

Abordagens Heurísticas para o Problema de Escalonamento de Contêineres em Orquestradores Kubernetes

Camilla Martins, Pedro Nuno de S. Moura, Adriana C. F. Alvim, Sidney C. de Lucena

Programa de Pós-Graduação em Informática (PPGI)

Universidade Federal do Estado do Rio de Janeiro (UNIRIO)

Av. Pasteur 458, Urca – Rio de Janeiro, RJ – Brasil

camilla.martins@edu.unirio.br, {pedro.moura, adriana,
sidney}@uniriotec.br

RESUMO

Este trabalho propõe heurísticas para o problema de escalonamento de contêineres em orquestradores Kubernetes, modelado como problema de Programação Inteira inspirado no *Capacitated Facility Location Problem* (CFLP). A proposta visa minimizar custos de infraestrutura considerando custo de abertura de nós, alocação de *pods* e penalização por erros. Foram desenvolvidas uma heurística construtiva gulosa, busca local de melhor melhoria e meta-heurística GRASP. Experimentos demonstraram reduções de custo superiores a 90% comparadas ao Kube-scheduler em instâncias com 5000 *pods* e 200 nós, com tempos computacionais menores que métodos exatos.

PALAVRAS CHAVE. Kubernetes. Escalonamento de Contêineres. Heurísticas. Meta-heurísticas. Facility Location Problem.

Tópicos (Outras Aplicações em PO, Meta-heurísticas)

ABSTRACT

This work proposes heuristics for the container scheduling problem in Kubernetes orchestrators, modeled as an Integer Programming problem inspired by the Capacitated Facility Location Problem (CFLP). The proposal aims to minimize infrastructure costs by considering the cost of opening nodes, allocating pods, and error penalties. A greedy constructive heuristic, a best-improvement local search, and a GRASP metaheuristic were developed. Experiments found cost reductions of over 90% compared to Kube-scheduler on instances with 5000 pods and 200 nodes, with computational times lower than exact methods.

KEYWORDS. Kubernetes. Container Scheduling. Heuristics. Metaheuristics. Facility Location Problem.

Topics (Other Applications in PO, Metaheuristics)

1. Introdução

A computação em nuvem transformou o cenário de Tecnologia da Informação (TI) proporcionando elasticidade, escalabilidade e redução de custos operacionais. Uma das técnicas fundamentais nesse contexto é a virtualização baseada em contêineres, que permite agrupar códigos de aplicações e suas dependências, possibilitando execução rápida e confiável em diferentes ambientes computacionais. Nesse cenário, a ferramenta Kubernetes [Kubernetes, 2025b] consolidou-se como o principal orquestrador de contêineres do mercado, responsável por automatizar a implantação, o dimensionamento e o gerenciamento desses contêineres.

No ambiente Kubernetes, os contêineres são agrupados em unidades chamadas *pods*, que são distribuídos em máquinas virtuais ou físicas, denominadas nós, formando um cluster. O escalonamento adequado desses *pods* nos nós disponíveis representa um desafio significativo de otimização, pois impacta diretamente no custo e desempenho da infraestrutura em nuvem. O componente responsável por esta alocação é o Kube-scheduler [Kubernetes, 2025a], que utiliza algoritmos predefinidos para determinar em qual nó cada *pod* deve ser executado.

Embora o Kube-scheduler padrão seja eficiente para muitos cenários, ele não foi projetado para otimizar explicitamente a relação custo/desempenho da infraestrutura. Em trabalho anterior [Martins et al., 2023], foi proposta uma abordagem baseada em Programação Inteira (PI) para o problema de escalonamento de contêineres, modelando-o como uma variante do *Capacitated Facility Location Problem* (CFLP). Essa abordagem demonstrou resultados significativamente melhores em termos de custo da solução quando comparada ao algoritmo padrão do Kubernetes. No entanto, por utilizar métodos exatos de otimização, o tempo computacional necessário para resolver instâncias de entrada de grande porte (grande número de nós e *pods* a serem agendados no *cluster*, por exemplo 10000 *pods* e 200 nós, onde exigem alta demanda performática para suas alocações) pode ser proibitivo para aplicações em tempo real.

Neste contexto, o presente trabalho propõe e avalia o emprego de abordagens heurísticas para o problema de escalonamento de contêineres em orquestradores Kubernetes, visando obter soluções de boa qualidade em tempo computacional reduzido. São desenvolvidas uma heurística construtiva, uma busca local e uma meta-heurística baseada em GRASP [Feo e Resende, 1995], explorando estruturas de vizinhança apropriadas para o problema.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica e trabalhos relacionados; a Seção 3 detalha a formulação do problema; a Seção 4 descreve as heurísticas propostas; a Seção 5 apresenta os experimentos computacionais realizados; a Seção 6 apresenta os resultados obtidos e discute os resultados; e, finalmente, a Seção 7 apresenta as conclusões e direções para trabalhos futuros.

2. Fundamentação Teórica e Trabalhos Relacionados

2.1. Kubernetes e o Escalonamento de Contêineres

O Kubernetes é um sistema de código aberto para automatizar a implantação, o dimensionamento e o gerenciamento de aplicações em contêineres [Kubernetes, 2023]. Sua arquitetura é composta por um conjunto de nós (máquinas virtuais ou físicas) organizados em um *cluster*. Os contêineres são encapsulados em *pods*, a menor unidade gerenciável no Kubernetes, que podem conter um ou mais contêineres.

