Universidade Federal do Rio Grande do Sul
Instituto de Informática

INF05010 - Otimização Combinatória
Profª Luciana Salete Buriol

Resolução do problema de Ávores de Steiner com a meta-heurística Busca Tabu

Felipe Mathias Schmidt
Gabriel Baruffi Veras
João Luiz Grave Gross

Porto Alegre, 4 de dezembro de 2013.

1 Problema: Árvores de Steiner

O Problema das Árvores de Steiner foi desenvolvido por Jakob Steiner. Trata-se de um problema NP-Completo, onde:

- Entrada: Um grafo $G = (V, A)$ não direcionado com vértices V, arestas A e custos $c_a \geq 0$ para $a \in A$.
- Solução: Um subgrafo conexo mínimo que inclui um dado conjunto de vértices necessários $T \subseteq V$.
- Objetivo: Minimizar: $\sum_{a \in A} c_a$

Dado um grafo $G = (V, A)$ conectado não direcionado, com vértices V, arestas A e custos $c_a \geq 0$, onde $a \in A$, se tem como objetivo gerar um subgrafo conexo mínimo que inclui um dado conjunto T de vértices, onde $T \subseteq V$, com custo mínimo.

O problema de Otimização de Redes Urbanas de Gás é um exemplo do problema de minimização de Árvores de Steiner. Nele busca-se um traçado ótimo de uma rede urbana de distribuição de gás, visando a criação de redes com arquitetura em árvore de forma que todos os pontos sejam atendidos e os custos de implementação sejam minimizados. Temos que G é um grafo $G = (V, A)$ onde $V = \{v1, ..., vn\}$ representa os pontos de passagem da rede de distribuição, os quais estão associados a uma planta urbana e $A = \{e1, ..., em\}$ que representa as vias de ligação entre os pontos de V. Cada elemento de *A* possui um custo c_i. Se deseja conectar todos os pontos de consumo de gás, minimizando o custo de infra-estrutura.

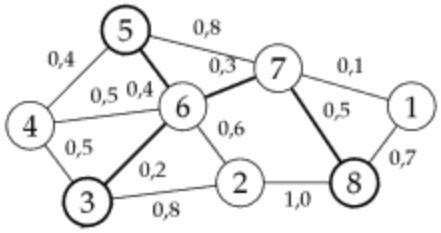

Figura 1: Exemplo de Árvore de Steiner para o subconjunto de vértices $T = \{3, 5, 6, 7, 8\}$

2 Formulação Matemática do Problema

Para um grafo não direcionado G = (V, E), com um conjunto de nós terminais T que devem ser conectados, criamos um grafo bidirecionado B = (V,A) e usamos uma formulação baseada em fluxo de multi-commodities com as seguintes variáveis:

c_e : custo da aresta $e \in E$
x_e : 1 se aresta $e \in E$ está na solução, 0 caso contrário
$f^k_{i,j}$:

- 1 se fluxo da comodidade 'k' entre nós i e j presente na solução, 0 caso contrário
- $\forall k \in T/\{1\}, \quad \forall i,j \in A$

Com elas, formulamos o problema matematicamente da seguinte forma:

Minimize

$$\sum_{e \in E} c_e x_e$$

subject to.

$$\sum_{i \in V} f^k_{ij} - \sum_{i \in V} f^k_{ji} = \begin{cases} -1 & \text{if } j = 1 \\ 1 & \text{if } j = k \\ 0 & \text{otherwise for } j \in V \setminus \{1\} \end{cases}$$ for every 'k' in T - {1}

$f^k_{ij} \leq x_e$ for every edge e = (i,j), commodity k

$f^k_{ij} \geq 0$ for every i,j in A, k in T

x_e integer

Neste sistema, $\forall k \in T$ / {1} geramos um fluxo *f*. Para exemplificar, vamos instanciar um problema simples e resolver o sistema para melhor entendê-lo:

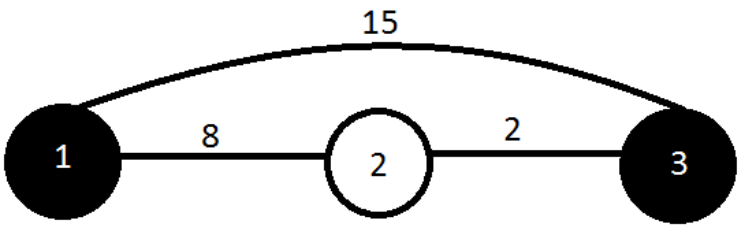

Figura 2: Grafo exemplo de instância do problema

e.g: Para o seguinte grafo:

Temos vértices $V = \{1,2,3\}$, terminais $T = \{1,3\}$, arestas $E = \{1\text{-}2, 2\text{-}3, 2\text{-}3\}$ com pesos $c_e = \{8,2,15\}$ respectivamente, geramos o seguinte sistema:

$$\min \quad 8*X_{1,2} + 2*X_{2,3} + 15*X_{1,3}$$

$$\begin{aligned}
s.a \quad & f^3_{2,1} + f^3_{3,1} - f^3_{1,2} - f^3_{1,3} = -1 && //j = 1 && (1)\\
& f^3_{1,2} + f^3_{3,2} - f^3_{2,1} - f^3_{2,3} = 0 && //j = 2 && (2)\\
& f^3_{1,3} + f^3_{2,3} - f^3_{3,1} - f^3_{3,2} = 1 && //j = 3 && (3)\\
& f^3_{1,2} \leq X_{1,2} && && (4)\\
& f^3_{2,3} \leq X_{2,3} && && (5)\\
& f^3_{1,3} \leq X_{1,3} && && (6)\\
& f^k_{i,j} \geq 0 \qquad \forall k \in T/\{1\}, \ \forall i,j \in A && && (7)
\end{aligned}$$

Para satisfazer todas as restrições, por (1) temos $f^3_{1,2} = 1$, que nos leva em (4) a ter $X_{1,2} = 1$; em (3) precisamos que $f^3_{2,3} = 1$, implicando em $X_{2,3} = 1$; com estes valores já estabelecidos de $f^k_{i,j}$ temos que os demais devem ser de valor 0 (zero) para que todas restrições sejam satisfeitas.

Em linhas gerais, o que o sistema faz é, partindo de um dado nó terminal (no caso, o nó 1), tentar atingir todos os outros terminais - gerando um fluxo/caminho - minimizando o custo das arestas escolhidas para compor a solução.

3 Formulação em GLPK

Para execução do glpk é usado o comando *“glpsol -d dados.stp -m modelo.mod”.*

3.1 Modelo GLPK

```
# Start Node
param StartNode, integer, >0;

# Number of Nodes
param NumNodes, integer, >0;

# Number of Edges
param NumEdges, integer, >0;

# Number of Terminals
param NumTerminals, integer, >0;

# Set of Nodes
set Nodes := {1..NumNodes};

# set of Edges: start_node, end_node, edge_cost
set Edges, dimen 3;

# set of Terminals
set Terminals, dimen 1;

# Costs
set Costs := setof{(i,j,c) in Edges} c;

# Undirected Edges of the graph
set E := setof{(i,j,c) in Edges} (i,j);
set Etwisted := setof{(i,j,c) in Edges} (j,i);

# Bidirected Edges of the graph
set B := E union Etwisted;

# Assignment
var x{E}, binary;

# Flows
```

```
var f {(i,j) in B, k in Terminals diff {StartNode}}, binary;

# Objective Function
minimize Obj: sum{(i,j,c) in Edges} c * x[i,j];

s.t. flows{j in Nodes, k in Terminals diff {StartNode}}:
      (sum{(i,j) in B} f[i,j,k]) - (sum{(i,j) in B} f[j,i,k]) =
      (if j = StartNode then -1 else if j = k then 1 else 0);

s.t. implication{(i,j) in E, k in Terminals diff {StartNode}}:
f[i,j,k] <= x[i,j];

s.t. implicationReverse{(i,j) in E, k in Terminals diff {StartNode}}:
f[j,i,k] <= x[i,j];

s.t. notNegative{(i,j) in B, k in Terminals diff {StartNode}}:
f[i,j,k] >= 0;

end;
```

3.2 Exemplo de Dados de Entrada

```
param StartNode := 1;

param NumNodes := 3;

param NumEdges := 2;

# Edges: start_node, end_node, edge_cost
set Edges :=
1 2 8
2 3 2
1 3 15;

param NumTerminals := 2;

set Terminals :=
1
3;

end;
```

4 Meta-heurística: Busca Tabu

A metaheurística Busca Tabu (BT) foi inicialmente desenvolvida por Glover (1986) como uma proposta de solução para problemas de programação inteira. A partir de então, o autor formalizou esta técnica e publicou uma série de trabalhos contendo diversas aplicações da mesma. (SUBRAMANIAN, 2011). A Busca Tabu tem se mostrado uma heurística eficiente na resolução de vários problemas e atualmente trata-se de uma técnica definitivamente consolidada.

Em linhas gerais, a Busca Tabu é um procedimento adaptativo, de busca local, que permite movimentos de piora (quando não há melhora) para escapar de ótimos locais. Por ser um procedimento de busca local ele é baseado na noção de vizinhança. A cada iteração, a solução atual *s* muda para outra que seja sua vizinha no espaço de busca, isto é, para uma solução *s'* que difere de *s* por uma modificação. Partindo de uma solução inicial *s 0* , um algoritmo Busca Tabu explora, a cada iteração, a vizinhança *N(s)* da solução corrente *s*. A melhor solução de *N(s)* é escolhida segundo uma função de avaliação *f(s)*, tornando-se a nova solução atual, mesmo que *f(s') > f(s)* em um problema de minimização.

