminho mínimo, pois o que temos ao final é um grafo com pesos mínimos nos vértices que apontam de forma inversa para o vértice de onde veio. O método shortWay é o responsável por esta tarefa, ele toma o vértice final e percorre o caminho inverso indicado pelos ponteiros até chegar ao vértice de origem, além também de acumular o custo. Ao final ele retorna a estrutura da figura 14:

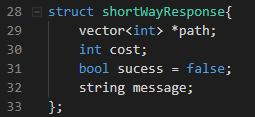


Figura 14: Estrutura de caminho mínimo retornada

Podemos ver no retorno da estrutura que a mesma nos traz o custo e o caminho, bem como uma variável para aferirmos o sucesso da função e uma string contendo uma mensagem em caso de erro.

Todos os detalhes da implementação podem ser encontrados em graph.cpp nas linhas 336 à 472. Todo o projeto pode ser baixado em:

<https://github.com/HigorFerreira/BFS_MPI>

# Processamento em paralelo

Agora que temos todos os algoritmos prontos para o uso, utilizaremos o processamento em paralelo com o OpenMPI para os nós servos da nossa rede de Clusteres processem nossas requisições de caminho mínimo. Faremos isso através do envio de mensagens pelo *Message Passing Interface (MPI)*. Primeiro, inicializaremos o grafo no nó comandante, depois, o distribuiremos entre os diversos nós para que cada nó tenha uma cópia exata do grafo. Uma vez com todas as cópias em sincronia, os nós já estarão aptos a receber requisições do comandante. É importante salientar que cada caminho mínio encontrado gera uma modificação no grafo, o que deverá ser replicado entre todos os nós.

## Inicialização

Primeiramente, devemos inicializar o MPI. Observe a figura 15

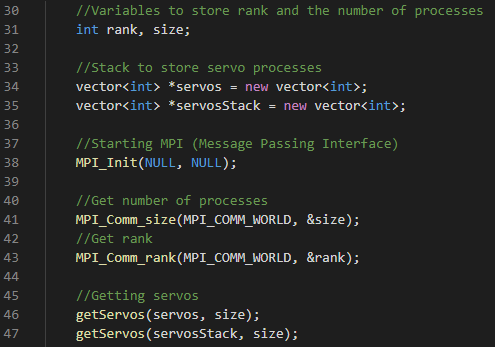


Figura 15: Inicializando o MPI

Para fazer o controle das mensagens as variáveis rank e size são de suma importância, pois através de rank sabemos em qual nó estamos, e com size sabemos quantos nós estão conectados, observe que as linhas 41 e 43 da figura 15 modificam as respectivas variáveis. Na figura 16, podemos observar a geração de nosso grafo e o envio do mesmo para os demais nós

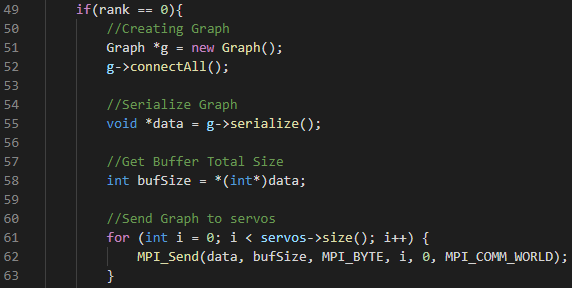


Figura 16: Gerando grafo, e enviando

## Serialização

Para enviar o nosso grafo entre os nós, um item na linha 55 da figura 16 é de suma importância. A serialização basicamente toma todo o grafo gerado aleatoriamente é o constrói em uma grande sequência de bytes denominada buffer. Desta forma, os bytes que representam o nosso grafo podem ser enviados sequencialmente para o nó desejado. Observe na figura 17 como é a estrutura de buffer do grafo:

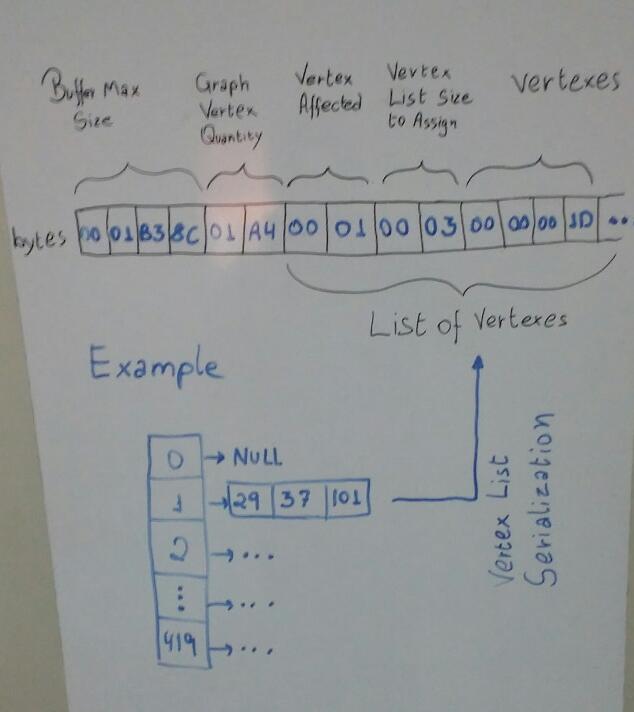


Figura 17: Grafo serializado

Observe que os primeiros quatro byte armazenam o tamanho total do buffer, os próximos dois bytes armazenam a quantidade total de vértices válidos e os demais bytes podem ser pensados como pequenos sub-bufferes que armazenam um dado vértice e sua respectiva lista. Ao chegar no nó servo, o método mout interpreta esse buffer e remonta o grafo no servo.

Basicamente, toda a comunicação se dá através desse esquema, ao criar e replicar o grafo, tanto os nós servos como o comandante entram em um loop infinito que aguarda requisições que serão processadas no mesmo estilo de mensagem acima.

# Referências

<https://pt.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs>

# Referências Bibliográficas

ROSEN KENNETH H., Matemática Discreta e Suas Aplicações, 6ª ed

CORMEN THOMAS H., Algoritmos Teoria e Pratica, 3ª ed