Uma Biblioteca de Grafos para a Linguagem MiniZinc

Claudio Cesar de Sá
Departamento de Ciência da Computação – Universidade do Estado de Santa Catarina (UDESC)
Rua Paulo Malschitzki, 200 - Campus Universitário Prof. Avelino Marcante – 89.219-710 – Joinville – SC – Brazil
claudio.sa@udesc.br

Marcos Creuz Filho
Departamento de Ciência da Computação – Universidade do Estado de Santa Catarina (UDESC)
Rua Paulo Malschitzki, 200 - Campus Universitário Prof. Avelino Marcante – 89.219-710 – Joinville – SC – Brazil
marcos.creuz.filho@gmail.com

Adriano Fiorese
Departamento de Ciência da Computação – Universidade do Estado de Santa Catarina (UDESC)
Rua Paulo Malschitzki, 200 - Campus Universitário Prof. Avelino Marcante – 89.219-710 – Joinville – SC – Brazil
adriano.fiorese@udesc.br

RESUMO

Esse artigo apresenta uma biblioteca de funções e predicados sobre grafos para linguagem MiniZinc. Esta linguagem destina-se a construção de modelos computacionais nas áreas de otimização combinatória ou discreta, programação por restrições e pesquisa operacional. Estas áreas abordam problemas combinatórios os quais pertencem a classe NP de problemas. Muitos dos problemas NPs são modelados com estruturas complexas, tais como grafos. Neste contexto, há uma motivação em se construir uma biblioteca de grafos, pois esta é uma estrutura computacional fundamental destinada à aplicações computacionais em geral. O uso desta biblioteca é ilustrada com dois problemas clássicos da área de grafos: verificar se um grafo é completo e obter o conjunto de cliques$-k$. Finalmente, o problema do cliques$-k$ é modificado para o problema do clique máximo, onde alguns experimentos são realizados.

PALAVRAS CHAVE. Linguagens orientadas a modelos. Biblioteca de grafos. Programação por restrições. Otimização combinatória.

Tópicos: TAG - Teoria e Algoritmos em Grafos, OC - Otimização Combinatória, EDU - PO na Educação

ABSTRACT

This article presents a library of functions and predicates on graphs to MiniZinc language. This language aims at creating computational models to combinatorial or discrete optimization, constraint programming and operational research. These areas approach combinatorial NP problems. Many of such NP problems are modeled as complex structures, such as graphs. In this context, there is a motivation to develop a graphs library, due to this to be a fundamental structure aiming at general computer applications. The usage of this library is demonstrated by two classical problems on graphs: checking if a graph is complete and obtaining the $k-$clique set. Finally, the $k-$clique problem is modified to the problem of maximum clique, where some other experiments were done.

KEYWORDS. Modelling languages. Graphs library. Constraint Programming. Combinatorial Optimization.

1. Introdução

Em nosso cotidiano, nos deparamos com problemas difíceis de serem implementados e cujo tempo de resposta nem sempre são satisfatórios. Outros ainda, fáceis de serem implementados com linguagens de programação apropriadas, mas de difíceis soluções, como o problema da mochila binária Papadimitriou [1994]; Sipser [2012]. Por exemplo, em uma pesquisa de uma passagem aérea entre dois pontos, felizmente temos respostas em segundos. Em geral, dado dois pontos, origem e destino da viagem, e dada a disponibilidade de horários de uma única companhia, não se tem muitas combinações possíveis. Contudo, se todas companhias fossem integradas, e que voce pudesse trocar de avião e de companhia em alguns pontos intermediários, ai sim, este número de possibilidades cresceria. Neste caso, o tempo de resposta cresce mais rapidamente que o tamanho destas entradas, isto é, um crescimento exponencial.

Para estes problemas, uma resposta com um algoritmo em tempo polinomial da ordem $\mathcal{O}(t(n)^c)$, ainda é desconhecido. Aqui, a medida de complexidade é a função assintótica big-$\mathcal{O}$ onde $t(n)$ é o custo ou tempo dispendido do cálculo de uma instância do problema com a entrada de tamanho n. A constante c em $\mathcal{O}(t(n)^c)$ é uma constante tal que $c > 0$ Papadimitriou [1994]; Sipser [2012].

Assim, estes problemas combinatoriais são onipresentes no mundo real e efetivamente requisitam soluções aceitáveis em tempo e recurso dispendido. As soluções apresentam aspectos quantitativos e qualitativos tais como: esta solução é ótima? Esta é única ou há muitas que atendem os requisitos? Ou ainda, demonstre que o problema admite uma solução exata. Em geral, estes são relacionados há problemas de: escalonamentos, planejamento, jogos, atribuição, diagnósticos, multi-critérios, etc. Invariavelmente, estes encontram-se com uma complexidade dentro das classes NP e NP-difíceis Papadimitriou [1994]; Sipser [2012].

