ALGORITMO E ESTRUTURA DE DADOS II

Conjuntos Disjuntos – Lab 2

Aluno: Gabriel Nobrega de Lima
E-mail: gabriel.nobrega.lima@gmail.com
Professor: David Correa Martins Jr

Sumário

Implementação da Estrutura de Conjuntos Disjuntos

Neste laboratório foram desenvolvidas estruturas de dados que implementam as operações de criação (makeSet), busca (findSet) e união (unionSet) em conjuntos disjuntos. O projeto foi desenvolvido em linguagem C++ utlizando o compilador g++ e o editor de textos Kate. Para a correção de códigos e identificação de bugs foram utlizados o GDB (GNU Debugger) e o Valgrind.

Arquitetura

Como um dos requisitos do laboratório é a utilização de boas práticas de programação, toda a estrutura de dados implementada neste projeto foi orientada a interfaces (classes puramente abstratas). Além disso procurou-se encapsular toda a estrutura de conjuntos como uma biblioteca, separando-as da aplicação em si. Estas estruturas podem ser encontradas no diretório .../src/setlib do projeto.

Como as estruturas de dados de conjuntos disjuntos possuem funções bem definidas, isto é, devem sempre implementar união, busca e criação foi definida a interface SetManager. Esta interface força todas as estruturas a manter uma compatibilidade mínima possibilitando a utilização de polimorfismo. Parte da interface pode ser vista no fragmento de código abaixo:

```
class SetManager{
public:
  virtual bool makeSet(int n) = 0;
  virtual int findSet(int n) = 0;
  virtual bool unionSet(int x, int y) = 0;
        …
}
```

Após definir uma interface comum foi implementado a estrutura de conjuntos disjuntos utilizando listas encadeadas e floresta (árvores).

A implementação com listas encadeadas utilizou a interface definida por SetManager, como mostra o fragmento abaixo:

```
class LinkedListSet: public SetManager{
        …
}
```

Cada nó da lista encadeada é representado pela classe LinkedNode e permite a identificação do próximo nó da lista (next) e o representante do conjunto (set). Isto pode ser observado no fragmento de código abaixo:

```
class LinkedNode{
public:
```

```
        Set *set; //Conjunto
        LinkedNode *next;    //Próximo nó
        LinkedNode();
};
```

Perceba que o representante é um ponteiro para a classe Set. Na verdade Set define não só o representante, mas mantêm ponteiros para o inicio e fim da lista, o número de elementos no conjunto e o valor de indice do nó representante. A classe pode ser vista logo abaixo:

```
class Set{
public:
        LinkedNode *head; // Inicio da lista (representante)
        LinkedNode *tail; // Fim da lista
        int nodeIndex; // Indice do nó(Didático)
        int size; // Número de nós
};
```

A estrutura de dados que utiliza árvores (floresta) também implementou a interface SetManager, como mostrado abaixo:

```
class ForestSet: public SetManager{
        ...
}
```

Como cada nó da árvore na implementação de uma floresta tem propriedades ligeiramente distintas aos nós da lista encadeda foi criada a classe ForestNode para representa-la. Neste caso cada nó mantêm um ponteiro para o elemento superior. Todo nó também possui uma estimativa de quantos nós estão associados a ele e o seu indice no vetor.

```
class ForestNode{
public:
        ForestNode *next; // Próximo nó da árvore
        int rank; // Número de nós na árvore
        int nodeIndex;   // Indice do nó(Didático)
         ForestNode();
};
```

A arquitetura proposta manteve tanto a estrutura de dados com listas encadeadas e a de árvores com uma interface pública similar. Isto acabou facilitando a codificação da classe de aplicação UnionFindApp. Esta classe passou a utilizar polimorfismo para intercambiar entre cada uma das estruturas de maneia transparente. O exemplo de código abaixo mostra a vantagem da arquitetura utlizada:

```
processSet(new LinkedListSet(4));
processSet(new ForestSet(4));

void processSet(SetManaget *setManager){
        setManager->makeSet(0);
        setManager->makeSet(1);
        setManager->makeSet(2);
        setManager->makeSet(3);
```

```
        setManager->unionSet(1, 2);
        setManager->unionSet(1,3);
        setManager->unionSet(1, 0);
}
```

Perceba que é possivel criar um método genérico que realize operações no conjunto sem ao menos ter idéia de qual conjunto de fato o usuário está utilizando. O método processSet executa a operação para qualquer tipo de estrutura de dados de conjuntos disjuntos. Isto reduz drásticamente o número de código, além de manter uma API pública uniforme.

Aplicação

Após definir a arquitetura e implementar as estruturas de dados foi desenvolvida a classe UnionFindApp. Ela está contida no diretório .../src e modela uma aplicação que cumpre os requisitos definidos nas questões A, B, C, D e E do laboratório 2.

A classe UnionFindApp permite o usuário processar as operações de união de conjuntos disjuntos a partir de arquivos de entrada com o formato especificado no enunciado da prática 2. O usuário pode definir qual estrutura utilizar (listas encadeadas ou floresta), o tipo de heurística e a coleta de dados estátisticos (operações de ponteiro) salvo em formato *.csv.

