

Algoritmo ILS-VND para o Problema da Interseção Máxima de k -Subconjuntos

Hélder Silva Ferreira

Instituto de Computação, Universidade Federal de Alagoas
Av. Lourival Melo Mota, Maceió, 57072-900, Alagoas, Brasil
hsfl@ic.ufal.br

Rian G. S. Pinheiro

Instituto de Computação, Universidade Federal de Alagoas
Av. Lourival Melo Mota, Maceió, 57072-900, Alagoas, Brasil
rian@ic.ufal.br

RESUMO

Dada uma coleção L de n subconjuntos de um conjunto R de elementos, o Problema da Interseção Máxima de k -Subconjuntos (k MIS) consiste em encontrar $L' \subseteq L$ com $|L'| = k$ tal que a interseção dos subconjuntos em L' seja máxima. Este trabalho propõe uma meta-heurística *Iterated Local Search* (ILS) para o k MIS. Ela utiliza duas estruturas de vizinhança, exploradas via uma estratégia *Variable Neighborhood Descent* (VND) com uso inovador de estruturas de dados para acelerar a fase de busca local, melhorando, assim, a performance. Os testes computacionais comprovam a superioridade desta proposta em relação aos algoritmos da literatura, encontrando em média soluções de qualidade superior às do estado da arte.

PALAVRAS CHAVE. k MIS, ILS, Meta-heurísticas.

Tópicos (Meta-heurísticas, Otimização Combinatória)

ABSTRACT

Given a collection L of n subsets of a finite set of elements R , the Maximum Intersection of k -Subsets problem (k MIS) consists of finding $L' \subseteq L$ with $|L'| = k$ such that the intersection of the subsets in L' is maximum. This work proposes an *Iterated Local Search* (ILS) metaheuristic to k MIS. The proposed metaheuristic relies on two neighborhood structures, which are explored using a *Variable Neighborhood Descent* (VND) strategy and makes innovative use of data structures to speed up the local search phase and thus improve the performance. Computational tests prove the superiority of this proposal over algorithms in the literature, finding on average solutions of higher quality than the state of the art.

KEYWORDS. k MIS, ILS, Metaheuristics

Paper topics (Metaheuristics, Combinatorial Optimization)

1. Introdução

O problema da Interseção Máxima de k -Subconjuntos (k MIS) pode ser definido da seguinte forma: dada uma coleção $L = \{S_1, \dots, S_n\}$ de n subconjuntos de um conjunto finito de elementos $R = \{e_1, \dots, e_m\}$, e um inteiro positivo k , o objetivo é encontrar $L' \subseteq L$ com $|L'| = k$ tal que os conjuntos de L' tenham o número máximo de elementos em comum, ou seja, $|\bigcap_{S \in L'} S|$ seja máximo. O problema pode ser modelado como um grafo bipartido $G = (L \cup R, E)$ no qual existe uma aresta (S_i, e_j) se e somente se $e_j \in S_i$.

Por exemplo, a Figura 1 ilustra uma instância do k MIS tal que $R = \{e_1, e_2, e_3, e_4, e_5\}$ e $L = \{S_1, S_2, S_3, S_4\}$, com $S_1 = \{e_1, e_2, e_3\}$, $S_2 = \{e_1, e_2, e_3, e_5\}$, $S_3 = \{e_2, e_5\}$ e $S_4 = \{e_3, e_4\}$. Para $k = 2$, deve-se encontrar uma combinação de dois subconjuntos que gere a interseção máxima. $L' = \{S_1, S_3\}$, com $|S_1 \cap S_3| = 1$, é uma possível solução. No entanto, $L' = \{S_1, S_2\}$, com $|S_1 \cap S_2| = 3$, é melhor, e, para a instância, é uma solução ótima.

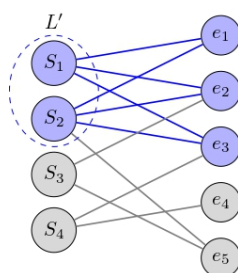


Figura 1: Exemplo de instância do k -MIS.

Para $k = 2$, a interseção máxima de k subconjuntos será obtida por S_1 e S_2 , com cardinalidade 3

O interesse pelo k MIS surgiu em Vinterbo [2002], em que se estuda a aplicação do problema no controle de privacidade de dados. No contexto de dados hospitalares, L é o conjunto dos pacientes de um hospital e R , o conjunto dos dados dos pacientes. Para divulgar a quantidade máxima de dados sem violar a privacidade dos pacientes, um dado só pode ser divulgado se no mínimo k pacientes o possuem em comum, sendo k suficientemente grande para preservar a privacidade. Na biologia, existe uma aplicação do k MIS na análise de DNA Microarray: deseja-se identificar k genes com o máximo número de características em comum, conforme Nussbaum et al. [2010]. Ainda, Bogue [2014] estuda uma aplicação do problema em sistemas de recomendação: L são artistas musicais e R , pessoas interessadas em música; um artista é relacionado a uma pessoa se a pessoa é fã do artista. Então, o objetivo é selecionar k artistas musicais que tenham o número máximo de fãs em comum para, por exemplo, criar um evento musical.