Enquanto a maioria dos processos de busca guarda essencialmente o valor da melhor solução obtida até o momento, a Busca Tabu arquiva as informações das últimas soluções visitadas. Tais informações são utilizadas para determinar a escolha da nova atual solução a ser escolhida em *N(s)*. O arquivamento consiste em proibir a escolha de alguns elementos de *N(s)* com a intenção de impedir o retorno a uma solução já visitada anteriormente. O não veto de determinados movimentos pode fazer com que o algoritmo cicle. Um artifício criado para proibir esses movimentos é a Lista Tabu *T*, uma lista contendo as últimas *n* soluções encontradas, todas distintas umas às outras. A lista é implementada como uma FIFO, ou seja, o primeiro elemento a entrar é o primeiro a sair. A saída de um elemento ocorre toda vez que uma solução distinta às presentes na lista tabu é encontrada, e esta, por sua vez, é inserida na lista.

Os principais parâmetros de controle do método são o tamanho da Lista Tabu, a quantidade de elementos de *N(s)* testadas a cada iteração e o número máximo de iterações sem melhora na função objetivo (condição de parada).

4.1 Características da Aplicação

Nesta sessão iremos abordar características da aplicação para o problema de árvores de steiner com a meta-heurística de busca tabu. Falaremos sobre quais entidades foram utilizadas para representar as informações das instâncias, bem como as suas respectivas estruturas de dados. A geração de uma solução inicial trivial e de sua vizinhança de soluções, os parâmetros utilizados para a busca tabu e o critério de parada também serão abordados aqui com mais detalhes.

4.1.1 Representação do Problema e Estrutura de Dados Usada

Como entrada para a aplicação temos diversos arquivos de texto, estruturados de acordo com o exemplo mostrado na seção 3.2. Cada arquivo representa uma instância do problema de árvores de steiner. Os problemas representados nesses arquivos, possuem soluções ótimas conhecidas e variam em quantidade de nodos, quantidade de arestas e também nos nodos pertencentes ao conjunto de terminais.

A leitura de cada instância é feita com um parser simples, capturando informações sobre nodos, arestas e nodos terminais. Para a representação do grafo da instância foi criada uma estrutura de dados chamada *Graph* que implementa uma tabela de adjacência através de uma lista de listas. Cada lista corresponde a um nodo e ela possui as arestas que ligam este nodo aos demais. Em cada interligação também é armazenado o custo da aresta.

O conjunto de nodos terminais é armazenado em uma estrutura chamada *Solution.* Nela são adicionadas em uma lista o conjunto de nodos terminais, bem como as arestas que compõem uma solução.

4.1.2 Solução Inicial

A solução inicial é obtida em um método iterativo de inserção e remoção de arestas em uma entidade *Solution.* A medida que as arestas são inseridas é realizado um teste para ver se a solução possui ciclo. Se houver um ciclo, então a aresta recém inserida é removida e novas arestas são inseridas, testadas e eventualmente removidas. O processo continua até que se tenha uma árvore conectando todos os nodos terminais.

Durante a inserção e remoção das arestas o custa da árvore é atualizado, somando o custo da aresta no caso de inserção ou diminuindo o custo da aresta em caso de remoção. Essa solução é utilizada como semente para o algoritmo da meta-heurística de busca tabu.

4.1.3 Vizinhança e Escolha do Vizinho

A vizinhança de soluções é gerada a partir de uma solução inicial. Uma função iterativa consulta arestas da tabela de adjacência e as insere na solução inicial caso não pertençam à solução, ou as remove caso pertençam. A nova solução é avaliada e se não contiver loops, for conexa e conectar todos os nodos terminais ela será inserida na vizinhança. O processo continua até que todas as arestas da lista de adjacência tenham sido testadas na solução inicial, adicionando as arestas caso não pertençam a solução, ou as removendo caso pertençam.

É importante ressaltar que a geração da vizinhança não leva em consideração o custo total de cada solução encontrada. Elas, independente do custo final, são adicionadas à vizinhança e posteriormente o algoritmo da busca tabu da o tratamento adequado às novas soluções encontradas.

