LVI SBPO SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL
DE 04 ATÉ 07 NOVEMBRO 2024
FORTALEZA | CEARÁ

UMA ANÁLISE DO GRASP COM FILTRO DE BLOOM PARA IMPEDIR EXPLORAÇÃO DE DUPLICATAS APLICADO AO CVRP

Israel Pereira de Souza
Instituto de Computação - Universidade Federal Fluminense
Av. Gal. Milton Tavares de Souza, s/n, Boa Viagem – Niterói - RJ, 24210-346
israelsouza@id.uff.br

Igor Machado Coelho
Instituto de Computação - Universidade Federal Fluminense
Av. Gal. Milton Tavares de Souza, s/n, Boa Viagem – Niterói - RJ, 24210-346
imcoelho@ic.uff.br

RESUMO

Meta-heurísticas são formas eficientes de resolver problemas de alta complexidade, porém, pela natureza dos problemas de otimização combinatória, é muito custoso armazenar todas as soluções geradas pelo alto custo computacional do espaço exponencial de soluções. Desta forma, diversos desses algoritmos podem reavaliar soluções já avaliadas, aplicando novamente rotinas de alto custo em soluções já exploradas. O filtro de Bloom é uma ferramenta eficiente e com baixo consumo de memória para mapear grandes conjuntos de dados. Neste trabalho, propomos o uso do filtro de Bloom com a meta-heurística *GRASP*, aplicado ao Problema de Roteamento de Veículos Capacitados (*CVRP*). Experimentos demonstraram que a estratégia com o filtro de Bloom foi eficiente, com baixo custo de memória e tempo computacional. Para $\alpha = 0.05$ (um parâmetro bem usado do *GRASP* e que também pode gerar muitas duplicatas), o ganho de tempo chega a 93% do tempo sem filtro.

PALAVRAS CHAVE. Filtro de Bloom, GRASP, CVRP.

Logística e Trasportes, Metaheurísticas e Otimização Combinatória.

ABSTRACT

Metaheuristics are efficient ways to solve high-complexity problems, however, due to the context of optimization problems it is very costly to store all the generated solutions for the high storage demand of the exponential space of solutions. Thus, many metaheuristics can reevaluate solutions that were already evaluated, applying again highly costly routines in already explored solutions. The Bloom filter is an efficient tool with low memory consumption to map big databases. In this work, we propose the use of the Bloom filter with GRASP metaheuristic applied to the Capacitated Vehicle Routing Problem (CVRP). Experiments presented that the strategy with the Bloom filter is efficient with low memory usage and low computational time. For $\alpha = 0.05$ (a parameter with high usage in GRASP, that can also lead to many duplicates), the time gain reaches 93% of the time without the filter.

KEYWORDS. Bloom filter. GRASP. CVRP.

Logistics and transport, Metaheuristics, and Combinatorial Optimization.

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

1. Introdução

Problemas de otimização combinatória são conhecidos pela sua complexidade. Meta-heurísticas são técnicas de otimização projetadas para solucionar problemas complexos, em que a busca por soluções ótimas no espaço de soluções é desafiadora. Porém, pelo alto custo para analisar as soluções já geradas, que compõem um espaço exponencial de soluções, diversos métodos não avaliam se a solução já foi explorada, aplicando novamente rotinas de alto custo computacional como as estratégias de busca local. O Problema de Roteamento de Veículos, do inglês *Vehicle Routing Problem* (*VRP*) é um problema conhecido de otimização combinatória e seu objetivo é minimizar custos para entrega de demandas para clientes, minimizando custos de transporte. O *VRP* possui impacto significativo em sistemas logísticos, otimizando tráfego e reduzindo trajetórias de transporte que podem reduzir diretamente o custo dos produtos.

Trabalhos recentes de meta-heuristicas aplicadas ao *VRP* envolvem: colônia de formigas para solucionar o *VRP* com drones [Huang et al., 2022], algoritmo genético para solucionar o *CVRP* [Mohammed et al., 2017] e um algoritmo baseado em Evolução Diferencial para solucionar o *CVRP* [Souza et al., 2023]. Trabalhos recentes envolvendo a meta-heurística *GRASP* aplicada ao VRP envolvem: um algoritmo *GRASP* para o *VRP* verde com estações capacitadas privadas [Bruglieri et al., 2022], um algoritmo de *GRASP* reativo com *VND* para solucionar o VRP com múltiplos depósitos [de Souza et al., 2022] e um algoritmo *GRASP* com *VND* para minimizar consumo energético de drones no *VRP* com coleta e entrega [Kyriakakis et al., 2023].

O filtro de Bloom é uma estrutura de dados probabilística para verificação da existência de um elemento em um conjunto. O filtro não apresenta falsos negativos, ou seja, não irá informar que um elemento não está, quando na verdade ele está no conjunto. Porém, o filtro pode apresentar falsos positivos, em que pode informar que o elemento está, quando na verdade não está no conjunto. Ele é uma forma compacta e de baixo custo de memória e tempo para representação de um conjunto.