O fragmento de código abaixo mostra como um usuário poderia processar as uniões especificadas pelo arquivo “uf1.txt”:

```
UnionFindApp app;

app.setInputDataFile(“uf1.txt”);
app.setOutputDataFile(“saida_uf1.txt”);
app.setDataStructureType(UnionFindApp::LINKEDLIST_DATA_STRUCTURE);
app.setHeuristic(UnionFindApp::WEIGHTED_UNION_HEURISTIC);
app.setViewStateStart(1000);
app.setViewStateInc(1000);
app.setViewStateSize(20);
app.setStatisticMode(true, “estatistica.csv”);

app.start();
```

Perceba que o fragmento de código utiliza a estrutura de listas encadeadas com heurística de união ponderada. Serão impressos 20 estados, onde o primeiro estado será aquele após a realização de 1000 uniões, o segundo a pós 2000 e assim sucessivamente. A saída dos estados será armazenada no arquivo “saida_uf1.txt”. Foi ainda ativada o modo de coleta de estátistica que cria o arquivo “estatistica.csv”, armazenando o número de operações de ponteiro obtidas a cada 1000 uniões. Este arquivo contêm em cada linha a sintaxe “númeroro de uniões, número de operações de ponteiro”. O usuário poderia alterar a estrutura de dados para floresta simplemente utilizando a flag UnionFindApp::FOREST_DATA_STRUCTURE no

método setDataStructureType(). A heurística poderia ser modificada através do método setHeuristic().

Interface com o usuário

A interace com o usuário é feita através da utilização de parâmetros na execução da aplicação. Basicamente a função main contida em .../src /main.cpp recebe os parâmetros, os identifica e configura um objeto da classe UnionFindApp para executar a tarefa. Para isso foram definidos diferentes tipos parâmetros, listados na Tabela 1. Cada parâmetro irá orientar a função main a configurar corretamente a classe UnionFindApp.

Parâmetro	Exemplo	Descrição
-i	./unionfindapp -i uf2.txt	Define o arquivo de entrada com as operações de união. Padrão: uf.txt
-o	./unionfindapp -i uf2.txt -o saida.txt	Define o arquivo de saída, onde sao imprimidos os estados de cada conjunto. Padrão: saida.txt
-t	Lista encadeada: ./unionfindapp -i uf2.txt -o saida.txt -t linkedlist. Floresta: ./unionfindapp -i uf2.txt -o saida.txt -t forest	Tipo de estrutura de dados: listas encadeadas ou floresta. Padrão: linkedlist
-h	Para listas encadeadas(-t linkedlist): default: Sem heuristica. weighted: União ponderada. ./unionfindapp -i uf2.txt -o saida.txt -t linkedlist -h weighted Para floresta(-t forest): default: Sem heuristica compweighted: Uniao ponderada e compressao de caminhos. ./unionfindapp -i uf2.txt -o saida.txt -t forest -h compweighted	Heuristica utilizada pela estrutura de dados. Padrão: default

-sstart	./unionfindapp -t forest -h compweighted -i uf1.txt -sstart 1000 -sinc 1000 -ssize 20	Captura o estádo dos conjuntos a partir da união de número especificado. Padrão: 1
-ssize	./unionfindapp -t forest -h compweighted -i uf1.txt -sstart 1000 -sinc 1000 -ssize 20	Captura o total de estádos especificados. Padrão: 100
-sinc	./unionfindapp -t forest -h compweighted -i uf1.txt -sstart 1000 -sinc 1000 -ssize 20	A cada quantas uniões o estado dos conjuntos deve ser capturado. Padrão: 1
-statistics	./unionfindapp -t forest -h compweighted -i uf1.txt -statistics statUF2LinkedListDefault.txt -sstart 1000 -sinc 1000 -ssize 20	Imprime as atualizações de ponteiro nas operações de união no arquivo especificado. Este arquivo possuirá o forma csv. Cada linha possuí a seguinte composição: x,y x: Define a união y:Número de operações de ponteiro Padrão: Não ativado

Tabela 1 – Parâmetros utilizados na execução da aplicação UnionFindApp.

Testes

Conforme as estruturas foram sendo implementadas foram executados testes unitários com base arquivos com número de operações reduzido. Isto ajudou a encontrar problemas que provavelmente passariam desapercebidos caso fossem utilizados somente os arquivos de entrada disponibilizados (que possuem elevado número de operações).

Bugs e Vazamento de Memória

Todos as execuções do programa realizadas neste trabalho foram feitas utilizando o valgrind. Em nenhuma delas foi detectada qualquer vazamento ou leak de memória. Para utilizar o valgrind instale-o e chame a aplicação como no exemplo abaixo:

valgrind --tool=memcheck --leak-check=yes ./unionfindapp -t linkedlist -h weighted -i uf1.txt -statistics StatisticUF1LinkedWeighted.csv -sstart 1000 -sinc 1000 -ssize 20

Resolução dos Exercícios com o Unionfindapp

Nesta seção serão mostrados os passos necessários para a obtenção das respostas requeridas nos exercícios A, B, C, D e E da prática 2. A solução dos exercícios será reaizada utilizando o aplicativo unionfindapp previamente desenvolvido.

Item A

O objetivo deste item é a realização das uniões de conjuntos disjuntos contidos nos arquivos "uf1.txt" e "uf2.txt". A estrutura de dados utilizada deve ser a de lista encadeada sem qualquer herística de união. São gravados 20 estados, sendo o primeiro estado após 1000 uniões, o segundo após 2000 uniões, o terceiro após 3000 uniões e assim sucessivamente até que após 20000 uniões se obtenha o vigésimo estado. Este problema pode ser solucionando executando o aplicativo unionfindapp como mostrado abaixo:

```
./unionfindapp -t linkedlist -h default -i uf1.txt -o outuf1LinkedDefault.txt -sstart 1000 -sinc 1000 -ssize 20
```

```
Numero de unioes:50000
Estrutura de dados: Lista Encadeada
Heuristica: Padrao
Arquivo de Entrada:uf1.txt
Arquivo de Saida:outuf1LinkedDefault.txt
Processando...
Escrevendo o estado 1000...
Escrevendo o estado 2000...
Escrevendo o estado 3000...
Escrevendo o estado 4000...
Escrevendo o estado 5000...
Escrevendo o estado 6000...
Escrevendo o estado 7000...
Escrevendo o estado 8000...
Escrevendo o estado 9000...
Escrevendo o estado 10000...
Escrevendo o estado 11000...
Escrevendo o estado 12000...
Escrevendo o estado 13000...
Escrevendo o estado 14000...
Escrevendo o estado 15000...
Escrevendo o estado 16000...
Escrevendo o estado 17000...
Escrevendo o estado 18000...
Escrevendo o estado 19000...
Escrevendo o estado 20000...
```

```
A captura de estados foi concluida...
A captura de operacoes de ponteiro esta desativada. Deseja continuar o processamento?
Sim(1), Nao(0):0
Concluido...
```

Perceba que após a captura de todos os estados requeridos existe ainda muitas instruções de união pendentes no arquivo “uf1.txt”. Como o aplicativo terminou de processar todos os estados requeridos ele pergunta se o usuário deseja continuar o processamento das uniões restantes. Como nosso objetivo já foi atingido foi requisitado para que o programa não continuasse com o processamento através da opção 0.

