UM FRAMEWORK PARA GERAÇÃO DE CIRCUITOS ELEMENTARES EM GRAFOS DIRIGIDOS

Rodrigo Alves Randel
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
rodrigorandel@gmail.com

Thiago Correia Pereira
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
thiagocorreiap@gmail.com

Leandro Rochink Costa
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
rochink@gmail.com

Daniel Aloise
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
aloise@dca.ufrn.br

Caroline Thennecy de Medeiros Rocha
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
caroline.rocha@ect.ufrn.br

RESUMO

Neste trabalho é proposto um framework para a geração de circuitos elementares em grafos dirigidos com características ditadas por um conjunto de restrições. A proposta toma como base o algoritmo concebido por Johnson em 1975 para a enumeração de circuitos elementares de grafos dirigidos. A modificação proposta ao algoritmo de Johnson visa aumentar a sua flexibilidade e aplicabilidade, incorporando a possibilidade de adicionarmos filtros de restrições para os circuitos enumerados.

PALAVRAS CHAVE. Grafos. Algoritmos. Otimização Combinatória.

Área Principal: Otimização Combinatória; Teoria e Algoritmos em Grafos

ABSTRACT

This work proposes a framework for generating elementary circuits in directed graphs with characteristics given by a set of constraints. The proposal is based on the algorithm proposed by Johnson(1975). The suggested modification to Johnson's algorithm aims to increase its flexibility and applicability, incorporating the possibility of adding constraint filters for the desired enumerated circuits.

KEYWORDS. Graphs. Algorithms. Combinatorial Optimization.

Main Area: Combinatorial Optimization; Theory and Algorithms on Graphs

1. Introdução

A teoria dos grafos é uma ferramenta poderosa de modelagem de problemas da Pesquisa Operacional (PO). Devido à vasta importância prática da modelagem em grafos para a PO, muitos algoritmos baseados na teoria dos grafos foram desenvolvidos para a resolução de seus problemas.

Problemas importantes na teoria dos grafos consistem em encontrar um ciclo (um circuito no caso de problemas dirigidos) com uma determinada característica. Por exemplo, no problema do caixeiro viajante [Applegate et al., 2011], deseja-se encontrar o ciclo de custo mínimo passando por todos os vértices do grafo. Este ciclo ótimo é naturalmente "elementar" pois qualquer solução contendo subciclos é subótima.

Diversos problemas de otimização combinatória utilizam como subrotina a enumeração de todos os circuitos elementares de um grafo dirigido $G = (V, A)$, por exemplo: para evitar impasses em sistemas industriais fléxiveis [Zhang and Judd, 2008]; no problema de orientação [Leifer and Rosenwein, 1994]; no problema de reabastecimento de postos de combustíveis com vários depósitos [Cornillier et al., 2012]; e no problema de reabastecimento de postos de combustíveis [Cornillier et al., 2009]. Esta enumeração só pode ser alcançada em tempo computacional razoável para grafos pequenos, uma vez que o número total de circuitos elementares é dado pela expressão [Johnson, 1975] :

$$\sum_{i=1}^{|V|-1} \binom{|V|}{|V|-i+1} (|V|-i)!$$

Johnson [Johnson, 1975] propôs um algoritmo para a enumeração de todos os circuitos elementares de um grafo dirigido. O algoritmo executa em tempo $O((|V| + |A|)(c + 1))$, sendo c o número de circuitos elementares do grafo.

A proposta do presente trabalho é apresentar um framework de geração de cicuitos elementares que modifica o algoritmo de Johnson para que o mesmo enumere apenas circuitos elementares obedecendo um determinado conjunto de restrições definidas pelo usuário. Esta modificação visa aumentar a aplicabilidade do algoritmo a diferentes cenários da PO. Por exemplo, para o problema de roteamento de veículos, pode-se adicionar uma restrição de carga aos circuitos gerados. Uma vez que o número de combinações é restringido para os elementos do grafo, o algoritmo aumenta sua eficiência em cenários com maior número de restrições. No sentido de tornar o código flexível e customizável, este artigo propõe um framework para a adição das restrições.

Este artigo está estruturado da seguinte forma: a Seção 2 descreve o algoritmo de Johnson. Na Seção 3, apresentamos nosso framework. Os resultados computacionais são apresentados na Seção 4. Finalmente, as considerações finais são dadas na Seção 5.

2. Algoritmo de Johnson

A ideia do algoritmo de Johnson é encontrar todos os Circuitos Elementares (CE) em um digrafo $G = (V, A)$. CE são caminhos em que o primeiro e último vértice são idênticos e todos os outros vértices só aparecem uma única vez. Dois CE são ditos idênticos se um é uma combinação cíclica do outro.

Para encontrar todos os CE, o algoritmo parte de um nó raiz *s* que pertence ao subgrafo induzido (e fortemente conexo) por *s* e todos os vértices maiores do que *s*, evitando assim, repetição de circuitos elementares previamente encontrados por iterações anteriores do algoritmo (o algoritmo assume que todos os vértices são numericamente identificados de 1 a n). O Algoritmo 1 apresenta o pseudo-código do algoritmo de Johnson.