Neste contexto, os problemas cuja estruturação de uma solução algorítmica exata ou aproximada, e que conduzam há um significativo número de espaço de estados são de interesse de pesquisa por áreas diversas. Assim, um problema que exija uma construção algorítmica, cuja complexidade ultrapasse a ordem de complexidade $\mathcal{O}(2^{t(n)^c})$, este é um problema exponencial difícil. Estes problemas são conhecidos como problemas das classes NPs, não-polinomiais por uma Máquina de Turing não-determinística Papadimitriou [1994]; Sipser [2012].

Diversas pesquisa se voltam a estes problemas, áreas como Pesquisa Operacional (PO), Programação por Restrições (PR) Rossi et al. [2006]; de Sá [2016], Inteligência Artificial (IA), a Computação Evolucionária (CE), etc. Um dos objetivos recorrentes destas áreas é a busca de metodologias, algoritmos e linguagens que resolvam estes problemas de modo eficiente. Um algoritmo é eficiente há um problema é quando o mesmo apresenta um resultado em um tempo aceitável ao mesmo.

Uma destas áreas que ataca estes problemas NPs, é conhecida como a Programação por Restrições (PR) Apt [2003]; de Sá [2016], onde os problemas combinatoriais são alguns exemplos das classes NPs Rossi et al. [2006]; Dechter [2003]. A estratégia da PR é *filtrar* e consequentemente *encolher* o espaço de estados para encontrar uma solução válida, ou soluções, ou algumas, ou a solução ótima, ou comprovar a inexistência de uma solução válida. Adicionalmente, outras heurísticas são aplicadas durante a fase de busca e exploração das árvores remanescentes no espaço de estados do problema, detalhes em Apt [2003]; Dechter [2003].

Quanto as soluções de problemas, estes precisam ser modelados e resolvidos por alguma linguagem de programação. Adicionalmente, alguns destes problemas são modelados com estruturas complexas, tais como tabelas, grafos, árvores, etc, aplicados há problemas tais como caixeiro viajante, atribuição de tarefas, escalonamentos, etc Papadimitriou [1994].

Nesta direção, esta a contribuição deste artigo é apresentar uma biblioteca com predicados e funções sobre grafos para linguagem MiniZinc Nethercote et al. [2007]; Stuckey et al. [20xx]; Community [2015]. A motivação desta biblioteca é a inexistência da mesma no Minizinc bem como em *solvers* clássicos da área da pesquisa operacional. Atualmente, esta biblioteca encontra-se

em fase de uso e melhoramentos.

A organização das seções deste artigo se encontram da seguinte forma: na seção 2 é apresenta a linguagem Minizinc e características. Na seção 3 a biblioteca proposta é discutida no tocante a seu uso e problemas. Na seção 4 a biblioteca é utilizada para ilustrar a solução de dois problemas clássicos: verificar se um grafo é completo, e clique$-k$ Papadimitriou [1994]; Sipser [2012]. Na seção 5 o problema problema clique$-k$ é modificado para um problema de otimização, cujo objetivo é encontrar o clique máximo de k. Alguns experimentos, *benchmarks*, são realizados sobre clique máximo (K_{max}) e resultados são discutidos.

2. Fundamentação

Devido a limitação de espaço, a motivação em torno de tópicos tais como: otimização combinatória, modelagem, programação por restrições, etc, é aqui omitida. Contudo, apresenta-se a ferramenta computacional que pode ser usada nestas áreas, no caso, a linguagem MiniZinc.

2.1. A Linguagem MiniZinc

A linguagem MiniZinc surgiu como uma proposta de se criar uma linguagem padrão para modelagem de problemas na área de Otimização Combinatória Nethercote et al. [2007]. A proposta é que um mesmo modelo expresso nesta linguagem possa ser computado por outros *solvers*, de outras linguagens como GECODE, Gurobi, Jacop, etc. Além disto, suas caracterísitcas são a simplicidade[1], a expressividade e a facilidade de implementação Nethercote et al. [2007]. A simplicidade se encontra principalmente na sintaxe, e na expressividade dada por permitir modelar problemas de natureza diversas.

De acordo com Nethercote et al. [2007], antes da linguagem de programação MiniZinc, não existia uma linguagem padrão para a modelagem de problemas de programação por restrições. Praticamente cada *solver* possuia sua própria linguagem de modelagem. Isto dificultava a tarefa de realizar experimentos para comparar o desempenho de diferentes *solvers* para um determinado problema.

