



UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Escalonamento de Contêineres em Orquestrador Kubernetes Usando uma Abordagem de
Programação Linear

Camilla Martins

Orientador

Sidney Cunha de Lucena

Coorientadores

Pedro Nuno Moura

Carlos Alberto Vieira Campos

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2024

Escalonamento de Contêineres em Orquestrador Kubernetes Usando uma Abordagem de Programação Linear

CAMILLA MARTINS

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

Documento assinado digitalmente
 **SIDNEY CUNHA DE LUCENA**
Data: 29/05/2024 20:14:41-0300
Verifique em <https://validar.iti.gov.br>

SIDNEY CUNHA DE LUCENA — UNIRIO

Documento assinado digitalmente
 **CARLOS ALBERTO VIEIRA CAMPOS**
Data: 02/06/2024 19:45:22-0300
Verifique em <https://validar.iti.gov.br>

CARLOS ALBERTO VIEIRA CAMPOS — UNIRIO

Documento assinado digitalmente
 **PEDRO NUNO DE SOUZA MOURA**
Data: 29/05/2024 20:09:24-0300
Verifique em <https://validar.iti.gov.br>

PEDRO NUNO SOUZA DE MOURA — UNIRIO

Documento assinado digitalmente
 **MARCEL WILLIAM ROCHA DA SILVA**
Data: 03/06/2024 12:10:42-0300
Verifique em <https://validar.iti.gov.br>

MARCEL WILLIAM ROCHA DA SILVA — UFRRJ

Documento assinado digitalmente
 **Adriana Cesario de Faria Alvim**
Data: 03/06/2024 18:41:49-0300
Verifique em <https://validar.iti.gov.br>

ADRIANA CESÁRIO DE FARIA ALVIM — UNIRIO

Documento assinado digitalmente
 **KLEBER VIEIRA CARDOSO**
Data: 04/06/2024 08:19:39-0300
Verifique em <https://validar.iti.gov.br>

KLÉBER VIEIRA CARDOSO — UFG

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 2024.

Catálogo informatizada pelo(a) autor(a)

M Martins, Camilla
Escalonamento de Contêineres em Orquestrador Kubernetes
Usando uma Abordagem de Programação Linear / Camilla
Martins, Pedro Nuno Souza de Moura, Sidney Cunha de
Lucena. -- Rio de Janeiro, 2024.
92

Orientador: Sidney Cunha de Lucena.
Coorientador: Carlos Alberto Vieira Campos.
Dissertação (Mestrado) - Universidade Federal do Estado
do Rio de Janeiro, Programa de Pós-Graduação em Informática,
2024.

1. Kubernetes. 2. Escalonamento de contêineres. 3.
Pesquisa Operacional. I. Nuno Souza de Moura, Pedro II.
Cunha de Lucena, Sidney III. Cunha de Lucena, Sidney,
orient. IV. Alberto Vieira Campos, Carlos, coorient. V.
Título.

Dedico este trabalho aos meus pais, amigos, família e minha querida Umbanda.

Agradecimentos

Agradeço primeiramente aos meus pais Sérgio e Luciana por nunca terem me abandonado, sempre terem investido em mim e nos meus estudos, e confiado em mim a capacidade de seguir a carreira como profissional e estudante de Tecnologia da Informação.

Agradeço ao meu orientador Sidney Cunha de Lucena por todo suporte necessário e por toda tranquilidade que me passou ao longo destes anos de mestrado. Agradeço aos meus coorientadores, Pedro Nuno Moura e Carlos Alberto Vieira Campos por todos os ensinamentos, incentivo, conversas e dedicação em minha jornada acadêmica.

Agradeço a pessoas muito especiais por me auxiliarem em anos de muita luta, foco e vencida de obstáculos, especialmente estes de 2023 e 2024, independente de como e do tempo de estrada comigo, como Danielle Fernandes, Ana Paula Landvoigt Wendt, Cristiane Cardoso, Adriana Esposito, Aline Bezzoco, Pedro Amaral, Luana Moraes, Gabriel Galatoli, Gabriel Souza, Rafael Gomes, Mauricio Gomes, Lucas Meireles, Roberto Stoll, Maria Luiza Goveia, Ivanilda Dias, Samanta Damini, Roze Anne, Rebeca Prado, Tainara Almeida e Ana Carolina Fonseca. Gratidão pelo incentivo, pelos conselhos, pelo acolhimento e abrigo sempre tão caloroso e necessário para a continuada das minhas batalhas.

Agradeço aos dirigentes de minha casa de Umbanda querida, Casa de Zarithamy, Armando e Cristina Fernandes, aos irmãos da casa e à corrente mediúnica pela compreensão com minha jornada acadêmica, o amor, respeito e guiar dos meus caminhos, local este que tanto me salvou.

Agradeço ao meu trabalho e meus queridos colegas pela motivação de sempre e apoio, João Paulo Freire e Marcelo Fernandes.

Agradeço à minha terapeuta Rosana Rosa, por nunca ter me deixado desistir.

Agradeço ao meu avô, uma eterna estrela no céu que brilha, Victor da Penha Firmino, que hoje é um anjo da guarda a me guiar.

MARTINS, CAMILLA **Escalonamento de Contêineres em Orquestrador Kubernetes Usando uma Abordagem de Programação Inteira**. UNIRIO, 2024. 88 páginas. Dissertação de Mestrado. Departamento de Informática Aplicada, UNIRIO.

RESUMO

Grande parte dos serviços providos na Internet residem em ambientes virtualizados de provedores de nuvem, cujas infraestruturas de Tecnologia da Informação adotam, em sua grande maioria, a técnica de virtualização de contêineres para abrigar as diversas aplicações que devem rodar de forma independente e isolada. O gerenciamento desses contêineres é predominantemente realizado pela solução Kubernetes. Contudo, o algoritmo padrão de alocação de contêineres em máquinas físicas (nós), usado pelo Kubernetes, o Kube-scheduler, não é capaz de otimizar a relação custo/desempenho do ambiente em nuvem. Para solucionar isso, este trabalho propõe uma abordagem de Programação Inteira inspirada em uma formulação para o *Capacitated Facility Location Problem*, a fim de otimizar a distribuição de contêineres em nós de um *cluster* Kubernetes, através de uma formulação de programação linear, levando em consideração na sua função objetivo o custo de abertura de um nó, o custo de alocação de um *pod* a um nó e a taxa de punição em caso de erros encontrados nos *pods* dentro de considerados instantes de tempo. Os experimentos realizados mostram que a solução proposta nesta dissertação tem resultados significativamente positivos, com uma ordem de grandeza de diferença, em relação ao custo final da solução no *cluster* Kubernetes construído sobre a estratégia convencional de agendamento comparando através de três conjuntos de experimentos propostos, ao mesmo tempo em que garante que as demandas dos ditos *pods* sejam atendidas de forma eficiente.

Palavras-chave: Agendamento de Contêineres, Alocação de Pods, Programação Inteira, Kubernetes, Problema de Localização de Instalações

ABSTRACT

A large part of the services provided on the Internet reside in virtualized environments from cloud providers, whose Information Technology infrastructures mostly adopt the container virtualization technique to house the various applications that must run independently and isolated. The management of these containers is predominantly carried out by the Kubernetes solution. However, the standard container allocation algorithm on physical machines (nodes), used by Kubernetes, the Kube-scheduler, is not capable of optimizing the cost/performance ratio of the cloud environment. To solve this, this work proposes an Integer Programming approach inspired by a formulation for the *Capacitated Facility Location Problem*, in order to optimize the distribution of containers in nodes of a *cluster* Kubernetes, through a formulation of linear programming, taking into account in its objective function the cost of opening a node, the cost of allocating a *pod* to a node and the punishment rate in case of errors found in the *pods* within of considered instants of time. The experiments carried out show that the solution proposed in this dissertation has significantly positive results, with an order of magnitude difference, in relation to the final cost of the solution in *cluster* Kubernetes built on the conventional scheduling strategy comparing across three sets of proposed experiments, while ensuring that the demands of said *pods* are met efficiently.

Keywords: Container Scheduling, Pod Allocation, Integer Scheduling, Kubernetes, Facility Location Problem.

Sumário

1	Introdução	1
1.1	Motivação	1
1.2	Problema	3
1.3	Objetivo	3
1.4	Contribuição	4
1.5	Organização	4
2	Conceitos Fundamentais	5
2.1	Computação em Nuvem	5
2.1.1	DevOps	7
2.1.2	Docker	9
2.2	Kubernetes	11
2.3	Kube-scheduler	12
2.4	Problemas NP-Difíceis	14
2.5	Otimização Matemática	15
2.5.1	Programação Linear	16
2.5.2	Programação Inteira	17
2.5.3	<i>Capacitated Facility Location Problem</i>	19

2.5.3.1	Formulação de Programação Inteira Mista	20
2.5.3.2	<i>Branch-and-Bound</i>	21
2.5.4	Resolvedor MIP	22
2.5.5	Gurobi	23
3	Trabalhos Relacionados	25
3.1	Alocação de <i>pods</i>	25
3.2	Abordagens Baseadas em Programação Linear	28
3.3	Facility Location Problem	30
4	Modelagem	32
4.1	Problema	32
4.2	Formulação	34
5	Experimentos Computacionais	37
5.1	Ambiente computacional	37
5.2	Instâncias utilizadas	38
5.3	Implementação do Kube-scheduler	39
5.4	Implementação da Formulação	40
5.5	Etapas dos Experimentos	40
6	Resultados	43
6.1	Resultados do primeiro conjunto de estudos experimentais computacionais	43
6.2	Resultados do segundo conjunto de estudos experimentais computacionais	51
6.3	Resultados do terceiro conjunto de estudos experimentais computacionais	58
6.3.1	Resultados para uma alocação inicial de 5000 <i>pods</i>	58
6.3.2	Resultados para uma alocação inicial de 10000 <i>pods</i>	61

6.4	Discussão dos resultados	64
7	Conclusão	67
7.1	Considerações finais	67
7.2	Trabalhos futuros	69
	Referências Bibliográficas	70

Lista de Figuras

2.1	Comparação explicativa entre os tipos de modelos de computação em nuvem com o funcionamento de uma pizzaria para compreender computação em nuvem. Adaptada de [31]	6
2.2	Pirâmide de IaaS, SaaS e PaaS que ilustra plataformas usadas em cada um dos modelos. Adaptada de [46]	8
2.3	Representação do ciclo de vida de processos de DevOps. Adaptado de [20]	9
2.4	Comparação entre uma infraestrutura com uso de máquina virtual e de contêineres. Adaptado de [21]	10
2.5	Funcionamento de um cluster Kubernetes com seus componentes principais. Extraído de [22]	12
2.6	O lado (a) mostra a região viável para um programa linear. O lado (b) demonstra as linhas de contorno da função objetivo 2.5.1 para diferentes valores do lucro c , em que o ponto ótimo de lucro está em 1900. Extraído de [37]	17
2.7	Exemplo de instância do FLP, em que se deseja escolher a melhor forma de distribuir insumos aos clientes finais. Extraído de [45]	19
4.1	Funcionamento de cluster Kubernetes na tentativa de alocar pods a nós em diferentes instantes de tempo	35
6.1	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 10 nós no primeiro conjunto de estudos experimentais computacionais.	45