A entrada do algoritmo é a representação do digrafo em uma estrutura de adjacência A_G composta por uma lista de adjacência $A_G(v)$ para cada $v \in V$. A lista $A_G(v)$ contém u se, e apenas se, $(v, u) \in A$. Também será necessário possuir uma estrutura de adjacência idêntica a de A_G chamada B que irá armazenar vértices bloqueados para evitar que buscas infrutíferas sejam realizadas repetidas vezes.

O algoritmo, então, procede construindo caminhos elementares a partir do vértice s e, à medida que os outros nós vão sendo visitados, eles são adicionados em uma pilha. Adicionar um vértice v ao circuito elementar corresponde a chamar a função $CIRCUIT(v)$. Quando um vértice v é adicionado à pilha, ele é dito como bloqueado ($blocked(v) = true$) para que não seja utilizado novamente na construção dos circuitos. Em seguida é verificado se o nó atual v consegue alcançar o vértice raiz s, formando assim um circuito que pode ser salvo ou impresso. Em seguida, a lista $A_G(v)$ é percorrida e, para cada nó u da lista, se ele não estiver bloqueado, é chamada a função $CIRCUIT(u)$ de forma recursiva. No fim da função $CIRCUIT$, o vértice v é retirado do bloqueio e retirado da pilha. O pseudocódigo do algoritmo de Jonhson é mostrado abaixo.

Retirar do bloqueio não significa apenas fazer $blocked(v) \leftarrow false$. Na linha 16 do algoritmo é mostrada uma possível chamada ao método ***UNBLOCK***. Se a busca encontrou circuitos elementares a partir de v, então ele, e todos os vértices que estão em $B(v)$, podem ser desbloqueados. Em outras palavras, pode ser desbloqueado todo vértice w que não havia gerado circuitos elementares e que possui um arco (w, v) no subgrafo atual. Já que um vértice apenas é desbloqueado se for retirado da pilha de vértices antes de qualquer chamada ao método CIRCUIT, então nenhum vértice irá aparecer na pilha de vértices mais de uma vez e, consequentemente, a saída do algoritmo são apenas os circuitos elementares.

3. Framework Proposto

Essa seção irá abordar a elaboração do framework proposto. Na Seção 3.1, apresentamos o formato das restrições a serem filtradas pelo nosso framework. Na Seção 3.2 apresentaremos as estruturas de dados necessárias para implementação destas restrições. Na Seção 3.3 é apresentado o framework completo para adição dos filtros de restrições ao algoritmo de Johnson.

3.1. Filtro de Restrições

Quando estamos trabalhando com grafos e realizando buscas com objetivo de formar circuitos elementares é comum o interesse em adicionar restrições na construções desses elementos. No PRV, por exemplo, é comum haver custos associados aos arcos do grafo representados através de uma função $w : V \times V \rightarrow \mathbb{R}$.

Por exemplo, a fim de tornar mais clara a utilização de restrições durante a geração de um circuito elementar em um grafo, vamos ilustrar a modelagem de uma restrição aplicada ao PRV. Suponha que cada veículo possui uma capacidade máxima de transportar C unidades, e que existam n clientes, $v_1, v_2, \ldots, v_n \in V$, para serem atendidos onde cada um possui demanda $p_1, p_2...p_n$, respectivamente. Uma possível restrição é que o somatório das demandas dos clientes em uma rota R (i.e., um circuito) seja menor do que C.

$$\sum_{i:v_i \in R} p_i \leq C$$

Para que as restrições sejam filtradas é necessário levar-se em consideração os custos dos arcos do grafo. Desta forma, sempre que se desejar adicionar um vértice v a um circuito, os custos associados ao arco que irá atingir v precisam ser verificados. Seguindo essa estratégia é possível filtrar circuitos elementares levando em consideração diversas restrições do PRV, tais como o número máximo de clientes em uma rota ou o horário limite de retorno ao vértice base.

3.2. Estruturas de Dados

Nosso framework trabalha com matrizes de custo τ^i para cada restrição $i = 1, \ldots, m$, onde m é o número de restrições desejadas. Desta forma, τ^i_{vw} armazena o custo do arco (v, w) associado à *i*-ésima restrição. Para ficar mais claro, vamos exemplificar a utilização do vetor associado ao arco aplicando as três restrições citadas anteriormente para o problema do PRV - capacidade do veículo, número máximo de clientes e horário limite de retorno à base:

Suponha a construção de um circuito a partir de u levando a v. Ou seja, o vértice u faz parte do circuito em construção e desejamos saber se v também pode fazer parte do mesmo circuito