Esta linguagem apresenta um conjunto simplificado de sintaxe, e sua semântica segue um padrão ou paradigma de programação, no qual *restrições conjuntivas* declaram um problema a ser computado. Assim a semântica computacional (Φ) de um modelo de problema é definida por:

$$\Phi(M_p) = C_1 \wedge C_2 \wedge C_3 \wedge ... \wedge C_n \quad (1)$$

Um exemplo ilustrativo da sintaxe do código desta linguagem é dado por:

```
var -5 .. 15 : x;

constraint x > 10;          % C1
constraint x <= 15;          % C2
constraint x mod 3 == 0;      % C3

solve satisfy;

output ["VAR x = ", show(x) ];
```

O código é auto-explicativo, tal que a variável x apresenta um domínio entre -5 e 15, e o *modelo do problema*, M_p, é dado pelas seguintes restrições: $x > 10 \wedge x \leq 15 \wedge (x \mod 3 = 0)$. Neste caso, as respostas são 12 e 15. Mais exemplos são encontrados em `https://github.com/claudiosa/CCS/minizinc/`.

[1]Considera-se aqui um usuário com sólidos conhecimentos de matemática.

2.2. Importância e Motivação aos Grafos

As estruturas de grafos são onipresentes na modelagem de problemas difíceis. Os problemas difíceis e em geral *interessantes* apresentam um carácter exponencial na relação as suas entradas. Por exemplo, uma rede social de pessoas, em que se deseja saber quantas pessoas estão conectadas entre sim, após um nível n de profundidade na rede. Este é um caso clássico em que uma sugestão de amigo, pode surgir quando duas ou mais pessoas, tenham em comum, uma relação que não é ainda comum há um amigo comum. Neste caso, surge a sugestão de amizade.

Outro problema clássico, são as redes de suprimentos, abastecimentos, etc. Seja um conjunto de cidades que devem ter suas demandas supridas, seja por uma matéria-prima, energia elétrica, etc, qual procedimento de rota alternativa ocorre em caso de falha de uma das cidades provedoras de insumos?

Enfim, todos estes problemas apresentam uma modelagem, na qual invariavelmente a estrutura de dados é um grafo. Por definição, um grafo é composto por um conjunto de nós ou vértices que modelam estas entidade físicas e um conjunto de arestas que conectam estas entidades. Estas arestas podem representam uma relação entre dois ou mais nós. Por exemplo, a distância entre duas cidades A e B, nós ou vértices, tem um custo de X. Detalhes e definições podem ser encontradas em Diestel [2000]; Netto [1979].

Vale salientar que há muitos outros problemas de grafos os quais não são exponencias (NP-Completos e NP-Difíceis), mas com aplicações reais igualmente importantes, tais como: fluxo máximo de rede, caminho mínimo entre dois pontos, árvore geradora mínima, etc, e que são apenas polinomiais. Estes problemas se encontram na classe de problemas polinomiais, a qual se encontra dentro da classe NP Dasgupta et al. [2006].

3. A Biblioteca de Grafos

O objetivo desta biblioteca é facilitar a codificação e legibilidade de criar modelos em MiniZinc para solucionar problemas que envolvam grafos. Para isto, esta biblioteca oferece um conjunto de restrições relacionadas à teoria dos grafos, tais como: caminhos, ciclos, número cromático, conectividade, entre outras.

A biblioteca é composta de diversas funções e predicados[2]. Apesar de uma função possa conter diversas restrições, para o usuário da biblioteca há uma abstração de que está sendo aplicada uma única restrição. Deste modo, as funções e predicados da biblioteca também são considerados como *restrições* declarativas, disponíveis para linguagem.

Cada uma destas funções é implementada em arquivos separados em código MiniZinc (`.mzn`). Para que a cada modelo desenvolvido não seja necessário incluir um novo arquivo específico, na chamada de uma função, foi criado um único arquivo chamado *graphs.mzn*. Este arquivo, contém um *include* individual para todas as funções ou arquivos que constituem a biblioteca. Isto é feito pela inclusão da seguinte linha de comando no código fonte:

```
include "../lib/graphs.mzn";
```

O conceito de uso e a sintaxe é idêntica a biblioteca de restrições globais provida pelo MiniZinc. Todos os códigos desta biblioteca, suas funcionalidades, e exemplos, estão disponíveis em um repositório público que pode ser acessado pelo link `https://github.com/claudiosa/CCS/` → `minizinc` → `experimental_graph_library` ou de Sá e Filho [2016]. Todos estes códigos estão sujeitos à constantes atualizações, e no futuro, devem integrar o repositório da linguagem.