6.2	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 20 nós no primeiro conjunto de estudos experimentais computacionais.	46
6.3	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 50 nós no primeiro conjunto de estudos experimentais computacionais.	46
6.4	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 100 nós no primeiro conjunto de estudos experimentais computacionais.	47
6.5	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 200 nós no primeiro conjunto de estudos experimentais computacionais.	47
6.6	Tempo de execução da formulação de PI e do Kube-scheduler para 10 <i>Pods</i> alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.	48
6.7	Tempo de execução da formulação de PI e do Kube-scheduler para 50 <i>Pods</i> alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.	48
6.8	Tempo de execução da formulação de PI e do Kube-scheduler para 100 <i>Pods</i> alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.	49
6.9	Tempo de execução da formulação de PI e do Kube-scheduler para 200 <i>Pods</i> alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.	49
6.10	Tempo de execução da formulação de PI e do Kube-scheduler para 500 <i>Pods</i> alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.	49
6.11	Tempo de execução da formulação de PI e do Kube-scheduler para 1000 <i>Pods</i> alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.	50

6.12	Tempo de execução da formulação de PI e do Kube-scheduler para 5000 <i>Pods</i> alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.	50
6.13	Tempo de execução da formulação de PI e do Kube-scheduler para 10000 <i>Pods</i> alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.	51
6.14	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 10 nós no segundo conjunto de estudos experimentais computacionais.	53
6.15	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 20 nós no segundo conjunto de estudos experimentais computacionais.	53
6.16	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 50 nós no segundo conjunto de estudos experimentais computacionais.	54
6.17	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 100 nós no segundo conjunto de estudos experimentais computacionais.	54
6.18	Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 200 nós no segundo conjunto de estudos experimentais computacionais.	55
6.19	Tempo de execução da formulação de PI e do Kube-scheduler para 10 <i>Pods</i> alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.	55
6.20	Tempo de execução da formulação de PI e do Kube-scheduler para 50 <i>Pods</i> alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.	56
6.21	Tempo de execução da formulação de PI e do Kube-scheduler para 100 <i>Pods</i> alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.	56

6.22	Tempo de execução da formulação de PI e do Kube-scheduler para 200 <i>Pods</i> alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.	56
6.23	Tempo de execução da formulação de PI e do Kube-scheduler para 500 <i>Pods</i> alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.	57
6.24	Tempo de execução da formulação de PI e do Kube-scheduler para 1000 <i>Pods</i> alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.	57
6.25	Tempo de execução da formulação de PI e do Kube-scheduler para 5000 <i>Pods</i> alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.	57
6.26	Tempo de execução da formulação de PI e do Kube-scheduler para 1000 <i>Pods</i> alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.	58
6.27	Comparativo entre o custo das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI no terceiro conjunto de estudos experimentais computacionais, iniciados com 5000 <i>Pods</i>	60
6.28	Comparativo entre o tempo de execução das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI no terceiro conjunto de estudos experimentais computacionais, iniciados com 5000 <i>Pods</i>	60
6.29	Comparativo entre a abertura de nós das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI para determinada quantidade de nós no terceiro conjunto de estudos experimentais computacionais, iniciados com 5000 <i>Pods</i>	61
6.30	Análise comparativa entre o custo das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI para determinada quantidade de nós no terceiro conjunto de estudos experimentais computacionais, iniciados com 10000 <i>Pods</i>	63
6.31	Comparativo entre o tempo de execução das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI no terceiro conjunto de estudos experimentais computacionais, iniciados com 10000 <i>Pods</i>	63

6.32	Comparativo entre a abertura de nós das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI para determinada quantidade de nós no terceiro conjunto de estudos experimentais computacionais, iniciados com 10000 <i>pods</i>	64
------	--	----

Lista de Tabelas

4.1	Descrição das variáveis e coeficientes usados na formulação do problema.	34
6.2	Tabela 6.1: Resultados obtidos pela implementação do Kube-scheduler padrão e pela formulação de PI para as instâncias artificiais geradas. . . .	44
6.4	Tabela 6.2: Resultados obtidos pela implementação da formulação de Programação Inteira para as instâncias artificiais geradas no segundo conjunto de estudos experimentais computacionais.	52
6.6	Tabela 6.3.1: Resultados obtidos para as instâncias geradas no terceiro conjunto de estudos experimentais computacionais para uma quantidade inicial de 5000 <i>pods</i>	59
6.8	Tabela 6.3.2: Resultados obtidos para as instâncias geradas no terceiro conjunto de estudos experimentais computacionais para uma quantidade inicial de 10000 <i>pods</i>	62

Lista de Nomenclaturas

CFLP	Capacitated Facility Location Problem
VM	Virtual Machine
FLP	Facility Location Problem
MEC	Multi-Access Edge Computing
NFV	Network Functions Virtualization
UPF	User Plane Function
CFLP	Capacitated Facility Location Problem
PLIM	Programação Linear Inteira Mista
PLI	Programação Linear Inteira

1. Introdução

Neste capítulo, são descritas as motivações da dissertação, trazendo uma introdução ao tema e a elaboração do problema, com as devidas contribuições e objetivos deste trabalho, além de como se encontra a disposição de seus capítulos.

1.1 Motivação

Na área de Tecnologia da Informação e Comunicações (TIC), uma das técnicas mais empregadas atualmente por provedores de serviço em nuvem é a virtualização baseada em contêineres, capazes de agrupar códigos de uma aplicação e todas as suas dependências, possibilitando que essa aplicação seja executada de forma rápida e confiável, ainda que se mude o ambiente computacional.

Nos últimos anos, tem-se testemunhado um crescimento exponencial nas aplicações em nuvem, impulsionado por uma série de fatores-chave. Uma das principais razões por trás desse crescimento é a flexibilidade e escalabilidade oferecidas pela computação em nuvem. Empresas de todos os portes estão migrando para a nuvem devido à capacidade de dimensionar recursos de acordo com a demanda, evitando investimentos pesados em infraestrutura física, como demonstra [33].

Outro aspecto crucial é a acessibilidade global proporcionada pela nuvem. Com aplicativos e dados armazenados na nuvem, as empresas podem acessá-los de qualquer lugar do mundo, promovendo a colaboração remota e a disponibilidade contínua dos serviços [34]. A segurança também é uma preocupação central. Embora inicialmente tenha havido preocupações sobre a segurança dos dados na nuvem, os provedores têm investido em medidas de segurança avançadas, permitindo que as empresas implementem suas aplicações garantindo confiabilidade para o seu usuário final [35].

Com a pandemia da COVID-19, também houve uma aceleração da adoção da nuvem

como infraestrutura base. Com isso, percebeu-se a importância de ter sistemas e aplicativos acessíveis remotamente, impulsionando ainda mais a migração para a nuvem [36].

Com o avanço da implementação de ferramentas e soluções em nuvem, passou-se a conhecer e a se aprofundar em tecnologias que buscam dinamizar o dia a dia das empresas, trazendo redução de custos, escalonamento dos recursos, segurança, facilidade no provisionamento e adoção dessas soluções para além do gerenciamento de forma confiável e estável.

Atualmente, a maneira mais simples e comum de implementar e provisionar aplicações em larga escala em nuvem é através dos contêineres, que são unidades de software leves e isoladas que virtualizam um sistema operacional em menor tamanho, e podem ser replicados para atender a demanda necessária. A empresa mais famosa responsável por esta tecnologia de código aberto é a Docker [42].

No escopo das tecnologias e soluções de mercado voltadas para contêineres e seu gerenciamento, o Kubernetes [43] é, segundo a própria documentação oficial, um sistema de código aberto que automatiza, em larga escala, a implantação, o dimensionamento e o gerenciamento de aplicativos em contêineres. O Kubernetes trabalha com o conceito de *cluster*, que é a entidade que agrupa todas as máquinas virtuais gerenciadas por uma mesma instância do Kubernetes. Nessas máquinas virtuais, chamadas de nós, rodam os *pods*, onde são executadas as aplicações, sendo que um *pod* pode conter um ou mais contêineres. Entretanto, para se escalar contêineres em um ambiente Kubernetes, é preciso ter uma atenção especial com questões relacionadas ao dimensionamento dos nós e dos *pods*, assim como com a probabilidade de falhas desses componentes, pois isso pode impactar no bom funcionamento da infraestrutura em nuvem.

A motivação para o trabalho proposto é a possibilidade de redução de custos para um ambiente de infraestrutura em nuvem através da otimização do processo de decisão do orquestrador de contêineres que a gerencia. No caso, o algoritmo de otimização deverá buscar uma solução que minimize o custo de implantação de nós (máquinas virtuais) e que também seja capaz de atender a demanda de alocação de *pods* conforme a necessidade de recursos de cada *pod*, de maneira que eles consigam desempenhar suas funções com qualidade e com menor chance de falhas.

1.2 Problema

Um novo *pod* deve ser sempre alocado para o nó mais barato dentre aqueles disponíveis e que possuam recursos suficientes para acomodá-lo. Haverá sempre uma determinada quantidade de *pods* em execução para cada um dos nós ativos. À medida que as implementações de novas aplicações vão acontecendo e o orquestrador é demandado, a quantidade de *pods* em cada nó naturalmente se altera, em função do término de algumas execuções e das novas alocações que porventura tenham surgido. Então, avalia-se a quantidade necessária de nós para atender tais demandas conforme a capacidade individual dos nós existentes, a quantidade de *pods* existentes no *cluster*, os recursos em uso por cada *pod* e os recursos demandados pelos novos *pods* que foram gerados. Caso os nós tenham características físicas diferentes que afetem sua capacidade e seu custo de ativação, isso também deverá ser levado em consideração.

De maneira sintética, este problema pode corresponder à seguinte pergunta de pesquisa: qual é a alocação que permite ao *cluster* Kubernetes atender a demanda existente de *pods* utilizando o menor número de nós e incorrendo no menor custo possível? Ou ainda: qual é o número máximo de *pods* que uma infraestrutura orquestrada em *cluster* Kubernetes conseguiria trabalhar em alta demanda sem necessitar criar novos nós?

1.3 Objetivo

Esta dissertação tem como objetivo propor uma formulação para o problema de alocação de *pods* por nós de um orquestrador de contêineres Kubernetes utilizando como base matemática uma solução de programação inteira em tempo de execução e compará-la com a solução padrão do agendador de *pods* do Kubernetes, o Kube-scheduler [38], afim de testar sua eficiência.

Postula-se que a solução ótima para este problema envolve a alocação de *pods* a nós buscando minimizar o custo associado ao resultado final desta alocação, de acordo com a disponibilidade de recursos de cada nó.