Algoritmo 1 Algoritmo de Jonhson

```
1:  função CIRCUIT(v)
2:      Adiciona v na pilha
3:      blocked(v) ← true
4:      retorno ← false
5:      para ∀ w ∈ A_G(v) faça
6:          se w = s então
7:              Armazena o CE composto pelos elementos da pilha e s
8:              retorno ← true
9:          senão e se w não está bloqueado então
10:             se CIRCUIT(w) então
11:                 retorno ← true
12:             fim se
13:         fim se
14:     fim para
15:     se retorno = true então
16:         UNBLOCK(v)
17:     senão
18:         para ∀ w ∈ A_G(v) faça
19:             se v ∉ B(w) então Adiciona v em B(w)
20:         fim para
21:     Remove v da pilha
22:     retorne retorno
23: fim CIRCUIT
24:
25: função UNBLOCK(v)
26:     blocked(v) ← false
27:     para ∀ w ∈ B(v) faça
28:         Deleta w de B(v)
29:         se blocked(w) = true então
30:             UNBLOCK(w)
31:         fim se
32:     fim para
33: fim UNBLOCK
34:
35: função MAIN()
36:     pilha ← ∅
37:     n ← número de vértices
38:     s ← 1
39:     enquanto s < n faça
40:         A_G ← componente fortemente conexo induzido por {s, s + 1, s + 2...}
41:         se A_G ≠ ∅ então
42:             Desbloqueia todos os vértices
43:             CIRCUIT(s)
44:             s = s+1
45:         senão
46:             s = n
47:         fim se
48:     fim enquanto
49: fim MAIN
```

dado que $(u, v) \in A$. Vamos supor que $p_v = 5$ e que $w_{uv} = 50$ sejam a demanda do vértice v e o tempo de deslocamento do arco (u, v), respectivamente. Uma vez que as restrições consideradas são: capacidade do veículo, número máximo de clientes e horário limite de retorno à base, nesta ordem, os valores associados ao arco (u, v) em τ são: (5,1,50). Suponha agora que a restrição de capacidade máxima do veículo é 45, que no máximo 5 clientes possam ser servidos e que o tempo máximo de rota é 1200 minutos. Estes valores são armazenas no vetor δ onde δ^i representa o limite superior para a restrição i. Além disso, também é necessário definir um vetor β em que β^i

corresponde ao valor acumulado da *i*-ésima restrição para o circuito elementar em construção. A Figura 1 ilustra as estruturas de dados que acabamos de mencionar. No exemplo mostrado, o vetor β informa que, no momento, já foram gastas 30 unidades da capacidade do véiculo, 3 vértices já foram adicionados na rota e o tempo atual é de 500 minutos. Ao se testar a adição do vértice v à rota a partir de u verifica-se se $\beta^i + \tau^i_{uv} \leq \delta^i$, para $i = 1, \ldots, m$. Em caso afirmativo v pode ser adicionado à rota.

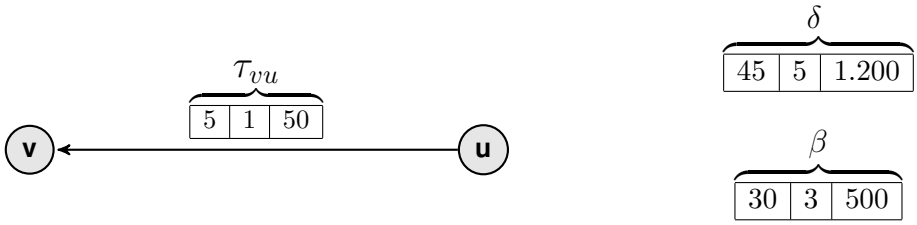

Figura 1: Representação do vetor associado ao arco, τ_{uv}, do vetor contendo os valores limites superiores das restrições, δ , e do vetor contendo os valores acumulados das restrições β.

O esquema descrito acima é exclusivo à restrições do tipo menor ou igual, uma vez que testamos se a soma $\beta^i + \tau^i_{uv}$ é menor ou igual ao elemento δ^i, para $i = 1, \ldots, m$. Entretanto, em muitas aplicações, é necessário modelar também restrições de maior igual e de igualdade. Para permitir que essas restrições sejam adicionadas é necessário definir os seguintes vetores:

1. ${}^{+}\delta$, ${}^{+}\beta$ e ${}^{+}\tau_{uv}$ irão armazernar os valores máximos, os valores acumulados e os custos do arco (u, v), resp., para testar restrições de menor ou igual;
2. ${}^{-}\delta$, ${}^{-}\beta$ e ${}^{-}\tau_{uv}$ irão armazernar os valores mínimos, os valores acumulados e os custos do arco (u, v), resp., para testar restrições de maior ou igual;
3. ${}^{=}\delta$, ${}^{=}\beta$ e ${}^{=}\tau_{uv}$ irão armazernar os valores exatos, os valores acumulados e os custos do arco (u, v), resp., para testar restrições de igualdade.