Como as funções desenvolvidas tratam com grafos, é necessário definir a forma de representação computacional padrão a ser utilizada. A matriz de adjacências foi escolhida como estrutura de representação de um grafo. Tal escolha foi feita pela simplicidade e pelo fato de o MiniZinc não proporcionar uma estrutura de listas e nem vetores de tamanho dinâmico, para construir

[2]No caso da linguagem MiniZinc, predicados são equivalente a funções booleanas. Contudo, a construção apresentam uma sintaxe diferenciada.

listas de adjacências. Assim, devido a ausência de vetores dinâmicos, adota-se a estrutura de matriz de adjacências para representar os grafos.

As funções da biblioteca permitem que os elementos das matrizes de adjacências sejam representados tanto por inteiros ou por booleanos. Quando os elementos da matriz são inteiros, considera-se que o valor 0 (zero) representa a ausência de uma aresta, enquanto que qualquer outro valor não nulo representa a presença de uma aresta, sendo o peso desta igual ao valor da posição na matriz. Quando o peso das arestas não é necessário, os elementos da matriz podem ser booleanos, sendo que o valor `true` indica a presença de uma aresta e o valor `false` indica a ausência.

A biblioteca construída contém diversas funções que simplificam o trabalho de construir modelos em MiniZinc. As funcionalidades oferecidas pela biblioteca são baseadas nas restrições globais disponibilizadas pela biblioteca Grasper of Grasper [2015] da linguagem ECLiPSe of ECLiPSe [2015] e nos conceitos clássicos sobre grafos. Assim, as funções desenvolvidas nesta biblioteca são dadas por:

- `order_fn`: obter a ordem de um grafo;
- `vertex_set`: retornar o conjunto de vértices de um grafo;
- `is_adj_matrix_square`: garantir que uma matriz de adjacências seja quadrada;
- `complete`: validar se um grafo é completo;
- `clique`: encontrar os cliques em um grafo;
- `vertex_colouring`: obter uma coloração de vértices para um grafo;
- `matching`: encontrar emparelhamentos em um grafo;
- `exists_path`: avaliar a existência de um caminho entre dois vértices;
- `path`: encontrar os caminhos entre dois vértices;
- `cycle`: encontrar os ciclos em um grafo;
- `connected`: verificar se um grafo é conexo;
- `strongly_connected`: verificar se um grafo é fortemente conexo;
- `hamiltonian_cycle`: obter um ciclo hamiltoniano.

Devido uma questão de espaço neste artigo, a discussão destas funções é restrita há duas destas. Estas são ilustradas na próxima seção.

4. Ilustrando o Uso da Biblioteca

Das várias funções disponíveis nesta biblioteca, foram escolhidos dois exemplos de uso. Outros exemplos de uso estão disponíveis no diretório `examples` do repositório da biblioteca desenvolvida de Sá e Filho [2016]. Um exemplo inicial é a verificação se um grafo é completo ou não, num segundo exemplo, o problma dos k cliques de um grafo.

4.1. Completo

Uma propriedade de ser obtida de um grafo é verificar se este é ou não completo. Isto é, se todos os pares possíveis de vértices do grafo estão relacionados por uma aresta. O valor do total de arcos é dado por #arcos$= \frac{n(n-1)}{2}$ onde n é o total de vértices.

Para este problema, a biblioteca oferece a funcionalidade do predicado `complete`, o qual permite verificar se um grafo é completo ou não. O código desenvolvido para o predicado `complete` é apresentado na tabela 1 (ver página 6)[3].

Este predicado recebe uma matriz de adjacências representando um grafo e o retorno é um valor `true` ou `false`, indicando se o grafo é completo ou não. A linha 3 apenas cria um conjunto contendo os índices desta matriz, para isto usa-se a função `vertex_set` do MiniZinc. Na linha

[3] As referências de páginas estão sendo feitas de modo redundante, devido as *flutuações* de referências as tabelas e figuras do LaTeX.

Tabela 1: Predicado para validar se um grafo é completo

```
1  predicate complete(array[int,int] of int: graph) =
2  let {
3    set of int: V = vertex_set(graph);
4  }
5  in (
6    is_adj_matrix_square(graph) /\
7    forall(u, v in V, where u < v) (graph[u,v] != 0 /\ graph[v,u] != 0)
8  );
```

6, um predicado é chamado para indicar se a matriz é quadrada ou não. Na linha 7, que garante que para todos os pares de vértices distintos do grafo, exista uma aresta não direcionada que liga estes vértices. Um exemplo simples de utilização deste predicado pode ser visto na tabela 2 (ver página 6). O predicado `complete` foi reusado na construção de outras funções disponíveis na biblioteca de Sá e Filho [2016].

Tabela 2: Exemplo de uso do predicado *complete*

```
1  include "../lib/graphs.mzn";
2
3  int: N;
4  array[1..N, 1..N] of int: adj_mat;
5
6  constraint complete(adj_mat);
7
8  solve satisfy;
```

4.2. Clique−k

Um problema correlato ao grafo completo é encontrar *cliques* deste grafo. Um clique é um subgrafo completo, isto é, um conjunto de vértices de um grafo em que todos estes vértices estão relacionados entre si diretamente. O tamanho de um clique, k é dado pela quantidade de vértices presentes neste sub-grafo completo.