O k MIS foi provado $\mathcal{NP}$ -difícil primeiramente em Vinterbo [2002]. O autor apresentou o problema da k -ambiguidade, definido da seguinte forma: seja $U = \{x_i\}_{i=1}^n$ um conjunto de objetos de interesse. Dado um inteiro positivo $k \leq n - 1$, deseja-se descobrir quanta informação a respeito de um elemento x_i deve ser descartada para torná-lo indiscernível dos demais k elementos. No artigo, o problema da k -ambiguidade é demonstrado equivalente ao da k -união mínima, por sua vez equivalente ao k MIS. A prova de que os problemas são $\mathcal{NP}$ -difíceis se dá por redução a partir do problema do Subgrafo Completo Bipartido Balanceado [Garey e Johnson, 1979].

Em Xavier [2012] é demonstrado que o problema é inaproximável, isto é, o k MIS não admite um algoritmo $\frac{1}{N^\epsilon}$ -aproximado, sendo N o tamanho da entrada e ϵ uma constante. Novamente demonstrou-se a complexidade do k MIS, desta vez por meio de redução do problema *Maximum Edge Biquique* (MEB), provado $\mathcal{NP}$ -difícil por Peters [2003].

Cinco modelos de programação linear inteira para encontrar soluções exatas para o k MIS são apresentados em Bogue [2014], dois dos quais baseiam-se em formulações desenvolvidas para

o MEB e uma, denominada M_{ARESTA} , é inteiramente nova. O autor introduziu, também, uma heurística construtiva gulosa, denominada k_{Inter} , e uma meta-heurística GRASP (*Greedy Randomized Adaptive Search Procedure*). A avaliação computacional foi feita utilizando instâncias geradas aleatoriamente e obtidas do serviço musical *online* LastFM.

Outra tentativa de resolver o $kMIS$ a partir de uma meta-heurística veio com Dias et al. [2020], em que, utilizando o VNS (*Variable Neighborhood Search*), diversas heurísticas e meta-heurísticas são combinadas com componentes dinâmicos para diversificar a busca. Uma contribuição importante do trabalho é o uso de *bitsets* para representar as instâncias e soluções, o que acelera as operações de conjuntos por meio do paralelismo de bits.

Finalmente, os autores de Casado et al. [2022] apresentaram uma meta-heurística GRASP, cuja fase de busca local foi substituída por uma meta-heurística de Busca Tabu. A escolha foi feita porque a Busca Tabu permite movimentos que geram soluções piores, com o que se espera escapar de ótimos locais. Os autores também modelam o problema por meio de *bitsets*. Os experimentos encontram soluções mais rápido e com melhor qualidade do que o VNS [Dias et al., 2020].

Este trabalho apresenta uma meta-heurística *Iterated Local Search* (ILS) [Lourengo et al., 2010] com duas estruturas de vizinhança, exploradas por meio de uma estratégia *Variable Neighborhood Descent* (VND) [Hansen et al., 2010] para o $kMIS$. A primeira vizinhança, $swap(1, 1)$, envolve remover um subconjunto da solução atual e inserir um outro em seu lugar. A segunda, $swap^*(2, 2)$, remove dois subconjuntos da solução atual e insere outros dois no lugar. Embora possam ser computacionalmente exigentes, as explorações são auxiliadas por estruturas de dados pré-calculadas cujo papel é reduzir o custo das avaliações e prevenir, por meio de filtros, buscas em espaços de solução pouco promissores.

O restante deste trabalho se estrutura da seguinte forma: a Seção 2 apresenta a meta-heurística proposta, debruçando-se, nas subseções, sobre a modelagem utilizada na implementação e sobre cada uma das fases do algoritmo. Os resultados obtidos são explorados na Seção 3, em que, por meio de experimentos computacionais, esta pesquisa é comparada com demais propostas da literatura. Por fim, na Seção 4, são extraídas algumas conclusões a respeito do trabalho e delineadas propostas de trabalhos futuros.

2. Algoritmo ILS proposto

Esta seção introduz o algoritmo ILS proposto. O ILS é uma meta-heurística bastante utilizada em problemas de otimização. Ela recebe uma solução inicial e melhora-a iterativamente por meio de procedimentos de perturbação e de busca local. Com a busca local, espera-se intensificar a busca em espaços promissores de solução, enquanto o papel da perturbação é diversificar a busca e potencialmente escapar de ótimos locais [Lourengo et al., 2010].

O pseudocódigo para o ILS pode ser visto no Algoritmo 1. A solução inicial é gerada (linha 2) e imediatamente refinada na função VND (linha 3), que executa a busca local. Em seguida, no loop, uma cópia da solução L^* se torna a solução atual, que entra na função de perturbação (linha 8), e então na busca local (linha 9). Então, a solução copiada L' entra na função *AcceptanceCriteria*, para ser avaliada. Ao final, L^* é retornada. Daqui em diante, o cálculo da cardinalidade da interseção resultante em L' será feito pela função *cost*. α é um parâmetro da Perturbação, explicado na Seção 2.4.

As subseções a seguir detalham a modelagem utilizada para representar as soluções cada uma das partes do algoritmo proposto.