O problema aqui tratado guarda similaridade com o *Capacitated Facility Location Problem* (CFLP), que é um problema sabidamente $\mathcal{NP}$ -Difícil existente na literatura??. Enquanto o CFLP busca as melhores localizações para novas instalações, nós buscamos o melhor nó para a alocação de um determinado *pod*. O objetivo de ambos os problemas é o mesmo: minimizar os custos e maximizar a eficiência do sistema, buscando a melhor combinação que leve a isso. A escolha do *pod* sempre deve privilegiar o nó com menor

custo, se possível, de acordo com sua disponibilidade de recursos, na tomada de decisão do algoritmo.

1.4 Contribuição

As contribuições desta dissertação consistem em:

1. Construção de uma modelagem baseada em Programação Inteira (PI) que permite identificar o que seria um nível ótimo de alocação de recursos em um *cluster* Kubernetes, considerando custos como de abertura de nó, alocação de *pods* e penalizações de erros, e buscando atender o maior número de *pods* possível;
2. Realização de experimentos computacionais comparando a formulação proposta à implementação do algoritmo padrão do Kubernetes em relação a um conjunto de instâncias artificialmente geradas;
3. Realização de experimentos computacionais considerando um horizonte de tempo, a fim de se comparar o uso de infraestrutura de ambas as abordagens;
4. Os resultados experimentais mostram que a formulação proposta proporciona soluções cujos valores são, em média, uma ordem de grandeza melhores do que as soluções obtidas pela implementação do algoritmo padrão do Kubernetes.

1.5 Organização

O Capítulo 1 apresenta a introdução da dissertação, trazendo sua motivação, o problema de pesquisa, os objetivos e as contribuições alcançadas. O Capítulo 2 aborda Computação em Nuvem, incluindo tópicos como DevOps, Docker e Kubernetes, além de se aprofundar em Otimização Matemática e conceitos de Programação Linear. Por sua vez, o Capítulo 3 debate os trabalhos relacionados que foram importantes para a construção desta pesquisa. O Capítulo 4 apresenta a modelagem e formulação proposta nesta pesquisa para a solução do problema. O Capítulo 5 descreve os cenários experimentais e como estes foram trabalhados. No Capítulo 6 são apresentados os resultados experimentais para a formulação proposta, comparando-a com o algoritmo padrão do Kubernetes. Por fim, no Capítulo 7 são apresentadas a conclusão, as limitações desta pesquisa e as sugestões para trabalhos futuros.

2. Conceitos Fundamentais

Neste capítulo, são descritos os conceitos necessários para o entendimento deste trabalho.

2.1 Computação em Nuvem

A computação em nuvem refere-se à entrega de serviços de computação, como armazenamento, processamento de dados, servidores, redes, análise, software e inteligência pelo intermédio da internet. Em vez de hospedar e gerenciar recursos localmente em hardware físico ou servidores dedicados, os usuários podem acessar esses serviços e recursos remotamente, por meio da internet.

Os serviços de computação em nuvem geralmente são fornecidos por provedores de serviços em nuvem, que mantêm infraestruturas e plataformas para permitir que os usuários acessem e utilizem recursos de computação sob demanda. Esses serviços são flexíveis, escaláveis e muitas vezes mais econômicos do que a infraestrutura tradicional de Tecnologia da Informação, pois os usuários pagam apenas pelos recursos que consomem.

Existem três principais modelos de serviço na computação em nuvem:

Infraestrutura como Serviço (*Infrastructure as a Service – IaaS*): Fornece recursos de infraestrutura virtualizados, como máquinas virtuais, armazenamento e redes. Os usuários têm controle sobre o sistema operacional, aplicativos e configurações, mas não precisam se preocupar com a manutenção do hardware subjacente.

Plataforma como Serviço (*Platform as a Service – PaaS*): Oferece uma plataforma de desenvolvimento e execução de aplicativos, eliminando a necessidade de gerenciar a infraestrutura subjacente. Os usuários podem se concentrar no desenvolvimento de aplicativos sem se preocupar com a infraestrutura.

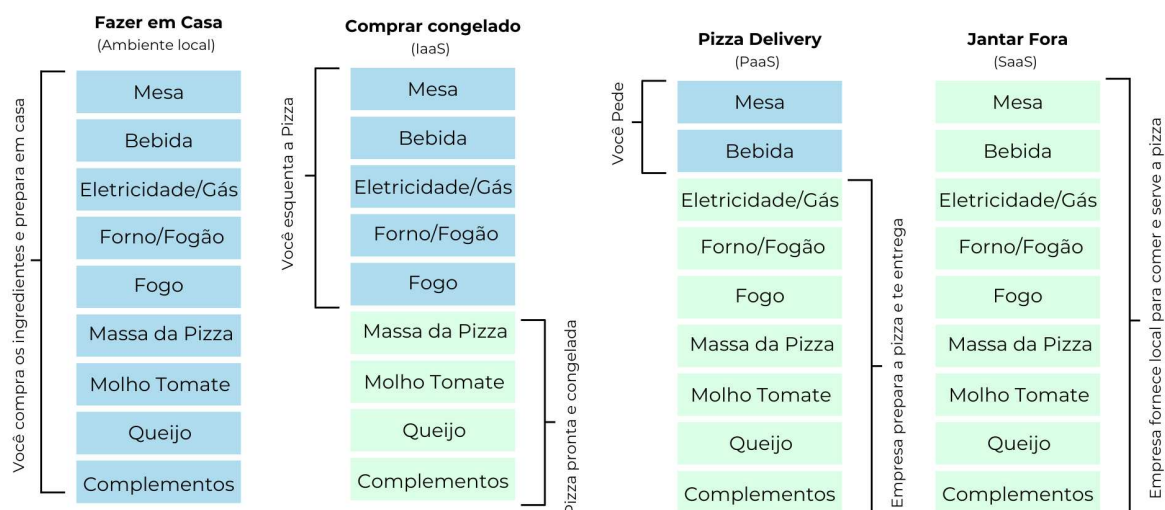


Figura 2.1: Comparação explicativa entre os tipos de modelos de computação em nuvem com o funcionamento de uma pizzeria para compreender computação em nuvem. Adaptada de [31]

Software como Serviço (*Software as a Service* – SaaS): Disponibiliza aplicativos de software prontos para uso na nuvem. Os usuários acessam esses aplicativos por meio de um navegador da web, sem a necessidade de instalação ou manutenção local.

A computação em nuvem proporciona benefícios como escalabilidade, flexibilidade, acessibilidade remota, custos reduzidos e atualizações automáticas. Ela tem sido fundamental para impulsionar a transformação digital em empresas e organizações, permitindo que elas se adaptem rapidamente às mudanças no ambiente de negócios.

Uma analogia interessante para compreender melhor a diferença entre esses três principais modelos é a analogia do *Pizza as a Service*, como mostra a Figura 2.1, em que, por meio do conceito de entrega de pizza, conseguimos entender o que cada um destes modelos faz em computação em nuvem, os comparando entre si.

No contexto da pizza, a IaaS (*Infrastructure as a Service*) representa a infraestrutura básica. Imagine que a massa, o queijo e o molho são os componentes essenciais da pizza. A IaaS fornece a infraestrutura fundamental, como servidores virtuais, armazenamento e redes.

O PaaS (*Platform as a Service*) adiciona os “toppings” à pizza. Os “toppings” representam a plataforma de desenvolvimento que é fornecida junto com a infraestrutura. Com PaaS, os desenvolvedores recebem não apenas os recursos de infraestrutura, mas

também ferramentas e serviços adicionais para desenvolver, testar e implantar aplicativos de maneira mais eficiente.

Por sua vez, no SaaS (*Software as a Service*), a pizza está pronta para ser consumida, assim como o software está pronto para uso. A analogia sugere que, com SaaS, os usuários obtêm acesso direto a aplicativos completos e prontos para uso, sem se preocupar com o desenvolvimento ou manutenção. A pizza está pronta para ser saboreada, bem como o software está pronto para ser utilizado.

Essa abordagem lúdica ajuda a simplificar os diferentes modelos de serviço em nuvem, tornando-os mais acessíveis, mesmo para aqueles que não estão familiarizados com os conceitos técnicos. No entanto, é importante ressaltar que *Pizza as a Service* consiste em uma metáfora humorística utilizada para fins didáticos e não é, portanto, um termo técnico oficial no contexto da computação em nuvem.

O verdadeiro conceito explicativo de IaaS, PaaS e SaaS se encontra na Figura 2.2, em que estão ilustrados alguns exemplos de serviços, plataformas e recursos utilizados para cada tipo de modelo de serviço de computação em nuvem.

Na base da pirâmide, temos os provedores de nuvem (IaaS), como a Amazon Web Services¹ e Google Cloud Platform², por exemplo. Já para as plataformas (PaaS), podemos encontrar ferramentas como o Docker [42], Kubernetes [43], dentre outros (não mencionados na imagem). E por fim, os serviços SaaS são aqueles que se encontram já prontos para uso pelo usuário final, como um armazenamento de fotos (Google Drive³), gerenciamento de clientes (SalesForce⁴), dentre outros. Quanto maior o nível em que se está na pirâmide, menor a dependência de manuseio e configuração pelo usuário final.

2.1.1 DevOps

Nesse contexto, se encaixa a área de DevOps que corresponde a um termo surgido por volta de 2007 cujo objetivo é unificar o mundo de desenvolvimento com o da infraestrutura, automatizando e otimizando tudo aquilo que é possível em termos de recursos operacionais com auxílio de código. Assim, essa área busca desenvolver com segurança, acelerar processos e trazer colaborações, trabalhando não somente a parte técnica, como também a humana, dentre outros fatores.

Existem várias ferramentas que auxiliam na propagação e aplicação de DevOps nas

¹<https://docs.aws.amazon.com>

²<https://cloud.google.com/docs>

³<https://drive.google.com/>

⁴<https://www.salesforce.com>

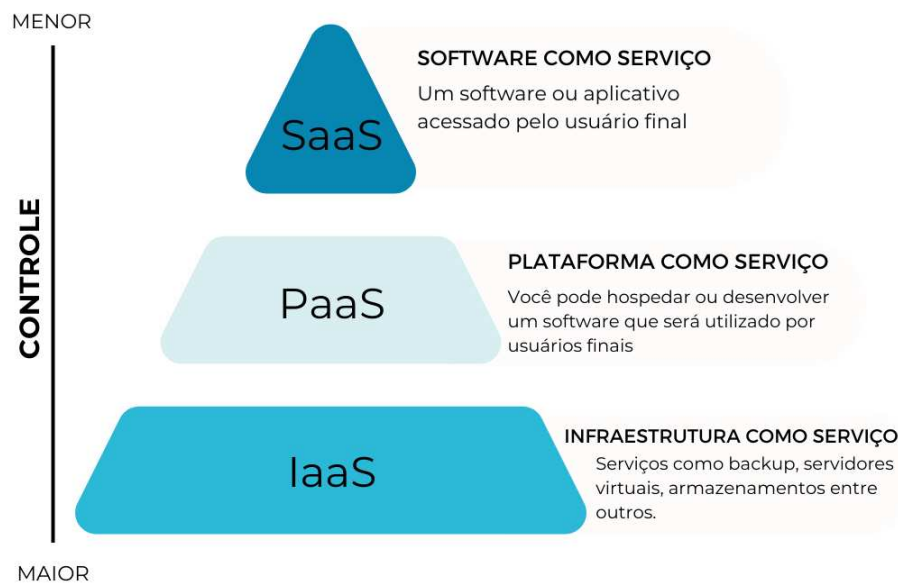


Figura 2.2: Pirâmide de IaaS, SaaS e PaaS que ilustra plataformas usadas em cada um dos modelos. Adaptada de [46]

equipes e na infraestrutura delas como um todo, como Docker e Kubernetes, que correspondem à principal estratégia de orquestração disponível no mercado atualmente, dentre outras. A cultura de colaboração, *feedback*, redução de trabalho manual, otimização, automação, provisionamento facilitado tornam estas ferramentas úteis e com um propósito dentro destas equipes de trabalho para serem utilizadas.

