

# Um Algoritmo VNS para o Problema da Máxima Interseção de $k$ Subconjuntos

**José Robertty de Freitas Costa,**

Curso de Engenharia de Computação, Universidade Federal do Ceará  
Campus de Quixadá, Quixadá, Ceará, CEP 63900-000  
robertty@alu.ufc.br

**Fabio Carlos Sousa Dias, Wladimir Araújo Tavares**

Curso de Ciência da Computação, Universidade Federal do Ceará  
Campus de Quixadá, Quixadá, Ceará, CEP 63900-000  
fabiocsd@lia.ufc.br, wladimir@lia.ufc.br

## RESUMO

Neste trabalho, estudamos o problema da máxima interseção de  $k$ -subconjuntos (kMIS). Dado um grafo bipartido  $G(L \cup R, E)$ , sendo  $R$  é o conjunto dos elementos,  $L$  o conjunto dos subconjuntos de  $R$ , e um inteiro  $k$ , o problema (kMIS) consiste em encontrar um subconjunto  $L' \subseteq L$  com  $|L'| = k$  tal que  $\cap(L')$  seja máxima, sendo  $\cap(L') = |\cap_{C \in L'} C|$ . Este problema possui aplicações importantes como, por exemplo, no controle de privacidade de dados de pacientes em hospitais. O algoritmo do estado da arte para o problema é o algoritmo Grasp Reativo apresentado em Bogue et al. [2014]. Neste trabalho, desenvolvemos duas heurísticas gulosas e um algoritmo de busca em vizinhança variável (VNS), com a busca local Descida em Vizinhança Variável (VND) e um algoritmo de segunda ordem para guiar a fase de agitação. Resultados computacionais mostraram que o novo algoritmo supera o algoritmo do estado da arte.

**PALAVRAS CHAVE.** VNS, Problema da Máxima Interseção de  $k$  Subconjuntos, Meta-Heurísticas.

**Área Principal:** Meta-Heurística

## ABSTRACT

In this work, we study the problem of the maximum intersection of  $k$ -subsets (kMIS). Given a bipartite graph  $G(L \cup R, E)$ , where  $R$  is the set of elements,  $L$  the set of subsets of  $R$ , and an integer  $k$ , the problem kMIS consists of finding a subset  $L' \subseteq L$  with  $|L'| = k$  such that  $\cap(L')$  is maximum, where  $\cap(L') = |\cap_{C \in L'} C|$ . This problem has important applications, for instance, patients' data privacy control in hospital. The state-of-the-art algorithm for the problem is the Reactive Grasp algorithm presented in Bogue et al. [2014]. In this work, we developed two greedy heuristics and a Variable Neighborhood Search (VNS) algorithm, with the local search Variable Neighborhood Descent (VND) and a second order algorithm to guide the agitation phase. Computational results showed that the new algorithm outperforms the state-of-the-art algorithm.

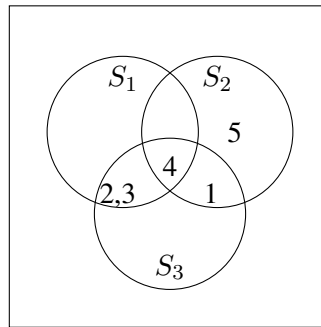
**KEYWORDS.** VNS, Maximum k-Subset Intersection Problem, Metaheuristics

**Main Area:** Metaheuristic

## 1. Introdução

O Problema da Máxima Interseção de  $k$ -subconjuntos (kMIS) pode ser definido da seguinte maneira: Dado uma coleção  $L = \{S_1, S_2, \dots, S_n\}$  de  $n$  subconjuntos de um conjunto  $R = \{e_1, \dots, e_m\}$  de  $m$  elementos e um inteiro  $k$ , temos que encontrar um subconjunto  $L' \subseteq L$  com  $|L'| = k$  tal que  $\cap(L')$  seja máxima, sendo  $\cap(L') = |\cap_{S \in L'} S|$ . O problema pode ser modelado por um grafo bipartido  $(\{1, \dots, n\} \cup R, E)$ , tal que  $\{i, e_j\} \in E$  se e somente se  $e_j \in S_i$ .

De forma ilustrativa suponha  $R = \{1, 2, 3, 4, 5\}$ ,  $L = \{S_1, S_2, S_3\}$ , onde  $S_1 = \{2, 3, 4\}$ ,  $S_2 = \{1, 4, 5\}$  e  $S_3 = \{1, 2, 3, 4\}$ . Para  $k = 1$ , basta encontrar o conjunto  $S_i$  com o maior cardinalidade. No nosso exemplo, a solução seria  $L' = \{S_3\}$ . Para  $k = 2$ , temos que encontrar a intersecção máxima entre dois conjuntos. No nosso exemplo, temos que  $|S_1 \cap S_2| = 1$ ,  $|S_1 \cap S_3| = 3$  e  $|S_2 \cap S_3| = 2$ . Logo, a solução seria  $L' = \{S_1, S_3\}$ . Abaixo está exposto o exemplo através do diagrama de Venn para melhor visualização.



O problema foi apresentado por Vinterbo [2002], dentro de uma aplicação que surge no controle da privacidade de dados de pacientes em hospitais. Seja  $L$  o conjunto de pacientes de um hospital e  $R$  um conjunto de dados dos pacientes. Para garantir a privacidade dos dados dos pacientes, é permitido mostrar algum dado somente se  $k$  pacientes possuam este mesmo dado, de maneira que  $k$  seja suficientemente grande para dificultar a identificação de um paciente. Para isso precisamos encontrar  $k$  pacientes que possuam a maior quantidade possível de dados em comum.

O problema kMIS tem uma estreita relação com o problema *Maximum Edge Biclique* (MEB). O problema MEB pode ser definido da seguinte maneira: dado um grafo bipartido  $G = (L \cup R, E)$ , temos que encontrar uma biclique  $B$  que possua a cardinalidade máxima de arestas. Uma biclique  $C = U_1 \cup U_2$  é um subconjunto de vértices tal que  $U_1 \subseteq L$  e  $U_2 \subseteq R$  tal que todo vértice de  $U_1$  é adjacente a todo vértice de  $U_2$ . O problema kMIS pode ser visto como o problema MEB com a restrição de que  $|U_1| = k$ .