O escalonamento de *pods* nos nós do *cluster* é realizado pelo Kube-scheduler, um componente central que determina em qual nó cada *pod* deve ser executado. O algoritmo padrão opera em duas fases: filtragem (identifica nós viáveis considerando recursos disponíveis e políticas de afinidade) e pontuação (classifica nós filtrados por critérios como utilização de recursos e balanceamento de carga) [Scheduler, 2023].

Embora o Kube-scheduler ofereça várias configurações como *NodeAffinity*, *TaintToleration* e *PodTopologySpread*, ele não foi projetado para minimizar explicitamente custos operacionais da infraestrutura em nuvem, motivando pesquisas por algoritmos mais eficientes em termos de custos.

2.2. Capacitated Facility Location Problem (CFLP) e Programação Inteira

A Programação Inteira (PI) é uma área da otimização matemática que lida com problemas onde algumas ou todas as variáveis de decisão devem assumir valores inteiros. Estes problemas são geralmente mais complexos computacionalmente que seus equivalentes com variáveis contínuas, frequentemente pertencendo à classe $\mathcal{NP}$ -difíceis. A PI utiliza técnicas como *branch-and-bound*, *cutting planes* e métodos heurísticos para encontrar soluções ótimas ou próximas do ótimo [Wolsey, 1998].

O Capacitated Facility Location Problem (CFLP) é um problema clássico de otimização combinatória que busca determinar quais instalações devem ser abertas e como atender a demanda de clientes de forma a minimizar o custo total, considerando capacidade limitada das instalações [Cornuéjols et al., 1983]. Formalmente, o CFLP pode ser definido como um problema onde deve-se: (i) selecionar um subconjunto de instalações para abrir, cada uma com custo fixo de abertura e capacidade limitada; (ii) determinar a alocação de cada cliente a uma instalação aberta, respeitando as capacidades; (iii) minimizar o custo total, incluindo custos fixos de abertura e custos variáveis de atendimento. Este problema é $\mathcal{NP}$ -difícil [Cornuéjols et al., 1990], justificando o uso de heurísticas para instâncias de grande porte. Tem amplas aplicações em localização de fábricas, armazéns, hospitais [Klose e Drex1, 2005].

2.3. Trabalhos Relacionados

Diversos trabalhos na literatura abordam o problema de escalonamento de contêineres em ambientes de nuvem. Zhang et al. [Zhang et al., 2017] propuseram um modelo de Programação Linear para escalonamento de contêineres no Docker Swarm, focando na redução do consumo energético. Zhou et al. [Zhou et al., 2018] desenvolveram uma estrutura de escalonamento baseada em Programação Inteira Mista (MIP) para computação em nuvem com contêineres, considerando a capacidade de utilização dos recursos.

Especificamente para o Kubernetes, Wojciechowski et al. [Wojciechowski et al., 2021] desenvolveram um plugin inspirado em Programação Linear que estende o Kube-scheduler padrão, verificando a capacidade dos nós para suportar novos *pods* segundo métricas de desempenho. Zhong e Buyya [Zhong e Buyya, 2020] propuseram uma estratégia de escalonamento heterogênea em *clusters* Kubernetes, objetivando redução de custos através do redimensionamento de tarefas, elasticidade do *cluster* e encerramento de nós subutilizados.

Embora existam diversos trabalhos abordando escalonamento de contêineres, poucos exploram especificamente heurísticas para o problema modelado como CFLP no contexto do Kubernetes. Por exemplo, Pham et al. [Pham et al., 2020] propuseram um algoritmo heurístico para escalonamento de contêineres com foco em eficiência energética, mas sem modelar o problema como CFLP. Kaur et al. [Kaur e Sachdeva, 2022] desenvolveram um framework para escalonamento sensível ao contexto em ambientes de IoT industrial usando Kubernetes, porém utilizando técnicas baseadas em aprendizado de máquina. Nosso trabalho se diferencia ao propor heurísticas específicas para o CFLP aplicado ao Kubernetes, considerando tanto a estrutura do problema quanto as particularidades do ambiente de orquestração de contêineres.

3. Formulação do Problema

Baseando-se no trabalho de Martins et al. [2023], o problema de escalonamento de contêineres em orquestradores Kubernetes pode ser modelado como uma variante do Capacitated Facility Location Problem (CFLP). Considera-se um conjunto de nós $N = \{n_1, n_2, \dots, n_m\}$ associados

a um cluster Kubernetes e um conjunto de *pods* $P = \{p_1, p_2, \dots, p_n\}$ que precisam ser alocados. Cada nó possui capacidade limitada de recursos (CPU, memória, etc.) e cada *pod* demanda determinada quantidade desses recursos. A alocação envolve decidir quais nós serão abertos e quais *pods* serão alocados a cada nó, com o objetivo de minimizar o custo total.

