Trabalho Prático 2

Alocação de Canais em Redes De Telefonia Celular

Aluno: Enzo Roiz

Matrícula: 2013062839

Índice

Descrição do problema

Uma nova empresa de telefonia móvel pretende prover serviços a baixo custo em Belo Horizonte. Nos serviços de telefonia móvel, tem-se um espectro de frequências destinados ao seu uso. Esse espectro por sua vez é dividido em canais, regulados pela Anatel, agência reguladora de serviços de telefonia. A empresa pretende então utilizar apenas um canal em uma *ERB* (Estação Rádio Base). O problema consiste então em, dadas as *ERB's* e seus raios de atuação, *ERB's* que contenham áreas de cobertura comum, ou seja, áreas de interferência, não funcionem num mesmo canal. A ideia é então alocar os canais nas *ERB's* de forma a se utilizar o mínimo de canais possível. Assim o programa desenvolvido deve ter como solução o número mínimo de canais necessários para o funcionamento correto dos serviços prestados.

Modelagem e Solução Proposta

O problema foi modelado utilizando grafos. As *ERB's* são então os vértices do grafo e as arestas ligam *ERB's* que tenham interferência, área de cobertura em comum. Considerou-se então o grafo como sendo não direcionado, pois se uma *ERB* “X” sofre interferência de outra *ERB* “Y”, então “Y” também sofre interferência de “X”, e não ponderado, pois o peso de cada aresta não nos interessa.

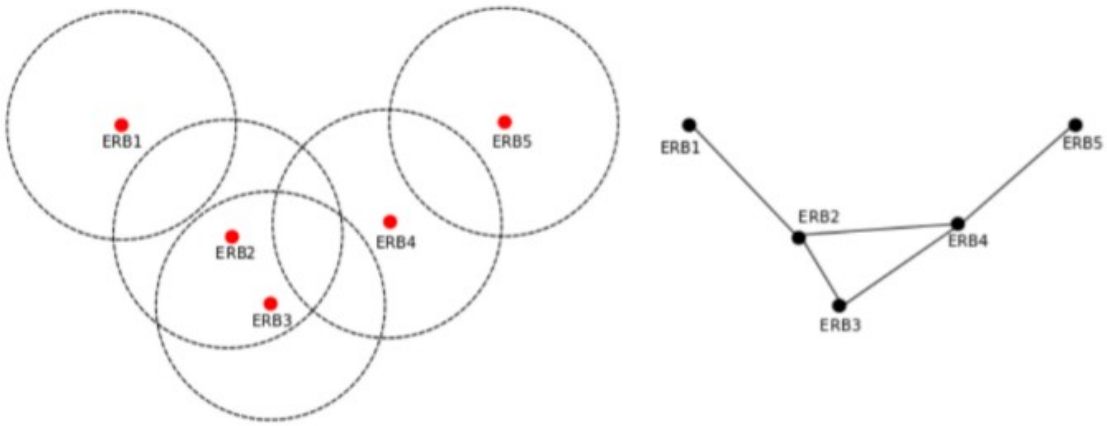

Figura 1 - Representação das áreas de interferência em ERB's usando grafo

Com o problema modelado, podemos perceber que vértices que estão ligados por uma aresta, não podem conter a mesma informação, no caso do problema, mesmo canal. Caímos então em um conhecido problema computacional, a *coloração de grafos,* onde a partir de um determinado grafo, damos cores aos vértices de modo que vértices ligados não tenham a mesma cor e que se use o mínimo de cores possível para colorir todo o grafo.

Implementação

Para este trabalho, serão desenvolvidos dois programas. Um contendo a solução ótima, ou seja, testará todas as possibilidades para atribuição de canais às *ERB's,* tendo como solução aquela que utilizar o menor número de canais. Outro contendo heurística, ou seja, uma solução aproximada do ótimo, sendo aceitável, visto o tempo que gasta, muito menor que o tempo necessário para a resolução do problema considerando-se o ótimo.

Como anteriormente dito, foi utilizada a estrutura de dados grafo para modelar o problema. Para a implementação da estrutura de dados foi utilizado então grafos por meio de listas de adjacência. Listas de adjacência consistem em um vetor de listas, com tamanho igual ao número de vértices. Assim cada vértice tem uma lista. Cada lista de um vértice "*u*" contém então os vértices adjacentes a ele, ou seja, os vértices aos quais "*u*" está ligado.

