Um algoritmo exato para o problema da Árvore Geradora Mínima de Grau Restrito

RUBEM KALEBE SANTOS

Natal, Brasil
Outubro de 2015

Sumário

1 Introdução

A priori, muitos problemas de otimização podem ser utilizados para modelar diversas aplicações práticas, principalmente nas áreas de roteamento, desenho de circuitos integrados, logística e telecomunicações. Esses problemas são importantes para garantir soluções eficientes e robustas. Daí, tem-se a necessidade de desenvolver algum método que seja capaz de encontrar, em tempo aceitável, solução para tais problemas.

A dificuldade encontrada em desenvolver soluções algorítmicas eficientes para tais problemas é o caso de muitos deles serem provados pertencerem a classe NP-completo ou NP-difícil, das quais sabemos que, provavelmente, não existe algoritmo que encontre uma solução em tempo polinomial. Então, mesmo sendo problemas práticos e próximos da realidade, não é possível desenvolver soluções algorítmicas eficientes para um conjunto de dados realístico, como um conjunto de vértices representando as casas de um bairro.

Neste trabalho será abordado o problema da Árvore Geradora Mínima de Grau Restrito (também conhecida como Árvore Geradora Mínima com restrição de grau e, em inglês, *Degree-constrained minimum spanning tree* – DCMST), que pode ser muito útil na área de, por exemplo, telecomunicações, quando necessário conectar vértices com um custo mínimo e com uma restrição de grau afim de possibilitar um "balanceamento" das conexões do grafo.

Neste trabalho serão discutidos os conceitos que envolvem tal problema, técnicas de soluções, as soluções propostas e os resultados computacionais obtidos. As soluções propostas aqui são exatas e por isso podem não ser aplicáveis em situações do cotidiano. Todavia, servem como uma introdução a análise desse problema específico, abrindo caminho para futuras implementações mais eficientes. Além disso, os conceitos discutidos aqui podem ser aplicáveis em outros problemas do gênero.

O trabalho está sendo mantido em um repositório *online* e pode ser acessado a partir deste *link*: https://github.com/rubemkalebe/DCMST.

2 Descrição do problema

Nesta seção será apresentada uma formulação do problema da Árvore Geradora Mínima de Grau Restrito. Porém, antes serão discutidos conceitos introdutórios da Teoria dos Grafos que são necessários para o entendimento do problema e da solução proposta.

2.1 Conceitos sobre grafos

Um grafo é um par ordenado $G = (V, E)$ composto por um conjunto V de vértices e um conjunto E de relações binárias sobre V, cada uma delas conhecida como aresta. Dizemos que uma aresta relaciona, ou conecta, um par de vértices do conjunto V e quando ela é do tipo $e = (v, v)$, com $v \in V$, é denominada *laço*.

Pode-se ter grafos *direcionados* (ou orientados) ou *não direcionados* (ou não orientados). As arestas em um grafo direcionado são pares ordenados, ou seja, a ordem entre dois vértices conectados por uma aresta é importante. Neste caso é comum representar a aresta como uma flecha apontando de um vértice para outro. As arestas em um grafo não direcionado são pares não ordenados, os quais são comumente representados através apenas de um seguimento de linha conectando dois vértices.

O *grau* $d(v)$ de um vértice $v \in V$ é o número de arestas que incidem em v. Nos grafos direcionados existe a diferença entre *grau de entrada* e *grau de saída* de um vértice.

Um grafo é dito *valorado* quando existe uma função que associa um custo a cada aresta $e \in E$.

Um *caminho* de v_1 para v_k é uma sequência de vértices v_1, v_2, ... , v_k, conectados pelas arestas (v_1, v_2), (v_2, v_3), ... , (v_{k-1}, v_k). Um caminho é chamado *simples* se cada vértice aparece uma única vez. Um vértice u é dito ser alcançado por v se existe um caminho de v para u. Um *ciclo* é um caminho no qual o primeiro e o último vértice são o mesmo.

Um grafo não orientado é dito *conexo*, se existe um caminho entre qualquer par de vértices distintos do grafo. Uma *árvore* $T = (V, E)$ é um grafo que é **acíclico** e **conexo**. Um vértice v da árvore é uma folha se possuir grau menor ou igual a 1. Uma floresta é um conjunto de árvores, ou seja, um grafo acíclico.