A função objetivo considera três componentes de custo, como o custo de abertura de nós (α_i), que é o custo associado à ativação de um nó i , o custo de alocação de *pod* a nó (β_i), que está relacionado ao custo para alocar um *pod* qualquer ao nó i e, por fim, o custo de penalidade por erros (γ_i), que é a penalização aplicada a um nó i pela quantidade de erros registrados nos *pods* alocados a ele.

A formulação de Programação Inteira é dada por:

$$\min \sum_{i=1}^n \left(\alpha_i x_i + \beta_i \sum_{j=1}^m y_{i,j} + \gamma_i \sum_{j=1}^m e_j y_{i,j} \right) \quad (1)$$

Sujeito a:

$$\sum_{i=1}^n x_i \geq 1 \quad (1.1)$$

$$y_{i,j} \leq x_i \quad i = 1, \dots, n, \quad j = 1, \dots, m \quad (1.2)$$

$$\sum_{j=1}^m y_{i,j} \geq 1 \quad i = 1, \dots, n \quad (1.3)$$

$$\sum_{j=1}^m u_j y_{i,j} \leq U_i \quad i = 1, \dots, n \quad (1.4)$$

$$x_i \in \{0, 1\} \quad i = 1, \dots, n \quad (1.5)$$

$$y_{i,j} \in \{0, 1\} \quad i = 1, \dots, n, \quad j = 1, \dots, m \quad (1.6)$$

onde x_i é uma variável binária que indica se o nó i está aberto (1) ou não (0); $y_{i,j}$ é uma variável binária que indica se o *pod* j está alocado ao nó i (1) ou não (0); u_j representa a quantidade de recursos utilizados pelo *pod* j ; U_i representa a capacidade de recursos do nó i ; e, por fim, e_j representa a quantidade de erros registrados no *pod* j .

A restrição (1.1) garante que pelo menos um nó esteja aberto. A restrição (1.2) assegura que um *pod* só pode ser alocado a um nó aberto. A restrição (1.3) define que cada nó aberto deve receber pelo menos um *pod*. A restrição (1.4) assegura que a quantidade de recursos utilizados pelos *pods* alocados a um nó não exceda sua capacidade. As restrições (1.5) e (1.6) definem o domínio das variáveis de decisão.

4. Heurísticas Propostas

Para resolver o problema de escalonamento de contêineres em Kubernetes modelado como CFLP, propomos três abordagens heurísticas complementares: uma heurística construtiva gulosa, procedimentos de busca local e uma meta-heurística GRASP. Esta estratégia multicamadas visa obter soluções de boa qualidade em tempo computacional reduzido, sendo mais adequada para aplicações práticas que os métodos exatos de otimização.

4.1. Representação da Solução

Uma solução para o problema é representada por duas estruturas principais, sendo a primeira um vetor binário indicando quais nós estão abertos e a segunda um vetor de alocação indicando a qual nó cada *pod* está designado.

Formalmente, seja b um vetor de tamanho n onde $b_i = 1$ se o nó i está aberto e $b_i = 0$ caso contrário. Seja a um vetor de tamanho m onde $a_j = i$ se o *pod* j está alocado ao nó i , ou $a_j = -1$ se o *pod* j não está alocado a nenhum nó.

O custo total de uma solução (b, a) é calculado como:

$$Custo(b, a) = \sum_{i=1}^n \alpha_i b_i + \sum_{j=1}^m \beta_{a_j} + \sum_{j=1}^m \gamma_{a_j} e_j \quad (2)$$

onde α_i representa o custo de abertura do nó i , β_{a_j} representa o custo de alocação do *pod* j ao nó a_j , γ_{a_j} representa o fator de penalidade por erros no nó a_j , e e_j representa a quantidade de erros registrados no *pod* j .

4.2. Heurística Construtiva Gulosa

A heurística construtiva proposta segue uma abordagem gulosa baseada no princípio de tomada de decisões localmente ótimas. Esta estratégia constrói uma solução de forma incremental, onde a cada passo seleciona-se a opção que minimiza o custo imediato sem considerar consequências futuras. O algoritmo implementa uma estratégia de construção sequencial onde cada *pod* é alocado ao nó que resulta no menor custo adicional, respeitando restrições de capacidade. Esta abordagem é particularmente eficaz para o CFLP pois explora a estrutura do problema onde decisões de abertura de nós e alocação de *pods* são interdependentes. O Algoritmo 1 descreve o pseudocódigo da heurística.

Algoritmo: 1: Heurística Construtiva Gulosa

Require: Conjunto de nós N , conjunto de *pods* P , custos α, β, γ , capacidades U , uso de recursos u , erros e

Ensure: Solução viável (b, a)

```

1: Inicializar  $b[i] \leftarrow 0$  para todo  $i \in N$ 
2: Inicializar  $a[j] \leftarrow -1$  para todo  $j \in P$ 
3: Ordenar nós  $N$  por custo de abertura  $\alpha$  crescente
4: for cada pod  $j \in P$  do
5:    $melhorCusto \leftarrow \infty$ 
6:    $melhorNo \leftarrow -1$ 
7:   for cada nó  $i \in N$  do
8:     if temCapacidadeSuficiente( $i, j, u_i, U_i, a$ ) then
9:        $custo \leftarrow \text{calcularCustoAlocacao}(i, j, \alpha, \beta, \gamma, e, b)$ 
10:      if  $custo < melhorCusto$  then
11:         $melhorCusto \leftarrow custo$ 
12:         $melhorNo \leftarrow i$ 
13:      end if
14:    end if
15:  end for
16:  if  $melhorNo \neq -1$  then
17:     $a[j] \leftarrow melhorNo$ 
18:     $b[melhorNo] \leftarrow 1$ 
19:  end if
20: end for
21: return  $(b, a)$ 