O item A também solicitou a execução das uniões contidas no arquivo “uf2.txt” utilizando estrutura de listas encadeadas sem qualquer heurísticade de união. Isto pode ser realizado executando o aplicativo unionfindapp como mostrado abaixo:

```
./unionfindapp -t linkedlist -h default -i uf2.txt -o OutUF2LinkedDefault.txt -sstart 1000 -sinc 1000 -ssize 20

Numero de unioes:50000
Estrutura de dados: Lista Encadeada
Heuristica: Padrao
Arquivo de Entrada:uf2.txt
Arquivo de Saida:OutUF2LinkedDefault.txt
Processando...
Escrevendo o estado 1000...
Escrevendo o estado 2000...
Escrevendo o estado 3000...
Escrevendo o estado 4000...
Escrevendo o estado 5000...
Escrevendo o estado 6000...
Escrevendo o estado 7000...
Escrevendo o estado 8000...
Escrevendo o estado 9000...
Escrevendo o estado 10000...
Escrevendo o estado 11000...
Escrevendo o estado 12000...
Escrevendo o estado 13000...
Escrevendo o estado 14000...
Escrevendo o estado 15000...
Escrevendo o estado 16000...
Escrevendo o estado 17000...
Escrevendo o estado 18000...
Escrevendo o estado 19000...
Escrevendo o estado 20000...
A captura de estados foi concluida...
A captura de operacoes de ponteiro esta desativada. Deseja continuar o processamento?
Sim(1), Nao(0):0
Concluido...
```

Vale ressaltar aqui que a visualização dos arquivos de saída em editores de texto como o Kate não são adequados. Editores de texto convencionais possuem um número limitado de caracteres por linha e quando este limite é ultrapassado ocorre uma quebra de linha

automática. Ou seja, nestes editores pode ser que elementos pertencentes a um estado assumam múltiplas linhas e não só uma como o enunciado requisita.

Item B

Neste item foi solicitada a mesma tarefa do Item A, porém utilizando a heurística de união ponderada em conjunto com a estrutura de listas encadeadas.

Para processar as operações de união do arquivo “uf1.txt” com conjuntos disjuntos implementados com listas encadeadas e heurística de união ponderada foi utilizado o aplicativo unionfindapp da seguinte forma:

```
./unionfindapp -t linkedlist -h weighted -i uf1.txt -o OutUF1LinkedWeighted.txt -sstart 1000 -sinc 1000 -ssize 20

Numero de unioes:50000
Estrutura de dados: Lista Encadeada
Heuristica: Uniao Ponderada
Arquivo de Entrada:uf1.txt
Arquivo de Saida:OutUF1LinkedWeighted.txt
Processando...
Escrevendo o estado 1000...
Escrevendo o estado 2000...
Escrevendo o estado 3000...
Escrevendo o estado 4000...
Escrevendo o estado 5000...
Escrevendo o estado 6000...
Escrevendo o estado 7000...
Escrevendo o estado 8000...
Escrevendo o estado 9000...
Escrevendo o estado 10000...
Escrevendo o estado 11000...
Escrevendo o estado 12000...
Escrevendo o estado 13000...
Escrevendo o estado 14000...
Escrevendo o estado 15000...
Escrevendo o estado 16000...
Escrevendo o estado 17000...
Escrevendo o estado 18000...
Escrevendo o estado 19000...
Escrevendo o estado 20000...
A captura de estados foi concluida...
A captura de operacoes de ponteiro esta desativada. Deseja continuar o processamento?
Sim(1), Nao(0):0
Concluido...
```

Para processar as operações de união do arquivo “uf2.txt” com conjuntos disjuntos implementados com listas encadeadas e heurística de união ponderada foi utilizado o aplicativo unionfindapp da seguinte forma:

```
./unionfindapp -t linkedlist -h weighted -i uf2.txt -o OutUF2LinkedWeighted.txt -sstart 1000 -sinc 1000 -ssize 20

Numero de unioes:50000
```

```
Estrutura de dados: Lista Encadeada
Heuristica: Uniao Ponderada
Arquivo de Entrada:uf2.txt
Arquivo de Saida:OutUF2LinkedWeighted.txt
Processando...
Escrevendo o estado 1000...
Escrevendo o estado 2000...
Escrevendo o estado 3000...
Escrevendo o estado 4000...
Escrevendo o estado 5000...
Escrevendo o estado 6000...
Escrevendo o estado 7000...
Escrevendo o estado 8000...
Escrevendo o estado 9000...
Escrevendo o estado 10000...
Escrevendo o estado 11000...
Escrevendo o estado 12000...
Escrevendo o estado 13000...
Escrevendo o estado 14000...
Escrevendo o estado 15000...
Escrevendo o estado 16000...
Escrevendo o estado 17000...
Escrevendo o estado 18000...
Escrevendo o estado 19000...
Escrevendo o estado 20000...
A captura de estados foi concluida...
A captura de operacoes de ponteiro esta desativada. Deseja continuar o processamento?
Sim(1), Nao(0):0
Concluido...
```