Um *subgrafo* de um grafo $G = (V, E)$ é um grafo $H = (U, F)$ tal que

$U \subseteq V$ e $F \subseteq E$. Uma *árvore geradora* de um grafo não direcionado G é um subgrafo de G que é uma árvore e contém todos os vértices de G. Perceba que um único grafo pode possuir várias árvores geradoras.

2.2 Representações de grafos

Pode-se listar três formas comuns de representar um grafo $G = (V, E)$ computacionalmente:

- **Lista de arestas:** modo simples de representar um grafo utilizando apenas uma lista, ou *array*, de $|E|$ arestas, cada uma delas relacionando os vértices em V que estão conectados.
- **Matriz de adjacência:** representa um grafo com $|V|$ vértices através de uma matriz binária $|V| \times |V|$, na qual a entrada na linha i e coluna j é 1 se e somente se a aresta (i, j) estiver no grafo.
- **Lista de adjacência:** combina os dois métodos anteriores, pois armazena um *array* dos vértices adjacentes a cada vértice $i \in V$. Geralmente, temos um *array* de $|V|$ listas de adjacência, uma lista por vértice.

2.3 Problema da Árvore Geradora Mínima

Considere um grafo não direcionado $G = (V, E)$ onde para cada aresta $(u, v) \in E$, tem-se um custo $c(u, v)$ associado. Deseja-se encontrar um subconjunto $T \subseteq E$ que conecta todos os vértices de G e cuja soma total dos seus custos

$$c(T) = \sum_{(u,v) \in T} c(u, v)$$

seja a menor possível.

Como T é acíclica e conecta todos os vértices de G, forma uma árvore geradora de G. A árvore T encontrada desta forma é uma árvore geradora de custo mínimo de G.

Um exemplo simples de aplicação para uma Árvore Geradora Mínima seria o caso de uma empresa de televisão a cabo com interesse em realizar o cabeamento de uma vizinhança de clientes. Dentre as diversas possibilidades de se conectar uma residência a outra, alguns caminhos podem possuir custo maior, por necessitarem de cabos de comprimento

mais longo, devido a algum tipo de obstáculo que deve ser contornado, por exemplo; esses caminhos podem ser representados por arestas com custos mais elevados. Uma árvore geradora para o grafo que interliga a vizinhança seria um subconjunto dos caminhos possíveis que não possui ciclos, mas garante a conexão entre todas as residências. A árvore geradora mínima que conecta todas as residências seria aquela que garantiria o menor custo possível para realizar todas as conexões [dS10].

O primeiro algoritmo para se encontrar uma árvore geradora mínima foi desenvolvido pelo cientista Otakar Borüvka, em 1926 [Bor26], que tinha como objetivo conectar a rede elétrica de Morávia, região localizada no leste da atual República Tcheca. Posteriormente, surgiram outros algoritmos capazes de resolver esse tipo de problema, tal como Kruskal [Kru56] e Prim [Pri57], que, assim como Borüvka [Bor26], resolveram o problema em tempo polinomial.

Conforme o aumento e a diversidade da demanda em situações práticas envolvendo árvores geradoras mínimas, foram surgindo variações do problema que consistem do problema fundamental acrescido de restrições. Dentre estas variações, é possível destacar alguns exemplos:

- Árvore Geradora Mínima de Grau Restrito;
- Árvore Geradora Estocástica;
- Árvore Geradora Mínima Quadrática;
- Árvore Geradora Mínima Probabilística.

O foco deste trabalho é o problema da Árvore Geradora Mínima de Grau Restrito, que será especificado na próxima subseção.

2.4 Problema da Árvore Geradora Mínima de Grau Restrito

Dado um grafo não orientado $G = (V, E)$, com custo c_{ij} associado com cada aresta e_{ij}, para toda $e_{ij} \in E$, uma Árvore Geradora Mínima com restrição de grau consiste em uma árvore geradora mínima em que o grau d_i de um vértice i, para todo $i \in V$, é menor ou igual a uma constante [NH80].

A versão de decisão deste problema foi provada pertencer a classe NP-completo [GJ79] e por isso, provavelmente, não será possível encontrar um algoritmo que resolva-o e execute em tempo polinomial.

Basicamente, podemos resolver este problema de duas formas: através de um algoritmo exato ou através de um algoritmo aproximativo. De uma maneira geral, métodos que produzem soluções exatas, ou boas, são computacionalmente mais custosos, enquanto que os que produzem uma solução rapidamente, podem estar muito distantes da solução ótima.