Dessa forma, se queremos adicionar um vértice v, a partir do vértice u, temos que testar as restrições:

$$
\begin{aligned}
{}^{+}\beta^i + {}^{+}\tau^i{}_{uv} &\leq {}^{+}\delta^i, \text{ para todo } i = 1, \ldots, |{}^{+}\delta|; \\
{}^{-}\beta^i + {}^{-}\tau^i{}_{uv} &\geq {}^{-}\delta^i, \text{ para todo } i = 1, \ldots, |{}^{-}\delta|; \\
{}^{=}\beta^i + {}^{=}\tau^i{}_{uv} &= {}^{=}\delta^i, \text{ para todo } i = 1, \ldots, |{}^{=}\delta|.
\end{aligned}
$$

Para facilitar a notação, iremos definir como:

- Δ a estrutura de dados contendo os vetores ${}^{+}\delta$,${}^{-}\delta$ e ${}^{=}\delta$;
- Γ_{uv} a estrutura de dados contendo os vetores ${}^{+}\tau_{uv}$,${}^{-}\tau_{uv}$ e ${}^{=}\tau_{uv}$;
- Θ a estrutura de dados contendo os vetores ${}^{+}\beta$,${}^{-}\beta$ e ${}^{=}\beta$;

Ainda assim, existe uma classe de restrições presente em aplicações práticas que não são cobertas por esse esquema. Por exemplo, ao trabalharmos com janelas de tempo, cada vértice possui limitantes diferentes exigindo outras estruturas de dados que sejam capazes de testar, de forma local ao vértice, essas restrições.

Por exemplo, suponha que cada cliente v_i possui uma janela de tempo $[B_i, E_i]$ para ser atendido. Logo, toda vez que um veículo se deslocar até o cliente v_i, é necessário incrementar o horário atual com o tempo de deslocamento e verificar se o novo horário está dentro do intervalo $[B_i, E_i]$. Uma vez que os valores B_i e E_i são exclusivos ao cliente v_i, não é viável que essa restrição seja tratada pelos vetores em Δ, porque dessa forma todos os vetores Γ_{uv} ($\forall (u, v) \in A$) teriam que adicionar

valores nulos referentes a essa restrição com exceção do arcos que atingem o vértice do cliente v_i, constituindo um grande desperdício de memória.

Para modelar as restrição locais ao vértice nosso framework utiliza vetores exclusivos a cada vértice. Assim, a cada vértice $v \in V$ está associado:

1. um vetor ${}^{+}\alpha_v$ para armazenar os valores máximos das restrições de menor ou igual;
2. um vetor ${}^{-}\alpha_v$ para armazenar os valores mínimos das restrições de maior ou igual e;
3. um vetor ${}^{=}\alpha_v$ para armazenar os valores exatos das restrições de igualdade.

Iremos denotar como Λ_v a estrutura de dados que agrupa os vetores ${}^{+}\alpha_v$, ${}^{-}\alpha_v$ e ${}^{=}\alpha_v$

Para o exemplo da janela de tempo, o valor E estará armazenado em ${}^{+}\alpha$ e o valor B em ${}^{-}\alpha$. Os vetores em Θ agora também irão armazenar os valores acumulados para testar as restrições implementadas em Λ_v ($\forall v \in V$). Além disso, os vetores em Γ_{uv} ($\forall (u,v) \in A$) também precisarão armazenar os custos dos arcos utilizados para testes das restrições locais de cada vértice.

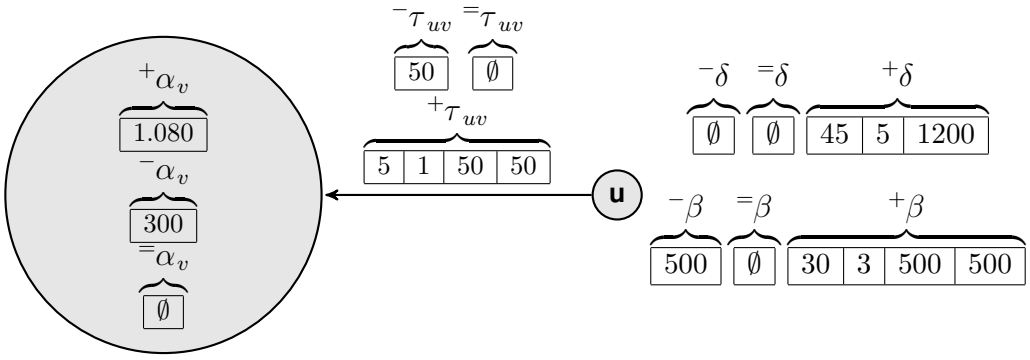

Figura 2: Representação das estruturas de dados necessárias para implementar todas as restrições da aplicação

A Figura 2 exibe, em detalhes, a representação dos vetores necessários para modelar as restrições. Nesse exemplo, o vetor ${}^{+}\alpha_v$ armazena o valor do fim da janela de tempo (1.080 minutos) e o vetor ${}^{-}\alpha_v$ informa que o início da janela de tempo do cliente é em 300 minutos. O custo de deslocamento do arco (u, v) é de 50 minutos e, para essas restrições serem testadas, esse valor foi concatenado no fim dos vetores ${}^{+}\tau_{uv}$ e ${}^{-}\tau_{uv}$. Por último o valor acumulado de 500 minutos precisa também ser adicionado nos vetores β^{+} e β^{-}. Como não há nenhuma restrição que testa a igualdade, os vetores ${}^{=}\alpha_v$, ${}^{=}\tau_{uv}$, ${}^{=}\delta$ e ${}^{=}\beta$ são todos vazios.