Os contêineres, que serão explicados na Seção 2.1.2, estão bastante relacionados com DevOps, na medida em que facilitam a colaboração e organização de ambientes necessárias para o trabalho de mais baixo nível. Dessa forma, os contêineres foram grandes catalisadores para a disseminação da cultura DevOps, na qual a ideia consistia em criar código para construir infraestrutura de maneira ágil, rastreável e padrão.

Como destacado pela Figura 2.3, o DevOps tem um conceito de infinito, em que se abordam a entrega contínua e a implementação contínua, que são metodologias para se disponibilizar ao cliente final uma aplicação de qualidade contemplando o que está escrito na roda do infinito, a saber: teste, planejamento, implementação, monitoramento,

Conceito Dev Ops

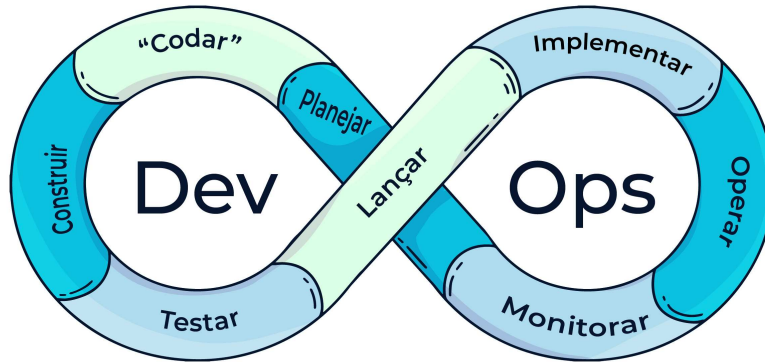


Figura 2.3: Representação do ciclo de vida de processos de DevOps. Adaptado de [20]

operação, lançamento, o código em si e seu preparo para produção.

2.1.2 Docker

Segundo a empresa Docker, um contêiner pode ser definido como uma unidade de *software* capaz de agrupar o código de uma aplicação e todas as suas dependências, possibilitando que o respectivo aplicativo seja executado de forma rápida e confiável, ainda que se mude o ambiente computacional. O conceito de contêineres vem da ideia de *cgroup*, que é um conceito dentro do sistema operacional Linux⁵ de isolar a execução de recursos, facilitando o teste de aplicações.

Com a evolução dos anos, um aprimoramento deste conceito foram os contêineres, cuja grande difusora foi a empresa Docker, que em 2013 resolveu abrir seu código-fonte. O contêiner vem da ideia de que podemos escalar mais recursos e atender uma demanda realizada sem precisar operar tanta capacidade e quantidade de infraestrutura, como memória, núcleos de CPU, dentre outras. Pelo fato de um contêiner consistir em um único processo dentro de uma execução de hardware muito menor do que seria uma máquina virtual ou física inteira, com sistema operacional completo e outros atributos, ele facilita usos em alta escala.

Como podemos perceber na Figura 2.4, uma infraestrutura levantada apenas com con-

⁵<https://www.linux.org/>

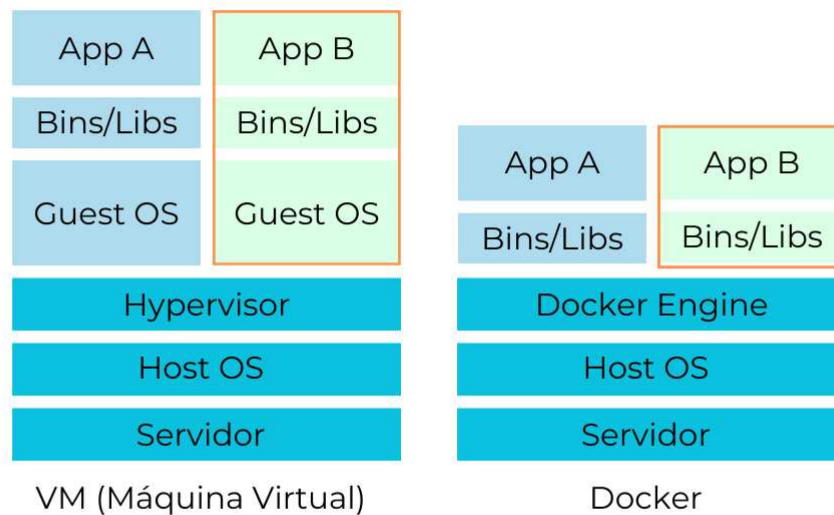


Figura 2.4: Comparação entre uma infraestrutura com uso de máquina virtual e de contêineres. Adaptado de [21]

têineres demanda muito menos configurações e recursos. Assim, são necessários somente a infraestrutura física, um sistema operacional que carregue a aplicação dos contêineres e os contêineres propriamente ditos. Já no caso de uma máquina virtual, além da infraestrutura física, é necessário o sistema operacional, um controlador dessas máquinas virtuais (chamado de hypervisor) e ainda as próprias máquinas, onde dentro delas também vão existir sistemas operacionais sendo monitorados. Apenas desta forma é possível executar aplicações nas máquinas virtuais, o que tende a ser mais complexo e pesado de se trabalhar em relação aos contêineres, além de obviamente aumentar o custo das operações de infraestrutura.

Essa característica dos contêineres, de serem leves e consumirem menos recursos, facilita o escalonamento de aplicações, haja vista que dessa forma é possível executar mais aplicações sobre uma mesma plataforma computacional (hardware mais sistema operaci-

onal hospedeiro) e, assim, atender um número maior de demandas recebidas.

Finalmente, é importante destacar que a empresa Docker atuou não apenas como uma grande facilitadora e difusora do conceito de contêineres, mas também do próprio conceito de DevOps, ao contribuir para automatizar e otimizar recursos de infraestrutura e servir aplicações de maneira mais simples e colaborativa, premissas essas que estão dentro da cultura DevOps.

2.2 Kubernetes

No escopo das tecnologias e soluções de mercado voltadas para a “orquestração” de contêineres, o Kubernetes é, segundo a própria documentação oficial, um sistema de código aberto que automatiza, em larga escala, a implantação, o dimensionamento e o gerenciamento de aplicativos em contêineres. Podemos utilizar contêineres de diferentes empresas em sua escalabilidade, mas a junção de Docker com Kubernetes é aquela que tem sido amplamente utilizada e que possui implementações mais bem sucedidas, contando vasta documentação.

O Kubernetes trabalha com o conceito de *cluster*, que é a entidade que agrupa todas as máquinas físicas gerenciadas por uma mesma instância do Kubernetes. Nessas máquinas físicas, chamadas de nós, rodam os *pods*, onde são executadas as aplicações. Os *pods* correspondem à menor unidade do ambiente, em que ficam armazenados os contêineres que ficarão dentro do *cluster*. Assim, podem ser armazenados um ou mais contêineres em um *pod*, que fica responsável por fazer a ponte no que diz respeito às configurações de escalabilidade e demanda de uso de uma aplicação. Já os nós são máquinas (virtuais ou físicas) que estão associadas a uma máquina principal denominada painel de controle (*control plane*), que são escalonadas de acordo com a necessidade do *cluster* e que rodam os *workloads* do Kubernetes.

Quando se escala contêineres em um ambiente Kubernetes, especialmente em um cenário de nuvem, diversos fatores podem impactar no bom funcionamento da infraestrutura, tais como performance, recursos, falhas, entre outros.

A Figura 2.5 ilustra brevemente o funcionamento do Kubernetes, com todos os seus componentes e parâmetros de controle que o fazem estar disponível para uso. Um dos principais é o plano de controle, encarregado de distribuir os *pods* pelos nós disponíveis conforme as necessidades de escalabilidade e demanda de uso (*workloads*) das aplicações. A forma na qual se dá o escalonamento de contêineres, ou *pods*, num ambiente

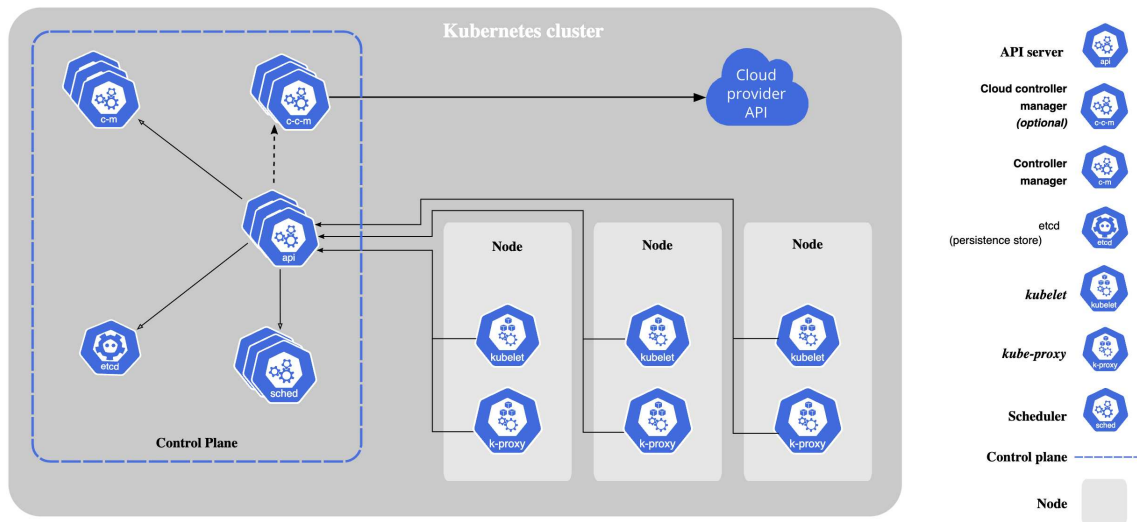


Figura 2.5: Funcionamento de um cluster Kubernetes com seus componentes principais. Extraído de [22]

Kubernetes, especialmente na nuvem, pode impactar significativamente o custo para o provimento dos serviços, assim como o desempenho geral percebido pelos usuários.

Além do plano de controle, existem também os seguintes componentes: (i) o kube-proxy, responsável pelas ligações de *pods* por meio da rede interna do *cluster*; (ii) o kubelet, que fica dentro de cada nó e é responsável pelo gerenciamento dos nós do *cluster*; (iii) o etcd, que é um banco de metadados do Kubernetes, com informações primordiais e sensíveis do *cluster*; e (iv) API-Server, que seria o principal meio de comunicação e referência entre o controlador do *cluster* e o próprio *cluster*.