Neste trabalho será apresentado um algoritmo exato que soluciona o problema em questão. Todavia, como o número de computações cresce exponencialmente (*explosão combinatória*), para instâncias maiores do problema, a utilização desse algoritmo torna-se inviável devido ao elevado tempo gasto para encontrar uma solução ou até mesmo pela possibilidade de levar o computador a situações de instabilidade, como o travamento.

3 Descrição da solução

Este problema já vem sendo bastante explorado na literatura e várias soluções foram propostas. A primeira solução proposta foi apresentada em [NH80], nos anos 80, e era baseada em uma técnica de *branch-and-bound*. No mesmo trabalho também foram apresentados dois algoritmos baseados em técnicas de Programação Linear que conseguem encontrar soluções para instâncias de até 100 vértices.

Devido ao fato das soluções exatas serem bastante limitadas pelo tamanho da instância, várias soluções heurísticas vêm sendo propostas, como em [BDZ12], onde é apresentada uma solução baseada na heurística de colônia de formigas. Em [ZG97] foi apresentado o primeiro algoritmo genético para o problema.

Neste trabalho foi necessário implementar um algoritmo exato e a técnica escolhida foi *backtracking*. Daí, faz-se necessário revisarmos esta técnica para depois analisarmos a solução. Além disso, também foi necessário implementar um programa validador para checar se as soluções encontradas eram realmente válidas. Esses tópicos serão discutidos nesta seção.

3.1 *Backtracking*

Backtracking é uma técnica caracterizada por ser um refinamento da busca por força bruta (enumeração exaustiva, busca exaustiva) pelo fato de que várias soluções podem ser eliminadas sem serem explicitamente examinadas.

Essa técnica pode ser aplicada para problemas que admitem o conceito de "solução candidata parcial", onde a cada passo do algoritmo podemos gerar uma solução parcial ao adicionar um novo elemento, e que exista um teste relativamente rápido para verificar se uma candidata parcial pode ser completada como uma solução válida. Esse teste faz com que, se aplicável, o uso de *backtracking* seja mais eficiente do que uma enumeração total, já que ele é capaz de eliminar soluções inválidas.

Perceba também que o teste faz com que um algoritmo que faz uso de *backtracking* gere apenas soluções válidas, ao contrário de algoritmos de força bruta que geram todas as soluções e depois verificam se elas são válidas.

O *backtracking* pode ser pensado como uma *busca em profundidade*

em um grafo implícito e garante corretude enumerando todas as possibilidades e nunca visitando um estado (solução parcial) mais de uma vez.

3.2 Validador

Para checar a validade da solução encontrada pelo algoritmo foi implementado um programa validador. Basicamente, ele faz as seguintes verificações no grafo gerado:

(i) Checa se o grafo é cíclico (de acordo com a definição de árvore);

(ii) Checa se o grafo é conexo (de acordo com a definição de árvore);

(iii) Checa a restrição de grau nos vértices (de acordo com a formulação do problema);

(iv) Checa se o custo encontrado foi calculado corretamente.

Sobre a implementação, é importante destacar que:

- Os grafos são armazenados em uma lista de arestas;
- O grau de cada vértice é mantido em um vetor de inteiros, com uma posição para cada vértice;
- Uma estrutura de conjuntos disjuntos (*Union-Find*) é usada para checar se o grafo é conexo e/ou cíclico (só foram consideradas propriedades que nos interessam no problema da DCMST);
- O programa deve receber duas entradas: o arquivo do caso de teste usado para gerar a solução e o arquivo contendo o grafo e custo da solução encontrada.

Os algoritmos utilizados para checar se o grafo é conexo e verificar a existência de ciclo são apresentados em forma de pseudocódigo nos algoritmos 1 e 2, respectivamente.