2.1. Representação das soluções

Seguindo Dias et al. [2020] e Casado et al. [2022], utilizam-se *bitsets* para representar as instâncias e soluções do $kMIS$. Um *bitset* é um *array* binário cujos elementos só podem ser 0 ou 1. Na implementação do ILS para o $kMIS$, cada $S \in L$ tem o próprio *bitset* B de tamanho $|R|$, em

Algorithm 1 Iterated Local Search (ILS)

```

1: function ILS( $G = (L \cup R, E), k$ )
2:    $L^* \leftarrow \text{Construction}(L, k)$ 
3:    $L^* \leftarrow \text{LocalSearch}(L^*)$ 
4:    $i \leftarrow 1$ 
5:   while stop condition not reached do
6:      $\alpha \leftarrow \text{Select-}\alpha()$ 
7:      $L' \leftarrow L^*$ 
8:      $L' \leftarrow \text{Perturb}(L', \alpha)$ 
9:      $L' \leftarrow \text{VND}(L')$ 
10:     $L^*, i \leftarrow \text{AcceptanceCriteria}(L', L^*, i)$ 
11:    Atualizar  $\alpha$  a cada 10 iterações
12:  end while
13:  return  $L^*$ 
14: end function

```

que a posição p indica a presença ou ausência do elemento $e_p \in R$ em S . Uma solução L' também tem um *bitset* B' com a interseção resultante dos subconjuntos adicionados.

Com *bitsets*, o cálculo da interseção entre dois subconjuntos S_i e S_j se dá em uma operação AND entre B_i e B_j , feita por meio de paralelismo de bits para alcançar maior eficiência. Caso o número de elementos seja menor ou igual ao comprimento da palavra da CPU, o cálculo é feito em tempo constante; caso contrário, divide-se o *bitset* em *arrays* do tamanho da palavra da CPU e a operação é feita em cada um dos *arrays*. Essa modelagem acelera consideravelmente o processo de busca e já obteve sucesso em outros problemas, como mostrado em Segundo et al. [2011] e em Komosko et al. [2015].

2.2. Fase de Construção

Para a construção da solução inicial, utiliza-se a versão estendida do algoritmo guloso `kInter` [Bogue, 2014]. O `kInter` original consiste no seguinte: dada uma solução parcial L' , é selecionado para ser adicionado a L' um subconjunto $S \in L \setminus L'$ tal que $\text{cost}(L' \cup S)$ seja máximo. A escolha gulosa ocorre k vezes, ao fim das quais se tem uma solução viável. No início, $L' = \emptyset$ e, seguindo o critério, o primeiro subconjunto inserido é o de maior cardinalidade.

A versão estendida do `kInter` foi proposta por Dias et al. [2020], com o seguinte funcionamento: primeiramente, o conjunto $L = \{S_1, S_2, \dots, S_n\}$ é ordenado de tal forma que $|S_1| \geq |S_2| \geq \dots \geq |S_n|$. Então, o `kInter` é executado n vezes. Na i -ésima execução, L' é inicializada com S_i . Uma execução do `kInter` será interrompida caso $\text{cost}(L') \leq \text{cost}(L_{best})$, sendo L_{best} a melhor solução encontrada até o momento. Ao final, retorna-se a solução de maior custo entre as n geradas. Originalmente, o `kInter` estendido possui uma fase de *Path-Relinking*, mas ela não foi incorporada na implementação do ILS.

2.3. Busca Local

A busca local é usada após a construção da solução inicial e após a fase de perturbação. O procedimento de busca local proposto utiliza uma estratégia VND, na qual uma nova estrutura de vizinhança é sistematicamente selecionada para continuar a busca sempre que a estrutura atual não melhora a solução atual; caso uma solução melhor seja encontrada, ela retorna para a primeira estrutura. Isso se repete até que não ocorram novas melhorias Hansen et al. [2010]. O Algoritmo 2 descreve a busca local implementada.

Duas estruturas de vizinhança baseadas em movimentos de troca são exploradas. Elas são detalhadas nas subseções abaixo, junto com as respectivas estruturas de dados auxiliares.

Algorithm 2 VND

```

1: function VND( $L'$ )
2:    $k \leftarrow 1$ 
3:   while  $k \leq 2$  do
4:      $L'' \leftarrow \text{LocalSearch}(k, L')$ 
5:     if  $\text{cost}(L'') \leq \text{cost}(L')$  then
6:        $k \leftarrow k + 1$ 
7:     else
8:        $k \leftarrow 1$ 
9:        $L' \leftarrow L''$ 
10:    end if
11:  end while
12:  return  $L'$ 
13: end function

```

2.3.1. Estrutura de vizinhança **swap** (1, 1)

A estrutura de vizinhança **swap** (1, 1) consiste na remoção de um subconjunto $S_i \in L'$ e na adição de um $S' \in L \setminus L'$. Para acelerar o procedimento e prevenir tanto a realização de cálculos redundantes quanto a avaliação de movimentos incapazes de melhorar a solução atual, utiliza-se um *array* μ de k elementos associado a L' . Para um subconjunto $S_i \in L'$, $\mu[i]$ guarda a interseção resultante de L' após a remoção de S_i , ou seja, $\mu[i] = \bigcap_{S \in L' - \{S_i\}}$. μ é calculado antes do início da exploração da vizinhança.