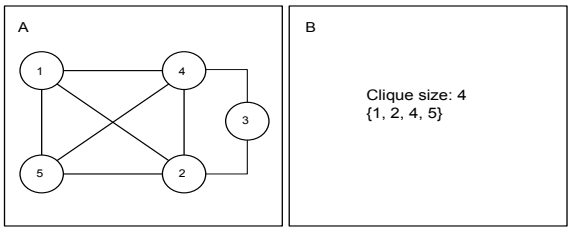

Figura 1: Exemplo de grafo com um clique de tamanho 4

Seja o exemplo do grafo da figura 1 (ver página 6). Na figura 1, há um clique de tamanho 4, sendo que este é formado pelo conjunto de vértices $\{1, 2, 4, 5\}$, visto que todos estes vértices estão conectados entre si.

A biblioteca construída disponibiliza uma função para encontrar cliques. Esta função é chamada de *clique* e funciona de modo semelhante ao predicado *complete*. Esta função é apresentada na tabela 3 (ver página 7).

Tabela 3: Função para encontrar cliques

```
1  function var set of int: clique(array[int,int] of int: graph, var int: size) =
2  let {
3    set of int: V = vertex_set(graph);
4    var set of V: clique_set;
5    array[V] of var bool: bool_set;
6
7    constraint
8              is_adj_matrix_square(graph);
9    constraint
10             assert(dom(size) subset 1..order(graph),
11             "The domain of the clique size must be [1..order(graph)]", true);
12   constraint
13             card(clique_set) == size;
14   constraint
15             link_set_to_booleans(clique_set, bool_set);
16   constraint
17            forall(u,v in V, where u!=v) (
18            (bool_set[u] /\ bool_set[v]) -> (graph[u,v] != 0 /\ graph[v,u] != 0)
19            );
20 }
21 in (
22     clique_set
23    );
```

Assim como no predicado de grafo `complete`, a entrada é dada por uma matriz de adjacências. O valor k de um inteiro indica o tamanho do clique, sendo que este é uma variável de decisão. Pois temos que saber se k pode ser encontrado ou não a partir de um grafo. O retorno da função clique é um conjunto de inteiros, sendo que este contém todos os vértices que fazem parte do clique-k encontrado.

As principais diferenças desta função em relação ao predicado *complete* estão nas linhas 13 e 17. Na linha 13, restringe-se a cardinalidade do conjunto de vértices que fazem parte do clique a ser exatamente igual ao parâmetro informado. A restrição presente na linha 17 faz com que todos os vértices que sejam parte do conjunto do clique-k, estejam conectados entre si, garantindo que o conjunto de vértices escolhido de fato forma um clique. Ou seja, um sub-grafo completo.

Tabela 4: Exemplo de uso da função *clique*

```
1  include "graphs.mzn";
2
3  int: N;
4  array[1..N, 1..N] of int: adj_mat;
5  var set of 1..N: clique_vertexes;
6  var 1..N: clique_size;
7
8  constraint clique_vertexes = clique(adj_mat, clique_size);
9
10 solve maximize clique_size;
11
12 output["Clique size:" ++ show(clique_size) ++ "\n" ++ show(clique_vertexes)];
```

Um exemplo de uso do modelo que utiliza a função *clique* é encontrar o maior clique de um grafo. Este exemplo está na tabela 4 (ver página 7). Como exemplo, seja o grafo da figura 1,

cuja matriz de adjacência é uma entrada apresentada na tabela 2 (ver página 8).

Figura 2: Exemplo de entrada

```
/*
 1---4---
 |\ /|   |
 | X |   3
 |/ \|   |
 5---2---
*/
N=5;
adj_mat = [|
    0, 1, 0, 1, 1,|
    1, 0, 1, 1, 1,|
    0, 1, 0, 1, 0,|
    1, 1, 1, 0, 1,|
    1, 1, 0, 1, 0 |];
```

Para este exemplo, uma das saídas para um clique-3 e um clique-4 é dada na tabela 3 (ver página 8).

Figura 3: Saídas de clique-3 e um clique-4

```
----------
Clique size:3
{1,2,4}
----------
Clique size:4
{1,2,4,5}
----------
```

Lembrando, que o MiniZinc é uma linguagem completa, isto é, apresenta todas as respostas possíveis no espaço de busca. Neste exemplo, apenas **uma** saída está sendo apresentada, para $k = 3$ e $k = 4$.