Heurística

A ideia da heurística é aproximar a solução daquela considerada a solução ótima. Para tanto a heurística implementada funciona da seguinte forma.

```
IncolorGrafo(); //Define todos os vértices não estarem coloridos

CriaListaParaCor(ListaDeColoridos); //Cria uma lista vazia para inserir os vértices coloridos
                                    //pela cor "c"

Enquanto não coloriu todos os vértices (VérticesColoridos != NumVertices)

    Para cada vértice "u" do grafo

        Se o vértice "u" não estiver colorido

            Algum vértice adjacente a "u" está na lista de coloridos pela cor "c" ?

            Se não:

                ColoreVertice(c);//Colore o vértice com a cor c

                InsereVerticeLista(ListaDeColoridos, u);//Insere o vértice "u" na
                                                        //lista de vértices
                                                        //coloridos pela cor c

                VérticesColoridos ++;

    c++;//Aumenta a cor em uma unidade. Fora do for, dentro do while;

    EsvaziaListaColoridos(ListaDeColoridos);//Esvazia lista de vértices coloridos pela cor c.
                                            //Fora do for, dentro do while
```

Assim, enquanto os vértices não foram todos coloridos, para os vértices não coloridos, é olhada a lista de vértices adjacentes a ele. Caso nenhum dos adjacentes do vértice em questão esteja na lista da cor “c” atual, ou seja, nenhum dos adjacentes está colorido com a cor c, colore-se então o vértice com a cor “c”, o insere na lista e conta mais um vértice colorido. Após percorrer todos os vértices, dentro do *loop “for”*, muda-se então a cor e esvazia a lista de vértices coloridos com a cor c anterior. Assim a lista de vértices coloridos com uma determinada cor “c” sempre estará vazia quando os vértices forem ser percorridos.

A analogia feita com coloração de grafos tem intenção de facilitar o entendimento. No caso, os vértices seriam as *Estações de Rádio Base* e as cores dos vértices seriam os canais de cada *ERB.*

Em alguns casos, a heurística tem o mesmo resultado do algoritmo ótimo, mas em outros não, por se tratar de um algoritmo que tende a aproximar o resultado do ótimo. Como vimos o algoritmo percorre todos os vértices a fim de “colori-los”, então depende da ordem da entrada para percorrer os vértices. Segue um exemplo no qual a heurística falha.

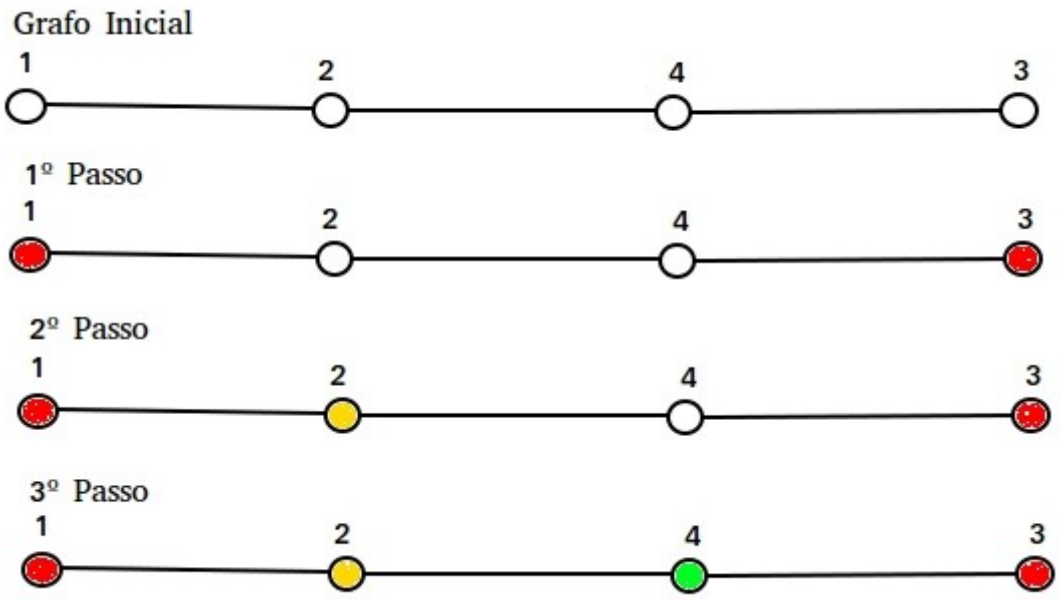

Figura 2 – Exemplo de falha na heurística

No grafo apresentado, o fato de a ordem dos vértices estar invertida para os vértices 3 e 4, implica em um uso maior de cores do que o necessário, já que o grafo acima poderia ser colorido com 2 cores. É colorido o vértice 1 de vermelho. O vértice 2 não é colorido de vermelho por ter adjacente com esta cor. O vértice 3 é também colorido de vermelho. Muda-se a cor para amarelo. O vértice 2 é colorido de amarelo. O vértice 4 não, pois tem adjacente com cor amarela. Muda-se a cor para verde e finalmente o vértice 4 é colorido.