Em Xavier [2012], o problema kMIS é demonstrado ser  $\mathcal{NP}$ -difícil realizando uma redução a partir do MEB, mostrado ser  $\mathcal{NP}$ -difícil por Peeters [2003]. Em Bogue et al. [2013], três formulações de programação linear inteira e uma heurística para kMIS são apresentadas. Duas dessas formulações são obtidas a partir de adaptações do problema MEB apresentadas em Acuña et al. [2011] e uma nova formulação desenvolvida diretamente para o problema kMIS denominada kInter. Resultados computacionais mostraram que a formulação kInter obteve resultados superiores às formulações adaptadas. Em Bogue et al. [2014], um procedimento efetivo de pré-processamento para redução do tamanho da entrada e uma heurística GRASP são apresentados. Além disso, uma formulação de programação inteira usando desigualdades de clique. Resultados computacionais em instâncias obtidas de aplicações reais confirmaram a efetividade dos algoritmos propostos.

Neste trabalho apresentamos duas novas heurísticas gulosas, obtidas estendendo a heurística gulosa apresentada em Bogue et al. [2013], e um algoritmo VNS (Busca em Vizinhança Variável) adicionando um procedimento de intensificação (*path-relinking*) e um processo de sintonização (reativo) encontrados no algoritmo GRASP Reativo, que foi apresentado em Bogue et al. [2014]. O artigo será organizado da seguinte maneira. Na seção 2, apresentamos o algoritmo GRASP Reativo com o procedimento de *path-relinking*. Na seção 3, apresentamos as duas novas heurísticas gulosas

propostas para kMIS. Na seção 4, apresentamos o algoritmo VNS. Na seção 5, apresentamos as instâncias de testes utilizadas nos nossos experimentos computacionais. Na seção 6, discutiremos os resultados computacionais obtidos pelos nossos algoritmos.

## 2. Algoritmo GRASP

Nesta seção, apresentaremos o algoritmo GRASP Reativo desenvolvido por Bogue et al. [2014]. Algumas técnicas desse algoritmo foram utilizadas em nosso trabalho como o procedimento *path-relinking* e o mecanismo de reatividade.

O GRASP é composto por duas fases: uma fase de construção baseada em uma heurística aleatória e uma fase de busca local aplicado na solução obtida na primeira fase. Após a fase de busca local, também é aplicado um procedimento de intensificação, chamado de *path-relinking*, que procura adicionar atributos das melhores soluções encontradas até o momento na solução da busca local na esperança de gerar soluções de boa qualidade.

### 2.1. Fase de Construção

A fase de construção do algoritmo GRASP consiste em construir iterativamente uma solução viável  $L'$  para o problema. Sendo inicialmente  $L' = \emptyset$ , em cada iteração, um subconjunto é escolhido aleatoriamente da chamada Lista Restrita de Candidatos (LRC) para ser adicionado em  $L'$ . Esse processo se repete até que  $|L'| = k$ .

Considerando uma solução parcial  $L'$ , ou seja, com  $|L'| < k$ , a Lista Restrita de Candidatos é composta pelos subconjuntos  $S \in L \setminus L'$  com os maiores custos incrementais  $c(S)$ , onde  $c(S) = |\bigcap_{C \in L' \cup \{S\}} C|$ . O tamanho de LRC depende de um parâmetro  $\alpha_i \in [0, 1]$ , que indica o nível de “ganância” do algoritmo, sendo que  $\alpha_i \in \{\alpha_i, \dots, \alpha_v\}$ . Dado  $\alpha_i$ , LRC é formado pelos  $S \in L \setminus L'$  tal que  $c(S) \in [c^{min} + \alpha_i \times (c^{max} - c^{min}), c^{max}]$ , onde  $c^{max} = \max_{S \in L \setminus L'} c(S)$  e  $c^{min} = \min_{S \in L \setminus L'} c(S)$  representando o maior e o menor custo incremental dos elementos que não estão em  $L'$ . Note que se  $\alpha_i = 0$  então todos os subconjuntos em  $L \setminus L'$  estão em LRC. A medida que  $\alpha_i$  cresce, o tamanho de LRC decresce. Na subseção 2.4, iremos apresentar o método dinâmico que os autores utilizaram para encontrar o melhor valor para  $\alpha_i$ . O Algoritmo 1 sumariza os passos realizados na fase de construção.

---

#### Algoritmo 1: Fase de Construção GRASP

---

```

Entrada:  $\alpha_i, k$ 
Saída:  $L' \subseteq L$ 
1 início
2    $L' \rightarrow \emptyset$ 
3   enquanto  $|L'| < k$  faça
4     Construa LRC usando o valor de  $\alpha$ 
5     Selecione aleatoriamente um subconjunto  $S \in LRC$ 
6      $L' \rightarrow L' \cup \{S\}$ 
7     Atualize os custos incrementais dos elementos em  $L \setminus L'$ 
8   retorna  $L'$ 

```

---

### 2.2. Fase de Busca Local

Na fase de busca local é feita uma busca dentro de uma vizinhança da solução  $L'$  encontrada pela fase de construção. A vizinhança de  $L'$  utilizada pelos autores é obtida removendo um subconjunto  $S$  de  $L'$  e adicionando um subconjunto de  $L \setminus L'$ .

Diversas implementações da busca local são possíveis. Por exemplo, toda a vizinhança poderia ser analisada e a solução de melhor qualidade poderia ser retornada, ou ainda, parar a busca após encontrar a primeira solução com custo melhor que  $L'$ . A estratégia adotada pelos autores foi parar a busca após encontrar a primeira solução com melhor qualidade.

### 2.3. Procedimento de Intensificação

Após a fase da busca local, utiliza-se um procedimento chamado de *Path-relinking* que foi originalmente proposto por Glover [1997] como uma estratégia para incorporar na solução atual atributos das melhores soluções encontradas até o momento. Durante esse procedimento, um conjunto das melhores soluções encontradas pelo Grasp, denotado por *Elite*, é mantido. O procedimento *Path-relinking* constrói um caminho de soluções viáveis conectando a solução  $L'$  encontrada

pela busca local, chamada aqui de solução origem, até uma solução  $L'' \in Elite$ , solução destino, gerando várias soluções intermediárias. Para gerar uma solução intermediária desse caminho é realizado um movimento, onde um movimento consiste em remover um subconjunto da solução intermediária anterior não pertencente ao destino e adicionar outro do destino não pertencente solução intermediária anterior. Em outras palavras, os movimentos consistem de todas as trocas possíveis para transformar a solução origem na solução destino.