Uma vez que os vetores em Θ e Γ_{uv} armazenam os valores relacionados às restrições globais (limitantes em Δ) e às restrições locais a cada vértice $v \in V$ (limitantes em Λ_v), é preciso ter cuidado ao percorrer esses vetores para testar corretamente cada restrição. Dessa forma, sempre que desejamos adicionar um vértice v ao circuito, a partir de um vértice u, precisamos realizar os seguintes testes:

T1 ${}^{+}\beta^i + {}^{+}\tau^i{}_{uv} \leq {}^{+}\delta^i$, para todo $i = 1, \ldots, |{}^{+}\delta|$;

T2 ${}^{-}\beta^i + {}^{-}\tau^i{}_{uv} \geq {}^{-}\delta^i$, para todo $i = 1, \ldots, |{}^{-}\delta|$;

T3 ${}^{=}\beta^i + {}^{=}\tau^i{}_{uv} = {}^{=}\delta^i$, para todo $i = 1, \ldots, |{}^{=}\delta|$.

T4 ${}^{+}\beta^i + {}^{+}\tau^i{}_{uv} \leq {}^{+}\alpha^i{}_v$, para todo $i = |{}^{+}\delta| + 1, \ldots, |{}^{+}\delta| + |{}^{+}\alpha_v|$;

T5 ${}^{-}\beta^i + {}^{-}\tau^i{}_{uv} \geq {}^{-}\alpha^i{}_v$, para todo $i = |{}^{-}\delta| + 1, \ldots, |{}^{-}\delta| + |{}^{-}\alpha_v|$;

T6 ${}^{=}\beta^i + {}^{=}\tau^i{}_{uv} = {}^{=}\alpha^i{}_v$, para todo $i = |{}^{=}\delta| + 1, \ldots, |{}^{=}\delta| + |{}^{=}\alpha_v|$.

Para tornar o framework ainda mais genérico, foram introduzidos operadores lógicos que podem ser utilizados para modificar e combinar restrições. Os operadores são:

1. *AND*(&) – realiza uma interseção do resultado de duas restrições
2. *OR*(||) – realiza uma união do resultado de duas restrições;
3. *NOT*(!) – realiza uma negação do resultado de uma restrição.

Por exemplo, se queremos testar uma diferença, não é preciso criar outros vetores para trabalhar apenas com restrições de diferença, basta realizar uma operação de negação sobre a restrição já estruturada nos vetores de igualdade (**T3** e **T6**). Se existe uma restrição de igualdade, R, e queremos adicionar um vértice v apenas se a restrição R resultar em uma diferença, ou seja, se a condição imposta por R não for verdadeira, basta negar o resultado do teste do **T6**, por exemplo. Seguindo o mesmo raciocínio, podemos implementar uma restrição de $>$ apenas negando o resultado dos testes **T1** ou **T4** e uma restrição $<$ negando o resultado dos testes **T2** ou **T5**.

Além da operação de *NOT*, podemos agregar as restrições utilizando os operadores *AND* e *OR*. Sabemos que o número total de restrições no problema é dado pela soma dos tamanhos dos vetores:

$$m = |^{+}\beta| + |^{-}\beta| + |^{=}\beta|,$$

cada restrição pode ser rotulada como R_i com $i = 1, 2...m$. Por exemplo, se temos uma aplicação com 3 restrições, R_1, R_2 e R_3, e queremos adicionar um vértice v apenas quando:

$$(R_1 \;||\; !R_2) \;\&\; R_3$$

então é necessário definir uma estrutura de dados que permita combinar restrições de forma arbitrária. Para calcular esse tipo de expressão utilizamos uma estrutura de árvore como mostrada na Figura 3. O valor de cada nó da árvore será o resultado da verificação da restrição através do fluxo associado, ou a combinação das restrições filhas através da operação indicada pelo nó. O resultado da expressão $(R_1 \;||\; !R_2) \;\&\; R_3$ será obtida no momento que toda a árvore é mudada

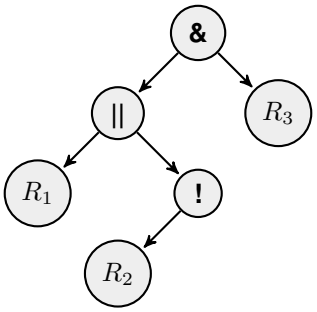

Figura 3: Representação da estrutura em árvore necessária para utilizar restrições lógicas sobre as restrições

A verificação das restrições é realizada como proposto no Algoritmo 2:

(a) Inicialmente é definido a estrutura de um nó da árvore de restrições. Cada nó pode possui ou uma operação (op) que será realizada sobre os filhos (L) ou uma restrição que será testada pelo fluxo de teste associado (R).
(b) na primeira linha da função é informado o vértice v que está sendo testado, a estrutura contendo os limites locais do vértice, Λ_v, a estrutura com os custos do arco incidente no vértice v, Γ_{uv}, a estrutura contendo os vetores acumuladores, Θ, e o nó $raiz$ da árvore que será observado nessa iteração;