O filtro de Bloom tem sido usado com cache web [Alexander et al., 2014; Rottenstreich e Keslassy, 2014], monitoramento de rede [Li et al., 2016] e sincronização de conteúdo [Luo et al., 2016]. Trabalhos recentes do filtro de Bloom combinado com metaheurísticas envolvem o uso de metaheurísticas para otimizar o filtro de Bloom para detecção de spam [Gibson et al., 2020; Arulanand e Premalatha, 2014; Natarajan et al., 2012].

Neste trabalho apresentamos um estudo do comportamento da meta-heurística *GRASP* (do inglês, *Greedy Randomized Adaptive Search Procedures*) com *Variable Neighborhood Descent (VND)*, utilizando também o filtro de Bloom para verificação das soluções já geradas pela meta-heurística e então, não reaplicar estratégias de busca local pelo alto custo computacional. Instâncias conhecidas do problema são utilizadas e o ganho em tempo é avaliado.

Este trabalho é estruturado da seguinte forma: a seção 2 descreve o filtro de Bloom. A seção 3 descreve o Problema de Roteamento de Veículos Capacitados e apresenta sua formulação matemática. A seção 4 descreve a meta-heurística *GRASP*. A seção 5 apresenta o algoritmo proposto e analisado, descrevendo as chaves utilizadas no filtro e as estratégias de vizinhança aplicadas no *VND*. A seção 6 apresentam os experimentos computacionais conduzidos com o algoritmo proposto. Por fim, a seção 7 conclui este trabalho.

2. Filtro de Bloom

O filtro de Bloom, proposto por Burton Bloom [Bloom, 1970], é uma estrutura de dados probabilística para verificação da existência de um elemento em um conjunto. O filtro apresenta uma forma probabilistica compacta de representação de um conjunto, que pode fornecer falsos positivos (informar que um elemento está no conjunto, quando não está), mas não reporta falsos negativos (informar que o elemento não está, quando ele está) [Tarkoma et al., 2011].

A definição matemática do filtro, apresentada em [Mitzenmacher, 2001], é descrita na forma: um filtro de Bloom para representação do conjunto $S = \{s_1, s_2, \ldots, s_n\}$ é descrito por

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

um vetor de m bits, com todos os bits inicialmente com zero. O filtro utiliza k funções de hash independentes $h_1, h_2, \ldots, h_k$, cada uma com raio $\{0, 1, \ldots, m-1\}$. Para cada elemento a ser inserido no conjunto $s \in S$, os bits $h_i(s)$ são configurados como 1, com $1 < i < k$. Um bit pode ser configurado como 1 múltiplas vezes, porém só terá efeito na primeira vez. Para checar se um elemento x pertence ao conjunto S, verificamos se todos os bits $hi(x)$ são 1, $1 < i < k$. Em caso afirmativo, assumimos que x pertence a S, podendo errar com certa probabilidade. Em outro caso, não há possibilidade que x pertença a S. Para diversas aplicações, este é um erro aceitável, desde que a probabilidade seja suficientemente baixa [Mitzenmacher, 2001].

Um exemplo de filtro de Bloom, com $m = 32$ bits, três funções hash (h_1, h_2, h_3) e $n = 3$ elementos é dado pela Figura 1. Neste exemplo, uma verificação é feita para reportar se o elemento w pertence ao conjunto. Porém, como $h_3(w)$ é 0, w não pertence ao conjunto.

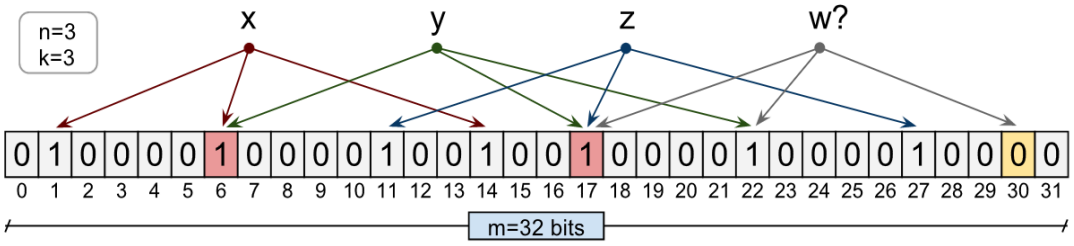


Figura 1: Exemplo de conjunto representado pelo filtro de Bloom.
FONTE: [Tarkoma et al., 2011].

Para a inserção de um elemento no filtro, a probabilidade de um bit arbitrário não ser configurado como 1 é $(1 - 1/m)^k$. Para n objetos, a probabilidade é $(1 - 1/m)^{kn}$ e então, a probabilidade de um bit ser configurado como 1 é $1-(1-1/m)^{kn}$. Desta forma, para um elemento considerando k funções de hash, a probabilidade do filtro reportar que o elemento está presente, quando este não está, é dado pela Equação (1) [Tarkoma et al., 2011].

$$((1 - \frac{1}{m})^{kn})^k \tag{1}$$

O principal benefício do filtro de Bloom é seu uso eficiente de memória. Ao contrário de armazenar explicitamente todos os elementos do conjunto, o filtro de Bloom requer apenas espaço para armazenar os bits do vetor, tornando-o particularmente útil em cenários onde a memória é limitada. Seu uso, porém, está restrito a informar se um elemento pertence ou não ao conjunto. No entanto, é importante notar que há probabilidade em que o Filtro de Bloom retorne falsos positivos, ou seja, indicar que um elemento está presente no conjunto quando na verdade não está. Esta probabilidade aumenta à medida que o número de elementos no conjunto aumenta ou ao utilizar um número de bits m menor.