Funções

Para a implementação do trabalho foram usadas as seguintes funções:

void FLVazia(TipoLista *);

short Vazia(TipoLista);

void Insere(TipoItem *, TipoLista *);

void Retira(TipoApontador, TipoLista *, TipoItem *);

void EsvaziaLista(TipoLista *);

As funções acima fazem operações sobre listas encadeadas. Inicializa uma lista, verifica se está vazia, insere e retira itens e esvazia uma lista encadeada.

void FGVazio(TipoGrafo *);

void InsereAresta(int *, int *, TipoGrafo *);

void LiberaGrafo(TipoGrafo *);

As funções acima listadas fazem operações sobre um grafo. Inicializa um grafo, insere uma aresta ligando dois vértices e libera a memória alocada para a construção de um grafo.

void ConstroiGrafo(TipoEstacao *, int, TipoGrafo *);

int TemInterferencia(TipoEstacao, TipoEstacao);

Estas funções são utilizadas para construir o grafo baseado nas posições das *ERB's* e seus raios de atuação. Na função *ConstroiGrafo()* é feita a iteração para cada vértice com os demais. Assim, cada vértice é comparado aos demais dentro da função *TemInterferencia()* visando buscar interferências. Para verificar interferência, é comparada a distância entre as *ERB's* e a soma dos raios de atuação. Caso a distância entre as *ERB's* seja maior que a soma dos raios, não há interferência. Caso contrário há interferência e é inserida uma aresta no grafo. Porém como o grafo é não direcionado, então são inseridas duas arestas. Uma de “X” para “Y” e outra de “Y” para “X”. Os vértices nunca são comparados 2 vezes, visando diminuir a complexidade da função. Por exemplo, se os vértices 1 – 4 já foram verificados, não é feita nova verificação para 4 – 1.

int Cores(TipoGrafo);

int AdjacenteColorido(TipoLista, TipoLista);

As funções acima são responsáveis por “colorir” o grafo. O seu funcionamento foi descrito através de pseudo-código ao explicar a heurística adotada. A função *AdjacenteColorido()* faz parte da função *Cores()* e busca por adjacentes coloridos ao vértice que está sendo verificado.

Análises

Complexidade temporal

As funções que operam sobre listas são **O(1)**, por se tratarem de funções que independem do tamanho da entrada, com exceção à função *EsvaziaLista()* que é **O(c)**, onde **c** é o número de vértices coloridos com uma determinada cor.

As funções que operam sobre grafo têm complexidade: *FGVazio(),* **O(v)**, onde **v** é o número de vértices do grafo. *InsereAresta(),* **O(1)** e *LiberaGrafo(),* **O(a)**, onde **a** é o número de arestas existentes no grafo.

A função *ConstroiGrafo()* contém a função *TemInterferencia().* A complexidade da função *TemInterferencia()* é **O(1)**. A função *ConstroiGrafo()* tem complexidade da ordem de **O(n²),** apesar da ligeira melhoria implementada, evitando comparar vértices duas vezes.

A função com maior complexidade é então a função *Cores().* Nela são feitas diversas iterações a fim de “colorir” o grafo. Sendo elas:

1 – Enquanto o grafo não estiver colorido. Pior caso → **O(v).**

2 – Para cada vértice não-colorido do grafo → **O(v).**

3 – Verifica lista de adjacentes→ **O(v-1)** → **O(v).**

4 – Verifica lista de vértices coloridos com cor “c” → **O(v-1)** → **O(v).**

Logo teremos aparente complexidade da ordem de **O(v⁴).** Porém podemos perceber no pior caso, os vértices serem coloridos um por vez. Ou seja, a cada passada no grafo realizada pela função, apenas um vértice será colorido. Assim, teremos **O(v)** por colorir apenas um vértice por vez. **O(v)** por percorrermos a lista de vértices. **O(v)** por percorrer a lista de adjacências que poderá conter no máximo **v-1** arestas. Porém teremos **O(1)** ao verificar a lista de vértices coloridos com a cor “c”, uma vez que como colorimos um vértice por vez, esta lista estará sempre vazia. Então a complexidade real será **O(v³).**

Os testes foram feitos utilizando entradas desconsiderando *ERB's* concêntricas ou casos onde a área de atuação de uma *ERB* engloba toda área de atuação de outra *ERB*.

Foi considerado um espaço (X,Y) de 70 x 70, com vértices aleatórios e raios com tamanho máximo de 3. O código da função que gera os testes está comentada no código.