4.1.4 Algoritmo da Busca Tabu: funcionamento e condição de parada

A seguir podemos ver o pseudo-código do algoritmo da aplicação, incluindo a função que implementa a meta-heurística Busca Tabu:

```
BuscaTabuIterativa(s, g, lstTabu)
      enquanto iterações < tabuSearchMAX faz
            adiciona s à lstTabu
            cria vetor neighbors de soluções vizinhas à solução inicial
            para cada solução i de neighbors faz
                  se i < s and i != soluções da lista tabu, então
                        se i < ianterior
                              guarda i como melhor solução até agora
                        end-se
                  end-se
            end-para cada
            se foi encontrada solução melhor que s?, então
                  s = i
                  atualiza solução ótima na lstTabu com s
                  iterações = 0
            end-se, senão
                  procura menor solução de neighbors
                  se foi encontrada menor solução de neighbors?, então
                        s = menor solução de neighbors
                        atualiza solução ótima na lstTabu com s
                        se solução ótima lstTabu foi atualizada, então
                              iterações = 0
                        end-se
                  end-se
                  se não foi encontrada solução em neighbors?, então
                        s = lstTabu.otima
                  end-se
            end-senão
      end-enquanto
      return lstTabu.otima
end-function
```

```
SteinerTree_TabuSearch()
      captura tempo inicial t1
      lê arquivo de entrada contendo a instância
      cria grafo g e o inicializa com a tabela de adjacência
      cria solução trivial s, contendo uma solução inicial
      seta a solução ótima da busca tabu como sendo a solução inicial s
      inicializa lstTabu, lista de tabus, com zero elementos
      BuscaTabu(s, g, 0, lstTabu)
      captura tempo final t2
      calcula tempo de execução t2 - t1
end-function
```

A meta-heurística Busca Tabu recebe como parâmetros três valores: uma solução inicial (s), o grafo do problema contendo a tabela de adjacência, a quantidade de iterações. Depois da criação da vizinhança e da solução inicial ter sido inserida na lista tabu (lstTabu), o algoritmo procura pela melhor solução da vizinhança, considerando que esta solução não pode estar presente na lista tabu. Se uma solução melhor for encontrada, então a quantidade de iterações é zerada, o valor da solução ótima na lista tabu é atualizado e a solução inicial da próxima iteração será a solução encontrada. Caso não se encontre uma solução vizinha menor que a inicial, é procurada a menor solução vizinha, sem se preocupar com a solução inicial. Se uma menor solução vizinha for encontrada e que não pertença à lista tabu, então esta será a solução inicial da próxima iteração e se a solução ótima da lista tabu for atualizada, então a quantidade de iterações é zerada, senão segue normalmente. Caso não se consiga achar nenhuma solução menor que a atual e que não pertença à lista tabu, então a solução ótima da lista tabu vira a solução inicial. Por fim, quando estourar a quantidade de iterações máxima, é retornada a solução ótima da lista tabu.

5 Análise de Resultados

Após a execução da metaheurística e do solver GLPK durante 1 hora (3600 segundos) para cada entrada chegamos nos dados apresentados nos sub-tópicos seguintes. Para tal, criamos um bat de execução para o solver com linhas de comando para execução de cada instância, e.g: *"glpsol -d b14.stp -m SteinerTree.mod --tmlim 3600 >> b14out.txt"*. A máquina usada para os testes dispõe de um processador quad-core de 2.33Ghz e 4GB de memória RAM.

5.1 Considerações sobre o GLPK

Na execução do solver tivemos alguns casos que devem ser mencionados. Para a instância *c01* o glpk precisamos de apenas 35,8 segundos para chegar na solução ótima, para a instância *i160-003* esse tempo foi de 55,5 segundos; a instância *d10* não conseguiu ser executada como mostrado na figura 3. Na execução das instâncias *e20* e *hc10p* o solver ficou aproximadamente 2 horas sem nem conseguir gerar as restrições necessárias - figura 4 mostra o estado da execução. Nos casos citados anteriormente, assumimos que o glpk não conseguiu chegar em nenhuma resposta no tempo estipulado.

```
c:\glpk-4.52\w32>glpsol -d d10.stp -m SteinerTree.mod
GLPSOL: GLPK LP/MIP Solver, v4.52
Parameter(s) specified in the command line:
 -d d10.stp -m SteinerTree.mod
Reading model section from SteinerTree.mod...
SteinerTree.mod:55: warning: final NL missing before end of file
55 lines were read
Reading data section from d10.stp...
d10.stp:2516: warning: final NL missing before end of file
2516 lines were read
Generating Obj...
Generating flows...
Generating implication...
Generating notNegative...
Generating implicationReverse...
Model has been successfully generated
glp_alloc: no memory available
Error detected in file ..\src\env\alloc.c at line 91
```

Figura 3: instância d10 com estouro de memória

```
c:\glpk-4.52\w32>glpsol -d e20.stp -m SteinerTree.mod --tmlim 900
GLPSOL: GLPK LP/MIP Solver, v4.52
Parameter(s) specified in the command line:
 -d e20.stp -m SteinerTree.mod --tmlim 900
Reading model section from SteinerTree.mod...
SteinerTree.mod:55: warning: final NL missing before end of file
55 lines were read
Reading data section from e20.stp...
e20.stp:63766: warning: final NL missing before end of file
63766 lines were read
Generating Obj...
Generating flows...
```