5. Um Experimento

Nesta seção é explorado o uso da função clique-k nesta biblioteca de Sá e Filho [2016], para um problema NP-difícil: o clique-máximo Papadimitriou [1994]; Sipser [2012]. Neste problema, ocorre uma otimização a qual é alcançada após uma busca exaustiva à todos os resultados de clique-k. O maior k é o K_{max}, ou clique-máximo. Os casos de testes foram retirados de Dharwadker [2016].

Neste sítio é apresentado um algoritmo em tempo polinomial para este problema para o caso de n vértices, com vértices de grau mínimo igual a δ e clique máximo de no mínimo $\frac{n}{n-\delta}$. Ou seja, o grau δ definido para o grafo vai definir uma quantidade mínima de cliques máximos (K_{max}).

As amostras retiradas de Dharwadker [2016] são propositais e representam grafos clássicos da área, bem como os casos difíceis para o clique máximo. Embora seja apresentado um algoritmo polinomial em Dharwadker [2016], este é restrito para certas situações das escolhas de

δ e partir de um número mínimo de cliques máximos que se deseja. Em outras palavras, isto não apresenta elementos suficientes para demonstrar que $P = NP$ ou que $P \neq NP$. Mas, algumas amostras são clássicas da área e relevantes no contexto da teoria do grafos.

O códigos fontes, scripts, amostras, destes experimentos se encontram em `https://github.com/claudiosa/CCS/experiments/clique_article`. O ambiente de programação foi sob um notebook dual-core, velocidade 1.2 GHz, 3 Gb RAM, ambiente Debian-Linux. A versão do MiniZinc foi a 2.02, sob o seu uso *default de* solver: o `mzn-g12fd` Community [2015]; Marriott e Stuckey [2015]. Nenhuma modificação substancial foi feita sobre a função do clique, exceto pela opção em maximizar o valor de k. Isto é, o maior clique existente num dado grafo.

Os experimentos para o estudo do clique máximo tem seus resultados apresentados na tabela 5 (ver página 9). As colunas desta tabela indicam: a quantidade de vértices de cada grafo (N_{vert}), seguido pelo tempo de processamento (T) em segundos, o tamanho máximo do cique encontrado (K_{max}), e sua solução.

Tabela 5: Experimentos com grafos de N vértices

N_{vert}	T (seg.)	K_{max}	Solução
4	0.094	4	$\{1 \ .. \ 4\}$
6	0.115	3	$\{1, 2, 3\}$
8	0.098	3	$\{1, 4, 8\}$
10	0.117	4	$\{1, 4, 8, 9\}$
20	0.1194	10	$\{11 \ .. \ 20\}$
30	0.2193	15	$\{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30\}$
40	0.2428	12	$\{4, 6, 9, 16, 17, 19, 20, 21, 27, 31, 36, 38\}$
50	0.4408	14	$\{11, 12, 13, 14, 17, 20, 23, 25, 26, 29, 32, 35, 38, 39\}$
60	334.7568	20	{3,4,9,10,14,16,21,23,25,29,33, 35,39,40,45,47, 49,54,56,58 }
70	2.1352	14	$\{11, 22, 27, 37, 40, 41, 44, 47, 54, 57, 59, 60, 67, 69\}$
80	3.7672	14	$\{7, 9, 10, 22, 25, 26, 27, 28, 31, 35, 36, 39, 41, 44\}$
90	9.011	19	{27,31,36,37,42,47,51,52,54,56,57,59,61,62,72,78,84, 85,88 }
100	15.2853	17	{4,6,7,12,13,15,16,18,21,22,24,25,27, 30,39,84,86}
150	744.9133	26	{91,94,97,103,106,107,108,109,112,113,115,117, 118,126,127,130,132,133,136,138,139,141,142, 145,147,148}
200	4179.8565	18	{2, 11, 12, 85, 87,96, 101, 106,108,109, 114,115, 119,122, 123,126, 131, 136 }

As amostras com $N = 40, 50, 70, 80, 90, 100, 150$ e 200 foram retirados da parte inicial do grafo 450 do sítio `http://www.dharwadker.org/clique/` ou Dharwadker [2016]. A amostra com um grafo de 450 vértices não executou em tempo aceitável para os testes realizados neste hardware disponível. Uma heurística com melhoramentos deve ser aplicada considerando o valor δ, a estimativa de um valor mínimo de K_{max}, bem como os parâmetros do `search` do MiniZinc. Estas amostras podem ser consideradas como aleatórias, considerando os resultados apresentados na tabela 5 (ver página 9). Por outro lado, de modo proposital, no sentido de que um mesmo grafo foi aumentando gradativamente até o limite de $N = 200$.