Variações do filtro de Bloom têm sido propostas na literatura. Fan et al. [2000] introduziram os *Counting Bloom filter* para uso em conjunto com cache Web, adicionando suporte para deleção de elementos. Para isso, ao invés de utilizar um vetor de bits 0,1, é apresentado um vetor com m contadores e cada inserção incrementa seus contadores e de forma análoga, cada deleção decrementa seu contadores, para cada função hash. O elemento não existirá no conjunto caso algum de seus contadores seja zero. Mitzenmacher [2001] propôs filtro de Bloom comprimidos para tráfego entre proxies. Guo et al. [2009] propuseram filtro de Bloom dinâmico, possibilitando representar conjuntos dinâmicos com menor consumo de memória e com suporte adicional para deleção de elementos utilizando contadores.

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

3. O Problema de Roteamento de Veículos Capacitados

O Problema de Roteamento de Veículos (PRV), do inglês *Vehicle Routing Problem (VRP)*, é um problema conhecido de otimização combinatória para minimizar custos de entregas de demandas aos clientes, com impacto econômico de transporte. Dantzig e Ramser em 1959 foram os primeiros a introduzir o *VRP* com proposta de uma formulação matemática e algorítmica para modelar a entrega de gasolina a estações de serviço [Dantzig e Ramser, 1959]. Em 1964, Clarke e Wright melhoraram tal estratégia com a proposta de uma heurística gulosa e efetiva [Clarke e Wright, 1964]. Posteriormente, o problema foi derivado em diversas variações, como PRV com janelas de tempo, múltiplos depósitos, coleta e entrega, entregas repartidas, entregas periódicas, entre outras.

O *VRP* é um problema *NP-hard* [Lenstra e Kan, 1981] com diversas aplicações no mundo real. Sua versão clássica, conhecida como PRV Capacitado, do inglês *Capacitated Vehicle Routing Problem (CVRP)*, modela rotas ótimas para entrega de demandas, onde cada veículo deve trafegar apenas uma rota. Cada rota deve iniciar seu trajeto e retornar ao depósito, visitando um subconjunto de clientes [Braekers et al., 2016]. O objetivo do *CVRP* é minimizar o custo de um conjunto de rotas que serão trafegadas pelos veículos, respeitando as capacidades dos veículos.

3.1. Definição matemática

O Problema de Roteamento de Veículos Capacitados, do inglês *Capacitated Vehicle Routing Problem* (CVRP), pode ser definido por um grafo completo e unidirecionado $G = (V, E)$, onde $V = \{c_0, c_1, \ldots, c_N\}$ é o conjunto de vértices e $E = \{(c_i, c_j) : c_i, c_j \in V, i < j\}$ é o conjunto de arestas. Vértices $c_1, \ldots, c_N$ representam os clientes e cada cliente c_i é associado a uma demanda não negativa q_i. O Vértice q_0 representa o depósito, em que uma frota de veículos homogêneos de capacidade Q devem partir. O depósito não possui demanda associada. Cada aresta $(c_i, c_j) \in E$ é associada com um custo de trajeto não negativo ou distância d_{ik}

O CVRP tem por objetivo minimizar um conjunto de rotas de veículos de forma que: (a) cada roda inicia e finaliza no depósito, (b) cada cliente é visitado apenas uma vez, (c) a demanda total de uma rota (a soma das demandas dos clientes daquela rota) não ultrapassam a capacidade do veículo Q e (d) o número de rotas não ultrapassam o número de veículos K. Em problemas reais, um veículo pode executar múltiplas rotas, porém ele é considerado algoritmicamente como múltiplos veículos. O modelo matemático do CVRP é apresentado abaixo [İlhan, 2021; Chen et al., 2006]:

Parâmetros de entrada:

N: número de clientes.

K: número máximo de veículos.

Q: capacidade máxima de cada veículo.

d_{ij}: distância entre clientes c_i e c_j, com $d_{ij} = d_{ji}\ \forall i, j \in \{0, 1, \ldots, N\}$.

q_i: demanda do cliente $c_i \in V$.

Variáveis de decisão:

$$x_{ij}^k = \begin{cases} 1, & \text{se o veículo } k \text{ trafega do cliente } c_i \text{ ao cliente } c_j, \\ 0, & \text{em outro caso.} \end{cases}$$

$Minimize$:

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

$$Cost = \sum_{k=1}^{K} \sum_{i=0}^{N} \sum_{j=0}^{N} d_{ij} x_{ij}^{k} \tag{2}$$

subject to:

$$\sum_{k=1}^{K} \sum_{i=0}^{N} x_{ij}^{k} = 1, \quad j = 1, 2, ..., N, \tag{3}$$

$$\sum_{k=1}^{K} \sum_{j=0}^{N} x_{ij}^{k} = 1, \quad i = 1, 2, ..., N, \tag{4}$$

$$\sum_{i=0}^{N} x_{ih}^{k} - \sum_{j=0}^{N} x_{hj}^{k} = 0, \quad k = 1, 2, ..., K; h = 1, 2, ..., N, \tag{5}$$

$$\sum_{j=0}^{N} q_j \left(\sum_{i=0}^{N} x_{ij}^{k} \right) \leq Q, \quad k = 1, 2, ..., K, \tag{6}$$

$$\sum_{j=1}^{N} x_{0j}^{k} \leq 1, \quad k = 1, 2, ..., K, \tag{7}$$

$$\sum_{i=1}^{N} x_{i0}^{k} \leq 1, \quad k = 1, 2, ..., K, \tag{8}$$

$$x_{ij}^{k} \in \{0, 1\}, \quad i \neq j; i, j = 0, 1, \ldots, N; k = 1, 2, ..., K. \tag{9}$$

A função objetivo (2) minimiza a distância total trafegada por todos os veículos. As restrições (3) e (4) garantem que cada cliente é visitado apenas uma vez. A restrição (5) garante a continuidade da rota. A restrição (6) garante que a demanda total de qualquer rota não exceda a capacidade máxima do veículo Q. As restrições (7) e (8) garantem que cada veículo é usado não mais que uma vez. Por fim, a restrição (9) a binaridade da variável de decisão, ou seja, que só assuma valores inteiros 0 ou 1.

4. GRASP

O algoritmo, do inglês, *Greedy Randomized Adaptive Search Procedures (GRASP)* [Feo e Resende, 1995] é uma meta-heurística multistart com construção e procedimentos de busca local. Em cada iteração do *GRASP*, uma solução é construída utilizando um algoritmo guloso randomizado, que é posteriormente submetido aos procedimentos de busca local. A melhor solução é apresentada como solução do algoritmo. O algoritmo *GRASP* iterativamente avalia um conjunto de elementos a serem inseridos na solução, checando também a viabilidade da inserção. Os elementos de menor custo incremental são inseridos em uma lista ordenada chamada lista de candidatos restrita, do inglês *restricted candidate list (RCL)*, que é limitada pelo parâmetro α em porcentagem. Como exemplo, quando $\alpha = 0.1$, a lista terá apenas os 10% melhores candidatos.

O próximo elementos a ser inserido na solução em construção é selecionado aleatoriamente da *RCL*. Desta forma, um dos melhores candidatos é selecionado para fazer parte da solução, mas não necessariamente o melhor candidato. A lista *RCL*, então, é atualizada e os elementos são

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

reavaliados até que a solução se torne completa. Se por fim a viabilidade não for atendida, os procedimentos de reparação podem ser aplicados e/ou novas soluções podem ser geradas para obtenção de soluções viáveis. O *GRASP* demanda apenas dois parâmetros de entrada: a restrição de tamanho da lista *RCL* e o número máximo de iterações do algoritmo *GRASP* ($max_iterations$) e pode ser implementado facilmente de forma paralela [Potvin e Gendreau, 2018].

O pseudocódigo do *GRASP* é dado por Algoritmo 1 e Algoritmo 2 [Potvin e Gendreau, 2018].

Algorithm 1: `GRASP`

```
Input: max_iterations, α
Output: MelhorSolução
for it ← 1 to max_iterations do
    Solução ← GreedyRandomizedSolution(α)
    if Solução não é viável then
        Solução ← Repara(Solução)
    end if
    Solution ← Busca_Local(Solução)
    MelhorSolução ← Update(Solução, MelhorSolução)
end for
return MelhorSolução
```

Algorithm 2: `GreedyRandomizedSolution`

```
Input: α
Output: Solução
Solução ← ∅
Inicializa o conjunto com os elementos candidatos
Avalia o custo incremental dos elementos candidatos
while existe ao menos um elemento candidato do
    RCL ← ConstroiRCL(α)
    s ← Seleciona um elemento aleatório da RCL
    Solução ← Solução ∪ {s}
    Atualiza o conjunto de elementos candidatos
    Reavalia os custos incrementais
end while
```

5. Algoritmo proposto

Este trabalho tem por objetivo analisar o comportamento e verificar se há ganho ao combinar o uso do filtro de Bloom em problemas com grandes espaços de soluções, removendo então o recálculo de procedimentos custosos de melhoria de solução como as buscas locais em soluções já analisadas. Para isso, escolhemos um problema conhecido de otimização combinatória, o *CVRP*, que é descrito na Seção 3. Como solução, analisamos o comportamento da meta-heurística *GRASP*, descrito na seção 4. O *GRASP* possui um comportamento interessante pois seu parâmetro α, que controla o tamanho da lista *RCL*, também controla diretamente a probabilidade de geração de soluções duplicadas. Desta forma, menores valores de α possuem uma maior probabilidade de geração de soluções duplicadas.