```

A função $\text{temCapacidadeSuficiente}(i, j, u_i, U_i, a)$ verifica se o nó i tem capacidade disponível para acomodar o $pod\ j$, considerando os $pods$ já alocados a este nó. A função $\text{calcularCustoAlocacao}(i, j, \alpha, \beta, \gamma, e, b)$ calcula o custo de alocar o $pod\ j$ ao nó i , considerando o custo de abertura do nó (caso ainda não esteja aberto), o custo de alocação e a penalidade por erros. Este cálculo representa uma parcela do custo total definido em $\text{Custo}(b, a)$. A complexidade deste procedimento é $O(1)$, mas é chamado $O(n \cdot m)$ vezes durante a execução completa da heurística construtiva.

A heurística construtiva começa ordenando os nós pelo custo de abertura (linha 3). Depois, para cada pod , procura o nó que resulte no menor custo adicional para alocação, respeitando as restrições de capacidade (linhas 4-20). Se encontrar um nó viável, aloca o pod a esse nó e marca o nó como aberto (linhas 17-18).

4.3. Busca Local

A partir da solução inicial gerada pela heurística construtiva, aplicam-se procedimentos de busca local para melhorar a solução. A busca local é uma técnica de refinamento que explora sistematicamente o espaço de soluções vizinhas à solução atual, buscando melhorias incrementais. Esta abordagem é fundamentada no conceito de otimalidade local, onde uma solução é considerada localmente ótima se nenhum movimento na vizinhança definida resulta em melhoria.

4.3.1. Estrutura de Vizinhança

A vizinhança de uma solução é definida pelo conjunto de soluções que podem ser obtidas movendo-se um pod de seu nó atual para outro nó. Formalmente, dada uma solução (b, a) , um vizinho é uma solução (b', a') obtida alterando-se a alocação de um único $pod\ j$ de $a_j = i$ para $a'_j = k$, com $i \neq k$. Os vetores b e b' são atualizados adequadamente, considerando-se a abertura de novos nós ou o fechamento de nós que fiquem vazios.

4.3.2. Busca Local com Melhor Melhoria

A estratégia de melhor melhoria implementa uma exploração exaustiva da vizinhança, avaliando todos os movimentos possíveis antes de selecionar aquele que resulta na maior redução de custo. Esta abordagem, embora computacionalmente mais intensiva que estratégias de primeira melhoria, tende a produzir soluções de melhor qualidade ao evitar decisões míopes. O critério de parada adotado corresponde à convergência a um ótimo local, onde nenhum movimento na vizinhança resulta em redução de custo. O Algoritmo 2 descreve esta abordagem.

Algoritmo: 2: Busca Local com Melhor Melhoria

Require: Solução inicial (b, a) , parâmetros do problema

Ensure: Solução localmente ótima (b, a)

```

1:  $\text{melhorou} \leftarrow \text{verdadeiro}$   $\text{melhorou}$ 
2:  $\text{melhorou} \leftarrow \text{falso}$ 
3:  $\text{melhorDeltaCusto} \leftarrow 0$ 
4:  $\text{melhorPod} \leftarrow -1$ 
5:  $\text{melhorNoDestino} \leftarrow -1$ 
6: for cada  $pod\ j \in P$  do
7:    $\text{noAtual} \leftarrow a[j]$ 
8:   for cada nó  $i \in N, i \neq \text{noAtual}$  do
9:     if  $\text{temCapacidadeSuficiente}(i, j, u_i, U_i, a)$  then
10:       $\text{deltaCusto} \leftarrow \text{calcularVariacaoCusto}(j, \text{noAtual}, i, \alpha, \beta, \gamma, e, b, a)$ 
11:      if  $\text{deltaCusto} < \text{melhorDeltaCusto}$  then
12:         $\text{melhorDeltaCusto} \leftarrow \text{deltaCusto}$ 
13:         $\text{melhorPod} \leftarrow j$ 

```

```

14:          $melhorNoDestino \leftarrow i$ 
15:     end if
16: end if
17: end for
18: end for
19: if  $melhorDeltaCusto < 0$  then
20:      $melhorou \leftarrow verdadeiro$ 
21:      $a[melhorPod] \leftarrow melhorNoDestino$ 
22:     Atualizar  $b$  conforme necessário
23: end if return  $(b, a)$ 

```

A complexidade de uma iteração completa da Busca Local com Melhor Melhoria é $O(n \cdot m)$, onde n é o número de nós e m é o número de pods. Isto ocorre porque exploram-se $O(n \cdot m)$ vizinhos potenciais (cada pod pode ser potencialmente realocado a cada um dos outros nós) e, para cada movimento candidato, o cálculo da variação de custo possui complexidade $O(1)$ para as operações aritméticas básicas. O algoritmo converge para um ótimo local após um número de iterações que, no pior caso, é limitado por $O(m)$, resultando em uma complexidade total de $O(n \cdot m^2)$ no pior caso, embora na prática o número de iterações tenda a ser significativamente menor.