Um algoritmo força bruta para preencher μ seria custoso; por isso, ele é feito conforme uma estratégia eficiente inspirada na solução para o problema de programação competitiva *Product of Array Except Self* (PAES)¹: dado um *array* A de n inteiros, deve-se construir um novo *array* p , também de tamanho n , tal que $p[i]$ seja igual ao produto de todos os elementos de A exceto $A[i]$. Isso deve ser feito em $\mathcal{O}(n)$ e sem usar o operador de divisão.

É fácil perceber que, feitas as devidas adaptações, trata-se do mesmo problema encontrado para preencher μ ; a diferença é que, para μ , em vez do produto de inteiros, deseja-se calcular a interseção de conjuntos. Para calcular μ conforme a solução do PAES, observa-se, primeiro, que $\mu[i]$ é a interseção dos subconjuntos pertencentes a L' de S_1 até S_{i-1} , chamada *prefixo*, com a interseção dos subconjuntos de S_{i+1} até S_k , chamada *sufixo*. Utilizando a técnica de Programação Dinâmica [Cormen et al., 2022], ambas interseções são calculadas previamente e em seguida acessadas para obter μ .

Na Programação Dinâmica, problemas são resolvidos através da combinação de soluções recursivas de subproblemas, os quais se sobrepõem. Cada subproblema é resolvido apenas uma vez e tem a solução guardada em uma tabela, o que evita o trabalho de recalculá-la sempre que um novo subproblema for resolvido.

Com isso em mente, pode-se definir o *prefixo* recursivamente:

$$\text{prefixo}[i] = \begin{cases} R & \text{se } i = 1, \\ \text{prefixo}[i-1] \cap S_{i-1} & \text{se } i > 1. \end{cases}$$

Ou seja, como o *prefixo* armazena, na posição i , $S_1 \cap \dots \cap S_{i-1}$, é possível usar o próprio *prefixo*, na posição anterior, para calcular a posição i . Quando $i = 1$, não existe S_{i-1} ; dessa forma, $\text{prefixo}[1]$ recebe R , que serve de conjunto neutro.

¹<https://www.enjoyalgorithms.com/blog/product-of-array-except-self>

Pode-se definir recursivamente também o `sufixo`:

$$\text{sufixo}[i] = \begin{cases} R & \text{se } i = k, \\ \text{sufixo}[i+1] \cap S_{i+1} & \text{se } i < k. \end{cases}$$

O `sufixo` armazena, na posição i , $S_{i+1} \cap \dots \cap S_k$, portanto a posição $i+1$ do array pode ser usada para calcular a posição i . Como para $i = k$ não existe S_{i+1} , `sufixo`[k] recebe R como conjunto neutro.

Assim, é possível definir μ a partir das duas interseções calculadas:

$$\mu[i] = \text{prefixo}[i] \cap \text{sufixo}[i]$$

Computacionalmente, `prefixo` e `sufixo` são *arrays* de k posições, calculados cada qual em um loop. Portanto, preencher μ com todas as soluções parciais de L' menos um subconjunto é uma operação feita em $\mathcal{O}(k)$.

O Algoritmo 3 mostra a exploração da vizinhança `swap(1, 1)`. Tendo calculado μ , procede-se para o loop principal. A cada iteração, um subconjunto $S_i \in L'$ é escolhido como candidato a ser removido da solução atual. A interseção resultante de $L' - \{S_i\}$ já está calculada em $\mu[i]$. Apenas soluções parciais cuja interseção tenha cardinalidade maior do que o custo de L' são exploradas, pois estas são as únicas com potencial de gerar uma solução viável melhor do que a atual; assim, é possível ignorá-las e reduzir o número de iterações (linha 3). Esse é um mecanismo de filtro que impede iterações infrutíferas. Ao conhecimento dos autores, o uso de uma estrutura de dados semelhante a μ , calculada de forma eficiente para servir de filtro na exploração da vizinhança, é uma abordagem inédita para o k MIS.

Dentre as possíveis opções para a escolha de qual subconjunto inserir em L' , optamos pelo critério de *first improvement*, isto é, o primeiro subconjunto que aumente a função objetivo será escolhido, pois esta opção gasta menos tempo. Note-se que μ também é utilizado na avaliação da inserção (linha 7), o que evita calcular $L' - \{S_i\}$. A complexidade do `swap(1, 1)` é $\mathcal{O}(k \cdot |L|)$ no pior caso. No entanto, observa-se que o filtro possibilitado por μ reduz bastante o custo computacional da busca.

Algorithm 3 `swap(1, 1)`

```

1: function SWAP-11( $L'$ )
2:   for  $S_i \in L'$  do
3:     if  $|\mu[i]| \leq \text{cost}(L')$  then                                 $\triangleright \mu$  ajuda a evitar soluções parciais infrutíferas
4:       continue
5:     end if
6:     for  $S' \in L \setminus L'$  do
7:       if  $|\mu[i] \cap S'| > \text{cost}(L')$  then
8:          $L' \leftarrow \text{swap}(L', S_i, S')$ 
9:       return  $L'$                                                      $\triangleright$  First improvement
10:    end if
11:  end for
12: end for
13: return  $L'$ 
14: end function

```

2.3.2. Estrutura de vizinhança $\text{swap}^*(2, 2)$

Nesta vizinhança removem-se dois subconjuntos $\{S_i, S_j\} \in L'$ e outros dois subconjuntos $\{S', S''\} \in L \setminus L'$ são adicionados. Não são testadas todas as combinações possíveis de $\{S', S''\}$, por isso a estrutura foi chamada $\text{swap}^*(2, 2)$. Como na vizinhança anterior, aqui também uma estrutura de dados ajuda a acelerar a exploração: o *array* bidimensional ϕ , cuja posição $\phi[i][j]$ guarda a interseção resultante de $L' - \{S_i, S_j\}$, ou seja, $\phi[i][j] = \bigcap_{S' \in L' - \{S_i, S_j\}}$.