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

Para implementação do filtro de Bloom, é preciso definir as funções de hash que serão utilizadas. Neste projeto, para solucionar o *CVRP*, definimos quatro funções hash. Uma solução para o CVRP pode ser representada por uma permutação de clientes e é representado na forma $individual = [c_{x_1}, \ldots, c_{x_N}]$, $x_i \in [1, 2, \ldots, N]$.

$$key_1 = \left\lfloor 0.5 + \sum_{i=1}^{N} individual[i] * i * \pi \right\rfloor \qquad key_3 = std :: \texttt{hash}(individual)$$

$$key_2 = \left\lfloor 0.5 + \sum_{i=1}^{N} individual[i] * \pi^i \right\rfloor \qquad key_4 = \texttt{Cost}(individual)$$

A função hash key_1, calcula o somatório do identificador do cliente, dado por $individual[i]$, multiplicado a sua posição i na solução e a constante π, para maior dispersão dos dados na hash. O valor final é arredondado. A função hash key_2, calcula o somatório do identificador do cliente, multiplicado pela constante π (para maior dispersão dos dados na hash), elevado a posição do cliente na solução. O valor final é arredondado. A função hash key_3, é uma biblioteca padrão do *C++*, em que fornece um valor inteiro para uma string (gerada pela concatenação de todos identificadores dos clientes na solução). Por fim, a função hash key_4, é fornecida pelo cálculo do custo do indivíduo, dado pela Equação (2). A posição efetiva do elemento na hash, dado sua chave, é calculado pelo resto da divisão inteira do valor da chave pelo tamanho do filtro de Bloom ($key_{bloom_k} = key_k \ \% \ m$, $k = 1, 2, 3, 4$).

Para implementar a busca local do algoritmo *GRASP*, escolhemos o algoritmo *Variable Neighborhood Descent (VND)* [Potvin e Gendreau, 2018]. Definimos também 6 vizinhanças, 4 *intra-route* (as modificações alteram apenas uma rota), e 2 *inter-route* (as modificações alteram mais de uma rota). **Métodos *intra-route***: *2-opt*: troca dois arcos, mantendo o tuor; *2-swap*: troca dois clientes da mesma rota; *or-opt*$_1$: reposiciona um cliente na rota; *or-opt*$_2$: reposiciona dois clientes contíguos na rota. **Métodos *inter-route***: *swap*$_{1,1}$: troca dois clientes de rotas diferentes; *shift*$_1$: remove um cliente de uma rota e insere em outra rota. O algoritmo *VND* e *Best Improvement*, dois algoritmos clássicos, podem ser encontrados em [Potvin e Gendreau, 2018].

O Algoritmo 3 apresenta o pseudocódigo do algoritmo proposto, que utiliza a meta-heurística *GRASP* com *VND*, combinando também o filtro de Bloom, para solucionar o *CVRP*.

Algorithm 3: `GRASP+BloomFilter`

Input: *max_iterations*, α
Output: *MelhorSolução*
for *it* ← *1 to max_iterations* **do**
  Solução ← `GreedyRandomizedSolution(`α`)`
  if *not* `PossivelmenteContém`*(BloomFilter, Solution)* **then**
    BloomFilter ← `Adiciona`(*BloomFilter*, *Solução*)
    Solução ← `VND`(*Solução*)
    MelhorSolução ← `Update`(*Solução*, *MelhorSolução*)
  end if
end for
return *MelhorSolução*

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

6. Experimentos

Nesta seção discutimos os experimentos computacionais executados. Os algoritmos foram implementados em *C++*, compilados usando *g++* versão 11.4 em uma máquina com processador intel i7-10750H com 16GB de memória RAM e sistema Ubuntu 22.04. O pacote OptFrame [Coelho et al., 2011] foi utilizado, fornecendo a implementação das estratégias de vizinhança, algoritmo de *Best Improvement* e *VND*, descritos na seção 5.

O algoritmo foi testado utilizando 3 instâncias do benchmark *CVRP* do pacote A [Augerat et al., 1995]. As instâncias foram selecionadas de tamanho pequeno, médio e grande, relativas ao pacote. A menor instância (A-n32-k5) e a maior instância (A-n80-k10) foram selecionadas, com adição de uma instância intermediária A-n60-k9. As instâncias podem ser obtidas no [CVRPLIB, 2021]. O filtro de Bloom foi executado com tamanho $m = 10^7$ bits, tendo custo de 1.25 megabytes.

O algoritmo foi executado 10 vezes em cada instância, utilizando os 10 primeiros inteiros como semente, para a comparação de tempo ao utilizar o filtro de Bloom para não reaplicar o *VND*. Desta forma, o *VND* e as estratégias de vizinhança nunca são executadas em soluções duplicadas.