4.4. Meta-heurística GRASP

Para explorar mais eficientemente o espaço de soluções e escapar de ótimos locais, implementou-se a meta-heurística GRASP (*Greedy Randomized Adaptive Search Procedure*). O GRASP é uma meta-heurística multi-início que combina construção gulosa aleatorizada com busca local intensiva. A fase construtiva utiliza uma função gulosa aleatorizada para construir soluções iniciais diversificadas, enquanto a fase de busca local refina cada solução construída. Esta dualidade permite ao GRASP balancear intensificação (através da busca local) e diversificação (através da construção aleatorizada), características essenciais para escapar de ótimos locais e explorar diferentes regiões do espaço de soluções.

O GRASP é particularmente adequado para o problema em questão, pois combina a construção gulosa de soluções iniciais com uma fase de busca local, permitindo diversificar as soluções através de seu componente aleatório. A estratégia de Lista Restrita de Candidatos (RCL) permite controlar o *trade-off* entre qualidade e diversidade das soluções iniciais através do parâmetro α .

O Algoritmo 3 apresenta o pseudocódigo do GRASP proposto.

Algoritmo: 3: GRASP para o Problema de Escalonamento de Contêineres

Require: Parâmetros do problema, número máximo de iterações $maxIter$, parâmetro de aleatoriedade α

Ensure: Melhor solução encontrada (b_{best}, a_{best})

```

1:  $(b_{best}, a_{best}) \leftarrow \text{null}$ 
2:  $custo_{best} \leftarrow \infty$ 
3: for  $iter = 1$  até  $maxIter$  do
4:      $(b, a) \leftarrow \text{ConstrucaoAleatorizadaGulosa}(N, P, \alpha)$ 
5:      $(b', a') \leftarrow \text{BuscaLocal}(b, a)$ 
6:     if  $\text{calcularCusto}(b', a') < custo_{best}$  then
7:          $(b_{best}, a_{best}) \leftarrow (b', a')$ 
8:          $custo_{best} \leftarrow \text{calcularCusto}(b_{best}, a_{best})$ 

```

9: **end if**

10: **end forreturn** (b_{best}, a_{best})

A função *ConstrucaoAleatorizadaGulosa*(N, P, α) é uma variante da heurística construtiva gulosa que, em vez de escolher sempre o nó de menor custo para cada pod, cria uma lista restrita de candidatos (RCL) contendo os $\alpha\%$ melhores nós e seleciona aleatoriamente um deles. Isso introduz um componente de aleatoriedade controlada que permite explorar diferentes regiões do espaço de soluções.

O GRASP é particularmente adequado para este problema por várias razões: oferece um bom equilíbrio entre intensificação (através da busca local) e diversificação (através da construção aleatorizada); possui um parâmetro de ajuste da aleatoriedade (α) que permite controlar o *trade-off* entre qualidade e diversidade das soluções iniciais; é naturalmente paralelizável, uma vez que cada iteração é independente; não requer ajustes complexos de parâmetros como outras meta-heurísticas; e tem demonstrado excelente desempenho em problemas de localização de instalações. O parâmetro `maxIterations = 10` oferece um compromisso razoável entre encontrar boas soluções e manter um tempo de execução aceitável.

4.5. Implementação e Disponibilidade do Código

As heurísticas e meta-heurísticas propostas foram implementadas em linguagem Java. Todo o código-fonte, incluindo as implementações da heurística construtiva, busca local, GRASP, além dos códigos para geração de instâncias e análise de resultados, estão disponíveis em um repositório público no GitHub¹.

5. Experimentos Computacionais

Para avaliar o desempenho das heurísticas e meta-heurísticas propostas, foram realizados experimentos computacionais comparando-as entre si e com a abordagem exata baseada em Programação Inteira proposta por Martins et al. [2023], bem como com o algoritmo criado para simular o agendador padrão do Kube-scheduler.

5.1. Ambiente Computacional

Os experimentos foram realizados em um notebook Apple com processador Apple M2 Pro (12 núcleos), 16 GB de memória RAM e sistema operacional MacOS Sequoia 15.4.1. As implementações foram feitas em linguagem de programação Java (versão 22.0.2). Para executar a formulação de PI, foi utilizada a versão 12.0.1 do resolvidor Gurobi a partir de sua API para Java [Gurobi Optimization, 2025a], adotando-se como critério de parada o parâmetro `MIPGap` configurado em 0.01 [Gurobi Optimization, 2025b].

5.2. Instâncias de Teste

Foram geradas instâncias artificiais com diferentes números de nós e pods, considerando os seguintes pares: nós $N = \{10, 20, 50, 100, 200\}$ e pods $P = \{50, 100, 200, 500, 1000, 5000, 10000\}$.