Acima de tudo, há também um outro componente muito importante e que é foco desta dissertação, que é o Kube-scheduler, a ser tratado especificamente na próxima subseção.

2.3 Kube-scheduler

O Kubernetes, por padrão, vem acompanhado de um agendador de *pods* denominado de Kube-scheduler [38] ou Kubernetes Scheduler. Este faz a alocação entre um *pod*, que precisa ser disponibilizado, para um nó que tem condições de recebê-lo. Essa alocação é realizada com base em um critério de ranqueamento para os nós válidos e passíveis de receberem *pods*, que os enfileira segundo esse critério.

A criação de um *pod* costuma ser demandada por alguma aplicação em execução ou pela equipe de desenvolvedores e/ou operadores dos serviços em nuvem. Por intermédio de alguns parâmetros de configuração, sempre que um novo *pod* é criado, é possível as-

sociar prioridades a determinados nós, a critério dos administradores, que são usadas para determinar a alocação deste novo *pod*. Os parâmetros mais importantes são **NodeAffinity**, **TaintToleration**, **PodTopologySpread** e **InterPodAffinity**.

O **NodeAffinity** é utilizado para especificar as preferências e restrições relacionadas aos nós do *cluster*. É usado para definir onde um *pod* será executado, como em nós que tenham um determinado rótulo, ou para evitar que um *pod* seja executado em nós com um rótulo específico. O **InterPodAffinity** permite especificar as preferências de execução de alguns *pods* em relação a outros *pods*. Pode ser usado para que grupos de *pods* sejam executados num determinado nó ou em nós separados. O **PodTopologySpread** controla a distribuição dos *pods* em relação à topologia do *cluster*, ajudando a garantir alta disponibilidade e a tolerar falhas através da distribuição de *pods* de um mesmo grupo em diferentes nós do *cluster*. Por último, o **TaintToleration** permite que sejam especificados *taints* (marcas) nos nós do *cluster* para, em seguida, definir quais *pods* os “toleram”. Isso é útil em determinadas situações nas quais se deseja restringir quais *pods* podem ser executados em nós específicos.

Esses e outros rótulos, que podem ser consultados em [38], possuem pesos diferentes e, portanto, compõem um nível de hierarquização para decidir para onde um *pod* vai, de acordo com as configurações instaladas no Kube-scheduler vigente naquele *cluster* Kubernetes. Algumas configurações podem se sobressair em relação a outras e possuir mais peso na decisão de alocação de um *pod* a um nó. Dessa maneira, o processo de alocação de um *pod* a um nó passa por duas fases: de **filtro** e de **pontuação**. A parte de filtragem busca, de acordo com os filtros estabelecidos e a capacidade de recursos, quais são os nós que podem ter os *pods* associados, gerando uma listagem. Caso não haja nós na lista oferecida, o *pod* não pode ser agendado. A etapa de pontuação classifica o nó mais adequado a receber o *pod* de acordo com o ranqueamento que recebeu na listagem de nós.

Apesar dos vários tipos de rótulo que podem ser utilizados no Kube-scheduler, não há qualquer parâmetro disponibilizado por padrão pelo orquestrador que faça uma ponderação em relação a um ou mais custos definidos pelo usuário, tal como de abertura de um nó ou de alocação de *pod* a um nó, em uma infraestrutura física local ou em nuvem. O custo do processo de alocação de *pods* a nós pelo orquestrador é definido pelo preço de inserir determinado *pod* em uma máquina (nó) que estiver sendo disponibilizada. E também quantos *pods* aquele nó consegue suportar, a depender do tamanho da máquina que fora alocada para ser acoplada ao orquestrador.

Este trabalho busca propor uma solução que aborde o processo de alocação conside-

rando os diversos custos elencados, no qual se consiga alocar a maior quantidade possível de *pods* no menor número de nós, assim como considerando outras características que visem uma minimização dos custos. Este é um ponto muito importante, na medida em que a solução proposta busca permitir a economia de recursos da infraestrutura adotada.

2.4 Problemas NP-Difíceis

Define-se que um problema, na sua versão de decisão, é aquele para o qual se deseja verificar a veracidade ou não de uma questão. As instâncias de um tal problema contemplam somente dois tipos de solução: “SIM” ou “NÃO”. Por exemplo, o problema de determinar um “caminho curto” em um grafo, na sua versão de decisão, pode ser definido como: “Dado um grafo $G = (V, E)$, dois vértices $s, t \in V$ e um inteiro $k > 0$, deseja-se determinar se existe um caminho do vértice s ao vértice t que tenha comprimento menor do que k ”. Já um problema na sua versão de otimização é aquele para o qual se deseja verificar a existência e identificar a melhor solução possível. Por exemplo, o problema de determinar o “caminho curto” em um grafo, na sua versão de otimização, pode ser definido como: “Dado um grafo $G = (V, E)$ e dois vértices $s, t \in V$, deseja-se determinar o caminho mais curto do vértice s ao vértice t ”.

A classe $\mathcal{NP}$ -Completo de problemas é definida para problemas nas suas versões de decisão. Assim, formalmente, define-se que um problema π está na classe $\mathcal{NP}$ -Completo se satisfaz às duas seguintes propriedades [53]: (i) π está na classe $\mathcal{NP}$ e (ii) existe uma transformação polinomial de todo problema em $\mathcal{NP}$ ao problema π . Ressalta-se que um problema está na classe $\mathcal{NP}$ caso se verifique uma possível solução em tempo polinomial [56].

Por sua vez, a classe $\mathcal{NP}$ -Difícil de problemas é definida para problemas nas suas versões de otimização. Se um problema, na versão de decisão, está na classe $\mathcal{NP}$ -Completo, então a sua versão de otimização está na classe $\mathcal{NP}$ -Difícil. Formalmente, um problema π é $\mathcal{NP}$ -Difícil se ele satisfaz a propriedade (ii) supracitada, mas não se sabe ao certo se π está ou não em $\mathcal{NP}$ [56].

Seja π um problema $\mathcal{NP}$ -Difícil, então, pela definição da classe, existe uma transformação polinomial de todo problema em $\mathcal{NP}$ a π , o que implica que π é tão difícil quanto os problemas $\mathcal{NP}$ s mais difíceis. Esses são problemas para os quais não se conhece ainda um algoritmo polinomial que os resolva, isto é, que encontre uma solução ótima para toda instância do problema em tempo expresso por uma função polinomial. Dessa forma, abordar problemas $\mathcal{NP}$ -Difíceis pode ser desafiador, mas existem métodos que podem ser

adotados e que sacrificam, de certa maneira, ou o tempo de processamento ou a qualidade da solução, correspondendo aos exatos ou heurísticos/aproximados.

Este trabalho se ocupa de um problema $\mathcal{NP}$ -Difícil, que corresponde ao escalonamento de *pods* no contexto de um *cluster* Kubernetes, respeitando-se as devidas restrições. Na abordagem a esse problema, optou-se por trabalhar com Programação Inteira, que corresponde a um método exato.

2.5 Otimização Matemática

A otimização matemática é uma área da Computação que lida com a maximização ou a minimização de funções. O objetivo é encontrar o valor máximo ou mínimo de uma função em relação a um conjunto de variáveis, sujeito a certas restrições. Em termos matemáticos, supondo um problema de minimização, dada uma função $f : \mathcal{S} \rightarrow \mathcal{T}$, deseja-se encontrar um $\mathbf{x}$ pertencente ao domínio $\mathcal{S}$ que leve ao menor valor do contradomínio $\mathcal{T}$, satisfazendo as restrições, isto é:

$$\forall \mathbf{x}' \in \mathcal{S} \quad f(\mathbf{x}) \leq f(\mathbf{x}') \quad (2.1)$$

Em outras palavras, a otimização matemática trata de encontrar uma solução ótima para um problema, em relação a alguma função objetivo definida a fim de maximizar algum benefício ou minimizar algum custo, de acordo com o domínio abordado. Desse modo, as variáveis envolvidas na função objetivo podem representar medidas físicas, decisões de negócios, alocação de recursos, entre outras.

Em alto nível, existem dois tipos principais de problemas de otimização:

- **Otimização sem restrições:** Nesse caso, busca-se o máximo ou mínimo de uma função sem impor restrições específicas às variáveis.
- **Otimização com restrições:** Aqui, além de maximizar ou minimizar a função objetivo, existem restrições que as variáveis devem satisfazer. Essas restrições podem ser igualdades ou desigualdades, a depender do problema abordado. A solução envolve considerar as restrições junto com a função objetivo para encontrar valores que atendam a ambos os requisitos.

A área de Otimização se subdivide em diversas subáreas, a depender das características das variáveis, das restrições e da função objetivo adotadas na modelagem do Problema

de Otimização, tais como: Otimização Combinatória, Otimização Convexa, Otimização Não-Linear, Otimização Linear, Otimização Inteira, etc.

Tipicamente, a área da Computação tem interesse em problemas de Otimização Combinatória, que são aqueles que envolvem a maximização (ou minimização) de uma função objetivo $f(\cdot)$ cujo domínio é discreto. Assim, tais problemas possuem soluções que se compõem de um número discreto de elementos que levem à maximização (ou minimização) da função objetivo. Exemplos de problemas de Otimização Combinatória são: Problema do Caixeiro Viajante, Problema da Mochila, Problema de Roteamento de Veículos, Problema de Alocação de Tarefas, etc.

Nas próximas subseções, serão brevemente explicados alguns conceitos de Otimização Combinatória que foram utilizados neste trabalho.

2.5.1 Programação Linear

A Programação Linear (PL) é uma técnica de otimização utilizada para resolver problemas nos quais se busca maximizar ou minimizar uma função linear, sujeita a um conjunto de restrições lineares. Em outras palavras, tanto a função objetivo quanto as restrições são definidas por funções lineares.

Em 2.5.1, exibe-se um exemplo de formulação de PL retirado de [37]. Esta formulação de PL é um exemplo de maximização de lucro, em que a variável x_1 representa caixas de chocolate de um determinado sabor e x_2 , de outro. A produção destas caixas possui algumas restrições, a saber: a primeira restrição (1) determina que x_1 não pode ser fabricado mais de 200 vezes ao dia; já a segunda (2) define que x_2 não pode ser produzido mais de 300 vezes ao dia. Adicionalmente, a terceira restrição (3) aponta que a produção de ambos os chocolates juntos também não pode ultrapassar a marca de 400 chocolates ao dia, mas deve-se obrigatoriamente fabricar chocolates no dia, tal como definido pela quarta restrição (4) onde x_1 e x_2 são maiores ou iguais a zero. Nesse exemplo, o objetivo corresponde a encontrar os valores das variáveis de decisão x_1 e x_2 que maximizem a função objetivo, respeitando as restrições impostas.