Figura 4: instância e20 parada na execução

Para a instância *brasil58* também tivemos um *warning* referente à instabilidade numérica da solução, como mostrado na figura 5.

```
 101000: obj =  3.614730929e+004  infeas = 1.062e-002 (24)
Warning: numerical instability (primal simplex, phase I)
 101102: obj =  3.614725724e+004  infeas = 7.757e-003 (24)
Warning: numerical instability (primal simplex, phase I)
 101215: obj =  3.614733695e+004  infeas = 1.153e-002 (24)
Warning: numerical instability (primal simplex, phase I)
 101341: obj =  3.614738469e+004  infeas = 1.386e-002 (24)
Warning: numerical instability (primal simplex, phase I)
 101458: obj =  3.614729332e+004  infeas = 1.363e-002 (24)
 101500: obj =  3.614729595e+004  infeas = 1.340e-002 (24)
Warning: numerical instability (primal simplex, phase I)
 101571: obj =  3.614650750e+004  infeas = 1.660e-003 (24)
```

Figura 5: instância brasil58 instável

O que achamos bastante estranho é que nos casos em que o GLPK não conseguiu gerar as restrições ou não conseguiu gerar a solução relaxada inicial, o programa ficou executando por um tempo muito maior que o designado pelo parâmetro *tmlim,* de 3600 segundos. Ao que parece, este tempo inicial não é considero no countdown timer da implementação da versão do GLPK que usamos. Para o caso de estouro de memória, não sabemos com exatidão o tempo que o programa executou pois esta informações não foi passado para o *standard io* no momento do estouro.

5.2 Considerações sobre Busca Tabu

5.2.1 Busca Tabu Recursiva

Assim como para o GLPK, com nossa heurística tivemos casos de estouro de memória. A figura 6 mostra o momento da execução que isto acontece para a instância i160-033. O mesmo aconteceu para a instância c01, mc11, d10, hc10p e e20.

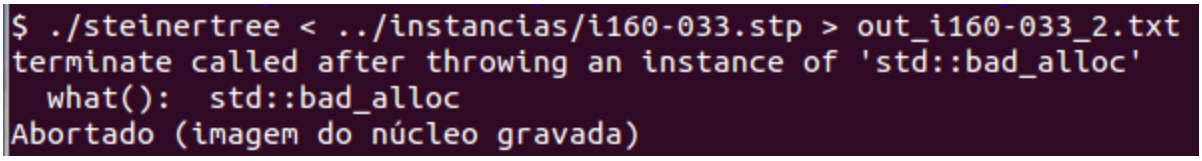

Figura 6: estouro de memória na heurística

Nesta etapa utilizamos uma função recursiva para implementar a meta-heurística Busca Tabu, porém em determinado ponto da execução havia tantas chamadas à mesma função que o empilhamento consumia memória suficiente para causar estouro. Por isso que para instâncias menores conseguimos executar o programa encontrando uma solução ótima. Já para instâncias com grafos maiores ocorria estouro.

5.2.2 Busca Tabu Iterativa

Devido aos problemas enfrentados com a implementação recursiva da Busca Tabu, desenvolvemos uma versão iterativa da função. Nesta nova implementação conseguimos eliminar o problema de estouro de memória para as instâncias testadas e obtivemos ganhos significativos, com um ganho médio de 40% no tempo de execução se comparada às instâncias executadas na versão recursiva. Já em relação às soluções ótimas geradas, não houve grandes mudanças.

5.2.3 Busca Tabu Iterativa Otimizada

Após executar as instâncias com a função Busca Tabu iterativa, observamos que a maior parte do processamento era dedicado a geração da vizinhança, logo, inserimos uma pequena otimização no código, limitando a quantidade de variações da solução inicial candidatas a serem vizinhas. Esse limite é determinado com um fator da forma 1/X, onde X determina a nova fração máxima de variações na solução inicial que podem gerar novos vizinhos. Por exemplo, para um problema no qual, a cada iteração, há 1000 variações na solução inicial gerando candidatos à vizinhança, um fator de ¼ limitaria as variações em 250. Dessa forma temos menos candidatos à vizinhança e consequentemente menos vizinhos a cada iteração. Assim, pelo algoritmo ter que realizar menos variações na solução inicial e também menos testes para ver se as variações geram soluções vizinhas, há um ganho de desempenho considerável.

É importante ressaltar que a quantidade de vizinhos gerados após a limitação não segue a exata mesma proporção da nova quantidade de variações, porém nos testes que executamos a redução da vizinhança se aproxima bastante desta proporção. E um cuidado que se deve ter é não utilizar um fator muito pequeno, ou seja X muito grande, pois senão a vizinhança será muito pequena e dessa forma há uma grande chance de que boas soluções vizinhas não sejam

geradas, impactando diretamente na eficiência do algoritmo em encontrar soluções ótimas de valor cada vez menor.