Para cada instância, foram sorteados valores para os parâmetros de capacidade dos nós (U_i), uso de recursos dos pods (u_j), custos de abertura de nós (α_i), custos de alocação (β_i), taxa de penalidade (γ_i) e quantidade de erros por pod (e_j). Para a capacidade dos nós (U_i), houve uma distribuição uniforme entre $\frac{|P|}{|N|} + 1$ e $2|P|$, onde $|P|$ é o número de pods e $|N|$ é o número de nós. Para o uso de recursos dos pods (u_j), uma distribuição uniforme entre 1 e 10. Em relação aos custos de abertura de nós (α_i), uma distribuição uniforme entre 1 e $4|N|$. E por fim, para os custos de alocação (β_i), uma distribuição uniforme entre 1 e $4|N|$. Para cada instância e algoritmo, realizaram-se 10 execuções independentes, registrando-se o valor médio da solução (custo total) e o tempo médio de execução em milissegundos. A fim de garantir a reprodutibilidade dos experimentos, foi adotada uma semente geradora de números aleatórios de valor 100 ao longo do algoritmo.

¹<https://github.com/camilla-m/kubernetes-scheduling-heuristics>

6. Resultados e Discussão

A Tabela 1 apresenta os resultados obtidos nos experimentos computacionais, comparando as diferentes abordagens em termos de custo total da solução e tempo de execução em milissegundos.

Tabela 1: Comparação de desempenho entre diferentes algoritmos para escalonamento de contêineres em Kubernetes

Pods	Nós	Kube-scheduler		PI		Heurística Gulosa		Busca Local		GRASP	
		Custo	Tempo (ms)	Custo	Tempo (ms)	Custo	Tempo (ms)	Custo	Tempo (ms)	Custo	Tempo (ms)
10	10	370	0	239	0	295	0	292	0	292	0
10	20	599	0	317	2	160	0	157	0	148	0
10	50	1986	0	229	2	634	0	634	0	493	1
10	100	3128	0	836	4	520	0	468	0	381	3
10	200	2206	0	724	3	1420	0	1420	0	1173	5
50	10	1302	0	760	8	705	0	702	0	625	0
50	20	2591	0	900	1	618	0	551	0	539	4
50	50	5273	0	1465	22	1889	0	1870	0	1241	3
50	100	12186	0	666	19	1676	0	1253	0	852	8
50	200	26364	0	1650	28	5978	0	5978	0	1870	21
100	10	1500	0	765	8	1669	0	1669	0	1626	0
100	20	5289	0	2136	9	1207	0	1117	0	1099	3
100	50	13797	0	895	11	3483	0	3322	0	2068	9
100	100	22904	0	2115	33	5131	0	4801	0	2783	16
100	200	31588	0	3610	45	10152	0	10030	0	2124	58
200	10	2741	0	1867	5	1952	0	1934	0	1919	0
200	20	7181	0	1595	3	2265	0	2197	2	2071	8
200	50	31449	0	1744	21	6994	0	6994	0	3819	36
200	100	22926	0	2707	68	5567	0	5567	0	2208	86
200	200	65642	0	3788	126	19994	0	18774	2	4200	200
500	10	10201	0	7751	5	7100	0	7100	0	7097	1
500	20	11137	0	4920	11	6124	0	4930	45	4915	69
500	50	38271	0	5617	20	15986	0	15986	0	5897	239
500	100	92963	0	7170	347	12855	0	9543	64	5904	511
500	200	222646	0	13546	333	53146	1	53146	0	4252	1266
1000	10	18924	0	11360	15	21767	1	21767	0	21725	1
1000	20	39035	0	9839	28	12119	1	12119	1	12085	97
1000	50	78688	1	12189	46	26091	1	25428	21	16155	810
1000	100	259152	0	9717	66	25289	2	25187	7	11150	1895
1000	200	325359	0	12647	128	81227	3	81227	1	11639	4994
5000	10	114336	6	80832	54	74741	34	74741	21	74620	9
5000	20	242665	6	129530	100	58309	34	58172	464	58027	4018
5000	50	611469	6	48807	199	146446	34	146446	22	58073	25436
5000	100	1268205	6	38734	695	116744	37	116744	24	61854	49490
5000	200	2577370	6	31849	1754	403046	42	403046	26	39035	184536
10000	10	109860	29	51139	95	101520	136	101520	87	101340	19
10000	20	496398	28	98610	185	137646	139	137646	91	137322	8336
10000	50	1125491	30	65163	426	297110	138	263097	82239	181288	80244
10000	100	1769065	27	137504	806	273595	146	273595	93	85692	192069
10000	200	2303908	26	136888	1963	784337	152	386876	254389	72555	3635737

6.1. Discussão e Análise dos Resultados

6.1.1. Qualidade das Soluções

A qualidade das soluções, medida pelo custo total obtido, demonstra uma clara hierarquia entre as abordagens avaliadas. O escalonador padrão do Kubernetes (Kube-scheduler) consistentemente apresenta as soluções de maior custo em quase todas as instâncias. Isso era esperado, pois este algoritmo não foi projetado com o objetivo explícito de minimizar custos de infraestrutura, mas sim para garantir distribuição adequada dos pods conforme outros critérios operacionais.