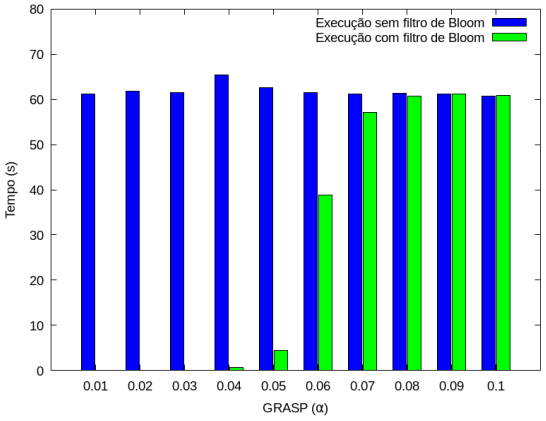
As Figuras 2a, 2b, 2c apresentam os gráficos de comparação de tempo do algoritmo na instâncias A-n32-k5, A-n60-k9 e A-n80-k10, respectivamente. As barras em verde representam as execuções que utilizaram o filtro de Bloom para não reexecutar o *VND* em soluções duplicadas geradas pelo algoritmo construtivo randomizado (*GreedyRandomizedSolution*) e as barras em azul representam as execuções sem o filtro de Bloom, executando então o algoritmo como proposto originalmente na literatura e reaplicando o *VND* em soluções duplicadas geradas pelo algoritmo construtivo randomizado, caso aconteçam. O eixo x representa os valores de $\alpha \in [0.01, 0.1]$ do *GRASP*. O eixo y apresenta a média aritmética de 10 execuções do tempo em segundos do algoritmo, ao executar todas as iterações, definidas por *max_iterations* = 5000.

Na Figura 2a, referente a instância A-n32-k5, os parâmetros $\alpha = \{0.01, 0.02, 0.03\}$ foram equivalentes ao algoritmo guloso, o ganho de tempo foi de aproximadamente 99.88% do tempo sem utilizar o filtro. A partir de $\alpha = 0.04$ soluções diversificadas passaram a ser geradas. Em $\alpha \in [0.04, 0.08]$ o ganho em tempo foi de 0.87% até 90%. Para $\alpha = \{0.09, 0.1\}$ houve um custo de tempo adicional de até 0.11% ao utilizar o filtro. Na Figura 2b, representando a instância A-n60-k9, apenas o parâmetro $\alpha = 0.01$ foi equivalente ao algoritmo guloso, tenho ganho de tempo ao utilizar o filtro até o parâmetro $\alpha = 0.05$, após este parâmetro, houve um custo adicional ao tempo sem filtro do algoritmo, ao utilizar o filtro de Bloom. Na Figura 2c, representando a instância A-n80-k10, o parâmetro $\alpha = 0.01$ foi equivalente ao algoritmo guloso, com ganho de 99.5% de tempo. Os demais valores de α nesta instância não apresentaram ganho expressivo de tempo por baixa quantidade de soluções duplicadas em 5mil soluções geradas.

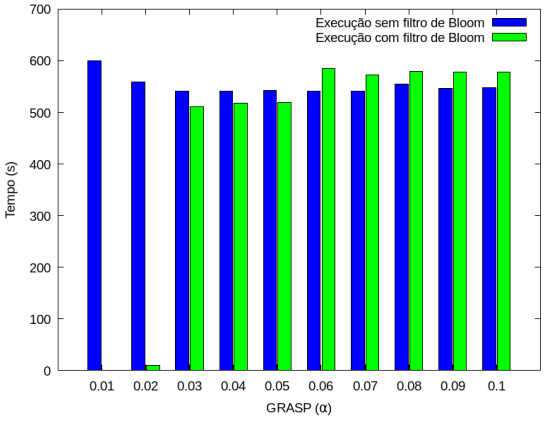
Como apresentado na seção 2, a probabilidade de falsos positivos do filtro de Bloom aumenta à medida que mais elementos são inseridos no filtro, desta forma, para testar os falso positivos executamos o algoritmo com 50 mil iterações como critério de parada, nas mesmas 3 instâncias selecionadas do pacote. A análise de duplicatas do algoritmo e falsos positivos foi conduzida e está apresentada nas Tabelas 1,2,3. A primeira coluna (α) apresenta os valores do parâmetro α utilizado na execução. A segunda coluna apresenta o número de duplicatas geradas pelo algoritmo. A terceira coluna apresenta o número de duplicatas reportadas pelo filtro de Bloom. A quarta e última coluna apresenta a porcentagem de falsos positivos reportados pelo filtro de Bloom.

A Tabela 1 apresenta o relatório de duplicatas para a instância A-n32-k5. Dado o tamanho da instância, os valores de $\alpha \in [0.01, 0.03]$ foram equivalente ao algoritmo guloso. Neste caso, houve 0% de falsos positivos. A partir de α 0.04 houve aleatoriedade nas soluções geradas, porém houveram 49948 soluções geradas como duplicatas, com 0% falsos positivos reportados. Nos demais valores de α, podemos ver que o número de duplicatas reais cai expressivamente (tendo

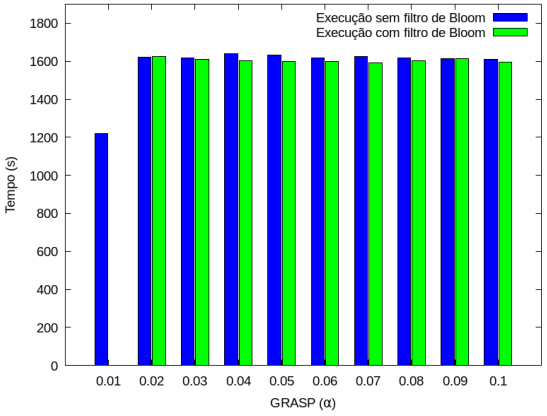
DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en



(a) Execução com a instância A-n32-k5



(b) Execução com a instância A-n60-k9



(c) Execução com a instância A-n80-k10

apenas 250 duplicatas reais). Para essa instância, com 50 mil soluções geradas pelo algoritmo, a maior quantidade de falsos positivos foi 2 (0.004%) no parâmetro $\alpha = 0.08$.