Como o cálculo em força bruta de ϕ seria demasiado custoso, uma estratégia análoga ao cálculo de μ foi adotada: em um loop externo, iterando $S_i \in L'$, gera-se $L'' = L' - \{S_i\}$; em seguida, calcula-se o *prefixo* e o *sufixo* com L'' . Ao final, para cada $S_j \in L''$, $\phi[i][j]$ será:

$$\phi[i][j] = \text{prefixo}[j] \cap \text{sufixo}[j]$$

A Seção 2.3.1 fornece mais detalhes sobre o cálculo do *prefixo* e do *sufixo*. Assim, em $\phi[i][j]$, a linha i refere-se à remoção do conjunto S_i e as colunas j referem-se às remoções dos conjuntos S_j restantes em $L' - \{S_i\}$. Deste modo, ϕ é calculado em $\mathcal{O}(k^2)$.

O Algoritmo 4 descreve o funcionamento da estrutura de vizinhança $\text{swap}^*(2, 2)$. Como ocorre no $\text{swap}(1, 1)$, aqui se evita a avaliação de soluções parciais infrutíferas por meio de ϕ (linha 4). Quanto às formas de se escolher os subconjuntos a serem inseridos em L' , optamos por testar, para $S_k \in L' \setminus L'$, a combinação de S_{k-1} e S_{k+1} e trocá-los pelos subconjuntos removidos na primeira melhoria encontrada. O loop que escolhe S_{k-1} e S_{k+1} o faz de maneira circular, para que posições inválidas não sejam acessadas. Também aqui é utilizado o critério de *first improvement*, isto é, a primeira combinação de $\{S_{k-1}, S_{k+1}\}$ que aumente o custo da solução é escolhida. Note o uso de ϕ também na avaliação da troca (linha 8), utilizando a solução parcial calculada previamente. O $\text{swap}^*(2, 2)$ tem complexidade $\mathcal{O}(k^2 \cdot |L|)$ no pior caso, mas também aqui o filtro, implementado por meio de ϕ , reduz o custo computacional.

Algorithm 4 swap^*-22

```

1: function SWAP*-22( $L'$ )
2:   for  $S_i \in L'$  do
3:     for  $S_j \in L' - \{S_i\}$  do
4:       if  $|\phi[i][j]| \leq \text{cost}(L')$  then           ▷  $\phi$  ajuda a evitar soluções parciais infrutíferas
5:         continue
6:       end if
7:       for  $S_k \in L \setminus L'$  do
8:         if  $|\phi[i][j] \cap S_{k-1} \cap S_{k+1}| > \text{cost}(L')$  then
9:            $L' \leftarrow \text{swap}(L', S_i, S_{k-1}, S_j, S_{k+1})$ 
10:          return  $L'$                                ▷ First improvement
11:        end if
12:      end for
13:    end for
14:  end for
15:  return  $L'$ 
16: end function

```

2.4. Perturbação

A fase de perturbação do ILS serve para diversificar a solução atual, na esperança de remover a busca de ótimos locais. Para o $k\text{MIS}$, na fase de perturbação é comum remover aleatoriamente

um certo número de subconjuntos da solução atual e em seguida inserir, também aleatoriamente, novos subconjuntos que antes estavam de fora.

Em Dias et al. [2020], uma estratégia de perturbação que incorpora conceitos do GRASP, *ShakingGrasp*, é apresentada. Tal estratégia é implementada no ILS, com algumas adaptações: no trabalho original leva-se em conta a troca de vizinhança do VNS implementado para decidir quantos subconjuntos serão removidos. No ILS, esse número é fixo para cada instância.

A parte do GRASP adaptada para a perturbação é a fase de construção. Para uma solução parcial L' , a cada iteração cria-se uma lista restrita de candidatos (LRC) composta pelos elementos $S \in L \setminus L'$ com maior custo incremental $c(S)$, com $c(S) = \text{cost}(L' \cup S)$. O elemento a ser inserido na solução perturbada L_p é escolhido aleatoriamente dentro dos elementos da LRC. Sendo o tamanho da LRC um fator relevante para conseguir equilíbrio entre a qualidade da solução e o tempo de execução, ele é associado a um parâmetro $\alpha \in [0, 1]$, que regula o quão aleatório ou guloso o algoritmo será. Dados um α e uma solução L' , a lista será formada pelos elementos S tais que: $c(S) \in [c^{\min} + \alpha(c^{\max} - c^{\min}), c^{\max}]$, em que $c^{\max} = \max_{S \in L \setminus L'} c(S)$ e $c^{\min} = \min_{S \in L \setminus L'} c(S)$ representam o valor do maior e do menor custo incremental, respectivamente, no caso de problemas de maximização [Bogue, 2014].