(c) em seguida é verificado se o nó em questão possui uma restrição, $raiz.R$, que deve ser verificada. Caso tenha, a restrição é testada utilizando as estruturas Δ (global ao algoritmo), Γ_{uv}, Λ_v e Θ. Note que é necessário atualizar os valores Θ durante o teste.
(d) se o nó não possuir uma restrição, então o retorno do nó será a combinação dos seus filhos através do operador definido em $raiz.op$;
(e) a linha 6 percorre todos os filhos do nó raiz atual;
(f) a linha 7 realiza a operação $raiz.op$ com o valor de todos os nós filhos obtidos através de uma chamada recursiva a função **verificarRestricoes**(v,Λ_v,Γ_{uv},Θ,l);
(g) na linha 9 o resultado da combinação dos nós filhos é retornado

Algoritmo 2 Algoritmo para verificação das restrições percorrendo a árvore

```
Nó {
  op – operação do nó (&,||,!)
  R – restrição do nó
  L – lista com os nós filhos
}

1:  função verificarRestricoes(v,Λ_v,Γ_uv,Θ,raiz) {
2:    se raiz.R != ∅ então
3:      retorne o resultado da restrição raiz.R para o vértice v.
4:    senão
5:      r – armazenará o resultado final do nó
6:      para ∀ l ∈ raiz.L faça
7:        r ← r {raiz.op} verificarRestricoes(v,Λ_v,Γ_uv,Θ,l)
8:      fim para
9:      retorne r
10:   fim se
11: fim função
```

O algoritmo 2 percorre a árvore de restrições em pós-ordem. O tempo de execução desse algoritmo é dado pelo tamanho da árvore. Se temos m restrições, temos pelo menos $m + 1$ vértices contendo todas as restrições e pelo menos um vértice para realizar uma operação de combinar todos as restrições. Cada operação envolvendo uma ou mais restrições também irá representar um vértice na árvore, logo, se temos ρ operações, esse algoritmo terá complexidade $O(m + \rho)$.

O Algoritmo 3 mostra como adicionar um vértice v ao circuito a partir do vértice u. Como entrada, o algoritmo deve receber o vértice que se deseja adicionar (v), o circuito atual em construção (C), estrutura contendo os limites locais do vértice $v(\Lambda_v)$, a estrutura com os custos do arco $(u, v)(\Gamma_{uv})$, a cópia da estrutura contendo os vetores acumuladores (Θ) e o nó raiz da árvore de restrições ($raiz$). O retorno do método deve ser um novo circuito contendo o vértice v ou $\emptyset$ caso alguma das restrições seja violada. Na linha 2 é verificado se o vértice v é compatível com o circuito atual em formação realizando uma chamada a função **verificarRestricoes**, ou seja, é verificado se, ao adicionar v, todas as restrições serão atendidas. Por último, uma novo circuito $C^{'}$ é criado, agora contendo o vértice v, sendo C' retornado em seguida.

Algoritmo 3 Adicionar elemento ao circuito

```
1: função addElement(v, C, Λ_v,Γ_uv,Θ,raiz)
2:    se ! verificarRestricoes(v,Λ_v,Γ_uv,Θ,raiz) então
3:       retorne ∅
4:    senão
5:       C' ← {C, v}
6:       retorne C'
7:    fim se
8: fim função
```

3.3. O Algoritmo de Johson Modificado

Mostramos agora como o algoritmo de Johnson está implementado em nosso framework de modo a computar todos os circuitos elementares de um digrafo obedecendo a um conjunto de restrições definidos pelo usuário.

Algoritmo 4 Construir Circuitos Elementares

```
1: função build(v, C,Θ)
2:    bloqueados(v) ← true
3:    para ∀ w ∈ A_G(v) faça
4:       se w = c[0] então
5:          Θ' ← Θ
6:          C' ← addElement(w, C,Λ_v,Γ_uv,Θ',raiz)
7:          se C' ≠ ∅ então
8:             Salva o circuito elementar C'
9:          fim se
10:      senão se ! bloqueados(w) então
11:         Θ' ← Θ
12:         C' ← addElement(w, C,Λ_v,Γ_uv,Θ',raiz)
13:         se C' ≠ ∅ então
14:            build(w, C',Θ')
15:         fim se
16:      fim se
17:   fim para
18:   bloqueados(v) ← false
19: fim função
```

O funcionamento do Algoritmo 4 se dá da seguinte forma:

(a) na linha 2 o vértice v é visitado e bloqueado;
(b) entre as linhas 3 e 18 são percorridas todas as arestas que saem do vértice v;
(c) na linha 4 é verificado se o vértice w é igual ao primeiro vértice do CE ($c[0]$), formando um circuito elementar, C';
(d) na linha 5 é criada uma cópia Θ' da estrutura Θ para manter intacto os valores em Θ, uma vez a função **addElement** pode modificar os valores, e eles precisam ser reutilizados para adicionar os outros vértices da lista $A_G(v)$ nas próximas iterações.
(e) na linha 6 o vértice base do CE é adicionado em C formando um CE completo, C'.
(f) a linha apenas checa se foi possível adicionar o vértice base em C, verificando se $C' \neq \emptyset$;
(g) na linha 8 o novo circuito elementar, C', foi em encontrado e pode ser impresso ou salvo;
(h) a linha 10 apenas verifica se o vértice w não está bloqueado;
(i) entre as linhas 11 e 15 o vértice w é adicionado ao circuito em construção e é realizada uma chamada recursiva a função **build** na tentativa de formar outros circuitos elementares a partir de w.
(j) Na linhas 18 o vértice v é desbloqueado para ser utilizado em novas buscas.