Segue abaixo o gráfico gerado:

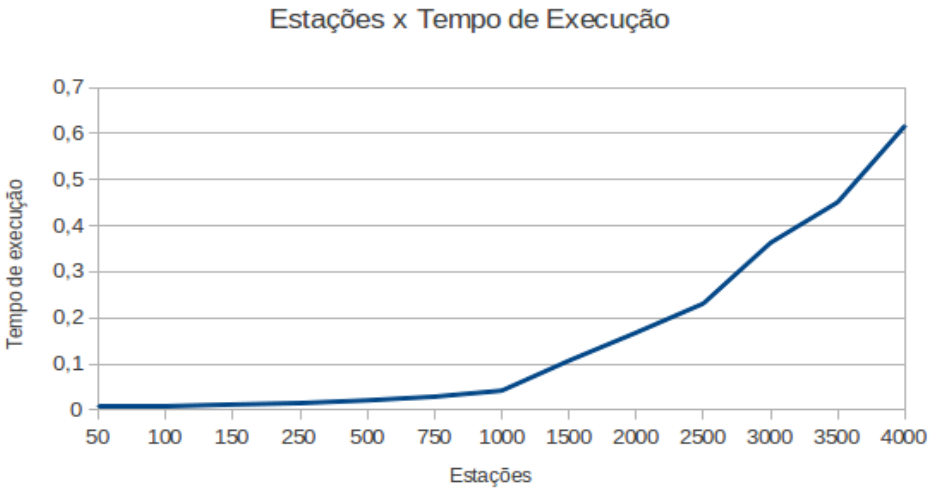

Figura 3 – Gráfico Estações x Tempo de Execução para heurística

Através do gráfico podemos perceber que o tempo de execução aumenta significativamente conforme a entrada (número de *ERB's*) aumenta. Porém segundo a especificação do trabalho, o número máximo de *ERB's* para a heurística é 4 mil. Assim o tempo para 4 mil estações é satisfatório, sendo menor que 1 segundo. Além disso o número de canais necessários para o funcionamento pleno do serviço de telefonia segundo a heurística para este número de estações é 15, o que também é um resultado satisfatório devido ao grande número de estações.

Complexidade espacial

A complexidade espacial trata-se do tamanho da memória utilizada em função da entrada para o funcionamento do programa. Assim, para heurística teremos: **O(v)** → para armazenar os dados de entrada das *ERB's,* coordenadas “x” e “y”, além do raio de atuação.

O(2 * a) → **O(a)** → para armazenar as arestas após construção do grafo. Como o grafo é não direcionado, é inserida uma aresta de ida e uma de volta.

Portanto, para a construção do grafo temos complexidade espacial **O(v+a).**

Para a função *Cores()* que avalia o número de canais mínimos necessários teremos:

O(v) → Uma vez que para armazenar os vértices já verificados pelo algoritmo eles são inseridos em uma lista encadeada. Assim, armazenamos **v** vértices na lista encadeada.

Logo a complexidade espacial do programa é **O((2*v) + (2*a)) → O(v+a).**

Solução Ótima

Como dito ao explicar a heurística, ela é uma maneira de se resolver o problema de “coloração de grafos”, porém depende de como está organizada a entrada para o seu funcionamento. Então, caso a entrada do programa, a ordem dos vértices, esteja correta, ao aplicarmos a heurística sobre estes vértices, teremos então a solução ótima. O que devemos fazer é então permutar os vértices, ou seja, ordená-los de forma a termos todas as combinações possíveis e aplicar a heurística em cima de todas as combinações.

Funções

void Permuta(TipoGrafo *, int *, int *, int);

void PermutaRecursiva(TipoGrafo *, int *, int *, int, int);

void TrocaPosicao(int *, int, int);

As funções acima fazem a permutação dos vértices por meio de recursividade. Definem todas as combinações possíveis para a ordem da entrada e então aplicam a heurística a cada uma delas, fornecendo assim o resultado ótimo.

void ConverteGrafo(TipoGrafo *, int *);

void RearranjaVertices(TipoGrafo *, int *, int);

Estas funções rearranjam os vértices quando é feita a permutação. Por exemplo, ao trocarmos o vértice 0 com o vértice 1, precisamos então mudar todos os vértices que referenciavam 1 como adjacente para 0 e os vértices que referenciavam 0 para 1. Isto é feito para todos os vértices, o que faz com que apesar de mudada a ordem de entrada, o grafo permaneça o mesmo.

Como o algoritmo utiliza a heurística para encontrar a solução ótima, as funções da heurística são também utilizadas para o programa que gera a solução ótima.