O algoritmo 5 mostra a fase de perturbação do ILS:

Algorithm 5 Perturb

```

1: function PERTURB( $L'$ ,  $\alpha$ )
2:    $q \leftarrow \lceil \sqrt{|L'|} \rceil$ 
3:   if  $q + k > |L|$  then
4:      $q \leftarrow \lceil \log(|L|) \rceil$ 
5:   end if
6:    $L_p \leftarrow \text{RemoveSubsets}(L', q)$  ▷ remove  $q$  subconjuntos aleatoriamente
7:   while  $|L_p| < k$  do
8:     Construir uma LRC com base no valor de  $\alpha$ 
9:     Selecionar aleatoriamente um elemento  $S$  da LRC
10:     $L_p \leftarrow L_p \cup \{S\}$ 
11:    Atualizar o custo incremental dos candidatos remanescentes
12:  end while
13:  return  $L_p$ 
14: end function
    
```

O valor de α é um parâmetro da fase de perturbação. Isto ocorre para que a escolha do tamanho da LRC seja dinâmico. Leva-se em conta as contribuições de Ribeiro et al. [2001], em que se mostra que usar um único α pode dificultar a geração de boas soluções. Como α interfere na qualidade e na diversificação da construção, seu valor é escolhido aleatoriamente de um conjunto finito de valores $X = \{\alpha_1, \alpha_2, \dots, \alpha_m\}$. Associa-se uma probabilidade p_i a cada α_i . Inicialmente, $p_i = 1/m$ para $i = 1, \dots, m$. No caso do ILS proposto, $X = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$. Para cada α_i , $\text{cont}[i]$ é o número de iterações em que α_i foi selecionado durante a fase de Perturbação e $\text{valores}[i]$ é a soma dos custos das soluções obtidas após a Busca Local nestas iterações. Seja $Z_{\max}$ o custo da melhor solução encontrada até o momento. A cada 10 iterações do ILS, as probabilidades p_i são atualizadas (linha 11 do Algoritmo 1) conforme os passos abaixo, se $\text{cont}[i] > 0, \forall i = 1, \dots, 10$. O objetivo é favorecer as escolhas dos α_i usados para obter as melhores soluções.

1. Para cada α_i , calcular a média $\text{avg}(i) = \frac{\text{valores}[i]}{\text{cont}[i]}$ e $Q_i = \frac{\text{avg}(i)}{Z_{\max}}$

2. Calcular $\sigma = \sum_{i=1}^m Q_i$
3. Calcular $p_i = \frac{Q_i}{\sigma}$ para obter valores entre $[0, 1]$.

2.5. Critério de aceitação

O Algoritmo 6 mostra o critério de aceitação de uma solução L' no ILS. No primeiro caso (linha 2), L' é melhor do que L^* e é, assim, escolhida como nova solução atual. A variável i é reiniciada com 1. Caso L' não seja melhor do que L^* , se $i < T$ (linha 5), i é incrementado e a solução se mantém a mesma. T é uma constante que limita o número de iterações pelas quais uma solução pode passar sem ser melhorada. Quando i ultrapassa T , executa-se um mecanismo simples para remover a busca do ótimo local, diversificando-a.

O mecanismo é uma adaptação do algoritmo `kInter`, apresentado por Bogue [2014], explicado na Seção 2.2. A adaptação consiste em, na primeira inserção da nova solução L^* , não utilizar o critério guloso, mas sim escolher um subconjunto $S \in L$ aleatoriamente. Isso altera as demais decisões gulosas e gera uma solução diferente da atualmente explorada. Gerada a solução viável, ela é refinada por meio de uma busca local. A solução de melhor custo encontrada globalmente pelo ILS não é substituída pela nova L^* gerada nessa fase.

Algorithm 6 AcceptanceCriteria

```

1: function ACCEPTANCECRITERIA( $L', L^*, i$ )
2:   if  $\text{cost}(L') > \text{cost}(L^*)$  then
3:      $L^* \leftarrow L'$ 
4:      $i \leftarrow 1$ 
5:   else if  $i \leq T$  then
6:      $i \leftarrow i + 1$ 
7:   else
8:      $S \leftarrow \text{RandomSubset}(L)$ 
9:      $L^* \leftarrow \{S\}$ 
10:    while  $|L^*| < k$  do
11:      Selecionar um subconjunto  $S \in L \setminus L^*$  tal que  $\text{cost}(L^* \cup \{S\})$  seja máximo
12:       $L^* \leftarrow L^* \cup \{S\}$ 
13:    end while
14:     $L^* \leftarrow \text{LocalSearch}(L^*)$ 
15:     $i \leftarrow 1$ 
16:  end if
17:  return  $L^*, i$ 
18: end function

```

3. Resultados computacionais

Nesta seção, discutem-se os resultados computacionais obtidos com o algoritmo proposto. Ele foi implementado na linguagem C++ e os testes foram executados em um AMD Ryzen 5 4500 6-Core com clock de 4,2 GHz, 16 GB de memória RAM, tendo o Ubuntu 22.04.3 LTS como sistema operacional.