A seguir uma tabela comparativa dos tempos obtidos nas execuções recursiva, iterativa e iterativa otimizada.

Tabela 1: Tempos de execução da Busca Tabu Recursiva x Busca Tabu Iterativa x Busca Tabu Iterativa Otimizada

Instância	Tempo Exec Recursivo (s)	Tempo Exec Iterativo (s)	Tempo Exec Iterativo Oitmizado(s)
b14	21	17	3.6
c01	*	11325	637.7
d10	*	running	31746
e20	*	running	*
mc11	*	16783	1872.6
brasil58	203	110	22.8
cc3-4p	54	17	2
hc10p	*	running	32410
i160-003	192	131	9.6
i160-033	*	141	10.5

Na coluna de execuções com a função recursiva as linhas marcadas com '*' indicam as instâncias que geraram estouro de memória, como foi explicado na seção 5.2.1. Na coluna de execuções com a função iterativa as linhas com o texto "running" indicam que as instâncias não convergiram, por terem um tempo de execução muito alto.

5.3 Solução Ótima vs. GLPK vs. Heurística

Na tabela 1 temos a comparação da solução ótima com a solução encontrada pelo solver após execução durante uma hora. As células com asterisco (*) indicam a falta de informação para o campo, seja por estouro de memória, seja por não gerar solução factível, seja por não ter conseguido gerar as restrições do problema em tempo hábil. Os tempos 3600+ indicam que a execução da instância demorou mais que 1 hora (3600 segundos), mas sem precisão exata do tempo usado.

Tabela 2: Solução Ótima vs. GLPK

Instância	Valor Ótimo	Sol. Encontrada	Tempo Exec (s)	Desvio %
b14	235	240	3600	-2.128
c01	85	85	36	0.000
d10	2110	estouro	*	*
e20	1342	running	*	*
mc11	11689	running	*	*
brasil58	13655	running	*	*
cc3-4p	2338	2350	3600	-0.513
hc10p	60679	running	*	*
i160-003	2297	2297	53	0.000
i160-033	2101	2101	1775	0.000

Tabela 3: Solução Ótima vs. Busca Tabu Recursiva

Instância	Valor Ótimo	Sol. Encontrada	Tempo Exec (s)	Desvio %
b14	235	243	21	-3.404
c01	85	estouro	*	*
d10	2110	estouro	*	*
e20	1342	estouro	*	*
mc11	11689	estouro	*	*
brasil58	13655	13943	203	-2.109
cc3-4p	2338	2553	54	-9.196
hc10p	60679	estouro	*	*
i160-003	2297	3085	192	-34.306
i160-033	2101	estouro	*	*

Tabela 4: Solução Ótima vs. Busca Tabu Iterativa

Instância	Valor Ótimo	Sol. Encontrada	Tempo Exec (s)	Desvio %
b14	235	250	17	-6.383
c01	85	104	11325	-22.353
d10	2110	running	running	*
e20	1342	running	running	*
mc11	11689	11994	16783	-2.609
brasil58	13655	13781	110	-0.923
cc3-4p	2338	2360	17	-0.941
hc10p	60679	running	running	*
i160-003	2297	2612	131	-13.714
i160-033	2101	2345	141	-11.614

Tabela 5: Solução Ótima vs. Busca Tabu Iterativa Otimizada

Instância	Valor Ótimo	Sol. Inicial	Melhor Sol. Encontrada	Tempo Exec (s)	Desvio %	Redução de Vizinhança
b14	235	511	237	3.6	-0.851	1/4
c01	85	2840	97	637.7	-14.118	1/4
d10	2110	5512	2223	31746	-5.355	1/8
e20	1342	13611	running	*	*	1/16
mc11	11689	33696	12052	1872.6	-3.105	1/4
brasil58	13655	135559	13950	22.8	-2.160	1/4
cc3-4p	2338	7801	2342	2	-0.171	1/4
hc10p	60679	107323	78160	32410	-28.809	1/16
i160-003	2297	17273	2396	9.6	-4.310	1/4
i160-033	2101	15240	2345	10.5	-11.614	1/4

5.4 GLPK vs. Heurísticas

Tabela 6: GLPK x Recursiva x Iterativa x Iterativa Otimizada

Instância	GLPK	Recursiva	Iterativa	Iterativa Otimizada
b14	240	243	250	**237**
c01	**85**	estouro	104	97
d10	*	estouro	running	**2223**
e20	*	estouro	running	running
mc11	*	estouro	**11994**	12052
brasil58	*	13943	**13781**	13950
cc3-4p	2350	2553	2360	**2342**
hc10p	*	estouro	running	**78160**
i160-003	**2297**	3085	2612	2396
i160-033	**2101**	estouro	2345	2345