O conjunto *Elite* tem um tamanho máximo denotado por  $MAX\_Elite$ . Quando o conjunto *Elite* atinge o seu tamanho máximo é adotado a seguinte política de substituição: a solução no conjunto *Elite* com a menor diferença simétrica em relação a solução encontrada neste procedimento é substituída por essa solução, onde a diferença simétrica é a quantidade de subconjuntos diferentes que existem entre as soluções.

Ribeiro et al.[2002] observaram que o procedimento *Path-relinking* atinge melhores resultados quando a solução origem é melhor que a solução destino, pois a vizinhança da solução origem é mais cuidadosamente explorada. Dado a solução origem  $x_i$  e a solução destino  $x_a$ , em cada iteração do *Path-relinking*, o procedimento computa todos os movimentos possíveis  $m$  escolhendo um que maximiza  $|\cap(x_i \oplus m)|$ , sendo  $x_i \oplus m$  representa a aplicação do movimento  $m$  na solução  $x_i$ . O procedimento encerra quando a solução  $x_i$  é transformada na solução  $x_a$ . O Algoritmo 2 apresenta a sequência de passos realizados pelo procedimento *Path-Relinking*.

---

#### Algoritmo 2: Path-Relinking

---

**Entrada:** a solução encontrada pela busca local  $x_i$ .  
**Saída:** melhor solução encontrada no caminho  $x^*$

```

1  início
2       $x_a$  uma solução do conjunto Elite escolhida aleatoriamente
3      se  $|\cap(x_i)| > |\cap(x_a)|$  então
4           $x_{origem} \leftarrow x_i; x_{destino} \leftarrow x_a; x^* \leftarrow x_i$ 
5      senão
6           $x_{origem} \leftarrow x_a; x_{destino} \leftarrow x_i; x^* \leftarrow x_a$ 
7      enquanto  $\exists$  movimento  $m$  faça
8          Compute o melhor movimento  $m$ 
9           $x_{origem} \rightarrow x_{origem} \oplus m$ 
10         se  $|\cap(x_{origem})| > |\cap(x^*)|$  então
11              $x^* \leftarrow x_{origem}$ 
12     retorna  $x^*$ 

```

---

## 2.4. Grasp Reativo

Uma boa escolha para  $\alpha$ , que define o tamanho de LRC, é um fator importante para conseguir um bom compromisso entre a qualidade da solução e o tempo de execução do algoritmo.

Prais e Ribeiro [2000] desenvolveram um método para que o algoritmo GRASP selecione o valor de  $\alpha$  dentro de um conjunto de valores pré-determinado. Seja  $A = \{\alpha_1, \alpha_2, \dots, \alpha_v\}$  a lista desses valores pré-determinado. No GRASP Reativo, o  $\alpha_i \in A$  é escolhido aleatoriamente com probabilidade  $P_i$ . Essas probabilidades são progressivamente modificadas para favorecer o tamanho da LRC que proporcione encontrar as melhores soluções.

As probabilidades  $P_i$  de selecionar  $\alpha_i \in A$  são inicializadas com  $\frac{1}{v}$ . Durante a execução do algoritmo, o número de soluções construídas com cada valor  $\alpha_i \in A$  é mantido em  $cont(i)$  e os custos da solução correspondentes (após o procedimento de intensificação) são acumulados em  $pontuacao(i)$ . Seja  $Z_{max}$  o custo da melhor solução encontrada até o momento. Para cada bloco de iteração, os valores de  $P_i$  são atualizados de acordo com os passos abaixo:

- Para cada valor  $\alpha_i \in A$ , calcule a média  $media(i) = \frac{pontuacao(i)}{cont(i)}$  e então  $Q_i = \frac{media(i)}{Z_{max}}$
- Calcule  $\sigma = \sum_{i=1}^v Q_i$
- Substitua cada  $P_i$  por  $\frac{Q_i}{\sigma}$  para obter valores em  $[0, 1]$ .

Depois de algumas iterações, o GRASP reativo tende a usar o valor de  $\alpha_i$  que obteve em média as melhores soluções. O Algoritmo 3 apresenta o Grasp Reativo.

---

**Algoritmo 3: Grasp Reativo**

---

**Saída:** melhor solução encontrada no caminho  $S_{max}$

```
1 início
2    $Z_{max} \leftarrow 0$ 
3    $Elite \leftarrow \emptyset$ 
4   para  $i \leftarrow 1$  até  $v$  faça
5     inicialize  $cont(i)$  e  $pontuacao(i)$  com 0
6      $P_i \leftarrow \frac{1}{v}$ 
7     para  $iter \leftarrow 1$  até  $MAX\_ITERACOES$  faça
8       Escolha  $\alpha_i \in A$  com probabilidade  $P_i$ 
9        $S \leftarrow Construcão(\alpha_i)$ 
10       $S \leftarrow BuscaLocal(S)$ 
11       $S \leftarrow PathRelinking(S)$ 
12      se  $|Elite| < MAX\_Elite$  então
13         $Elite \leftarrow Elite \cup S$ 
14      senão
15        Substituição usando a diferença simétrica.
16      se  $|\cap(S)| > Z_{max}$  então
17         $S_{max} \leftarrow S$ 
18         $Z_{max} \leftarrow |\cap(S)|$ 
19       $cont(i) \leftarrow cont(i) + 1$ 
20       $pontuacao(i) \leftarrow pontuacao(i) + |\cap(S)|$ 
21      se  $iter \bmod v = 0 \wedge \forall i, cont(i) \neq 0$  então
22        compute  $media(i)$  e depois  $Q_i$  para cada  $i = 1$  até  $v$ 
23        compute  $\sigma = \sum_{i=1}^v Q_i$  e atualize  $P_i$  para  $\frac{Q_i}{\sigma}$  para cada  $i = 1$  até  $v$ 
24  return  $S_{max}$ 
```

---

### 3. Heurísticas Gulosas

Nesta seção, apresentamos duas novas heurísticas gulosas obtidas estendendo a heurística gulosa apresentada em Bogue et al. [2013].