A abordagem baseada em Programação Inteira (PI) obteve soluções de excelente qualidade, especialmente em instâncias de pequeno (10-100 *pods*) e médio porte (200-500 *pods*). Para instâncias grandes (1000-10000 *pods*), a PI encontra soluções com custos significativamente menores que o escalonador padrão. Por exemplo, para a instância com 5000 *pods* e 200 nós, a solução

via PI apresenta custo 31.849, enquanto o Kube-scheduler atinge 2.577.370, representando uma redução impressionante de aproximadamente 98,8% no custo total. Similarmente, para 1000 *Pods* e 200 nós, a PI alcança custo 12.647 comparado a 325.359 do Kube-scheduler, uma melhoria de 96,1%.

Entre as abordagens heurísticas, observa-se que o GRASP consistentemente produz soluções de melhor qualidade quando comparado à Heurística Construtiva (HC) e à Busca Local (BL). Os resultados mostram que o GRASP supera significativamente as outras heurísticas em instâncias grandes. Por exemplo, na instância com 10000 *Pods* e 200 nós, o GRASP alcança custo 72.555, enquanto a Busca Local atinge 386.876 e a Heurística Gulosa 784.337, representando melhorias de 81,2% e 90,7%, respectivamente. Para a instância com 5000 *Pods* e 100 nós, o GRASP obtém custo 61.854 comparado a 116.744 das outras heurísticas, uma redução de 47,0%.

Notavelmente, em várias instâncias de grande porte (1000-10000 *Pods*), o GRASP chega a superar a PI em termos de qualidade da solução. Por exemplo, para 10000 *Pods* e 100 nós, a solução via GRASP tem custo 85.692 enquanto a PI atinge 137.504. Isso demonstra que a componente de aleatoriedade controlada e a exploração mais ampla do espaço de soluções do GRASP podem encontrar melhores soluções que a PI em casos onde o espaço de busca é vasto e complexo.

A relação de qualidade entre as abordagens pode ser resumida como: Kube-scheduler > Heurística Construtiva $\geq$ Busca Local > GRASP $\approx$ PI, em que ">"

A relação de qualidade entre as abordagens pode ser resumida como: Kube-scheduler > Heurística Construtiva $\geq$ Busca Local > GRASP $\approx$ PI, em que ">" indica custo maior (qualidade inferior) e " $\approx$ " indica desempenho comparável com vantagens situacionais.

6.1.2. Escalabilidade

O escalonador padrão do Kubernetes mantém tempos de execução consistentemente baixos mesmo para instâncias grandes, mas com significativa degradação na qualidade da solução. A PI, embora produza soluções de alta qualidade, apresenta crescimento acentuado no tempo computacional conforme o tamanho do problema aumenta - por exemplo, para 10000 *Pods* e 200 nós, o tempo chega a 1963ms.

As heurísticas propostas apresentam diferentes perfis de escalabilidade. A Heurística Construtiva e a Busca Local mantêm tempos de execução muito baixos mesmo para instâncias com 5000 ou 10000 *Pods* (34-42ms para HC e 21-26ms para BL para 5000 *Pods* e 200 nós), demonstrando excelente escalabilidade.

O GRASP, devido à sua natureza iterativa e exploratória, apresenta um comportamento de escalabilidade distinto. Para instâncias pequenas e médias (até 500 *Pods*), mantém tempos de execução razoáveis, na ordem de segundos a minutos. Contudo, para instâncias muito grandes (acima de 5000 *Pods*), o tempo computacional cresce substancialmente, chegando a mais de uma hora (3635737ms) para a maior instância testada (10000 *Pods* e 200 nós). Este comportamento era esperado devido à complexidade inerente do GRASP, que executa múltiplas iterações de construção e busca local.

É importante notar que, apesar do maior custo computacional, o GRASP oferece um excelente compromisso entre qualidade e tempo para instâncias de todos os tamanhos, proporcionando melhorias significativas em qualidade quando comparado ao Kube-scheduler, e rivalizando com PI em alguns casos, particularmente em instâncias muito grandes.

6.1.3. Eficiência Computacional

A eficiência computacional é um fator crítico para aplicações em tempo real de escalonamento em ambientes Kubernetes. Os resultados mostram que todas as abordagens propostas superam significativamente o escalonador padrão em termos de qualidade da solução, mas com diferentes perfis de eficiência computacional.

A Heurística Construtiva e a Busca Local demonstram excelente eficiência, com tempos de execução consistentemente baixos mesmo para instâncias grandes. A PI proporciona soluções de alta qualidade com tempos de execução aceitáveis para instâncias de tamanho pequeno a médio, mas seu desempenho pode se tornar limitado para instâncias muito grandes.

O GRASP, embora computacionalmente mais intensivo, apresenta um perfil de desempenho interessante. Para instâncias pequenas e médias, oferece um excelente equilíbrio entre qualidade e tempo. Por exemplo, para 500 pods e 200 nós, o GRASP alcança uma solução de custo 4.252 em 1.266ms, significativamente melhor que as outras abordagens. Em instâncias maiores, o custo computacional aumenta, mas a qualidade das soluções obtidas pode justificar o investimento adicional em tempo, especialmente em cenários onde o escalonamento não precisa ser feito em tempo real ou pode ser pré-calculado.