Item C

Para processar as operações de união do arquivo “uf1.txt” com conjuntos disjuntos implementados com árvores e sem qualquer heurística foi utilizado o aplicativo unionfindapp da seguinte forma:

```
./unionfindapp -t forest -h default -i uf1.txt -o OutUF1ForestDefault.txt -sstart 1000 -sinc 1000 -ssize 20

Numero de unioes:50000
Estrutura de dados: Floresta
Heuristica: Padrao
Arquivo de Entrada:uf1.txt
Arquivo de Saida:OutUF1ForestDefault.txt
Processando...
Escrevendo o estado 1000...
Escrevendo o estado 2000...
Escrevendo o estado 3000...
Escrevendo o estado 4000...
Escrevendo o estado 5000...
Escrevendo o estado 6000...
Escrevendo o estado 7000...
```

```
Escrevendo o estado 8000...
Escrevendo o estado 9000...
Escrevendo o estado 10000...
Escrevendo o estado 11000...
Escrevendo o estado 12000...
Escrevendo o estado 13000...
Escrevendo o estado 14000...
Escrevendo o estado 15000...
Escrevendo o estado 16000...
Escrevendo o estado 17000...
Escrevendo o estado 18000...
Escrevendo o estado 19000...
Escrevendo o estado 20000...
A captura de estados foi concluida...
A captura de operacoes de ponteiro esta desativada. Deseja continuar o processamento?
Sim(1), Nao(0):0
Concluido...
```

Para processar as operações de união do arquivo “uf2.txt” com conjuntos disjuntos implementados com árvores e sem qualquer heurística foi utilizado o aplicativo unionfindapp da seguinte forma:

```
./unionfindapp -t forest -h default -i uf2.txt -o OutUF2ForestDefault.txt -sstart 1000 -sinc 1000 -ssize 20

Numero de unioes:50000
Estrutura de dados: Floresta
Heuristica: Padrao
Arquivo de Entrada:uf2.txt
Arquivo de Saida:OutUF2ForestDefault.txt
Processando...
Escrevendo o estado 1000...
Escrevendo o estado 2000...
Escrevendo o estado 3000...
Escrevendo o estado 4000...
Escrevendo o estado 5000...
Escrevendo o estado 6000...
Escrevendo o estado 7000...
Escrevendo o estado 8000...
Escrevendo o estado 9000...
Escrevendo o estado 10000...
Escrevendo o estado 11000...
Escrevendo o estado 12000...
Escrevendo o estado 13000...
Escrevendo o estado 14000...
Escrevendo o estado 15000...
Escrevendo o estado 16000...
Escrevendo o estado 17000...
Escrevendo o estado 18000...
Escrevendo o estado 19000...
Escrevendo o estado 20000...
A captura de estados foi concluida...
A captura de operacoes de ponteiro esta desativada. Deseja continuar o processamento?
Sim(1), Nao(0):0
Concluido...
```

Item D

Para processar as operações de união do arquivo “uf1.txt” com conjuntos disjuntos implementados com árvores e heurística de união ponderada e compressão de caminhos, foi utilizado o aplicativo unionfindapp da seguinte forma:

./unionfindapp -t forest -h compweighted -i uf1.txt –o OutUF1ForestCompweighted.txt -sstart 1000 -sinc 1000 -ssize 20

```
Numero de unioes:50000
Estrutura de dados: Floresta
Heuristica: Uniao Ponderada e Compressao de Arvores
Arquivo de Entrada:uf1.txt
Arquivo de Saida: OutUF1ForestCompweighted.txt
Processando...
Escrevendo o estado 1000...
Escrevendo o estado 2000...
Escrevendo o estado 3000...
Escrevendo o estado 4000...
Escrevendo o estado 5000...
Escrevendo o estado 6000...
Escrevendo o estado 7000...
Escrevendo o estado 8000...
Escrevendo o estado 9000...
Escrevendo o estado 10000...
Escrevendo o estado 11000...
Escrevendo o estado 12000...
Escrevendo o estado 13000...
Escrevendo o estado 14000...
Escrevendo o estado 15000...
Escrevendo o estado 16000...
Escrevendo o estado 17000...
Escrevendo o estado 18000...
Escrevendo o estado 19000...
Escrevendo o estado 20000...
A captura de estados foi concluida...
A captura de operacoes de ponteiro esta desativada. Deseja continuar o processamento?
Sim(1), Nao(0):0
Concluido...
```

Para processar as operações de união do arquivo “uf2.txt” com conjuntos disjuntos implementados com árvores e heurística de união ponderada e compressão de caminhos, foi utilizado o aplicativo unionfindapp da seguinte forma:

./unionfindapp -t forest -h compweighted -i uf2.txt -o OutUF2ForestCompweighted.txt -sstart 1000 -sinc 1000 -ssize 20

```
Numero de unioes:50000
Estrutura de dados: Floresta
Heuristica: Uniao Ponderada e Compressao de Arvores
Arquivo de Entrada:uf2.txt
Arquivo de Saida: OutUF2ForestCompweighted.txt
Processando...
```

```
Escrevendo o estado 1000...
Escrevendo o estado 2000...
Escrevendo o estado 3000...
Escrevendo o estado 4000...
Escrevendo o estado 5000...
Escrevendo o estado 6000...
Escrevendo o estado 7000...
Escrevendo o estado 8000...
Escrevendo o estado 9000...
Escrevendo o estado 10000...
Escrevendo o estado 11000...
Escrevendo o estado 12000...
Escrevendo o estado 13000...
Escrevendo o estado 14000...
Escrevendo o estado 15000...
Escrevendo o estado 16000...
Escrevendo o estado 17000...
Escrevendo o estado 18000...
Escrevendo o estado 19000...
Escrevendo o estado 20000...
A captura de estados foi concluida...
A captura de operacoes de ponteiro esta desativada. Deseja continuar o processamento?
Sim(1), Nao(0):0
Concluido...
```

Item E

Neste item são comparados os números de operações de ponteiro realizadas por cada uma das estruturas de dados implementadas durante o processamento dos arquivos “uf1.txt” e “uf2.txt”. A computação das operações de ponteiro levará em consideração as atualizações e movimentações de ponteiro referentes membros da estrutura de dados de conjuntos disjuntos. As atualizações referem-se a modificação do endereço que o ponteiro aponta. A movimentação trata-se da ação de caminhar pelos nós de uma árvore ou lista encadeada. Para gerar as tabelas de número de operações de ponteiro por uniões utilize o parâmetro –statistics. Você pode visualizar alguns exemplos na ultima seção “Gerando arquivos *.csv com o programa”.