O ILS foi comparado com as duas últimas meta-heurísticas para o k MIS encontradas na literatura, a saber, o VNS apresentado em Dias et al. [2020] e o GRASP com Busca Tabu apresentado em Casado et al. [2022]. Os autores disponibilizaram os respectivos códigos-fontes.

As instâncias utilizadas foram as introduzidas em Dias et al. [2020]. Tratam-se de 238 instâncias divididas em três grupos: no primeiro, o número de subconjuntos é igual ao número de

Tabela 1: Comparação entre Reactive VNS e ILS.

Classe	Qnt.	Reactive VNS [Dias et al., 2020]				ILS			
		Best	Worst	Avg.	Time (s)	Best	Worst	Avg.	Time (s)
1	30	4.2	4	4.1	0.523	4.1	3.97	4.03	0.079
2	30	1	1	1	1.235	1	1	1	0.002
4	30	25.3	24.67	24.89	0.672	26	25.4	25.7	0.260
5	30	7.2	7.07	7.14	1.410	7.37	7.13	7.24	0.204
6	28	2.32	2.29	2.29	0.924	2.39	2.36	2.38	0.206
7	30	97.93	97.53	97.73	0.571	98.27	97.77	97.95	0.322
8	30	75.47	75.27	75.38	1.187	75.87	75.67	75.74	0.261
9	30	50.07	50.07	50.07	1.033	50.83	50.7	50.77	0.274
Avg.		33.19	32.99	33.08	0.945	33.49	33.26	33.36	0.201

elementos, isto é, $|L| = |R|$; no segundo, o número de subconjuntos é maior ($|L| > |R|$) e, no terceiro, é maior o número de elementos ($|L| < |R|$). Baseando-se na densidade das instâncias, também é possível dividi-las em classes: quanto maior a classe, maior a densidade. C_1 , C_4 e C_7 têm valores baixos de k , C_2 , C_5 e C_8 têm valores médios e C_6 e C_9 têm valores altos.

Para que as comparações entre os algoritmos sejam justas, todos foram executados no mesmo computador. Além disso, fixamos o tempo de execução dos três algoritmos em $\frac{k}{10}$ segundos para cada instância. Desejando evitar o viés produzido pelos elementos randômicos das implementações, os resultados foram computados a partir de 10 execuções independentes de cada algoritmo em cada instância. Seguindo esse protocolo, foram construídas as tabelas a seguir.

3.1. Comparação com o Reactive VNS

Esta subseção avalia a performance da proposta deste trabalho em comparação com a proposta anterior denominada *Reactive VNS* [Dias et al., 2020]. Trata-se de um algoritmo de *Variable Neighborhood Search* com elementos reativos.

A Tabela 1 mostra os resultados obtidos após os testes seguindo o protocolo descrito acima. Por uma questão de brevidade, os resultados são exibidos em relação às classes das instâncias. Assim, a coluna *Qnt.* mostra a quantidade de instâncias em uma classe; a coluna *Best* tem a média das melhores soluções obtidas pelos algoritmos nas instâncias de uma dada classe nas 10 execuções; *Worst* é a média dos piores valores obtidos por classe; *Avg.* é o valor médio encontrado por classe, independentemente de ser o melhor ou o pior. Já *Time (s)* exibe o tempo médio que uma execução do algoritmo leva para encontrar o seu melhor valor, dentro do intervalo de $\frac{k}{10}$ segundos. Na última linha estão as médias das colunas ponderadas pela quantidade de instâncias em cada classe.

Analisando inicialmente a melhor solução encontrada por cada algoritmo, verifica-se a superioridade do ILS, que atinge os melhores valores em 7 das 8 classes, enquanto que o VNS atinge os melhores, ou empata com o ILS, em 2 classes. O empate ocorre nas instâncias da Classe 2, para as quais todos os algoritmos encontram sempre soluções de custo 1. O ILS apresenta pior desempenho nas instâncias da Classe 1, que têm menor densidade e valores baixos de k . A proporção é a mesma quando se analisa a qualidade das piores soluções, bem como média encontrada por cada um deles. O ILS vence na média final de todas as colunas, inclusive em relação ao tempo para encontrar as melhores soluções, levando em média 0,2 s, em contraste com 0,94 s do VNS.

Tabela 2: Comparação entre GRASP e ILS.

Classe	Qnt.	GRASP [Casado et al., 2022]				ILS			
		Best	Worst	Avg.	Time (s)	Best	Worst	Avg.	Time (s)
1	30	4.23	4.13	4.21	0.212	4.1	3.96	4.03	0.079
2	30	1	1	1	0.009	1	1	1	0.002
4	30	25.37	25.03	25.2	0.359	26	25.4	25.7	0.260
5	30	7.4	7.27	7.34	0.429	7.37	7.13	7.24	0.204
6	28	2.43	2.36	2.41	0.51	2.39	2.36	2.38	0.206
7	30	98.07	97.73	97.9	0.435	98.27	97.77	97.95	0.322
8	30	75.53	75.27	75.41	1.045	75.87	75.67	75.74	0.261
9	30	50.17	50.1	50.13	0.662	50.83	50.7	50.77	0.274
Avg.		33.28	33.12	33.21	0.457	33.49	33.26	33.36	0.201

3.2. Comparação com o GRASP

Nesta subseção, avalia-se a performance do ILS em comparação com o GRASP apresentado em Casado et al. [2022]. Trata-se de um algoritmo *Greedy Randomized Adaptive Search* com uma Busca Tabu na fase de busca local.