O caso de $N = 60$ é uma amostra atípica, conhecida como grafo do complemento de Berge Dharwadker [2016]. Para este grafo se deseja um K_{max} mínimo de valor igual a 20. O de-

talhe é que este grafo apresenta muitos $K_{max} = 20$, precisamente foram encontrados 7200 cliques máximos neste grafo. Como ocorre em buscas completas em otimização, todos os máximos tiveram que serem encontrados pelo *solver*, afim de garantir que o maior clique tem valor 20. Contudo, ressalta-se que o conjunto potencial de soluções é da ordem de C_{60}^{20}, onde C é expressão clássica da análise combinatória C_n^k. Lembrar que respostas em combinatória a ordem dos valores não interessam. Por exemplo, uma resposta igual a $\{a, b, c\}$ é equivalente a $\{a, c, b\}$.

Ao compararmos os resultados da tabela 5 (ver página 9) com os valores encontrador em Dharwadker [2016], a numeração apresentada nos resultados dos vértices, difere de uma unidade 1. Isto deve-se ao fato da implementação feita em MiniZinc o grafo é numerado a partir de 1, enquanto que os resultados apresentados em Dharwadker [2016], os vértices são numerados a partir do 0, valor inicial de vetores na linguagem C++.

Claramente, a natureza exponencial dos resultados para este problema é mantida, mas é inequívoco que a biblioteca proposta é consistente e correta com os resultados, com respostas de tempo aceitáveis. Caso o objetivo deste trabalho fosse exibir desempenho, o que é desejável em casos práticos, algumas heurísticas poderiam ser aplicadas tais como: aumentar a capacidade de processamento da máquina, mudanças no código para casos específicos acelerar a busca, mudanças nos parâmetros do `search`[4] bem como testar outros *solvers* disponíveis ao MiniZinc.

Exemplificando destas idéias, usamos um computador do tipo desktop de 4 núcleos com 4 Gb RAM, 2.4 GHz, para a amostra $N = 60$. Em seguida, foi usado um *solver* que suporta paralelismo o `mzn-gecode`, usando 8 clones[5] (equivalente as *threads* da linguagem Java) nesta computação. Se fez uma comparação desta amostra neste computador com o *solver* `mzn-g12fd`, usado nos experimentos da tabela 5 (ver página 9). O tempo de processamento destes experimentos foram obtidos diretamente via o comando *time* do Linux. Estes resultados são dados por:

```
$time(mzn-gecode -p 8 -a clique_ex.mzn data_clique_60.dzn)
Clique size:20
{1,5,7,11,15,16,20,22,26,30,31,36,37,41,45,46,51,52,56,60}
----------
==========
real    1m23.637s


$time(mzn-g12fd  clique_ex.mzn data_clique_60.dzn)
Clique size:20
{3,4,9,10,14,16,21,23,25,29,33,35,39,40,45,47,49,54,56,58}
----------
==========
real    2m38.079s
```

O conjunto clique encontrado é um valor diferente do apresentado da tabela 5 ver página 9), devido o tipo do *solver*, neste caso foi o `mzn-gecode`. Como já explicado, há muitos cliques com $k = 20$, haja visto o tipo da amostra apresentada. Logo, este é apenas mais um valor possível do conjunto das 7200 soluções existentes para este grafo. Assim, a solução paralela (`1m23s`) para este caso foi quase duas vezes mais rápida que a sequencial (`2m38s`). Contudo, muitos testes deveriam ser feitos, sob diversas variações de parâmetros que a PR permite fazer, afim de validar alguma conclusão específica do processamento paralelo sob o sequencial.

[4] Aqui reside os dois fatores de impacto do paradigma da PR: sequência de *escolha das variáveis* e *seleção de valores* de seus domínios.

[5] Ver parâmetro `-p 8` no exemplo.

6. Conclusões

Os problemas NP-completos e NP-difíceis apresentam complexidades exponenciais, haja visto, que sua natureza combinatorial torná-os de difícil solução. Estes problemas são encontrados com exemplos reais, tais como: planejamento de agendas, escalonamento de tripulações ou motoristas, escalonamento de aeronaves e horários, escalonamento de recursos em geral, desdobramento de proteínas, algoritmos para encontrar um conjunto de amigos comuns entre si (exemplo do clique-k), redes de abastecimento, etc.

Estes problemas apresentam características intrínsecas em sua modelagem, cuja estrutura fundamental em geral são grafos. Os grafos são estruturas de dados gerais e de difícil implementação computacional. Nesta direção, as linguagens e algoritmos que resolvam problemas que envolvam grafos são de extrema importância. Contudo, há algoritmos de grafos que não necessariamente estejam nas classes NP-completo e NP-difícil.