Função objetivo: $\max x_1 + 6x_2$

sujeita às seguintes restrições:

$$x_1 \leq 200 \quad (1)$$

$$x_2 \leq 300 \quad (2)$$

$$x_1 + x_2 \leq 400 \quad (3)$$

$$x_1, x_2 \geq 0 \quad (4)$$

Como uma equação linear envolvendo x_1 e x_2 define uma reta no plano 2D, tem-se que as inequações lineares definem um semi-plano que corresponde à região em um dos lados da reta [37]. Dessa forma, todas as soluções viáveis desse PL estão contidas no conjunto definido pela interseção dos cinco semi-planos referentes às cinco restrições, conforme mostra a Figura 2.6.

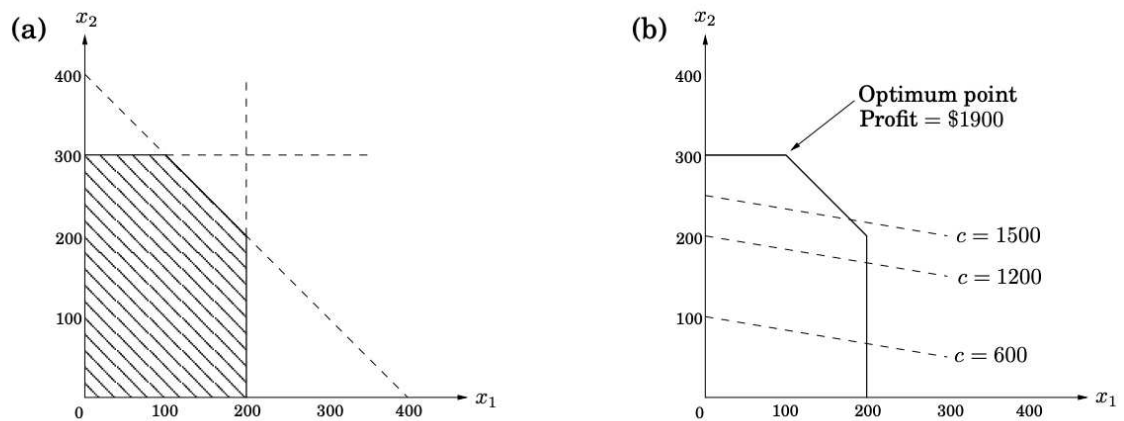


Figura 2.6: O lado (a) mostra a região viável para um programa linear. O lado (b) demonstra as linhas de contorno da função objetivo 2.5.1 para diferentes valores do lucro c , em que o ponto ótimo de lucro está em 1900. Extraído de [37]

A solução de problemas de Programação Linear pode ser realizada usando métodos específicos, como o método **Simplex** [37] ou o método de Pontos Interiores, dependendo da complexidade do problema. Alguns resolvidores MIP possuem biblioteca disponível para as principais linguagens de programação, como C, Java e Python, como é o caso do Gurobi⁶ e do CPLEX⁷.

2.5.2 Programação Inteira

A Programação Inteira (PI) é um ramo da otimização matemática que lida com problemas nos quais as variáveis de decisão assumem somente valores inteiros. Dessa forma,

⁶<https://www.gurobi.com/>

⁷<https://www.ibm.com/products/ilog-cplex-optimization-studio>

é utilizada para resolver problemas nos quais as soluções viáveis são restritas a números inteiros, em oposição a valores contínuos. Diz-se, pois, que os problemas de PI são formulados em que contextos em que é devido tomar decisões em situações discretas. O problema de PI é $\mathcal{NP}$ -Difícil, visto que diversos problemas $\mathcal{NP}$ -Difíceis, tais como o Problema do Caixeiro Viajante ou o Problema de Cobertura de Conjuntos, podem ser formulados por meio de PI [53].

A formulação geral de um problema de Programação Inteira pode ser representada da seguinte maneira:

$$\begin{aligned} &\text{Maximizar (ou Minimizar)} f(x) \\ &\text{sujeito a} \quad a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1 \\ &\quad a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2 \\ &\quad \vdots \\ &\quad a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m \\ &\quad x_1, x_2, \dots, x_n \text{ inteiros} \end{aligned}$$

ou, em alguns casos,

$$\begin{aligned} &\text{Maximizar (ou Minimizar)} \quad c_1x_1 + c_2x_2 + \dots + c_nx_n \\ &\text{sujeito a} \quad a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1 \\ &\quad a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2 \\ &\quad \vdots \\ &\quad a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m \\ &\quad x_1, x_2, \dots, x_n \text{ binários (0 ou 1)} \end{aligned}$$

Neste contexto, $c_1, c_2, \dots, c_n$ são coeficientes que representam a função objetivo a ser maximizada ou minimizada, e a_{ij} são os coeficientes das restrições. As variáveis $x_1, x_2, \dots, x_n$ são as variáveis de decisão, e as restrições definem as condições que devem ser satisfeitas.

Problemas de PI são comuns em situações onde as decisões tomadas precisam ser expressas como valores inteiros, como na alocação de recursos, programação de projetos, design de redes, entre outros. Pelo fato de ser $\mathcal{NP}$ -Difícil, a resolução de problemas de PI costuma ser mais complexa computacionalmente do que problemas de PL tradicional, visto que ainda não se conhece um algoritmo eficiente para resolver tais problemas de ma-

neira geral. Assim, em geral, a busca por soluções ótimas costuma envolver a exploração exaustiva de diferentes combinações de valores inteiros. Uma abordagem tradicional para abordar problemas de PI é por meio da técnica de *Branch-and-Bound*, a ser explicada na Seção 2.5.3.2, ou por Programação Dinâmica.

2.5.3 *Capacitated Facility Location Problem*

O *Facility Location Problem* (FLP) corresponde a um problema de Otimização Combinatória em que se deseja encontrar os locais em que novas instalações (fábricas ou armazéns) devem ser colocadas a fim de suprir insumos a determinados clientes. Mais especificamente, a decisão dos locais onde colocar as instalações deve levar em conta os custos de abertura dessas, de transporte aos clientes, dentre outros, associados ao problema. A Figura 2.7 ilustra um exemplo de instância do FLP.

Este problema guarda certa similaridade com o problema abordado nesta dissertação, tal como será apresentado no Capítulo 4. O *Capacitated Facility Location Problem* (CFLP) foi utilizado como base porque, enquanto este busca as melhores localizações para novas instalações para otimizar a operação fabril, nós buscamos através da implementação de nosso algoritmo o melhor nó para a alocação de um determinado pod. O objetivo de ambos os problemas é o mesmo: minimizar os custos (com base nas variáveis de decisão de custos fixos e variáveis, capacidade e demanda) e maximizar a eficiência do sistema, buscando a melhor combinação que leve a isso, conforme mostra um exemplo com galpões na Figura 2.7.

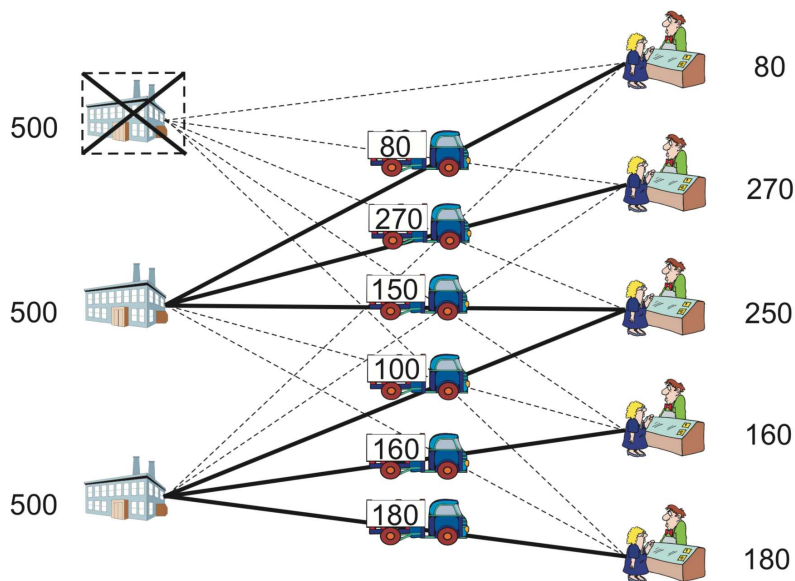


Figura 2.7: Exemplo de instância do FLP, em que se deseja escolher a melhor forma de distribuir insumos aos clientes finais. Extraído de [45]

O *Capacitated Facility Location Problem* (CFLP) é uma variante do FLP em que as instalações possuem uma capacidade máxima de insumos que podem prover aos clientes. Isto tende a aumentar a dificuldade do problema, pois introduz uma restrição adicional a ser considerada na obtenção de possíveis soluções, aumentando assim a natureza combinatoria do problema. O CFLP é considerado pela literatura um problema $\mathcal{NP}$ -Difícil, pois, segundo [4] não se conhece um algoritmo polinomial capaz de resolvê-lo em todos os casos. O CFLP também é considerado $\mathcal{NP}$ -Difícil por sua capacidade de modelar problemas de otimização combinatoria complexos, como o **Problema da Soma de Subconjuntos**, cuja solução eficiente é altamente improvável. O Problema da Soma de Subconjuntos busca encontrar, dentro de um conjunto de números inteiros, um subconjunto cuja soma seja igual a um valor referenciado a ele. Esse problema é conhecido por sua complexidade computacional e é utilizado como referência na classe $\mathcal{NP}$ -completo, segundo [53].

2.5.3.1 Formulação de Programação Inteira Mista

Uma das formas de abordar o CFLP é por meio de uma formulação de Programação Inteira Mista (*Mixed Integer Programming* - MIP). Tal formulação pode tipicamente apresentar variáveis de decisão da seguinte natureza: binárias, para representar a abertura ou não de uma dada fábrica; e contínuas, para indicar o percentual da demanda de um dado cliente que é atendido por uma fábrica. O exemplo abaixo mostra uma formulação retirada de [55] contendo as variáveis de decisão binárias x_i que indicam se uma instalação i está aberta ou não, assim como as variáveis contínuas $y_{i,j}$ que indicam a quantidade de produto transportada da fábrica i para o cliente j .

$$\text{Minimizar: } \sum_i \sum_j (c_{i,j} \cdot y_{i,j}) + \sum_i f_i \cdot x_i$$

com restrições:

$$\sum_i y_{i,j} = D_j \quad \forall j \quad (1)$$

$$y_{ij} \leq M \cdot x_i \quad \forall i, j \quad (2)$$

$$x_i \in \{0, 1\} \quad \forall i \quad (3)$$

$$y_{ij} \geq 0 \quad \forall i, j \quad (4)$$

Nessa formulação, os coeficientes utilizados possuem o seguinte significado: f_i se refere ao custo fixo de abrir a fábrica i , $c_{i,j}$ corresponde ao custo de transporte por unidade de produto entre a fábrica i e o cliente j e, por fim, D_j à demanda feita por cada cliente j

que precisa ser atendida.