É notável que, para certas classes de instâncias, o GRASP consegue encontrar soluções significativamente melhores que a PI em tempos comparáveis ou mesmo menores. Por exemplo, para 10000 pods e 10 nós, o GRASP encontra uma solução de custo 101.340 em apenas 19ms, enquanto a PI alcança 51.139 em 95ms. Esse comportamento sugere que o GRASP pode ser particularmente eficaz para certas configurações de *cluster*.

7. Conclusão e Trabalhos Futuros

Este trabalho avaliou diferentes abordagens para o problema de escalonamento de contêineres em orquestradores Kubernetes, modelado como uma variante do *Capacitated Facility Location Problem*. Os resultados experimentais confirmam a superioridade das abordagens propostas em relação ao algoritmo padrão do Kubernetes (Kube-scheduler), tanto em qualidade das soluções quanto em desempenho para diversas configurações de *clusters*.

A hierarquia observada em termos de custo das soluções (Kube-scheduler > Heurística Construtiva $\geq$ Busca Local > GRASP $\approx$ PI) está alinhada com as expectativas teóricas, validando a eficácia da modelagem como CFLP e das implementações propostas. Um resultado particularmente interessante foi o desempenho excepcional do GRASP em instâncias de grande porte, em que foi capaz de superar a abordagem baseada em Programação Inteira para vários casos, demonstrando o potencial das meta-heurísticas para problemas complexos de escalonamento. A aplicação das abordagens heurísticas propostas, particularmente o GRASP, pode levar a economias substanciais em termos de utilização de recursos e custos de infraestrutura em nuvem, com reduções de custo superiores a 90% em vários cenários quando comparadas ao escalonador padrão.

Para trabalhos futuros, identificaram-se algumas direções promissoras, como a integração das abordagens propostas diretamente ao Kubernetes aplicado em nuvem através de *plugins* ou extensões do Kube-scheduler que permitiria validar o desempenho das soluções em ambientes de produção com cargas de trabalho reais. Também pode-se fazer uma implementação de estratégias de paralelização para os algoritmos propostos para aumentar sua eficiência, ou ainda a aplicação de técnicas de aprendizado de máquina para prever padrões eficientes de alocação baseados em dados históricos.

Referências

- Cornuéjols, G., Nemhauser, G., e Wolsey, L. (1983). The uncapacitated facility location problem. Technical report, Cornell University Operations Research.
- Cornuéjols, G., Nemhauser, G., e Wolsey, L. (1990). *Discrete Location Theory*. John Wiley & Sons.

- Feo, T. A. e Resende, M. G. C. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133. URL <https://www2.imm.dtu.dk/courses/02719/grasp/grasp.pdf>.
- Gurobi Optimization (2025a). Java API Overview. https://www.gurobi.com/documentation/current/refman/java_api_overview.html.
- Gurobi Optimization (2025b). MIPGap. <https://support.gurobi.com/hc/en-us/articles/8265539575953-What-is-the-MIPGap>.
- Kaur, K. e Sachdeva, M. (2022). Context-aware scheduling in mapreduce: a compact review. *Concurrency and Computation: Practice and Experience*, 34(4):e6501.
- Klose, A. e Drexler, A. (2005). Facility location models for distribution system design. *European Journal of Operational Research*, 162(1):4–29.
- Kubernetes (2023). Kubernetes documentation. URL <https://kubernetes.io/docs/home/>. Acessado: 2023-10-15.
- Kubernetes (2025a). Kubernetes components. <https://kubernetes.io/docs/concepts/overview/components/>. Acessado em: 23 de abril de 2025.
- Kubernetes (2025b). Why you need kubernetes and what it can do. <https://kubernetes.io/docs/concepts/overview/#why-you-need-kubernetes-and-what-can-it-do>. Acessado em: 23 de abril de 2025.
- Martins, C., De Souza Moura, P., De Lucena, S., e Vieira Campos, C. (2023). Container scheduling in kubernetes clusters using a mixed integer linear programming approach. In *2023 15th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, p. 140–145.
- Pham, L. H., Hoang, Q. T., Nguyen-Thanh, N., Nguyen, N. X., e Ho, A. V. (2020). Lahc: A load-aware host selection method for docker container migration. *ETRI Journal*, 42(1):45–53.
- Scheduler, K. (2023). Scheduler documentation. URL <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>. Acessado: 2023-10-15.
- Wojciechowski, L., Opasiak, K., Latusek, J., Wereski, M., Morales, V., Kim, T., e Hong, M. (2021). Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, p. 1–9.
- Wolsey, L. A. (1998). *Integer Programming*. Wiley-Interscience, New York. ISBN 0471283665.
- Zhang, D., Yan, B., Feng, Z., Zhang, C., e Wang, Y. (2017). Container oriented job scheduling using linear programming model. In *2017 3rd International Conference on Information Management (ICIM)*, p. 174–180.
- Zhong, Z. e Buyya, R. (2020). Cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources. *ACM Trans. Internet Technol.*, 20. URL <https://doi.org/10.1145/3378447>.
- Zhou, R., Li, Z., e Wu, C. (2018). Scheduling frameworks for cloud container services. *IEEE/ACM Transactions on Networking*, 26:436–450.