#### 3.1. Heurística kInter

A heurística gulosa apresentada em Bogue et al. [2013], é uma heurística que a cada iteração seleciona um subconjunto  $S \in L \setminus L'$  tal que  $\cap(L' \cup \{S\})$  é máxima, onde  $L'$  é a solução parcial que está sendo construída. A solução parcial  $L'$  é inicializada com  $\emptyset$ . Observe que na primeira iteração, escolhemos o subconjunto  $S$  com a maior cardinalidade. Essa escolha gulosa é repetida  $k$  vezes, conforme Algoritmo 4. Em nosso trabalho, chamamos essa heurística de *kInter*.

---

**Algoritmo 4: HEURÍSTICA KINTER**

---

**Entrada:**  $L, R, k$   
**Saída:**  $L' \subseteq L$

```
1 início
2    $L' \leftarrow \emptyset$ 
3   enquanto  $|L'| < k$  faça
4     Selecionar um conjunto de  $S \in L \setminus L'$  tal que  $\cap(L' \cup \{S\})$  seja máxima.
5      $L' \leftarrow L' \cup \{S\}$ 
6  retorna  $L'$ 
```

---

#### 3.2. Heurística kInter Estendida

A Heurística *kInter Estendida* consiste em várias execuções da Heurística *kInter* apenas por modificar a solução parcial inicial. Considere inicialmente que o conjunto  $L$  está ordenado de acordo com a cardinalidade dos subconjuntos, ou seja,  $L = \{S_1, S_2, \dots, S_n\}$ , onde  $|S_1| \geq |S_2| \geq |S_3| \geq \dots \geq |S_n|$ . Na  $i$ -ésima execução inicializamos  $L'$  com o subconjunto  $S_i$ . Observe que a primeira solução encontrada será a mesma encontrada pela Heurística *kInter*. A solução com o melhor custo entre todas encontradas será retornada.

Para evitar a construção de soluções piores do que a melhor solução encontrada até o momento, durante a execução descartamos as soluções parciais  $L'$  tal que  $|\cap(L')| \leq |L_{max}|$ , onde  $L_{max}$  é a melhor solução encontrada até o momento.

### 3.3. Heurística kInter Estendida com Path-Relinking

Nesta heurística utilizamos a Heurística *kInter Estendida* juntamente com o procedimento de *Path-Relinking*. A cada iteração o *Path-Relinking* é invocado, ou seja, é construído um caminho conectando a solução viável encontrada pela *kInter Estendida* até uma solução pertencente ao conjunto *Elite*. A melhor solução encontrada no caminho será a solução da iteração. Aqui o conjunto *Elite* é formado pelas soluções de maiores diferença simétrica entre todas as soluções encontradas até o momento. A mesma política de substituição das soluções do conjunto *Elite* utilizada no GRASP será utilizada aqui.

### 4. Algoritmo Busca em Vizinhos Variável

Neste trabalho, desenvolvemos um algoritmo de Busca em Vizinhos Variável (Variable Neighborhood Search - VNS) (Mladenovic and Hansen [1997] Hansen et al. [2008]) para o *kMIS*. O VNS é uma meta-heurística que explora o espaço de soluções através de trocas sistemáticas de estruturas de vizinhança e a aplicação de um método de busca local.

No algoritmo VNS, é definida uma vizinhança para a fase de agitação, cujo objetivo é encontrar uma solução viável para a fase da busca local. Baseado na ideia da busca *q-opt*, amplamente utilizada na literatura para problemas de árvore geradora mínima com restrição de grau (Andrade et al. [2006]), definimos a vizinhança  $N_{VNS}^q$ ,  $1 \leq q \leq q_{max}$  responsável pela agitação do nosso VNS, onde  $q$  subconjuntos são removidos de forma aleatória da solução inicial e substituídos por outros  $q$  subconjuntos também escolhidos de forma aleatória dos subconjuntos não pertencentes a solução inicial. Após isso a busca local é aplicada. Se a solução encontrada pela busca local não melhora a solução atual então incrementamos o valor de  $q$ , ou seja, o VNS passa a explorar vizinhanças gradativamente maiores. Caso contrário, o valor de  $q$  passa a ser 1, ou seja, reiniciamos o nível de agitação.

O pseudocódigo do VNS pode ser visto no Algoritmo 5.

---

#### Algoritmo 5: Busca em Vizinhos Variável - VNS

---

```

Entrada: Solução Viável  $L'_{entrada}$ ,  $q_{max}$ ,  $q_{step}$ ,  $seqs$ 
Saída: Solução Viável  $L'$ 
1  $L' \leftarrow L'_{entrada}$ ;
2 para  $i \leftarrow 1$  até  $seqs$  faça
3    $q \leftarrow 1$ ;
4   enquanto  $q \leq q_{max}$  faça
5     Gere solução  $L'_{agitacao} \in N_{VNS}^q(L')$ ;
6      $\bar{L}' \leftarrow VND(L'_{agitacao})$ 
7     se  $|\bar{L}'| > |L'|$  então
8        $L' \leftarrow \bar{L}'$ ;
9        $q \leftarrow 1$ ;
10    senão
11       $q \leftarrow q + q_{step}$ ;
12 retorna  $L'$ ;

```

---

#### 4.1. Descida em Vizinhos Variável

O método Descida em Vizinhos Variável (Variable Neighborhood Descent - VND) é um método de busca local cuja sistemática de funcionamento é a mudança de estrutura de vizinhança. Partindo de uma solução inicial, o VND realiza a busca por uma solução de melhor qualidade dentro de uma estrutura de vizinhança inicial. Caso não encontre nenhuma solução de melhor qualidade, ocorre uma mudança na estrutura da vizinhança. Isso se repete até que todas as estruturas de vizinhanças pré-definidas sejam utilizadas, o VND termina sua execução, ou até o momento que uma melhora ocorre, e então ele retorna para a estrutura de vizinhança inicial.

A estrutura de vizinhança utilizada em nosso trabalho foi: Dado uma solução viável  $L'$  e um inteiro  $t$ , a  $vizinhanca(L', t)$  são todas as soluções viáveis que possuem até  $t$  subconjuntos diferentes de  $L'$ .

Observem que podemos ter diversas implementações da busca dentro dessa *vizinhanca*. Podemos percorrer toda a vizinhança e retornar a solução de melhor qualidade, ou podemos realizar

a buscar dentro de um subconjunto dessa vizinhança. Além de podermos parar a busca após encontramos a primeira solução de melhor qualidade. Nesse trabalho optamos por uma implementação que realiza a busca dentro de um subconjunto da *vizinhanca* sem parar a busca quando uma solução de melhor qualidade é encontrada.