A Tabela 2 apresenta o relatório de duplicatas para a instância A-n60-k9. Nesta instância, apenas o parâmetro $\alpha = 0.01$ foi equivalente ao algoritmo guloso. O algoritmo gerou duplicatas apenas até o valor $\alpha = 0.04$, tendo 0 duplicatas a partir de $\alpha = 0.05$. O maior número de falsos positivos foi 1 (0.002% das soluções geradas). A Tabela 3 apresenta o relatório de duplicatas para a instância A-n80-k10. Nesta instância, apenas o parâmetro $\alpha = 0.01$ foi equivalente ao algoritmo guloso. O algoritmo gerou duplicatas apenas até o valor $\alpha = 0.02$, tendo 0 duplicatas a partir de $\alpha = 0.03$. O maior número de falsos positivos foi 2 (0.004% das soluções geradas).

Em resumo, o filtro de Bloom apresentou baixo número de falsos positivos, tendo no máximo 0.006% de falsos positivos nos experimentos reportados, utilizando um baixo consumo de memória (1.25 megabytes). Ao mesmo tempo, conseguiu reduzir o tempo das execuções de todos os parâmetros $\alpha \in [0.01, 0.05]$ executados (com exceção de $\alpha = 0.02$ na instância A-n80-k10), se apresentando como uma estratégia eficiente para impedir o recálculo de rotinas custosas como as estratégias de busca local, em soluções que já foram exploradas.

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

α	Duplicatas reais	Duplicatas identificadas	Falsos positivos (%)
0.01	49999	49999	0%
0.02	49999	49999	0%
0.03	49999	49999	0%
0.04	49948	49948	0%
0.05	49584	49584	0%
0.06	43487	43487	0%
0.07	22258	22258	0%
0.08	3664	3666	0.004%
0.09	1261	1262	0.002%
0.1	250	251	0.002%

Tabela 1: Duplicatas na A-n32-k5.

α	Duplicatas reais	Duplicatas identificadas	Falsos positivos (%)
0.01	49999	49999	0%
0.02	49897	49897	0%
0.03	6458	6459	0.002%
0.04	7	7	0%
0.05	0	1	0.002%
0.06	0	1	0.002%
0.07	0	0	0%
0.08	0	1	0.002%
0.09	0	1	0.002%

Tabela 2: Duplicatas na A-n60-k9.

α	Duplicatas reais	Duplicatas identificadas	Falsos positivos (%)
0.01	49999	49999	0%
0.02	414	417	0.006%
0.03	0	1	0.002%
0.04	0	1	0.002%
0.05	0	1	0.002%
0.06	0	1	0.002%
0.07	0	2	0.004%
0.08	0	2	0.004%
0.09	0	1	0.002%

Tabela 3: Duplicatas na instância A-n80-k10.

7. Conclusão

Este trabalho introduz o uso do filtro de Bloom e estuda o comportamento ao aplicá-lo para evitar recálculo de rotinas custosas em metaheurísticas. Para conduzir esta análise, foi escolhido a meta-heurística *GRASP* com *VND* para solucionar o *CVRP*.

O filtro de Bloom, vastamente utilizado no contexto de cache Web, se demonstrou também muito proveitoso nesta aplicação. Reduzindo expressivamente o tempo em até 99.88% em casos extremos ($\alpha = 0.01$) e com baixo consumo de memória e custo computacional. Para $\alpha = 0.05$, um parâmetro bem usado do *GRASP* e que também pode gerar muitas duplicatas, o ganho de tempo ao utilizar o filtro chegou a 93.03% na instância A-n32-k5. O filtro se apresentou como uma ferramenta muito eficaz para buscas de duplicidade no espaço exponencial de soluções, com custo O(1) e baixíssimos números falsos positivos (a execução com maior quantidade de falsos positivos possuiu 0.006% de falsos positivos em 50mil soluções armazenadas, utilizando filtro de tamanho de 1.25 megabytes). O filtro fornece à meta-heurística uma pesquisa de duplicidade das solução gerada, com baixo custo de tempo e armazenamento, em um cenário que seria inviável armazenar e analisar as soluções por comparação, pelo alto custo computacional e inviabilidade da tratativa por exaustão (analisar todas as soluções armazenadas) do espaço exponencial.

Para trabalhos futuros, o comportamento do filtro pode ser analisado em outras metaheurísticas, como metaheurísticas populacionais, evolucionárias e de solução única. O comportamento do filtro também pode ser estudado ao aplicar no *VND* e em outras estratégias de vizinhanças, para evitar retrabalho dos movimentos.

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

Referências

Alexander, H., Khalil, I., Cameron, C., Tari, Z., e Zomaya, A. (2014). Cooperative web caching using dynamic interest-tagged filtered bloom filters. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):2956–2969.

Arulanand, N. e Premalatha, K. (2014). Bin bloom filter using heuristic optimization techniques for spam detection. *International Journal of Computer and Information Engineering*, 8(8):1472–1478.