3.3 Algoritmo proposto

Foi implementado um algoritmo que utiliza a técnica de *backtracking*, em linguagem **C++**, afim de resolver o problema. Também foi utilizada a *flag* de otimização `-O3` disponibilizada pelo compilador `g++`, que possibilitou um bom ganho de desempenho. A partir desse primeiro algoritmo

Algoritmo 1 Pseudocódigo que ilustra o algoritmo utilizado para checar se o grafo é conexo.

```
1: function isConnected
2:     cont ← 0
3:     for all vertex v ∈ V do makeSet(v)
4:     for all edge (u, v) ∈ E do
5:         if not sameComponent(u,v) then
6:             cont ← cont + 1
               union(u,v)
7:     if cont is equals to |V| − 1 then return true
8:     else
        return false
```

Algoritmo 2 Pseudocódigo que ilustra o algoritmo utilizado para checar se o grafo contém ciclo.

```
1: function isCyclic
2:     for all vertex v ∈ V do makeSet(v)
3:     for all edge (u, v) ∈ E do
4:         if sameComponent(u,v) then return true
           union(u,v)
       return false
```

simples foram derivadas duas soluções alternativas, também com uso de *backtracking*, mas com simplificações afim de acelerar o tempo de execução.

Primeiramente será mostrado o algoritmo "puro" e depois serão mostradas suas alternativas.

As implementações apresentadas logo mais também armazenam o grafo em uma lista de arestas e mantém o grau de cada vértice em um vetor de inteiros. Além disso, uma estrutura de conjuntos disjuntos (*Union-Find*) é usada para checar se o grafo é conexo e/ou cíclico.

3.3.1 *Backtracking* sem otimizações

As partes do programa implementado que são responsáveis por procurar a solução são apresentadas nos códigos 1, 2 e 3.

Código 1: Método responsável por procurar a solução

```
1  void Backtracking::findMinimum() {
2          Edge link[edges(Tree::getVertexMax())]; // Numero de rotas possiveis
3                                                  // equivalente ao somatorio de 1...n
4          Tree tree;
5          int length = initializeLinkVector(link);
6          Chronometer::start();
7          combinations(link, length, Tree::getVertexMax()-1, 0, tree); // Computa as
               combinacoes de arestas possiveis
8          Chronometer::stop();
9          executionTime = Chronometer::elapsedTime();
10 }
```

O código 1 é o responsável por procurar a solução, ele é o primeiro a ser invocado pelo método principal. Nele são invocados alguns métodos auxiliares e é calculado o tempo gasto no algoritmo de *backtracking*.

Código 2: Método responsável por formar todas as possíveis ligações entre vértices

```
int Backtracking::initializeLinkVector(Edge link[]) {
	int countEdges = 0; // Indice da aresta
	for(int i = 0; i < Tree::getVertexMax(); i++) {
		for(int j = i; j < Tree::getVertexMax(); j++) {
			if(i != j) {
				Vertex v1(i+1);
				Vertex v2(j+1);
				Edge e(v1, v2, countEdges, costMatrix->getElement(i, j));
				link[countEdges++] = e;
			}
		}
	}
	return countEdges;
}
```

O código 2 preenche um vetor com todas as possíveis ligações entre vértices do grafo. Perceba que os vértices são rotulados com valores inteiros começando por 1 e seguindo até n. As arestas criadas neste método são acessadas pelo algoritmo de *backtracking*.

Código 3: Método que simula as combinações de arestas

```
void Backtracking::combinations(Edge link[], int length, int size, int startPosition,
    Tree &tree) {
	if(size == 0) {
		if((solutions == 0) || (tree.totalCost() < bestTree->totalCost())) {
			bestTree->update(tree);
		}
		solutions++;
		return;
	}
	for(int i = startPosition; i <= length - size; i++) {
		if(tree.addEdge(link[i])) {
			combinations(link, length, size-1, i+1, tree);
			tree.removeEdge();
		} else {
			combinations(link, length, size, i+1, tree);
			return;
		}
	}
}
```

O código 3, principal responsável por achar a solução, é um algoritmo *backtracking* que basicamente computa combinações. O algoritmo vai preenchendo um vetor de $|V| - 1$ posições, começando pela primeira

posição. A cada execução ele tenta adicionar a aresta da posição i e se essa adição for possível (respeitar a restrição de grau e não gerar ciclo no grafo) ele adiciona a aresta (criando uma solução parcial) e segue para preencher o restante do vetor. Caso contrário, ele tenta adicionar a próxima aresta $(i + 1)$. Quando o vetor é completamente preenchido o custo é verificado. Se o custo do grafo gerado for o menor até o momento, ele armazena esse grafo. Caso contrário, ele ignora tal grafo. Independente do custo, após processar a solução incrementa-se o contador de soluções válidas e segue-se a busca por uma solução. O algoritmo pára quando não existirem mais alternativas para formar uma solução.