Em nossa implementação da *vizinhanca* utilizamos a ideia da heurística *kInter* apresentado na Seção 3. A cada iteração, selecionamos aleatoriamente um subconjunto para sair da solução inicial, gerando uma solução parcial. Selecionamos o subconjunto não pertencente a solução que gera a maior interseção com a solução parcial. A troca será efetuada somente se ela gera melhoria. Esse processo se repete  $t_{max}$  vezes, sendo permitido que subconjuntos que foram removidos em iterações anteriores sejam adicionados nas iterações posteriores e evitando que subconjuntos adicionados sejam removidos posteriormente. Na nossa implementação, consideramos  $t_{max} = 3$ .

O pseudocódigo do VND pode ser visto no Algoritmo 6:

---

**Algoritmo 6:** Descida em Vizinhança Variável - VND

---

**Entrada:** Solução Viável  $L'_{entrada}$   
**Saída:** Solução Viável  $L'$

```

1  $L' \leftarrow L'_{entrada};$ 
2  $t \leftarrow 1;$ 
3 enquanto  $t \leq t_{max}$  faça
4    $\overline{L'} \leftarrow vizinhanca(L', t);$ 
5   se  $|\overline{L'}| > |L'|$  então
6      $L' \leftarrow \overline{L'};$ 
7      $t \leftarrow 1;$ 
8   senão
9      $t \leftarrow t + 1;$ 
10 retorna  $L';$ 
```

---

## 4.2. VNS Reativo

Normalmente, no algoritmo tradicional do VNS, são inseridos outros procedimentos que visam melhorar seu desempenho. Nesta seção, apresentaremos alguns desses procedimentos que inserimos ao nosso VNS, alguns já amplamente utilizados na literatura, outros apresentados aqui. Dentre esses procedimentos, iremos utilizar o algoritmo da fase de construção do Grasp Reativo, por isso denominamos essa versão de VNS Reativo.

### 4.2.1. Alteração do $q\_step$

Desenvolvemos uma nova forma para o aumento da distância das vizinhanças na fase de agitação, que corresponde ao incremento da variável  $q$  no algoritmo 5, caso uma solução melhor qualidade não seja encontrada. Na literatura, normalmente se utiliza um valor fixo para o  $q\_step$ , escolhido depois de testes computacionais. Em nosso trabalho, o valor do  $q\_step$  é inicializado com 1 e será alterado depois de certa quantidades de iterações sem melhora na solução, denotada por  $qtd\_sem\_melhora$ . Quando  $qtd\_sem\_melhora$  supera  $\beta * q\_max$ , então o  $q\_step$  é incrementado por 1, onde  $0 < \beta \leq 1$  representa a proporção que desejamos manter o valor do  $q\_step$  inalterado sem que tenha ocorrido melhora. Após alguns testes computacionais, escolhemos o valor  $\beta = 0.4$ . O contador  $qtd\_sem\_melhora$  é inicializado com 0 na linha 4 do Algoritmo 5 e o Código 7 substitui as linha de 5-9 do Algoritmo 5.

Após testes computacionais verificamos que essa forma de atualização, além de permitir que o VNS encontre soluções de melhor qualidade, reduz o tempo computacional quando comparado com a forma tradicional.

### 4.2.2. Algoritmo de Segunda Ordem

Na fase de agitação do VNS, adicionamos o chamado Algoritmo de Segunda Ordem (SO) (Karnaugh [1976]), que visa guiar a fase de agitação do VNS.

De uma forma geral, o SO gera uma sequência de problemas restritos, onde cada problema restrito atual se relaciona com o anterior. Ao problema restrito atual  $T$  é adicionado uma nova

restrição que gera um novo problema restrito  $Q$ . Com isso, o conjunto de soluções viáveis de  $Q$  estará contido no conjunto de soluções viáveis de  $T$ .

Em nosso trabalho, uma restrição que será adicionada será a fixação de um subconjunto ao problema, ou seja, o problema restrito é definido como: encontrar a máxima intersecção de  $k$  subconjuntos com  $t < k$  subconjuntos já estão fixados.

Logo, na fase de agitação iremos remover  $q$  subconjuntos da solução candidata, gerando com isso um problema restrito inicial. O SO será utilizado para gerar uma solução viável, ou seja, selecionar  $q$  subconjuntos. Para isso, iremos utilizar a estratégia da fase de construção do Algoritmo GRASP Reativo, apresentado na Seção 2.1. Nossa ideia é que o problema restrito inicial seja passado como parâmetro ao algoritmo da fase de construção do Grasp Reativo, onde a cada iteração ele irá selecionar um novo subconjunto para entrar na solução, gerando assim um novo problema restrito, até que tenhamos uma solução viável para o problema.

---

**Algoritmo 7:** Atualização do  $q\_step$

---

```

1 se  $|\overline{L'}| > |L'|$  então
2    $L' \leftarrow \overline{L'}$ 
3    $q \leftarrow 1$ 
4    $qtd\_sem\_melhora \leftarrow 0$ 
5 senão
6    $qtd\_sem\_melhora \leftarrow qtd\_sem\_melhora + 1$ 
7   se  $qtd\_sem\_melhora \geq \beta * q\_max$  então
8      $q\_step \leftarrow q\_step + 1$ 
9      $qtd\_sem\_melhora \leftarrow 0$ 
10   $q \leftarrow q + q\_step$ 

```

---

#### 4.2.3. Aceitação da Solução

Normalmente a solução encontrada pela busca local somente será aceita se seu custo for melhor que da melhor solução encontrada. Em Hansen et al. [2000] é apresentado um novo critério de aceitação da solução retornada pela busca local no VNS, chamado de Skewed VNS. A ideia é a aceitação de uma solução que não melhore a solução corrente na esperança que na vizinhança dessa solução possa conter soluções de melhor qualidade. Na literatura existe diversos critérios de aceitação da solução, normalmente critérios que utilizam os custos e uma função que retorna uma diferença de estrutura entre as soluções. A ideia desses critérios é a aceitação de soluções piores de acordo com uma diferença de estrutura entre as soluções, ou seja, aceitaremos uma solução de pior qualidade se essa tiver estruturas bem distintas. Quanto mais diferente for as duas soluções, mais tolerante seremos com relação a qualidade da solução.