A Tabela 2 mostra os resultados obtidos após os testes. As colunas são as mesmas da Tabela 1, explicadas na Seção 3.1. Verifica-se novamente a vantagem do ILS no que diz respeito à qualidade das melhores soluções encontradas, obtendo os maiores valores em 5 das 8 classes; já o GRASP vence ou empata em apenas 4, sendo o empate na Classe 2, para a qual os algoritmos sempre encontram soluções de custo 1. O desempenho do ILS é inferior na Classe 1, que possui instâncias com menor densidade e valores baixos de k . O mesmo ocorre nas Classes 5 e 6, que apresentam densidades médias, sendo que a Classe 5 tem valores médios de k , enquanto que a Classe 6 possui valores altos. Semelhantemente ao que acontece com o VNS, a proporção se mantém quando comparamos os piores resultados e os resultados médios. O ILS também vence quando comparamos as médias finais. É evidente, ainda, a superioridade do ILS no tempo necessário para encontrar soluções de qualidade: o GRASP leva em média 0,45 s, mais do que o dobro do ILS.

Portanto, o ILS surge como uma proposta competitiva de meta-heurística para o k MIS, obtendo em pouco tempo soluções de melhor qualidade do que os métodos encontrados na literatura. Atribui-se à busca local implementada, com os ganhos de performance obtidos pelo uso dos arrays μ e ϕ (Seção 2.3), a grande responsabilidade por esses resultados.

4. Conclusão

Este trabalho apresenta um algoritmo *Iterated Local Search* (ILS) para o problema da Interseção Máxima de k -Subconjuntos (k MIS). São utilizadas duas estruturas de vizinhança na fase de busca local, executada por meio de uma estratégia de VND. Cada estrutura de vizinhança está associada a uma estrutura de dados auxiliar que, calculando de forma eficiente as soluções parciais a partir de uma solução atual, evitam a busca em espaços de solução infrutíferos e aceleram a avaliação de um movimento, reduzindo o tempo computacional exigido para encontrar um ótimo local. Os resultados dos experimentos confirmam a superioridade do método proposto em relação aos demais encontrados na literatura.

Os trabalhos futuros focarão em um estudo aprofundado das instâncias em que o ILS não obtém os melhores resultados, buscando implementar novas estruturas de vizinhança e métodos

híbridos de busca. O objetivo é aumentar a diversidade da busca sem interferir negativamente na performance do algoritmo.

Referências

- Bogue, E. T. (2014). O problema da máxima interseção de k-subconjuntos. Master's thesis, Universidade Federal de Campinas.
- Casado, A., Pérez-Peló, S., Sánchez-Oro, J., e Duarte, A. (2022). A grasp algorithm with tabu search improvement for solving the maximum intersection of k-subsets problem. *Journal of Heuristics*.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., e Stein, C. (2022). *Introduction to Algorithms*. The MIT Press.
- Dias, F. C. S., Tavares, W. A., e de Freitas Costa, J. R. (2020). Reactive vns algorithm for the maximum k-subset intersection problem. *Journal of Heuristics*.
- Garey, M. R. e Johnson, D. S. (1979). *Computers and Intractability*. W. H. Freeman and Compant, New York.
- Hansen, P., Mladenović, N., Brimberg, J., e Pérez, J. A. M. (2010). Variable neighborhood search. In Gendreau, M. e Potvin, J.-Y., editors, *Handbook of Metaheuristics, Second Edition*, p. 61–86. Springer, New York, USA.
- Komosko, L., Batsyn, M., San Segundo, P., e Pardalos, P. (2015). A fast greedy sequential heuristic for the vertex colouring problem based on bitwise operations. *Journal of Combinatorial Optimization*, 31.
- Lourenço, H. R., Martin, O. C., e Stützle, T. (2010). Iterated local search: Framework and applications. In Gendreau, M. e Potvin, J.-Y., editors, *Handbook of Metaheuristics, Second Edition*, p. 363–397. Springer, New York, USA.
- Nussbaum, D., Pu, S., e Jörg-Rüdiger Sack, T. U. . H. Z.-Z. (2010). Finding maximum edge bicliques in convex bipartite graphs. *Algorithmica*.
- Peters, R. (2003). The maximum edge biclique problem is np-complete. *Discrete Applied Mathematics*, 131 (3):651–654.
- Ribeiro, C. C., Uchoa, E., e Werneck, R. F. (2001). A hybrid grasp with perturbations for the steiner problem in graphs. *Inform Journal on Computing*.
- Segundo, P. S., Rodríguez-Losada, D., e Jiménez, A. (2011). An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38 (2):571–581.
- Vinterbo, S. A. (2002). A note on the hardness of the k-ambiguity. Technical report, Harvard Medical School.
- Xavier, E. C. (2012). A note on a maximum k-subset intersection problem. In *Information Processing Letters*.