Esse algoritmo é bastante intuitivo e não é tão robusto e eficiente. De fato, observou-se a possibilidade de fazer algumas otimizações, de onde saíram duas soluções alternativas, que apresentaram um bom desempenho (muito superior ao do *backtracking* "puro"). Essas soluções alternativas são abordadas a seguir.

3.3.2 *Backtracking* com passo de teste otimizado

No código 4 é apresentado um método que otimiza o algoritmo de *backtracking* do código 3. Perceba que a única alteração no código foi no condicional da linha 10.

Código 4: Método otimizado que simula as combinações de arestas

```
1  void OptimizedBacktracking::combinations(Edge link[], int length, int size, int
       startPosition, Tree &tree) {
2          if(size == 0) {
3                 if((solutions == 0) || (tree.totalCost() < bestTree->totalCost())) {
4                        bestTree->update(tree);
5                 }
6                 solutions++;
7                 return;
8          }
9          for(int i = startPosition; i <= length - size; i++) {
10                if((solutions == 0 || tree.totalCost() + link[i].getCost() <
                       bestTree->totalCost()) && tree.addEdge(link[i])) {
11                       combinations(link, length, size-1, i+1, tree);
12                       tree.removeEdge();
13                } else {
14                       combinations(link, length, size, i+1, tree);
15                       return;
16                }
17         }
18 }
```

Nesta solução a árvore é podada utilizando-se o custo da solução do momento (se uma solução já tiver sido encontrada). Essa simplificação trivial possibilitou um grande aumento de desempenho já que a restrição mais rígida ignora soluções que não podem gerar árvores de custo mínimo.

3.3.3 *Backtracking* com passo de teste otimizado e espaço de busca ordenado

Uma outra variação da solução também foi pensada para este problema. A ideia desta é ordenar (de forma crescente pelo custo) o vetor das possíveis arestas antes de rodar o algoritmo de *backtracking*, mas sem remover a simplificação apresentada anteriormente. Com isso, pode-se reduzir ainda mais o número de soluções válidas e, teoricamente, encontrar a melhor solução mais rapidamente. Na seção 5 vamos ver que essa solução nem sempre é mais rápida do que a apresentada no código 4, mesmo considerando um conjunto menor de soluções. Todavia, continua sendo bem mais rápida do que a solução apresentada no código 1.

A solução em questão é apresentada no código 5. Perceba que a única diferença é a adição de um método de ordenação. Esse é o `sort`, da biblioteca padrão do C++, e ele nos possibilita ordenar o vetor de maneira crescente pelo custo da aresta. Isso é garantido pela função `comp` que é requisitada pelo método `sort` e foi implementada pelo autor. Essa função também é mostrada no código 5. O método `combinations` é o mesmo que foi apresentado no código 4.

Código 5: Método responsável por procurar a solução em um espaço de busca ordenado de forma crescente de custo

```
1  bool comp(const Edge &e1, const Edge &e2) {
2         return e1.getCost() < e2.getCost();
3  }
4
5  void OrderedOptimizedBacktracking::findMinimum() {
6         Edge link[edges(Tree::getVertexMax())]; // Numero de rotas possiveis
7                                                 // equivalente ao somatorio de 1...n
8         Tree tree;
9         int length = initializeLinkVector(link);
10        std::sort(link, link+edges(Tree::getVertexMax()), comp); // A funcao comp garante
              ordenacao crescente de custo
11        Chronometer::start();
12        combinations(link, length, Tree::getVertexMax()-1, 0, tree); // Computa as
              combinacoes de arestas possiveis
13        Chronometer::stop();
14        executionTime = Chronometer::elapsedTime();
15 }
```

4 Instâncias do problema

É possível encontrar instâncias deste problema na literatura e elas são classificadas geralmente entre *euclidianas* e *não euclidianas* [ALM06]. Nas instâncias euclidianas são gerados pontos aleatórios no plano Euclidiano que passam a ser associados com os vértices de um grafo $G = (V, E)$. Os custos das arestas são obtidos a partir da equação da distância euclidiana entre os pontos, a saber:

$$d(p,q) = d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + ... + (q_n - p_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

Em distâncias não euclidianas os custos das arestas são gerados aleatoriamente.