Para isso iremos fazer distinção entre a melhor solução encontrada pelo VNS, chamada aqui de  $L^*$ , e a solução  $L'$  que será aplicada a fase de agitação. Apresentamos a seguir os critérios de aceitação que definimos em nosso trabalho.

Para o cálculo da diferença de estrutura entre duas soluções viáveis, iremos utilizar a diferença simétrica de Bogue et al. [2014], ou seja, é a quantidade de subconjuntos diferentes que existem entre as soluções.

No Algoritmo 8 apresentamos como é feita essa aceitação já incluindo o Algoritmo 7. Na linha 1 do Algoritmo 8, quando a solução  $\overline{L'}$  retornada pela busca local for melhor que  $L^*$ , atualizamos tanto  $L^*$  como  $L'$ . Quando  $\overline{L'}$  não for melhor que  $L^*$ , mas for melhor que  $L'$ , atualizamos  $L'$ .

Repare na condição da linha 11 que não permitimos que uma solução de pior qualidade seja escolhida para atualizar a solução  $L'$ , mas apenas soluções de mesma qualidade e que a diferença simétrica (função  $\delta$ ) seja maior que um erro  $\epsilon$  definido previamente. Optamos por não aceitar soluções de pior qualidade, como normalmente ocorre no Skewed VNS, mas apenas de mesmo valor, por que observamos que o GAP entre os ótimos conhecidos e as soluções encontradas mesmo pela heurística gulosa kInter, é baixo, logo temos um pequeno intervalo entre os custos das soluções encontradas durante o VNS e a melhor solução  $L^*$ . Logo, se consideramos na condição

da linha 11 como estritamente menor, teríamos poucas aceitações, o que tornaria o Skewed VNS inútil. Com essa escolha, permitimos que o VNS chegue em vizinhanças que não seriam exploradas. Isso foi provado pelos testes computacionais. Observamos também que a utilização da condição de menor ou igual foi indiferente, por isso optamos pela igualdade.

A tolerância  $\epsilon$  leva em consideração o valor do  $q_{max}$ ,  $q$  e  $k$ . A função *get\_erro\_quartil* recebe como parâmetro os valores  $q_{max}$  e  $q$  e calcula em qual quartil de  $q_{max}$  está  $q$ . Dependendo do quartil a função retorna 0.2, 0.3, 0.4, 0.5 para o quartil 1, 2, 3 e 4, respectivamente. Esse valor que a função retorna representa a proporção do valor de  $k$  que queremos permitir que a solução de mesma qualidade seja aceita. Assim, a tolerância será uma fração de  $k$ , uma vez que a diferença simétrica ( $\delta$ ) entre duas soluções é no máximo  $k$ . Com isso, a medida que o valor de  $q$  aumenta, aumentamos o erro de aceitação de soluções de mesma qualidade, ou seja, só iremos aceitar soluções de mesma qualidade se elas tiverem maior diferença simétrica.

---

**Algoritmo 8:** Aceitação da Solução

---

```

1 se  $|\overline{L'}| > |L^*|$  então
2    $L^* \leftarrow \overline{L'}$ ;
3    $L' \leftarrow \overline{L'}$ ;
4    $q \leftarrow 1$ ;
5    $qtd\_sem\_melhora \leftarrow 0$ 
6 senão
7   se  $|\overline{L'}| > |L'|$  então
8      $L' \leftarrow \overline{L'}$ ;
9   senão
10     $\epsilon = k * get\_erro\_quartil(q\_max, q)$ ;
11    se  $|\overline{L'}| = |L'|$  e  $\delta(L', \overline{L'}) > \epsilon$  então
12       $T \leftarrow \overline{L'}$ ;
13     $qtd\_sem\_melhora \leftarrow qtd\_sem\_melhora + 1$ 
14    se  $qtd\_sem\_melhora \geq \beta * q\_max$  então
15       $q\_step \leftarrow q\_step + 1$ 
16       $qtd\_sem\_melhora \leftarrow 0$ 
17     $q \leftarrow q + q\_step$ 
```

---

## 5. Instâncias de Teste

Neste trabalho foram geradas instâncias de teste, conforme proposto por Bogue et al. [2013]. Também foi utilizado o modelo de grafo descrito em Erdos and Renyi [1960], e para gerar classes de instâncias, que diferenciam-se entre si pela densidade do grafo e o valor de  $k$ , foi considerado a classificação dessas propriedades em três categorias: baixa, média e alta, conforme Tabela 1.

Assim como em Bogue et al. [2013], geramos 9 classes ( $C_1, \dots, C_9$ ), diferenciando-se pela sua densidade e  $k$ , de acordo com a classificação da Tabela 1. As três primeiras classes possuem baixa densidade, enquanto as três seguintes densidade média e as três últimas alta densidade. No entanto,  $C_1$ ,  $C_4$  e  $C_7$  possuem  $k$  baixo, já  $C_2$ ,  $C_5$  e  $C_8$   $k$  médio e  $C_3$ ,  $C_6$  e  $C_9$   $k$  alto.

| Classificação | densidade                       | $k$                             |
|---------------|---------------------------------|---------------------------------|
| Baixa         | $0.1 E  \leq \dots \leq 0.3 E $ | $0.1 L  \leq \dots \leq 0.3 L $ |
| Média         | $0.4 E  \leq \dots \leq 0.6 E $ | $0.4 L  \leq \dots \leq 0.6 L $ |
| Alta          | $0.7 E  \leq \dots \leq 0.9 E $ | $0.7 L  \leq \dots \leq 0.9 L $ |

Tabela 1: Classificação das propriedades das instâncias

Geramos 3 tipos de instâncias, o primeiro tipo formado por grafos bipartidos balanceados com cada lado da partição contendo 60, 80, 100, 140, 180, 200, 240, 280 e 300 vértices. Os dois outros tipos são formados por grafos não balanceados. No segundo tipo, temos que  $|R| = 0.8 * |L|$  com  $|L| \in \{60, 80, 100, 140, 180, 200, 240, 280, 300\}$ . No terceiro tipo temos que  $|L| = 0.8 * |R|$  com  $|R| \in \{60, 80, 100, 140, 180, 200, 240, 280, 300\}$ . Iremos identificar os tipos 1, 2 e 3 por  $|L| = |R|$ ,  $|L| > |R|$  e  $|L| < |R|$ , respectivamente.