Processamento do arquivo “uf1.txt”

O arquivo “uf1.txt” define 50000 elementos e 49999 operações de união. Embora o número de uniões seja grande, as operações descritas seguem um padrão bem simples, unir sucessivamente os conjuntos dos elementos em ordem crescente. Desta maneira após a primeira união sempre ocorrerá a união entre um conjunto com N>2 elementos e um conjunto com 1 elemento. Deve-se ressaltar que N inicialmente é 1 e é incrementado acada união, tornando o primeiro conjunto da união cada vez maior. Isto será interessante para analisar não

só o desempenho das estruturas de dados implementadas, mas as heurísticas utilizada conjuntamente.

O Gráfico 1 mostra um comparativo entre as estruturas de lista encadeada com heurística, floresta sem heurística e floresta com união ponderada e compressão de caminhos paras as uniões do arquivo “uf1.txt”. O Gráfico 2 mostra a estrutura de listas encadeadas sem heurística. Ela foi posta separadamente das demais curvas por apresentar um número de operações tão elevado que dificultava a visualização demais curvas.

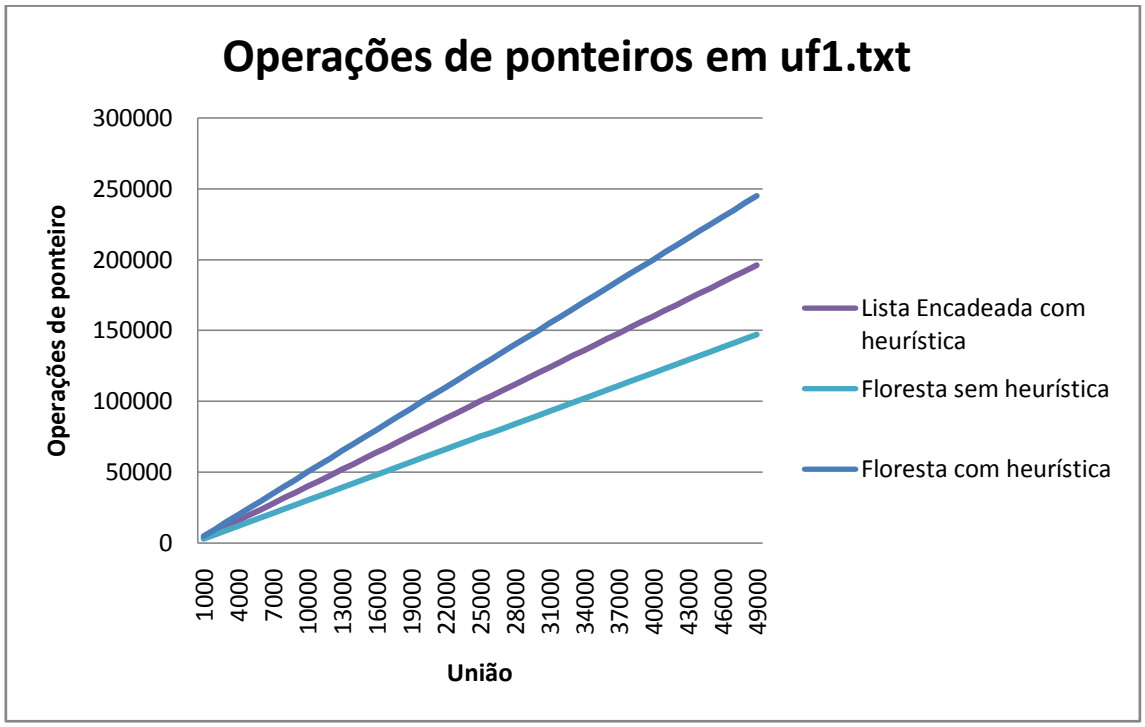

Gráfico 1 - Comparativo das estrturas de dados implementados com listas encadeadas com heurística de união ponderada, floresta sem heurística e floresta com heurística de união ponderada e compressão de caminhis.