A principal diferença do algoritmo do nosso framework em relação ao algoritmo de Johnson se dá quanto à política de desbloquear um vértice. No algoritmo de Johnson, se o vértice v foi infrutífero na busca por circuitos elementares ele não será removido do desbloqueio ao fim da função **CIRCUIT**. Ele será mantido em bloqueio até que algum vértice alcançado por v encontre um circuito elementar. Em nosso algoritmo essa política é mais relaxada pois adicionar um vértice v a partir de um vértice u pode ser totalmente diferente de adicionar v a partir de w devido às restrições de cada vértice. Então v não pode ser mantido em bloqueio ao fim da função **build**.

4. Resultados

Para testar o algoritmo proposto utilizamos duas instâncias propostas por Augerat *et al* em 1995 para o PRV disponíveis em `http://vrp.atd-lab.inf.puc-rio.br/index.php/en/`.

	Nº de clientes	**Capacidade Máxima do Veículo**
P-n19-k8	19	160
P-n20-k8	20	160

Tabela 1: Características das instâncias do PRV utilizadas

Nosso *framework* é utilizado para enumerar todas as rotas possíveis considerando a restrição de capacidade máxima do veículo e número máximo de clientes por rota. As instâncias *P-n19-k8* e *P-n20-k8* nos permitem testar o algoritmo proposto com problemas de ordem diferentes já que possuem 19 e 20 clientes, respectivamente.

O principal objetivo do *framework* de restrições proposto é filtrar as buscas realizadas nos dígrafos. Aplicar restrições durante a construção de circuitos pode reduzir signitivamente o número de circuitos finais, economizando significativamente tempo de processamento caso a seleção dos circuitos desejáveis fosse feita *a posteriori*.

Para exemplificar o poder que o filtro de restrições pode ter, utilizamos as instâncias apresentadas anteriormente modificando a capacidade do veículo e o número máximo de clientes por rota a fim de testar diferentes configurações. Inicialmente, alteramos a capacidade máxima do veículo para as duas instâncias, assumindo os valores 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120 e 130, conforme pode ser visto nas Tabelas 3 e 4. Em seguida, foi alterado o número máximo de clientes permitidos em uma rotas das instâncias, assumindo os valores de 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 e 15, de acordo com as Tabelas 5 e 6. Todas as instâncias foram executadas no ambiente mostrado pela Tabela 2.

Arquitetura:	64 bits
Processadores:	6 Intel(R) Xeon(R) CPU X5650 2.67GHz.
Memória RAM:	32 Gb
Sistema Operacional:	Linux Ubuntu 14.04

Tabela 2: Ambiente de execução das instâncias

Analisando as Tabelas 3, 4, 5 e 6 podemos observar que a medida que a capacidade e o número máximo de clientes permitidos em uma rota aumentam, o número total de rotas viáveis e o tempo de execução também se elevam. Em outras palavras, aplicar restrições durante a busca no grafo pode provocar grandes impactos no número de resultados e ser muito significativo para a solução do problema como um todo.

Capacidade	Rotas	Tempo(s)
20	35	0.00000691414
30	287	0.0000519753
40	2038	0.000374079
50	12102	0.00238895
60	68464	0.01337
70	370104	0.045799
80	1911666	0.20923
90	8832708	0.936344
100	39341628	4.25803
110	168744852	17.7517
120	665916420	68.7331
130	2533068060	263.79

Tabela 3: Resultados da Instância ***P-n19-k8*** ao alterar a capacidade máxima do veículo.

Capacidade	Rotas	Tempo(s)
20	148	0.0000150204
30	1592	0.000305891
40	15181	0.00304484
50	115657	0.0161819
60	864479	0.0983529
70	5621569	0.623109
80	32160061	3.71014
90	169610407	19.5176
100	843931063	92.4437
110	3920170111	427.123
120	16593437479	1795.33
130	67516543879	7300.29

Tabela 4: Resultados da Instância ***P-n20-k8*** ao alterar a capacidade máxima do veículo.

Max. Clientes	Rotas	Tempo(s)
3	4	0.000000953674
4	15	0.00000214577
5	64	0.00000286102
6	325	0.0000140667
7	1956	0.000154018
8	13699	0.00119901
9	64240	0.00469112
10	371529	0.0272911
11	1951300	0.1396
12	14912311	1.12814
13	69221904	5.44963
14	221762845	18.5566
15	1066574404	93.2458

Tabela 5: Resultados da Instância ***P-n19-k8*** ao alterar o número máximo de clientes por rota.

Max. Clientes	Rotas	Tempo(s)
3	4	0.000000953674
4	15	0.00000190735
5	64	0.00000286102
6	325	0.0000138283
7	1956	0.0000898838
8	13699	0.000653982
9	64240	0.0044601
10	416889	0.0290689
11	2883700	0.193064
12	18465511	1.39347
13	95636544	7.43555
14	719639245	57.5324
15	2333045764	206.304

Tabela 6: Resultados da Instância ***P-n20-k8*** ao alterar o número máximo de clientes por rota.