Em todas as instâncias foi aplicado o pré-processamento descrito em Bogue et al. [2013], que remove os elementos de  $R$  da instância que não estão em pelo menos  $k$  subconjuntos.

Destacamos que não conseguimos gerar instâncias da  $C_3$  viáveis, assim como para algumas instâncias de outras classes.

Todas as instâncias, assim como os códigos fontes desse trabalho encontram-se no repositório do GitHub no link: <https://github.com/fabiocsd/kMIS>.

## 6. Resultados Computacionais

Nesta seção iremos mostrar os resultados dos testes computacionais realizados neste trabalho.

Em todas as implementações realizadas neste trabalho utilizamos vetores de bits para representar a adjacência de um elemento  $S \in L$ . Utilizando essa representação podemos executar algumas operações realizando o paralelismo em nível de bits. O paralelismo de bits é uma forma de paralelismo alcançada quando representamos os dados em vetores de bits de tamanho  $w$ , onde  $w$  o tamanho da palavra do computador (por exemplo, 32 ou 64 bits). Operações específicas, como a interseção, podem ser executadas para um vetor de bits de tamanho  $w$  usando uma única instrução do processador.

Constatamos após testes computacionais que o uso do Bit Paralelismo torna o algoritmo bastante eficiente. O tempo chega a ser 20 vezes maior quando o Bit Paralelismo não está presente e aumenta de forma proporcional ao tamanho da instância.

Em Bogue et al. [2014], os autores não relatam a utilização de paralelismo de bits, mesmo assim, em nossa própria implementação do Grasp Reativo, utilizamos tal estrutura, pois após testes computacionais com o Grasp Reativo sem o paralelismo de bits, o tempo se tornou proibitivo para instâncias de tamanho grande.

Como não tivemos acesso a implementação original do Grasp Reativo (Bogue et al. [2014]), fizemos uma nova implementação do algoritmo.

Os parâmetros utilizados no VNS Reativo foram escolhidos após alguns testes computacionais. O  $seqs = 500$ , o  $q_{max}$  será igual a  $|L| - k - 4$  se  $k \geq |L|/2$ , ou  $q_{max} = k - 4$  se  $k \leq |L|/2$  e  $k + 4 \geq |L|/2$ . Essa forma de escolha do valor de  $q_{max}$  foi para garantir que a fase de agitação sempre encontre uma solução viável. A solução inicial para o VNS Reativo é a solução viável encontrada pela Heurística Gulosa *kInter Estendida com Path-Reliking* apresentada na Seção 3.

Todos os testes computacionais neste trabalho foram implementadas na linguagem C++ e executadas usando o sistema operacional Linux em um computador Intel Xeon(R), com 3.30 GHz e 8 GBytes de memória RAM.

Resolvemos primeiramente implementar a formulação apresentada em Bogue et al. [2013] com essas novas instâncias, com o fim de avaliarmos o nível de dificuldade das mesmas. Utilizamos o Solver Matemático CPLEX na versão 12.6.1, através da biblioteca Concert Technology. O tempo máximo de 1800 segundos (30 minutos) de processamento, com apenas uma única thread. Desativamos todas as opções de pré-processamento, heurísticas e geração de cortes do CPLEX, exceto a geração de corte de Cliques, conforme relatos feitos em Bogue et al. [2014].

De uma forma geral, a medida que passamos da classe  $C_1$  para a classe  $C_9$ , percebemos que o Solver Cplex consegue resolver mais instâncias, chegando a instância de tamanho 200 para o grupo  $|L| = |R|$  e a instância de tamanho 240 para os demais grupos, isso para a classe  $C_9$ . A classe  $C_1$  parece ser a classe de instâncias de maior dificuldade, já que o CPLEX conseguiu resolver apenas as 3 menores. As demais instâncias não foi possível encontrar a solução ótima, na maioria dos casos a execução termina devido a falta de memória.

Executamos as duas meta-heurísticas 10 vezes, para cada instância, computando a média das soluções e do tempo de processamento.

O cálculo do GAP será feito por  $(BUB - UB)/BUB$  onde  $BUB$  é o valor da melhor média de soluções encontrada pelas duas meta-heurísticas ou o valor da solução ótima, e  $UB$  o valor da média das soluções da meta-heurística. Além disso, iremos computar quantos vezes  $UB = BUB$  para algoritmo.

Na Tabela 2 realizamos uma comparação por grupo de instâncias e na Tabela 3 por classes. Em ambas tabelas, inserimos o número de vezes que o algoritmo encontra *BUB* (coluna *QTD*), a média e o desvio padrão dos GAP's. Na coluna *QTD* o valor estará em negrito se seu valor for maior ou igual a do outro algoritmo.

Quando comparamos por grupo de instâncias, observamos que no geral o VNS Reativo se mostra superior ao GRASP Reativo em todos os grupos, seja pela quantidade de melhores soluções encontradas na média, seja pelas suas qualidades, como nos mostra a média e o desvio padrão dos GAP's, sempre inferior, chegando a cerca de 57% menor, para o grupo  $|L| = |R|$ .

| Grupo       | Grasp Reativo |         |               | VNS Reativo |         |               |
|-------------|---------------|---------|---------------|-------------|---------|---------------|
|             | Qtd           | Média   | Desvio Padrão | Qtd         | Média   | Desvio Padrão |
| $ L  =  R $ | 55            | 0,02860 | 0,08735       | <b>72</b>   | 0,01641 | 0,07101       |
| $ L  >  R $ | 52            | 0,02604 | 0,06617       | <b>63</b>   | 0,01147 | 0,03826       |
| $ L  <  R $ | 50            | 0,01645 | 0,04587       | <b>74</b>   | 0,00915 | 0,04403       |

Tabela 2: Resultados por Grupo de Instâncias

Quando comparamos por classes, observamos que apenas nas classes  $C_6$  e  $C_9$  que o GRASP Reativo se mostra ligeiramente melhor que o VNS Reativo. Na  $C_2$  houve empate e nas demais classes, o VNS Reativo é bastante superior, destacando as classes  $C_4$  e  $C_5$ , onde ele se mostra quase duas vezes melhor.