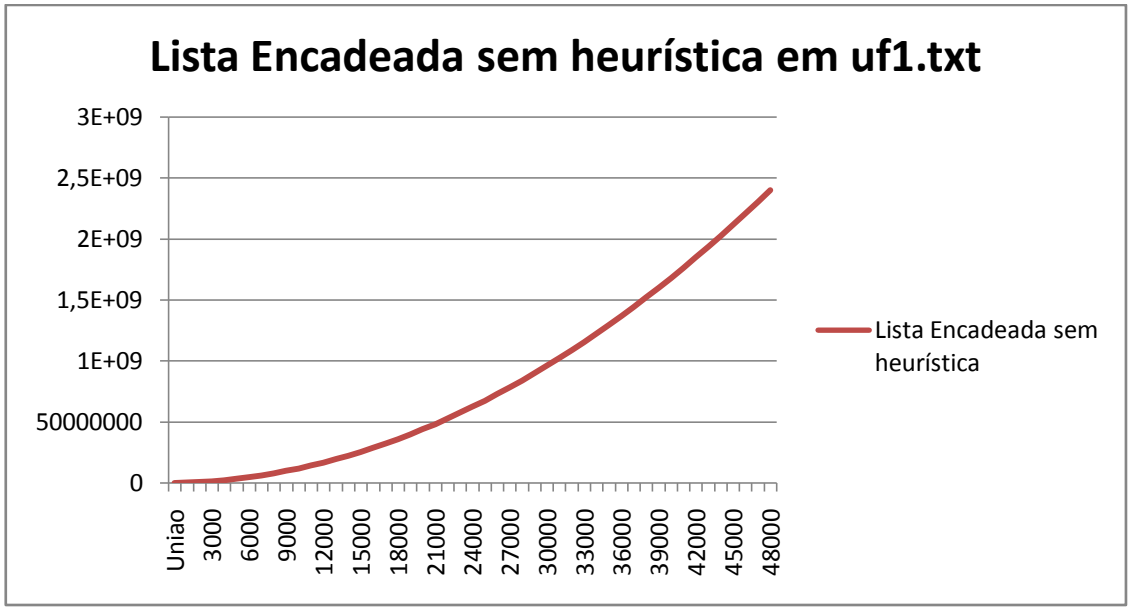

Gráfico 2 – Mostra o comportamento da estrutura de lista encadeada com união ponderada.

Visualizando os Gráficos 1 e 2 fica evidente que a estrutura que obteve o maior número de operações de ponteiro foi a lista encadeada sem heurística. Isto aconteceu porque em todos as uniões em "uf1.txt" ocorria o pior caso para união nesta estrutura, isto é, unir um conjunto maior com um conjunto menor. No caso, o conjunto menor sempre possuia apenas um elemento, enquanto o outro, um número de elementos que era incrementado a cada operação de união. A floresta com heurística obteve um resultado melhor. Mesmo assim a compressão de caminhos ao invés de ajudar acabou gerando uma operação de ponteiro a mais descenessária. Isto será analisado com maior detalhe nas seções seguintes. A lista encadeada com heurística obteve o segundo melhor resultado. O melhor resultado foi para a estrutura que implementa a floresta sem heurística. **A implementação de lista encadeada com heurística poderia ter obtido resultado similar ao das florestas sem heurística. Como será abordado nas seções seguintes uma atualização de ponteiro para a implementação de listas encadeadas poderia ter sido eliminada. Isto faria com que ela obtivesse aproximadamente o mesmo número de operações de ponteiro que a implementação de floresta sem heurística**. Como será visto, a inclusão de uma operação de ponteiro na estrutura de listas encadeadas permite listar todos os elementos pertencentes a um conjunto a partir do elemento representante. Isto não seria possivel caso fosse eliminado esta operação de ponteiro.

Como o arquivo "uf1.txt" é de facil compreensão as seções seguintes analisam os casos de união para cada estrutura.

Listas Encadeadas sem heurística

Na implementação com listas encadeadas sem heurística a primeira união gera 4 operações de ponteiro, a segunda 6, a terceira 8 e assim sucessivamente. Isto ocorre pois a cada união do conjunto X ao conjunto Y são gastos:

1 atualização do ponteiro “próximo” da cauda de Y, sendo apontando para a cabeça de X.
1 atualização do ponteiro da cauda de Y, sendo apontando para a cauda de X.

Estas duas atualizações sempre irão ocorrer na implementação de listas encadeadas, independentemente do número de elementos em cada conjunto. Existem ainda as atualizações que dependem do número de elementos de X, elas envolvem a atualização do ponteiro do representante para cada nó (elemento) em X. Estes ponteiros passam a apontar para o representante de Y. Por exemplo, na primeira união, onde Y possuia apenas 1 elemento houve a necessidade de apenas 1 atualização de representante e apenas 1 movimentação para identificar que a lista encadeada havia chegado ao fim. Somando as operações de lista que sempre ocorrem com as atualizações dos representantes temos 4 operações de ponteiro para a primeira união. Na segunda união, X possui 2 elementos e Y apenas 1, então serão necessários 2 movimentações e 2 atualizações de representante, somando com o gasto fixo de união das listas temos um total de 6 operações de ponteiro. Isto pode ser conferido no método internalUnion(x,y) da classe LinkedListSet localizada no diretório /src/setlib.

**** **Poderia ser economizada uma atualização de ponteiro retirando “1 atualização do ponteiro “próximo” da cauda de Y, sendo apontando para a cabeça de X”. Isto faria com que o número de operações fixas caisse de 2 para 1. Preferi no entanto, manter esta operação de ponteiro tornando possivel que se faça uma navegação do representante até todos os elementos do conjunto. Retirando esta atualização isto não seria viável.***

Listas Encadeadas com União Ponderada

Como cada lista possui o número de elementos esta heurística sempre irá fazer a união da lista menor na lista maior. Neste caso , em “uf1.txt” sempre são unidas uma lista com 1 elemento a uma lista com mais elementos. Isto garante que sempre ocorram 4 operações de ponteiro por união. Estas operações de ponteiro são similares às descritas na seção anterior para a união de dois conjuntos com apenas um elemento cada.

*****A mesma economia mencionada para listas encadeadas sem união ponderada poderia ser feita aqui. Diminuindo uma operação de ponteiro, cada união em “uf1.txt” exigiria apenas 3 operações de ponteiro. Isto equivaleria ao mesmo desempenho obtido pela implementação de floresta sem heurística.**

Floresta sem Heurística

Na floresta sem heurística sempre o conjunto X é unido ao conjunto Y. A primeira operação no processo de união é encontrar o representante de cada conjunto. Isto é feito partindo de cada nó representante da união e subindo até raiz. Por sorte o arquivo “uf1.txt” sempre realiza uma união entre o nó representante do conjunto X com o nó representante e único do conjunto Y. Isto confere a estrutura uma **um númerto constante de 3 operações** de ponteiro por união em “uf1.txt”:

1 operação para percorrer o próximo elemento de X (aquele mais acima na árvore) e verificar que é o elemento representante.