A formulação busca minimizar os custos de abertura de fábricas e de atendimento aos clientes, ao mesmo tempo que busca garantir o atendimento de toda demanda feita pelos clientes. Dessa forma, a primeira restrição (1) define que, para um dado cliente j , o somatório das quantidades de produto atendidas por cada fábrica i seja igual à demanda total D_j do cliente j . Por sua vez, a segunda restrição (2) indica que um cliente j só pode ser atendido por uma fábrica i caso essa fábrica i esteja aberta. A terceira restrição (3) define que as variáveis x_i somente assumem valores binários. Finalmente, a quarta restrição (4) define as variáveis $y_{i,j}$ assumem valores reais maiores ou iguais a zero, indicando que a quantidade de produtos transportados não deve ser negativa.

2.5.3.2 *Branch-and-Bound*

O *Branch-and-Bound* (B&B) [41] é um algoritmo usado para resolver problemas de Programação Inteira (PI) e MIPs. Esse método utiliza uma estratégia de divisão e conquista para explorar o espaço de busca de soluções e encontrar a solução ótima ou uma solução suficientemente boa para o problema.

A ideia básica do *Branch-and-Bound* é decompor o problema original em subproblemas menores, por meio da fixação do valor de algumas variáveis, resolver esses subproblemas e utilizar técnicas de poda (*pruning*) para eliminar ramos do espaço de busca que não satisfazem a um certo critério. Para tal, o algoritmo testa constantemente valores de limites inferiores e superiores, e utilizando-os na tomada de decisão.

O *Branch-and-Bound* é especialmente eficaz em problemas nos quais a estrutura do espaço de busca pode ser explorada de maneira inteligente. Ele é amplamente utilizado na abordagem a problemas de Otimização Combinatória, tais como design de redes, alocação de recursos, entre outros.

Existem diferentes estratégias de implementação do *Branch-and-Bound*, e suas eficiências podem depender das características específicas do problema em questão. Além disso, muitos resolvidores MIP, a serem explicados na Seção 2.5.4, incorporam métodos baseados em *Branch-and-Bound* para resolver problemas de PI ou MIP. Na abordagem proposta nesta, por exemplo, é aplicado um método derivado de *Branch-and-Bound* na abordagem a um problema formulado como MIP.

Conforme definido em [37], na abordagem a um problema de maximização ou de minimização, o *Branch-and-Bound* busca eliminar soluções que já se sabe que não atingirão o objetivo, decretando inicialmente uma base para garantir a eficiência do método. O

Algoritmo 1 mostra a estrutura geral do método de *Branch-and-Bound*.

Algoritmo 1: Estrutura geral do método de *Branch-and-Bound* (adaptado de [37]).

```
1: Defina  $S = \{P_0\}$  como o conjunto de subproblemas ativos
2:  $\text{melhorAteEntao} \leftarrow \infty$ 
3: while  $S$  não está vazio do
4:   Escolha um subproblema (solução parcial)  $P \in S$  e remova-o de  $S$ 
5:   Expanda  $P$  em subproblemas menores  $P_1, P_2, \dots, P_k$ 
6:   for cada  $P_i$  do
7:     if  $P_i$  é uma solução completa then
8:       Atualize  $\text{melhorAteEntao}$ 
9:     else if  $\text{limiteInferior}(P_i) < \text{melhorAteEntao}$  then
10:      Adicione  $P_i$  a  $S$ 
11:    end if
12:  end for
13: end while
14: return  $\text{melhorAteEntao}$ 
```

O Algoritmo 1 inicialmente define um conjunto de subproblemas ativos (linha 1). Na segunda linha, se define uma variável **melhorAteEntao** que guarda a melhor solução encontrada até o momento, que é inicializada com o valor infinito. Dentro do *while* (laço) que se inicia na linha 3, é escolhida uma possível solução parcial P dentro do conjunto S de subproblemas ativos, de modo que P é dividida em parte menores (linha 5). Para cada uma dessas partes P_i , caso P_i seja uma solução completa, atualiza-se a melhor solução conhecida até o momento (linhas 6-8). Caso P_i ainda não seja uma solução completa, verifica-se se o valor do limite inferior fornecido por P_i é menor do que o valor da melhor solução conhecida (limite superior) até o momento e, caso seja, adiciona-se P_i ao conjunto de subproblemas ativos S (linhas 9-10). Note-se que, caso o limite inferior fornecido por P_i seja maior do que o valor da melhor solução conhecida, então se desconsidera o ramo originado em P_i , o que corresponde a realizar uma poda (*pruning*). Esses passos são executados pelo algoritmo enquanto o conjunto S não estiver vazio. Finalmente, na linha 14, é retornada a melhor solução encontrada.

2.5.4 Resolvedor MIP

Um resolvedor MIP é um software projetado para resolver problemas de Programação Inteira Mista (MIP). Problemas MIP envolvem a otimização de uma função objetivo sujeita a restrições lineares, onde algumas das variáveis de decisão podem assumir apenas valores inteiros.

Os resolvedores MIP são ferramentas especializadas projetadas para encontrar solu-

ções eficientes para esses tipos de problemas. Eles implementam algoritmos avançados para explorar o espaço de busca e identificar a combinação ótima de valores para as variáveis de decisão envolvidas.

Mais especificamente, estes sistemas realizam a implementação de técnicas de B&B e suas variações, a fim de resolver problemas de Programação Linear Inteira em busca de um cenário ótimo. Essas são responsáveis por gerenciar as árvores de busca do B&B, escolher quais ramos explicar, calcular limites, adicionar cortes sob demanda, dentre outras funcionalidades.

Existem vários softwares comerciais dedicados a resolver problemas MIP, como CPLEX, Gurobi, SCIP, CBC (Coin-OR Branch and Cut), entre outros. Esses resolvers tipicamente fornecem uma *Application Programming Interface* (API) para que o desenvolvedor implemente a sua formulação e a submeta à resolução do resolver MIP.

2.5.5 Gurobi

O Gurobi [51] é uma das principais ferramentas de resolução MIP disponíveis atualmente no mercado e foi aquela adotada neste trabalho para a implementação e realização de experimentos com a formulação de IP proposta. O Gurobi contém um conjunto de métodos exatos e heurísticos à disposição para que sejam usados para a abordagem de problemas de Otimização Combinatória, tais como *Branch-and-Bound*, *Branch-and-Cut*, adição de cortes (*Cutting Planes*), Simplex, Método de Pontos Interiores, busca local, etc. Dessa forma, o Gurobi pode ser utilizado na resolução de problemas de Otimização Combinatória que se encontram em um espectro que vai de Programação Linear a Programação Quadrática Mista-Inteira.

Para resolver os problemas de otimização, o Gurobi utiliza o algoritmo de *Branch-and-Bound* com uma extensão que se utiliza de cortes adicionais para aumentar a eficiência da busca pela solução ótima, que recebe o nome de *Branch-and-Cut*.

Existem também outras características que associadas tornam o Gurobi uma ferramenta primorosa para encontrar a solução ótima. São elas, por exemplo, a adição de cortes, em que se restringe um conjunto de soluções viáveis, e os cortes ocorrem de maneira desigual e linear. O Gurobi já possui uma ampla gama de cortes disponíveis nativamente: cortes de cobertura, cortes de Gomory, cortes de clique, etc. É justamente o uso de cortes na abordagem ao problema que caracteriza a aplicação de *Branch-and-Cut* pelo resolver.

Também existem os cortes adicionados sob demanda (*On-Demand Cuts*), que ocor-

rem durante a solução do problema no *Branch-and-Cut*, e ajudam a encontrar a solução ótima, em um processo de refinamento dos resultados, além de trazer vantagens como paralelismo eficiente para explorar múltiplos núcleos de CPU e orquestradores. Ademais, o resolvidor também faz uso de um conjunto de heurísticas avançadas quando o problema abordado se mostra muito complexo.

3. Trabalhos Relacionados

Neste capítulo, são descritos os trabalhos disponíveis na literatura que se relacionam com o tema abordado nesta dissertação.

3.1 Alocação de *pods*

Dentre os trabalhos na literatura que possuem relação com o tema tratado nesta dissertação, destaca-se o trabalho de Zhang (et al.) [1] que aborda um problema de alocação de recursos em um cenário de *Multi-Access Edge Computing* (MEC) com mobilidade. No cenário abordado, veículos usuários de um aplicativo de transporte em carros de passeio precisam estabelecer conexões sem fio, enquanto se movimentam, a alguma base de rádio dentre as que abrigam os servidores MEC, conforme o raio de alcance dessas bases. Essas bases possuem capacidade limitada e precisam ser devidamente dimensionadas para evitar sobrecarregadas. O trabalho propõe um modelo de otimização para dimensionar a alocação de recursos nos servidores MEC instalados nas bases, buscando maximizar a disponibilidade de conexões para os veículos e minimizar latências. Há similaridades com o problema abordado nesta dissertação, porém a dinâmica na alocação de recursos tem natureza diferente da que é tratada neste cenário.

Nesse artigo, foi proposta a Formulação 3.1, que serviu de inspiração para esta dissertação, que busca maximizar os recursos computacionais dos servidores MEC, na tentativa de atender todos os veículos conectados a esses servidores de tal forma que o serviço não seja interrompido por alguma queda de conexão ou intermitência. Para que isso seja possível, a função objetivo adotada, dada por $-\alpha x_{i,t} + \beta u_{i,t} - \gamma q_{i,t}$, corresponde à estratégia de recompensa na abordagem de Aprendizado por Reforço Profundo (*Deep Reinforcement Learning*) [32] adotada, onde α representa o custo da máquina reservada (servidor MEC), β é o lucro para um veículo conectado e γ é a punição em caso de intermitências na conexão. Em outras palavras, a recompensa é representada pela utilidade dos recur-

os do sistema obtida a partir do lucro dos veículos conectados menos o custo de reserva de servidores do MEC e a punição por falhas de conexão. Tal problema, assim como o abordado nesta dissertação, também corresponde a um problema $\mathcal{NP}$ -Difícil [1].

$$\begin{aligned}
& \max && \sum (-\alpha x_{i,t} + \beta u_{i,t} - \gamma q_{i,t}) \\
& \text{sujeito a} && \\
& d_{i,j,t} y_{i,j,t} &\leq & D x_{i,t} & (3.1) \\
& \sum_{i \in I} y_{i,j,t} &\leq & 1 & (3.2) \\
& \sum_{i \in I} y_{i,j,t} &\leq & U x_{i,t} & (3.3) \\
& y_{i,j,t} &\leq & x_{i,t} & (3.4) \\
& x_{i,t}, y_{i,j,t} &\in & \{0, 1\} & (3.5)
\end{aligned}$$

Para fazer com que a formulação ficasse aderente ao cenário aplicado, foram adicionadas restrições, a saber: a Equação (3.1) limita a latência entre um veículo conectado e seu servidor MEC. Já a Equação (3.2) indica que um veículo conectado pode se conectar a, no máximo, um servidor por vez; por sua vez, a (3.3) garante que a utilização dos recursos não deve exceder a capacidade de um servidor MEC (restrição de capacidade); a Equação (3.4) implica que um veículo conectado só pode se conectar a um servidor MEC reservado; e, por fim, a Equação (3.5) especifica que as variáveis de decisão $x_{i,t}$ e $y_{i,j,t}$ que representam as reservas dos servidores MEC e as conexões de veículos conectados são binárias.