Neste trabalho foi implementado um gerador de casos de teste do tipo euclidiano, que trabalha com pontos gerados aleatoriamente em um plano cartesiano de duas dimensões. Baseado em análise de resultados, casos de teste euclidianos parecem mais fáceis de resolver do que os não euclidianos [ALM06].

As instâncias geradas são da forma:

```
<valor de n> <valor de d>
<custo v1--v2> <custo v1--v3> <custo v1--v4> ... <custo v1--vn>
<custo v2--v3> <custo v2--v4> ... <custo v2--vn>
<custo v3--v4> ... <custo v3--vn>
...
<custo v<n-1>--vn>
```

Ex. 1: Formato geral dos arquivos das instâncias.

```
5 2
5 10 15 2
21 2 45
53 12
13
```

Ex. 2: Exemplo de um arquivo válido, de acordo com o formato citado anteriormente.

Note que o custo da conexão de um vértice v_1 com v_2 é o mesmo custo da conexão do vértice v_2 com v_1, por isso, no arquivo é fornecido apenas o valor para $< v_1, v_2 >$.

5 Resultados computacionais

Afim de avaliar a eficiência dos algoritmos propostos, nesta seção serão apresentados os resultados da execução dos programas implementados.

Os testes foram realizados em um computador com processador Intel i3 2ª geração, 4 núcleos e frequência de 3.1 GHz, e 4 GB de memória RAM.

Nas tabelas que seguem, o algoritmo 1 corresponde ao *backtracking* "puro", sem nenhuma simplificação. O algoritmo 2 corresponde a versão otimizada e o algoritmo 3 corresponde a versão otimizada e com espaço de busca ordenado.

`n`	`d`	$Algoritmo_1$	$Algoritmo_2$	$Algoritmo_3$
5	2	7.3e-05s	3.1e-05s	2.8e-05s
5	3	0.000104s	6.1e-05s	2.9e-05s
5	4	0.000107s	4.1e-05s	2.9e-05s
7	2	0.004534s	0.000619s	0.000846s
7	3	0.013171s	0.000802s	0.000931s
7	4	0.010602s	0.000519s	0.000609s
7	5	0.007441s	0.00059s	0.000602s
7	6	0.008809s	0.000789s	0.000607s
9	2	0.269272s	0.011795s	0.015876s
9	3	1.84688s	0.006618s	0.0115s
9	4	2.32626s	0.004202s	0.005747s
9	5	2.43953s	0.025066s	0.021971s
9	6	2.57994s	0.014864s	0.020675s
11	2	45.3547s	0.363536s	0.544982s
11	3	857.386s	0.12304s	0.175352s
11	4	-	0.94497s	0.971935s
11	5	-	1.43194s	1.72668s
11	6	1592.94s	0.588514s	0.74642s
13	2	10280.9s	15.4542s	19.202s
13	3	-	18.7459s	29.5751s
13	4	-	30.044s	45.232s
13	5	-	14.0901s	28.7156s
13	6	-	16.1415s	30.1409s
15	2	-	2181.95	3822.26s
15	5	-	720.007s	1613.27s
15	6	-	372.899s	553.882s

Tabela 1: Tabela contendo o tempo gasto na execução dos programas. O traço "-" indica que o algoritmo não foi executado (quando não houve execução foi devido a prevista longa demora).

É notável que a execução do algoritmo 2 para $n = 15$ e $d = 2$ demorou cerca de 36 minutos, mesmo considerando poucas soluções. Isso se deve ao fato dele ainda precisar procurar várias outras prováveis soluções. O algoritmo sem simplificação demoraria bem mais do que esse período de tempo, como pode-se ver para $n = 13$ e $d = 2$, onde o algoritmo 1 leva

n	d	$Algoritmo_1$	$Algoritmo_2$	$Algoritmo_3$
5	2	60	2	1
5	3	120	10	1
5	4	125	11	1
7	2	2520	5	1
7	3	14070	9	1
7	4	16590	22	1
7	5	16800	11	1
7	6	16807	23	1
9	2	181440	9	1
9	3	3356640	23	1
9	4	4609080	25	1
9	5	4770360	72	1
9	6	4782456	31	1
11	2	19958400	32	2
11	3	1359666000	31	1
11	4	-	77	1
11	5	-	42	1
11	6	-	37	1
13	2	-	48	3
13	3	-	70	1
13	4	-	94	1
13	5	-	84	1
13	6	-	68	1
15	2	-	53	11
15	5	-	111	1
15	6	-	82	1

Tabela 2: Tabela contendo o número de soluções geradas pelos programas. O traço "-" indica que o algoritmo não foi executado (quando não houve execução foi devido a prevista longa demora).

quase 3 horas para encontrar a solução.