Os gráficos da Figura 4 apresentam o comportamento das instâncias quando aumentamos a capacidade máxima do veículo e o número máximo de clientes por rota. Devido a natureza combinatória do problema, notamos um comportamento exponencial quando o problema se torna menos restrito. Também fica notória a vantagem de aplicar restrições para problemas em grafos, como o PRV, porque podemos filtrar apenas rotas viáveis e em tempo muito inferior ao de percorrer todo o grafo.

5. Considerações Finais

Este trabalho apresentou um esquema genérico para adição de restrições em problemas em que se deseja enumerar circuitos elementares de um grafo dirigido com certas características. Este trabalho se baseou no algoritmo de Johnson de construção circuitos elementares, possibilitando a rápida modelagem de restrições a serem aplicadas durante o processo de busca no grafo. O frameowrk proposto também permite a utilização de operações lógicas para combinar e modificar restrições, aumentando o caráter genérico do algoritmo e permitindo que sejam testados vários tipos de condições diferentes ($>, \geq, <, \leq, =, != , \&, ||, !$)

Os resultados computacionais apresentados mostram como o tratamento *a priori* das restrições desejadas durante a fase de construção dos circuitos elementares acelera enormemente o processo de aquisição dos circuitos desejados quando comparado a uma estratégia a posteriori. A partir dos resultados obtidos, podemos destacar a contribuição desse trabalho à comunidade cientifica por se tratar de um algoritmo genérico, sendo escasso na literatura trabalhos dessa natureza.

Finalmente, cabe destacar que embora apresentado aqui dentro do conexto de obtenção de circuitos elementares em grafos dirigidos, nosso framework pode ser utilizado para ob-

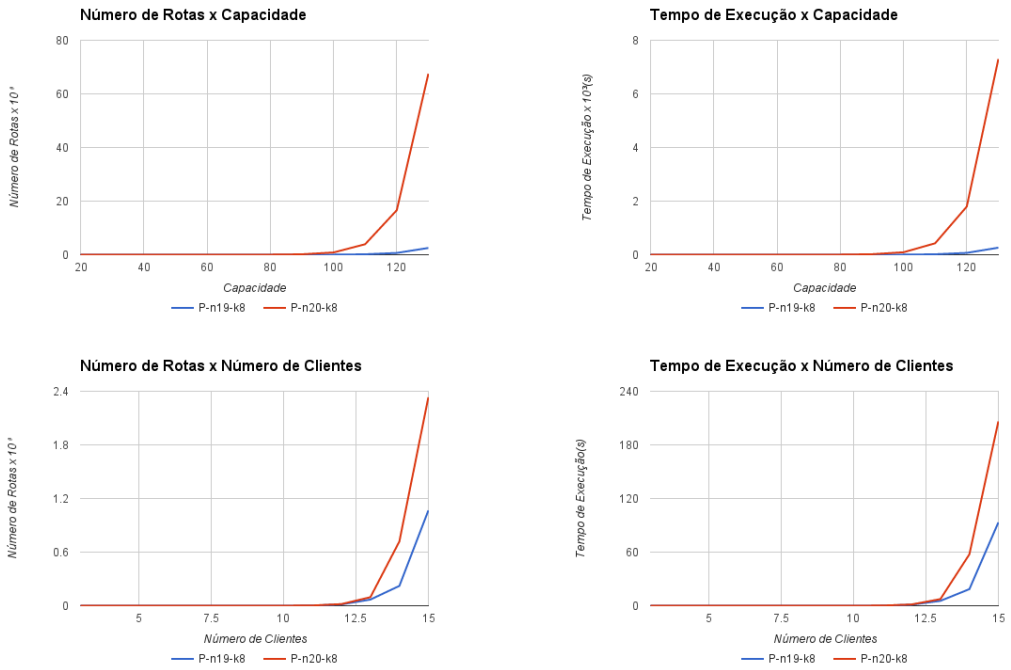

Figura 4: Comportamento das instâncias quando as capacidades máximas dos veículos o número máximo de clientes por rota são alteradas.

tenção/enumeração de outras estruturas em grafos (orientados ou não), tais como caminhos ou subgrafos.

Referências

Applegate, D. L., Bixby, R. E., Chvatal, V., and Cook, W. J. (2011). *The traveling salesman problem: a computational study*. Princeton University Press.

Cornillier, F., Boctor, F., and Renaud, J. (2012). Heuristics for the multi-depot petrol station replenishment problem with time windows. *European Journal of Operational Research*, 220(2):361 – 369.

Cornillier, F., Laporte, G., Boctor, F. F., and Renaud, J. (2009). The petrol station replenishment problem with time windows. *Computers and Operations Research*, 36(3):919 – 935.

Johnson, D. B. (1975). Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84.

Leifer, A. C. and Rosenwein, M. B. (1994). Strong linear programming relaxations for the orienteering problem. *European Journal of Operational Research*, 73(3):517 – 523.

Zhang, W. and Judd, R. P. (2008). Deadlock avoidance algorithm for flexible manufacturing systems by calculating effective free space of circuits. *International Journal of Production Research*, 46(13):3441–3457.