Assim, este artigo apresenta uma biblioteca de grafos para a linguagem MiniZinc Marriott e Stuckey [2015]; Community [2015]. Esta é orientada a modelos matemáticos com elevada grau de abstração e clareza de implementação. Atualmente o MiniZinc tem sido utilizado em áreas tais como a Programação por Restrições (PR), Otimização Combinatória e Pesquisa Operacional. Quanto ao MiniZinc, um de seus destaques é a elegância de sua modelagem, cuja natureza declarativa é complementada por expressões matemáticas, representando as restrições do problema dentro de um domínio de valores.

Assim, este artigo a biblioteca desenvolvida para grafos é ilustrada em dois casos de uso: a verificação se um grafo é completo ou não, construindo um predicado booleano; e uma função de implementação para clique-k. O algoritmo do clique-k foi explorado com alguns *benchmarks*, afim de exibir seu desempenho haja visto que uma otimização foi requerida: encontrar o clique máximo. Este problema é pertence a classe NP-difícil, os quais são os mais difíceis da classe dos NP-completos Papadimitriou [1994]; Sipser [2012].

A estrutura desta biblioteca segue os padrões da linguagem MiniZinc, dispõem de exemplos de uso, etc. Esta encontra-se disponível no sítio da referência: de Sá e Filho [2016].

Como trabalhos futuros e encaminhamentos, tem-se:

- Realizar testes massivos para avaliar seu desempenho e contorno das especificações;
- Aprimorar algumas estruturas internas, como por exemplo: trocar as estruturas *sets* por *arrays*. Embora sets apresente muitas facilidades, a mesma nem sempre é compatível com alguns *solvers*;
- Disponibilizá-la nas futuras distribuições do MiniZinc. Encontra-se em avaliação pela comunidade do MiniZinc;

7. Agradecimentos

A Peter J. Stuckey pelo apoio e críticas e Guido Tack pela ajuda em partes da implementação.

Referências

Apt, K. (2003). *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA. ISBN 0521825830. URL `http://portal.acm.org/citation.cfm?id=1237975\#`.

Community, M. (2015). Sítio do minizinc. Internet. URL `http://www.minizinc.org/`. `http://www.minizinc.org/`, acessada: 2015-10-30.

Dasgupta, S., Papadimitriou, C., e Vazirani, U. (2006). *Algorithms*. McGraw-Hill Education. ISBN 9780073523408.

de Sá, C. C. (2016). Curso de fundamentos de programação por restrições. Mídia eletrônica. URL http://www2.joinville.udesc.br/~coca/index.php/Main/ProgramacaoLogicaRestricao. http://www2.joinville.udesc.br/~coca/index.php/Main/ProgramacaoLogicaRestricao.

de Sá, C. C. e Filho, M. C. (2016). Repositório da biblioteca de grafos para linguagem minizinc. Internet. URL https://github.com/claudiosa/CCS/tree/master/minizinc/experimental_graph_library. Biblioteca de Grafos para Linguagem MiniZinc –https://github.com/claudiosa/CCS/tree/master/minizinc/experimental_graph_library, acessada: 2016-02-20.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Dharwadker, A. (2016). The clique algorithm (sítio dos benchmarks do clique). Internet. URL http://www.dharwadker.org/clique/. The Clique Algorithm –http://www.dharwadker.org/clique/, acessada: 2016-02-20.

Diestel, R. (2000). *Graph Theory*. Electronic library of mathematics. Springer New York. ISBN 9780387989761.

Marriott, K. e Stuckey, P. J. (2015). A minizinc tutorial.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., e Tack, G. (2007). Minizinc: Towards a standard cp modelling language. In Bessiere, C., editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, p. 529–543. Springer. ISBN 978-3-540-74969-1. URL http://dblp.uni-trier.de/db/conf/cp/cp2007.html#NethercoteSBBDT07.

Netto, P. (1979). *Teoria e modelos de grafos*. E. Blücher.

of ECLiPSe, C. (2015). Sítio do eclipse. URL http://eclipseclp.org/. http://eclipseclp.org/.

of Grasper, C. (2015). Sítio do grasper. internet. URL http://eclipseclp.org/doc/bips/lib_public/grasper/. http://eclipseclp.org/doc/bips/lib_public/grasper/.

Papadimitriou, C. M. (1994). *Computational Complexity*. Addison-Wesley, Reading, Massachusetts. ISBN 0201530821.

Rossi, F., Beek, P. V., e Walsh, T., editors (2006). *Handbook of constraint programming*. Elsevier.

Sipser, M. (2012). *Introduction to the Theory of Computation (3rd Edition)*. Cengage Learning, Boston, MA, USA. ISBN 113318779X.

Stuckey, P. J., Becket, R., Br, S., Brown, M., Fischer, J., B, M. G. D. L., e Marriott, K. (20xx). The evolving world of minizinc.