Análises

Complexidade temporal

Para a solução ótima, devido às permutações realizadas na entrada a fim de se obter o ótimo, temos **O(v!)** para esta operação. Como é aplicada heurística para cada combinação possível, a complexidade do algoritmo ótimo é então **O(v! * v³).**

Devido a esta alta complexidade o algoritmo ótimo tem funcionamento consideravelmente lento para entradas a partir de 11 *ERB's*. Segue abaixo o gráfico gerado:

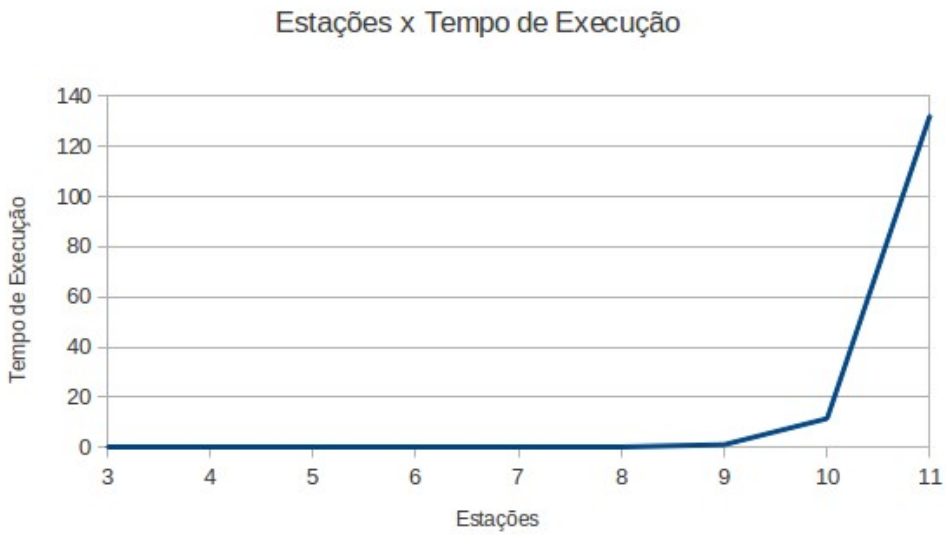

Figura 4 – Gráfico Estações x Tempo de Execução para solução ótima

Complexidade espacial

Como anteriormente visto, a complexidade para a montagem do grafo é **O(v+a).** O que varia então da heurística para o ótimo é que na lista de vértices "coloridos", são inseridos **v!** Itens, alocando memória adicional para estes **v!** itens. Apesar disso, a memória não "estoura", pois a cada passada que a heurística dá, colorindo novos vértices, esta lista é esvaziada, desalocando assim a memória. Conclui-se então que a complexidade espacial para o algoritmo ótimo é **O(v!).**

Utilitário Makefile

Para este trabalho, foram criados 2 projetos separados. Um para a heurística e outro para a solução ótima. Assim, 2 *main* foram criadas. Para a heurística *mainh.c* e para a solução ótima *maino.c.* A fim de compilar e executar o programa foram criadas então duas regras para o utilitário Makefile.

- *run_h* → compila o programa que gera a solução heurística e o executa para o arquivo de entrada *input.txt* e de saída *output.txt*.
- *run_o* → compila o programa que gera a solução ótima e o executa para o arquivo de entrada *input.txt* e de saída *output.txt*.

Assim, para compilar e executar os programas pela linha de comando basta digitar *make run_h,* para a heurística, e *make run_o,* para o ótimo.

Máquina utilizada

A máquina utilizada no desenvolvimento e testes dos algoritmos foi um notebook Dell Inspiron, com sistema operacional Linux 64 bits, com 4Gb de memória RAM DDR3 e processador Intel Pentium Dual Core, 2.3GHz. A implementação foi feita utilizando a IDE Code::Blocks 12.11.

Conclusão

Pudemos através deste trabalho perceber o alto custo para a obtenção da soluções ótimas para este tipo de problema. É perceptível também a relação custo-benefício que a heurística, neste caso, teve, sempre se aproximando do resultado ótimo, porém com custo computacional muito menor. Enquanto para a heurística obtivemos tempo hábil para cerca de 4 mil estações, para a solução ótima, a partir de 11 estações, fica impraticável a resolução do problema, demorando na ordem de minutos, para o programa gerar o resultado.

Referências

- Estrutura de dados grafo por meio de listas de adjacência
  - http://www2.dcc.ufmg.br/livros/algoritmos/implementacoes-07.php
- Função para permutação de vértices (adaptada)
  - http://www.ime.usp.br/~pf/mac0122-2003/aulas/permut.html