1 operação para percorrer o próximo elemento de Y (aquele mais acima na árvore) e verificar que é o elemento representante.

1 operação para atualizar o ponteiro “próximo” da raiz de X para apontar para a raiz de Y.

Apenas reforçando, este procedimento sempre ocorre considerando que estamos analisando as uniões do arquivo “uf1.txt”. Em outras casos certamente não teriamos o mesmo comportamento.

Floresta com heurística de União Ponderada e Compressão de Caminhos

Na floresta com heurística na primeira união o conjunto X é unido a Y, ou seja, 0 é unido a 1. Nas demais sempre é realizado uma união de Y em X, já que estamos utilizando união ponderada. Na composição de “uf1.txt” o primeiro conjunto será sempre maior após a primeira união.

Analisando a união(0,1) em, “uf1.txt”. A primeira operação realizada na união é encontrar os representantes de X e Y. Na primeira união é necessário apenas uma movimentação de ponteiro para verificar que o elemento ja é o nó representante. Como isto deve ser feito para X e Y, temos 2 movimentações. Identificando o representante, atualizamos o ponteiro “próximo” do representante de X para apontar para o representante de Y, tendo para isso mais 1 atualização de ponteiro. No total a primeira união requer apenas 3 movimentações de ponteiro.

Agora temos o conjunto com elementos 0 e 1, sendo 1 o representante. A operação seguinte descrita em “uf1.txt” é unir(1,2). Pela união pondera 2 será unido ao conjunto de 1. Novamente o algoritmo irá buscar a raiz da árvore para os dois conjuntos, assim como no caso anterior serão necessárias 3 atualizações.

Agora temos o conjunto com elementos 0,1 e 2 sendo 1 o representante. A próxima operação em “uf1.txt” é unir(2,3). Neste caso 2 não é mais o representante . O algoritmo necessitará de dois passos para encontrar o representante. Primeiro verifica que 2 não é a raiz, depois realiza mais outro passo encontrando propriamente a raiz. Como estamos utilizando compressão de caminhos o elemento dois terá seu ponteiro “próximo” atualizado para pontar para o representante (embora já o esteja fazendo). Até o momento temos 3 operações de ponteiro. Para encontrar o represetante de 3 é exigido mais um único passo, visto que ele é o próprio representante. Agora deve-se atualiza o ponteiro “próximo” do representante de 3 para apontar para o representante de 2. Totalizando 5 operações de ponteiro para a união(2,3) . Este mesmo caso é repetido no restantedas uniões descritas no arquivo “uf1.txt”.

Fica claro que a compressão de caminhos não está ajudando em nada na redução de operações de ponteiro nas uniões descritas em “uf1.txt”. Na verdade elas apenas aumentam o número de operações.

Processamento do arquivo “uf2.txt”

Este arquivo apresenta ter uniões aleatória o que dificulta uma análise mais detalhada dos resultados. Contudo, a análise realizada na seção anterior para o arquivo “uf1.txt” deve ser suficiente para a compreesão das estruturas de dados.

O Gráfico 3 mostra um comparativo entre as estruturas de lista encadeada com heurística e a floresta com união ponderada e compressão de caminhos paras as uniões do arquivo “uf2.txt”. Como podemos observar a floresta com heurística está realizando menor número de operações de ponteiro. Isto ocorre porque a compressão de caminhos está sendo utilizada efetivamente.

O Gráfico 4 mostra as curvas da estrutura de listas encadeadas sem heurística e floresta sem heurística. Elas foram postas separadamente por apresentarem um número de operações de ponteiro tão elevadas que dificultavam a visualização da demais curvas. Pode-se perceber melhor eficiência na floresta sem heurística do que nas listas encadeadas sem heurística. Embora sabemos que o número de operações de ponteiro poderia ser diimuído de 1 a cada união com listas encadeadas isto não é suficiente para o grau de disparidadeapresentado entre as curvas.