| Classes | Grasp Reativo |         |               | VNS Reativo |         |               |
|---------|---------------|---------|---------------|-------------|---------|---------------|
|         | Qtd           | Média   | Desvio Padrão | Qtd         | Média   | Desvio Padrão |
| C1      | 22            | 0.06333 | 0.11307       | <b>26</b>   | 0.02889 | 0.07605       |
| C2      | <b>29</b>     | 0.01667 | 0.09129       | <b>29</b>   | 0.01667 | 0.09129       |
| C4      | 11            | 0.01937 | 0.02925       | <b>26</b>   | 0.00418 | 0.01454       |
| C5      | 14            | 0.03376 | 0.05053       | <b>25</b>   | 0.02032 | 0.04891       |
| C6      | <b>26</b>     | 0.02381 | 0.09849       | 25          | 0.02048 | 0.06908       |
| C7      | 20            | 0.00902 | 0.02048       | <b>29</b>   | 0.00002 | 0.00012       |
| C8      | 19            | 0.01816 | 0.03399       | <b>25</b>   | 0.00146 | 0.00353       |
| C9      | <b>28</b>     | 0.00512 | 0.02450       | 27          | 0.00703 | 0.02730       |

Tabela 3: Resultados por Classes de todos os grupos

Na Tabela 4 estão as médias dos tempos computacionais por grupo de instâncias. Percebemos que o VNS Reativo foi muito mais rápido em todos os grupos, chegando a cerca de 80% mais rápido no grupo  $|L| < |R|$ .

| Grupo       | Grasp Reativo | VNS Reativo |
|-------------|---------------|-------------|
| $ L  =  R $ | 30,538        | 17,614      |
| $ L  >  R $ | 83,573        | 37,334      |
| $ L  <  R $ | 25,878        | 4,880       |

Tabela 4: Média dos Tempos por Grupo

## 7. Conclusão e Perspectivas

Neste trabalho apresentamos uma meta-heurística do tipo VNS, que denominamos de VNS Reativo, e duas heurísticas gulosas para o problema da máxima interseção de  $k$  subconjuntos. Os resultados computacionais foram comparados com o GRASP Reativo proposto por Bogue et al. [2014].

Os testes computacionais demonstraram que a nova forma de atualização do aumento da distância das vizinhanças na fase de agitação do VNS, além de permitir que o VNS encontre soluções de melhor qualidade, reduz o tempo computacional. Da mesma forma, ao incorporarmos ao VNS a fase de construção do GRASP Reativo, como um Algoritmo de Segunda Ordem, e juntamente com o Skewed VNS, guiou o VNS Reativo a encontrar vizinhanças com bons ótimos locais, como ficou demonstrado pelos excelentes resultados computacionais. Por fim, o uso do Bit Paralelismo demonstrou ser uma estrutura bastante eficiente nos algoritmos aplicados ao *kMIS*.

Os resultados computacionais do VNS Reativo se mostraram de alta qualidade, superando os resultados do GRASP Reativo, seja quando comparamos os custos das soluções encontradas quanto o tempo computacional.

Como trabalho futuro podemos alterar a forma de aceitação da solução, mais precisamente, que a tolerância  $\epsilon$  leve em consideração também a quantidade de iterações sem encontrar uma solução de melhor qualidade, algo parecido como fizemos na atualização dinâmica de  $q\_step$ . Podemos usar os resultados do nosso VNS nos algoritmos exatos de Bogue et al. [2014], além do desenvolvimento de novas formulações de programação inteira.

### Agradecimentos

Nós queremos agradecer aos revisores pelas importantes e uteis sugestões que certamente ajudaram a melhorar o artigo.

### Referências

- Acuña, V., Ferreira, C. E., Freire, A. S., and Moreno, E.** (2011). Solving the maximum edge biclique packing problem on unbalanced bipartite graphs. *Discrete Applied Mathematics*, 164:2–12. (in press).
- Andrade, R. C., Lucena, A., and Maculan, N.** (2006). Using lagrangian dual information to generate degree constrained spanning trees. *Discrete Appl. Math.*, 154(5):703–717.
- Bogue, E. T., de Souza, C. C., Xavier, E. C., and Freire, A. S.** (2013). O problema da máxima interseção de k-subconjuntos. In *Anais do XLV SBPO*, pages 2416–2425, Natal. Sobrapo.
- Bogue, E. T., de Souza, C. C., Xavier, E. C., and Freire, A. S.** (2014). An integer programming formulation for the maximum k-subset intersection problem. In *Combinatorial Optimization*, pages 87–99, Cham. Springer International Publishing.
- Erdos, P. and Renyi, A.** (1960). On the evolution of random graphs. *Publ. Math. Inst. Hungary. Acad. Sci.*, 5:17–61.
- Glover, F.** (1997). *Tabu Search and Adaptive Memory Programming — Advances, Applications and Challenges*, pages 1–75. Springer US, Boston, MA.
- Hansen, P., Jaumard, B., Mladenovic, N., and Pereira, A.** (2000). Variable neighborhood search for weight satisfiability problem. *Les Cahiers du GERARD - G-2000-62*.
- Hansen, P., Mladenović, N., and Moreno Pérez, J. A.** (2008). Variable neighbourhood search: methods and applications. *4OR*, 6(4):319–360.
- Karnaugh, M.** (1976). A new class of algorithms for multipoint network optimization. *IEEE Transactions on Communications*, 24(5):500–505.
- Mladenovic, N. and Hansen, P.** (1997). Variable neighborhood search. *Computers Operations Research*, 24(11):1097 – 1100.
- Peeters, R.** (2003). The maximum edge biclique problem is np-complete. *Discrete Applied Mathematics*, 131(3):651–654.
- Prais, M. and Ribeiro, C. C.** (2000). Reactive grasp: An application to a matrix decomposition problem in tdma traffic assignment. *INFORMS Journal on Computing*, 12(3):164–176.
- Ribeiro, C. C., Uchoa, E., and Werneck, R. F.** (2002). A hybrid grasp with perturbations for the steiner problem in graphs. *INFORMS Journal on Computing*, 14(3):228–246.
- Vinterbo, S.** (2002). A note on the hardness of the k-ambiguity problem. Technical report, Harvard Medical School, Boston, MA, USA.
- Xavier, E. C.** (2012). A note on a maximum k-subset intersection problem. *Information Processing Letters*, 112(12):471–472.