Tabela 7: Desvios (%) do GLPK x Heurísticas

Instância	Valor Ótimo	GLPK	Recursiva	Iterativa	Iterativa Otimizada
b14	235	-2.128	-3.404	-6.383	-0.815
c01	85	0	100	-22.353	-14.118
d10	2110	100	100	100	-5.355
e20	1342	100	100	100	100
mc11	11689	100	100	-2.609	-3.105
brasil58	13655	100	-2.109	-0.923	-2.16
cc3-4p	2338	-0.513	-9.196	-0.941	-0.171
hc10p	60679	100	100	100	-28.809
i160-003	2297	0	-34.306	-13.714	-4.31
i160-033	2101	0	100	-11.614	-11.614

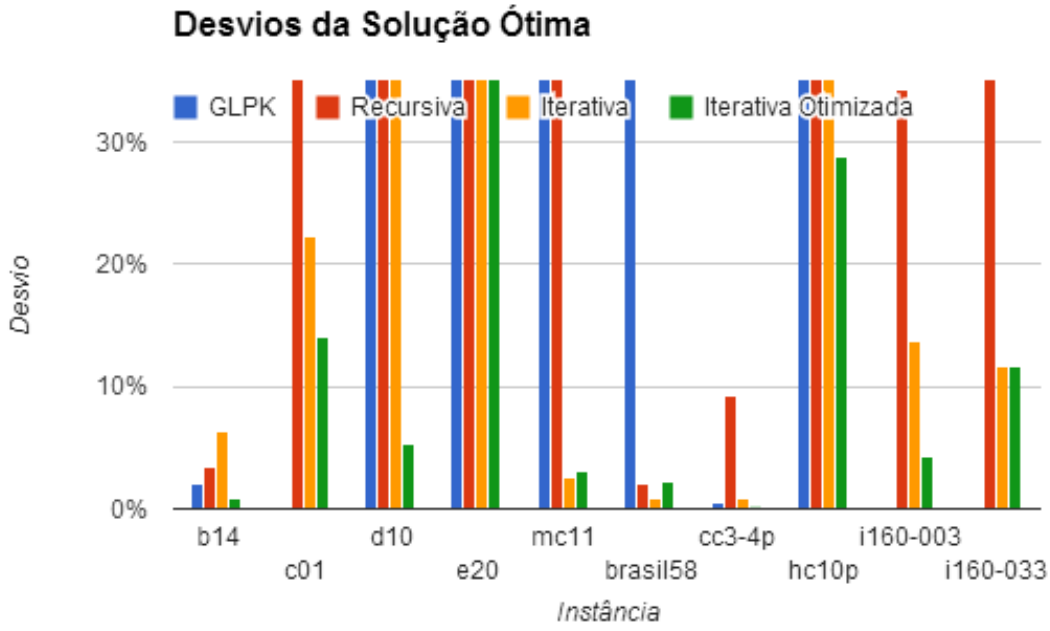

Figura 7: Dados da tabela 6 graficamente.

No gráfico, valores acima de 30% vão até 100%, representando que a execução não gerou nenhum resultado. O gráfico foi cortado em 30% para que se possa ter uma melhor visualização dos resultados obtidos.

5.5 Executando GLPK por 4h

Executamos o solver por aproximadamente 4 horas (14.400 segundos) para a instância *b14* e chegamos no valor 239; abaixo a imagem do prompt de comando com o estado final do glpk após o tempo citado.