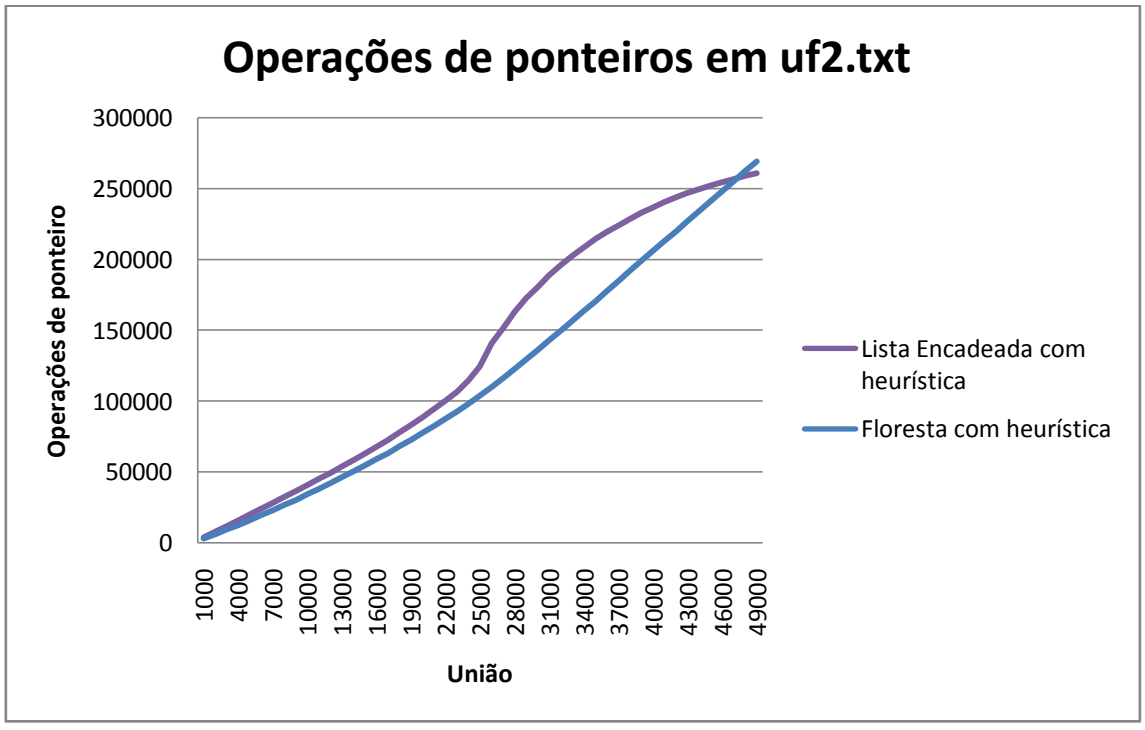

Gráfico 3 – Comparativo entre as estruturas de dados de lista encadeada com heurística de união ponderada e floresta com heurística de união ponderada e compressão de caminhos.

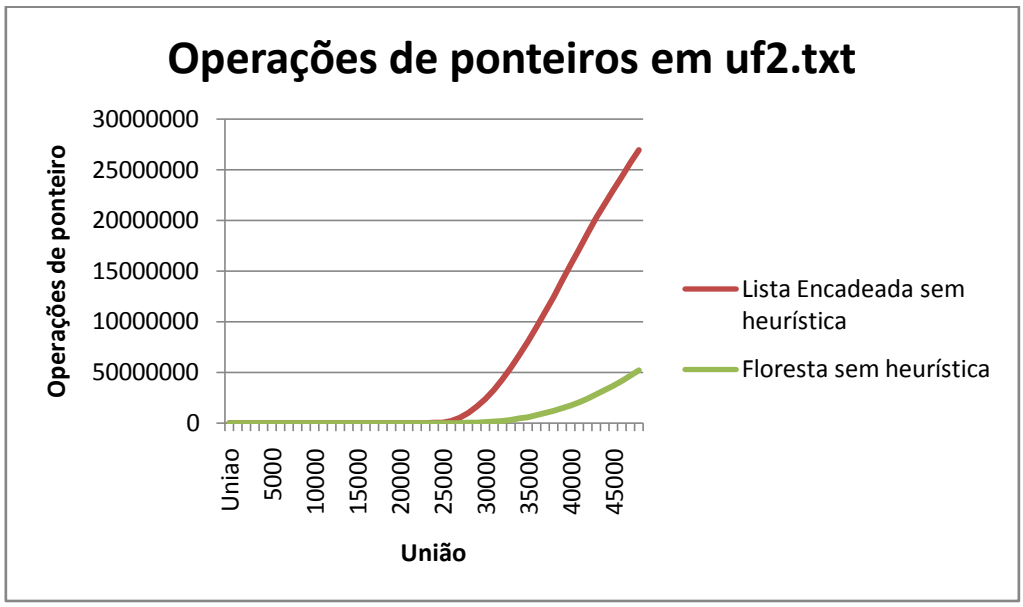

Gráfico 4 – Comparativo entre as estruturas de dados de lista encadeada sem heurística e floresta sem heurística.

Para o arquivo "uf2.txt" a estrutura de floresta com heurística ponderada e compressão de caminhos foi a mais eficiente. Em segundo lugar, a estrutura de listas encadeadas com heurística de união ponderada. Em terceiro, a estrutura de florestas sem heurística e em quarto a lista encadeada sem herurística.

*Gerando arquivos *.csv com o programa*

Embora tenha descrito os parâmetros de entrada do programa na seção "Interface de usuário", abaixo estão as chamadas utilizadas para a geração dos arquivos csv utilizados para a geração dos gráficos:

Gera o arquivo "StatisticUF1LinkedDefault.csv" para listas encadeadas com heurística ponderada aplicadas no arquivo "uf1.txt":

```
./unionfindapp -t linkedlist -h default -i uf1.txt -statistics StatisticUF1LinkedDefault.csv
```

Gera o arquivo "StatisticUF1LinkedWeighted.csv" para listas encadeadas com heurística ponderada aplicadas no arquivo "uf1.txt":

```
./unionfindapp -t linkedlist -h weighted -i uf1.txt -statistics StatisticUF1LinkedWeighted.csv
```

Gera o arquivo "StatisticUF1ForestDefault.csv" para floresta sem heurística aplicadas no arquivo "uf1.txt":

```
./unionfindapp -t forest -h default -i uf1.txt -statistics StatisticUF1ForestDefault.csv
```

Gera o arquivo “StatisticUF1ForestCompweighted.csv” para floresta sem heurística aplicadas no arquivo “uf1.txt”:

```
./unionfindapp -t forest -h compweighted -i uf1.txt -statistics StatisticUF1ForestCompweighted.csv
```

Para o arquivo “uf2.txt” foi apenas substituído o argumento de – statistics.

Conclusão

A implementação das estruturas de conjuntos disjuntos foi proveitosa para verificar que a eficiência de cada estrutura pode variar em relação ao tipo de operações de união que estão sendo realizadas. No processamento das operações de união do arquivo “uf1.txt” a estrutura de floresta com heurística ponderada e compressão de caminhos obteve o segundo pior resultado. Naquele caso a compressão de caminhos não era benéfica. No processamento das operações de união do arquivo “uf2.txt” a mesma estrutura obteve a melhor eficiência. A compressão de caminhos passou a torna-la mais eficiente neste caso. Foi mostrado também que curiosamente, para o arquivo “uf1.txt” a eficiência da floresta sem heurística e lista encadeada com união ponderada poderia obter eficiência similar. Por sua vez, a estrutura de listas encadeadas sem heurística obteve os piores resultados para os arquivos “uf1.txt” e “uf2.txt”.