Augerat, P., Naddef, D., Belenguer, J., Benavent, E., Corberan, A., e Rinaldi, G. (1995). Computational results with a branch and cut code for the capacitated vehicle routing problem.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

Braekers, K., Ramaekers, K., e Van Nieuwenhuyse, I. (2016). The vehicle routing problem: State of the art classification and review. *Computers & Industrial Engineering*, 99:300–313.

Bruglieri, M., Ferone, D., Festa, P., e Pisacane, O. (2022). A grasp with penalty objective function for the green vehicle routing problem with private capacitated stations. *Computers & Operations Research*, 143:105770.

Chen, A.-l., Yang, G.-k., e Wu, Z.-m. (2006). Hybrid discrete particle swarm optimization algorithm for capacitated vehicle routing problem. *Journal of Zhejiang University-Science A*, 7(4):607–614.

Clarke, G. e Wright, J. W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581.

Coelho, I. M., Munhoz, P. L. A., Haddad, M. N., Coelho, V. N., de Melo Silva, M., Souza, M. J. F., e Ochi, L. S. (2011). Optframe: a computational framework for combinatorial optimization problems. In *Agra, Agostinho and Doostmohammadi, Mahdi (2011) A Polyhedral Study of Mixed 0-1 Set. In: Proceedings of the 7th ALIO/EURO Workshop. ALIO-EURO 2011, Porto, pp. 57-59.*, p. 51.

CVRPLIB (2021). *Capacitated Vehicle Routing Problem Library*. `http://vrp.atd-lab.inf.puc-rio.br`. [Online; accessed September-2022].

Dantzig, G. B. e Ramser, J. H. (1959). The truck dispatching problem. *Management science*, 6(1): 80–91.

de Souza, I. P., Boeres, M. C. S., de Moraes, R. E. N., e Thompson, J. V. C. (2022). A reactive grasp algorithm for the multi-depot vehicle routing problem. In *International Conference on Computational Science and Its Applications*, p. 81–96. Springer.

Fan, L., Cao, P., Almeida, J., e Broder, A. Z. (2000). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293.

Feo, T. A. e Resende, M. G. (1995). Greedy randomized adaptive search procedures. *Journal of global optimization*, 6:109–133.

Gibson, S., Issac, B., Zhang, L., e Jacob, S. M. (2020). Detecting spam email with machine learning optimized with bio-inspired metaheuristic algorithms. *Ieee Access*, 8:187914–187932.

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en

Guo, D., Wu, J., Chen, H., Yuan, Y., e Luo, X. (2009). The dynamic bloom filters. *IEEE Transactions on knowledge and data engineering*, 22(1):120–133.

Huang, S.-H., Huang, Y.-H., Blazquez, C. A., e Chen, C.-Y. (2022). Solving the vehicle routing problem with drone for delivery services using an ant colony optimization algorithm. *Advanced Engineering Informatics*, 51:101536.

İlhan, İ. (2021). An improved simulated annealing algorithm with crossover operator for capacitated vehicle routing problem. *Swarm and Evolutionary Computation*, 64:100911.

Kyriakakis, N. A., Aronis, S., Marinaki, M., e Marinakis, Y. (2023). A grasp/vnd algorithm for the energy minimizing drone routing problem with pickups and deliveries. *Computers & Industrial Engineering*, 182:109340.

Lenstra, J. K. e Kan, A. R. (1981). Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227.

Li, Y., Miao, R., Kim, C., e Yu, M. (2016). {FlowRadar}: A better {NetFlow} for data centers. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, p. 311–324.

Luo, L., Guo, D., Wu, J., Rottenstreich, O., He, Q., Qin, Y., e Luo, X. (2016). Efficient multiset synchronization. *IEEE/ACM Transactions on Networking*, 25(2):1190–1205.

Mitzenmacher, M. (2001). Compressed bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, p. 144–150.

Mohammed, M. A., Abd Ghani, M. K., Hamed, R. I., Mostafa, S. A., Ahmad, M. S., e Ibrahim, D. A. (2017). Solving vehicle routing problem by using improved genetic algorithm for optimal solution. *Journal of computational science*, 21:255–262.

Natarajan, A., Subramanian, S., e Premalatha, K. (2012). A comparative study of cuckoo search and bat algorithm for bloom filter optimisation in spam filtering. *International Journal of Bio-Inspired Computation*, 4(2):89–99.

Potvin, J.-Y. e Gendreau, M. (2018). *Handbook of Metaheuristics*. Springer.

Rottenstreich, O. e Keslassy, I. (2014). The bloom paradox: When not to use a bloom filter. *IEEE/ACM Transactions on Networking*, 23(3):703–716.

Souza, I. P., Boeres, M. C. S., e Moraes, R. E. N. (2023). A robust algorithm based on differential evolution with local search for the capacitated vehicle routing problem. *Swarm and Evolutionary Computation*, 77:101245.

Tarkoma, S., Rothenberg, C. E., e Lagerspetz, E. (2011). Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155.

DOI: 10.59254/sbpo-2024-193325
https://proceedings.science/p/193325?lang=en