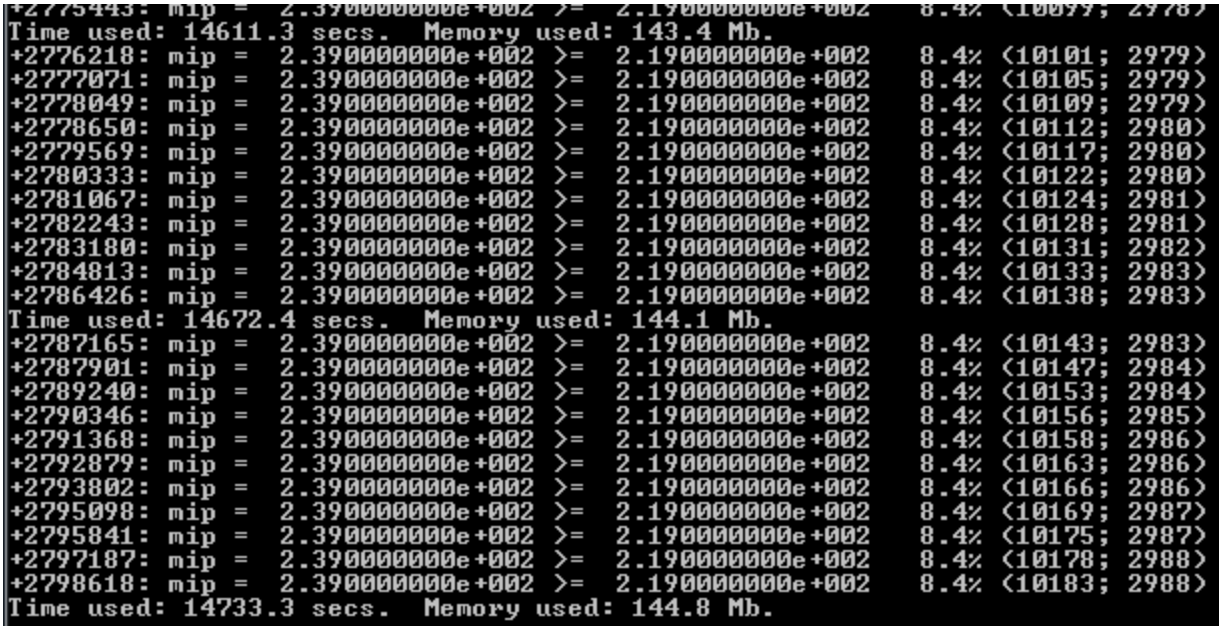

Figura 8: execução do solver por 4h.

Mesmo depois de todo este tempo, para uma instância relativamente pequena, não se chegou no resultado ótimo que é 235, evidenciando a dificuldade de resolução deste problema de forma ótima.

6 Conclusões

Com a implementação deste trabalho vimos como é importante levar em consideração mais de um tipo de solução para o mesmo problema, em especial quando os problemas são difíceis de ser resolvidos como os apresentados nesta parte da disciplina. Um dos pontos que chamou a atenção é que, para instâncias relativamente pequenas - como a *b14* usada para testes - podemos chegar a uma solução ótima em pouquíssimo tempo, evidenciando o poder computacional que temos atualmente e a eficiência do GLPK.

Outro ponto que deve constar aqui é que, para instâncias grandes, todos os detalhes devem ser pensados para que se consiga chegar a alguma resposta minimamente aceitável. Foi o caso da instância *d10*, que só gerou algum resultado com nossa heurística iterativa otimizada após um tempo de execução muito grande. Em uma empresa que trabalha com este tipo de problemas diariamente, uma otimização simples como a por nós implementada pode significar uma redução drástica de tempo que, por sua vez, pode acarretar em uma diminuição dos custos da geração destas solução, o que é bastante benéfico para a empresa.

7 Referências

- Steiner tree problem. Disponível em: http://en.wikipedia.org/wiki/Steiner_tree_problem. Acesso em: 17/10/2013.
- Solução problema Árvore Geradora Mínima (exercício 8). Disponível em: http://moodle.inf.ufrgs.br/mod/resource/view.php?id=43097.
  Acesso em: 24/10/2013.
- Árvore de Extensão Mínima (também chamada Árvore Geradora Mínima). Disponível em: http://pt.wikipedia.org/wiki/%C3%81rvore_de_extens%C3%A3o_m%C3%ADnima.
  Acesso em: 24/10/2013.
- Árvores de Steiner: Teoria, Geração Numérica e Aplicações. Disponível em: http://posmat.ufabc.edu.br/teses/MAT-2010%20-%20Wendhel%20Raffa%20Coimbra.pdf
  Acesso em: 30/10/2013.
- Problema da Árvore de Steiner. Disponível em: http://subversion.assembla.com/svn/puc_minas/4%C2%BA%20PERIODO/MATEMATICA%20COMPUTACIONAL/Semin%C3%A1rio%20Final/Apresenta%C3%A7%C3%A1o(real).ppt.
  Acesso em: 25/10/2013.
- Steiner Tree - Formulação com commodity. Disponível em: http://www.kellogg.northwestern.edu/faculty/chopra/htm/research/steiner-tsai-mar01.pdf.
  Acesso em: 25/10/2013.
- Ajuda na formulação GLPK do .stp (dados) e .mod (modelo). Disponível em: http://trac.astrometry.net/browser/trunk/projects/archetypes/glpk-4.31/examples/graph.mod?rev=9312.
  Acesso em: 30/10/2013.
- MathProg language by example. Disponível em: http://adrian.idv.hk/doku.php/studynotes/mathprog.
  Acesso em 27/10/2013.
- SUBRAMANIAN, Anand. 2011. Aplicação da metaheurística Busca Tabu ao problema de alocação de aulas a salas em uma instituição universitária. Disponível em: http://producaoonline.org.br/rpo/article/download/419/762. Acesso em: 30/11/2013.
- O problema de árvores de steiner em grafos não direcionados. Disponível em: http://pt.scribd.com/doc/108768748/25/O-Problema-de-Steiner-em-Grafos-nao-Direcionados. Acesso em: 25/11/2013.