Com as execuções realizadas podemos observar também como a complexidade do algoritmo proposto depende, e muito, do número de vértices. O valor do grau máximo não influencia tanto, pelo menos para os casos de teste utilizados com valores pequenos. Talvez, para números um pouco maiores de n o valor d possa fazer uma diferença maior, como visto na execução para $n = 15$ e $d = 6$, onde a execução foi bem mais rápida do que para $n = 15$ e $d = 2$.

Para as instâncias utilizadas neste trabalho, o algoritmo 3 foi bem eficaz. Todavia, ao utilizar outras instâncias com os mesmos valores para n e d (mudando somente os custos das arestas) comprovou-se que as soluções geradas não depende exclusivamente dos valores de n e d, mas dos custos das arestas também. Por isso, não é correto afirmar que esse algoritmo sempre vai encontrar a melhor solução na primeira árvore válida gerada.

Os valores encontrados comprovam que as simplificações fazem di-

ferença no *backtracking* e chamam a atenção para o uso de passos de teste que não sejam "frouxos". Para alguns casos de teste, o algoritmo 3 teve melhor desempenho do que o 2, porém na maioria das instâncias, o algoritmo 2 se saiu melhor. De fato, não podíamos esperar grande desempenho desses dois últimos algoritmos, já que eles ainda precisam percorrer um grande espaço de busca. Mas, como esperado, o algoritmo 3 tende a encontrar o melhor resultado mais rapidamente e considerar um espaço de soluções válida ainda menor do que o do algoritmo 2.

6 Considerações finais

O presente trabalho apresenta uma boa introdução ao problema da Árvore Geradora Mínima de Grau Restrito com a proposta de um algoritmo exato, e suas variantes, que abre caminho para futuras implementações mais eficientes. Além disso, nas implementações percebemos a importância de tentar otimizar a árvore de *backtracking*, o que pode causar um impacto significativo no desempenho do algoritmo.

Aqui também foram apresentados dois programas que podem servir em trabalhos futuros desse problema ou até de outros do gênero: o validador e o gerador de instâncias. Por estarem implementados em uma linguagem popular e disponíveis no repositório *online* apresentado na Introdução, eles podem ser úteis em outros trabalhos, podendo ser parcialmente ou completamente utilizados (desde que a fonte seja citada). Além disso, os casos de teste criados podem, e são úteis, em abordagens para este problema.

Espera-se, em futuros trabalhos, implementar soluções aproximativas que sejas eficientes e encontrem soluções próximas a solução ótima. Para tanto, pode-se utilizar heurísticas, como *Simulated annealing* ou *Colônia de formigas*, ou ainda implementar algum algoritmo genético. Em posse de alguma dessas implementações poderemos comparar as soluções quanto ao seu desempenho e verificar sua aplicabilidade em problemas do cotidiano.

7 Referências

[ALM06] Rafael Andrade, Abilio Lucena, and Nelson Maculan. Using lagrangian dual information to generate degree constrained spanning trees. *Discrete Applied Mathematics*, 154(5):703–717, 2006.

[BDZ12] Thang N Bui, Xianghua Deng, and Catherine M Zrncic. An improved ant-based algorithm for the degree-constrained minimum spanning tree problem. *Evolutionary Computation, IEEE Transactions on*, 16(2):266–278, 2012.

[Bor26] Otakar Borüvka. O jistém problému minimálním. 1926.

[dS10] Rafael Ferreira Barra de Souza. Algoritmos para o problema da árvore geradora mínima probabilística. 2010.

[GJ79] Michael R Garey and David S Johnson. Computers and intractability: a guide to the theory of np-completeness. 1979. *San Francisco, LA: Freeman*, 1979.

[Kru56] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[NH80] Subhash C Narula and Cesar A Ho. Degree-constrained minimum spanning tree. *Computers & Operations Research*, 7(4):239–249, 1980.

[Pri57] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957.

[ZG97] Gengui Zhou and Mitsuo Gen. A note on genetic algorithms for degree-constrained spanning tree problems. *Networks*, 30(2):91–95, 1997.