Botez (et al.) [10] tem como motivação apresentar um consumo de recursos de aplicativo de telemetria IoT (*Internet of Things*) em contêineres em diferentes cenários de nuvem, trazendo uma configuração de hardware física com roteadores, servidor *bare metal*, etc. Para abordar a questão dos contêineres, foi feita uma divisão da rede suportada 5G para separar funções de rádio, transporte e redes centrais, fatiando-a em busca de escalabilidade. Assim, o trabalho aborda o problema de escalabilidade de funções virtualizadas de rede, ou *Network Functions Virtualization* (NFV), em ambientes de computação em nuvem voltados para aplicações no estado da arte de redes 5G. Nesse trabalho, o Kubernetes é usado como orquestrador para escalonar as funções de rede (*User Plane Functions*, ou UPS), porém a solução não contempla um modelo em larga escala.

Foi apresentada [11] uma forma de otimizar o escalonamento de *workloads* sensíveis à latência no Kubernetes, visando diminuir o tempo de atendimento às requisições. Para tal, o gerenciamento é feito através de *Service Mesh* [9] para o agendamento dos *Pods*, implicando em uma diminuição de até 69% o tempo de resposta a depender do protocolo.

Um *Service Mesh* gerencia uma série de componentes de uma aplicação implementada que compartilham dados entre si, utilizando uma infraestrutura dedicada e usada por ela, conforme foi apontado em [49]. A camada adicional criada auxilia na comunicação dos serviços.

As requisições são fatiadas por topologia, em que topologias mais simples são rapidamente atendidas, mas topologias mais complexas também não demoram a ser contempladas. Esse artigo também menciona MEC como uma possibilidade por trás de Kubernetes e de melhorias dentro do 5G. O trabalho apresenta um *plugin* inspirado em Programação Linear que se propõe a ser uma extensão do Kube-scheduler padrão e busca verificar se um nó tem ou não capacidade para suportar a alocação de um novo *Pod*, segundo métricas de desempenho e seus níveis de prioridade definidas através de filtros e pontuações. Ademais, uma parte negativa desse trabalho é que, por envolver o uso de um *Mesh*, existe uma camada extra que sobrepõe o *cluster*, fazendo com que, mesmo que se ganhe de um lado latência, o próprio *Mesh* faz com que exista um pouco mais de latência por causa desta camada.

Finalmente, foi proposta [13] uma estratégia de alocação de *Pods* em contêineres heterogênea em um *cluster* Kubernetes, objetivando uma redução de custo, por meio de três etapas: a primeira etapa corresponde a otimizar a alocação de *Pods* nos nós existentes através do redimensionamento destas tarefas. A segunda seria a elasticidade do *cluster*, por meio de escalonamento automático. E por fim, a terceira se dá por encerrar os nós que não estiverem mais em uso ou com pouco consumo, redirecionando os *Pods* para outros nós.

Na abordagem proposta nesta dissertação, não há modificações do *cluster* ou dos contêineres em execução, com o algoritmo de escalonamento atuando apenas sobre novas demandas de alocação de *Pods*. Adicionalmente, a formulação proposta admite penalizações aos nós do *cluster* conforme a incidência de erros na execução dos *Pods* ocorresse.

3.2 Abordagens Baseadas em Programação Linear

Para além do agendamento de *pods*, foi investigado em [24] o panorama das técnicas de agendamento de *pods* mais recentes, por meio de um *survey* com o objetivo de inspirar mais trabalhos de pesquisa na área. Este trabalho está estruturado em torno da classificação de técnicas de escalonamento em quatro categorias com base no tipo de algoritmo de otimização empregado para gerar o escalonamento, a saber: modelagem matemática, heurística, meta-heurística e aprendizado de máquina. Para cada classe de algoritmos de escalonamento, se analisam e se identificam os principais prós e contras, juntamente com os principais desafios das técnicas disponíveis com base nas métricas de desempenho. Dentre os comparativos, as técnicas de modelagem matemática abordam o problema de escalonamento como um problema de otimização, como, por exemplo, PL ou MIP, que usam técnicas-padrão para encontrar a solução ideal para um determinado problema [25].

Existem outros exemplos de trabalhos que mostram o uso de modelagem matemática, como foi proposto [26] um modelo de PL para uso no escalonamento de contêineres dentro de um orquestrador Docker Swarm, que se demonstrou eficaz e efetivo dentro dos comparativos realizados. A diferença entre esse trabalho e o desta dissertação reside no fato de que o Docker Swarm é um orquestrador de contêineres mais simples em quesito de infraestrutura e gerenciamento.

O Kubernetes tem maior complexidade que o Docker Swarm no que se refere em questões como escalabilidade de recursos, onde o Kubernetes é projetado para trabalhar em larga escala. Também apresenta diferentes formas de implementação, como *auto-scaling*, *rollbacks* e *rollouts* automáticos, etc. Apresenta uma maior gama de *plugins* a serem instalados dentro do orquestrador, tem uma arquitetura com maior número de componentes e, conseqüentemente, tem uma curva de aprendizado mais íngreme.

Zhang (et al.) [26] tem sua formulação mais focada para a parte de economia de energia relacionada ao escalonamento de tarefas. Sendo assim, o foco é na redução do consumo energético dos recursos computacionais.

Mais especificamente, à medida que o Docker tem ganhado mais popularidade nos últimos anos, o fator de agendamento de contêineres se torna um papel crucial para a criação de aplicativos e serviços em contêineres. O artigo de [26] avalia questões como a conservação de energia do *host* do contêiner, os custos de extração da imagem do contêiner do registro de imagem para os *hosts* do contêiner e os custos de transição da rede de *workload* dos clientes para os *hosts* do contêiner são avaliados em combinação. Ao modelar o problema de escalonamento por meio de MIP, é proposto um escalonador bem eficaz

e adaptativo. Além disso, pode ser facilmente integrado às estruturas de orquestração de contêineres de código aberto, diferentemente da abordagem proposta nesta dissertação que, apesar da proposta futura ser a sua implementação dentro do próprio Kube-scheduler, ainda não tem sua operação dentro de um *cluster* real, tendo sido feita a validação apenas em instâncias artificiais.

Foi proposto [27] uma estrutura de agendamento de *pods* baseada em MIP para computação em nuvem baseada em contêineres. A formulação MIP busca resolver um problema NP-Difícil e leva em consideração, assim como nesta dissertação, a capacidade de utilização dos recursos, mas com o objetivo de executar os contêineres de maneira também parcial e trabalhando com tarefas que podem expirar ou não, através dos agendadores *off-line* e *on-line* criados pela solução. A formulação MIP é, diferentemente da que foi proposta nesta dissertação, totalmente focada para cenários de comparação de agendamento *off-line* e *on-line*, buscando encontrar o cenário ótimo em que se maximiza a quantidade de tarefas agendadas no *cluster*, independentemente de esse estar *on-line* ou *off-line*.

Da mesma forma, relata-se [28] uma formulação MIP com o uso de Docker e microsserviços, com a justificativa de que o algoritmo padrão utilizado por escalonadores de contêineres não é capaz de aproveitar bem suas vantagens de escalonamento de recursos computacionais e economia da infraestrutura. Com um melhor gerenciamento por meio do uso de microsserviços, a ideia do artigo é de reduzir os custos de alocação e de operação das aplicações. A proposta final é realizada no Docker Swarm, cujas diferenças para o ambiente adotado neste trabalho já foram apontadas anteriormente. A formulação tem um objetivo bem similar ao do nosso trabalho, em que se procura definir o custo dos nós e, em uma formulação de minimização, reduzir o custo de execução das aplicações, porém em um contexto de microsserviços, e levando também em consideração as bibliotecas e *frameworks* existentes para realizar o *download* deles para dentro dos contêineres a serem escalonados no *cluster* do Docker Swarm.

Por sua vez, apresentou-se [29] um controlador chamado “*Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem*” (KEIDS), para gerenciamento de contêineres em nós de nuvem de borda, levando em consideração a emissão de carbono e consumo de energia. Sua arquitetura é bastante similar à do trabalho de [1], visto que também apresenta uma infraestrutura que suporta computação de borda e de internet das coisas. É proposta então uma formulação MIP que visa minimizar a utilização de energia de nós de nuvem de borda em IoT e de emissão de carbono, usando estes custos como coeficientes na formulação, diferente da formulação proposta nesta dissertação que busca minimizar o custo de infraestrutura na execução dos *pods*.

3.3 Facility Location Problem

Por sua vez, em relação ao contexto histórico do domínio de Otimização Combinatória, o trabalho de Cornuéjols (et al.) [52], lançou as bases para abordagens eficazes na alocação de recursos em redes e definiu o *Facility Location Problem*. Este aborda o desafio de selecionar locais para instalações de galpões de estoque, considerando a minimização dos custos de operação e transporte, que corresponde a uma questão central em muitos problemas práticos de escalonamento e alocação. Analogamente, o trabalho de Walker (et al.) [7] traz uma introdução ao problema de escalonamento, definindo-o formalmente, apresentado diferentes variações de tal problema e apresenta quatro algoritmos comumente empregados na abordagem a problemas de escalonamento, em que três deles consistem em algoritmos construtivos que selecionam iterativamente a operação a ser escalonada, agindo de maneira gulosa e produzindo soluções razoáveis rapidamente, à medida que o último corresponde a uma formulação de Programação Inteira.

Dhall (et al.) [6] define o problema de escalonamento de tarefas de tempo crítico em sistemas de computação multiprocessadores, que corresponde a especificar uma ordem em que as requisições de um conjunto de tarefas devem ser executadas e o processador a ser utilizado, com o objetivo de cumprir todos os prazos com um número mínimo de processadores. São então propostas duas heurísticas que obtêm uma solução razoavelmente próxima da ótima e os seus comportamentos de pior caso são analisados.

Além desses, outro trabalho relevante é o de Kaur (et al.) [8], no qual é considerado o *Capacitated Facility Location Problem* em sua versão clássica e uma variante, de modo que foram feitos experimentos considerado o uso da estratégia de decomposição na formulação de Programação Linear Inteira Mista. O trabalho de Kuehn (et al.) [5] também trouxe impressões valiosas sobre o transporte de recursos em redes, embora seu foco tenha sido em problemas de transporte fixo, as técnicas de otimização discutidas são pertinentes para discutir processos de escalonamento.

Finalmente, em Fischetti (et al.) [3], é oferecida uma perspectiva atualizada sobre o *Uncapacitated Facility Location Problem*, considerando o reprojeto necessário nas estratégias de decomposição clássicas para que se tire proveito das novas tecnologias de hardware e software. A estratégia proposta permitiu uma aceleração na decomposição da Programação Linear Inteira Mista e foi possível resolver à otimalidade de instâncias da literatura que ainda estavam em aberto, isto é, para as quais não se conhecia ainda uma solução ótima.

Os trabalhos apresentados nesta seção destacam o uso da Programação Linear para re-

solver problemas de agendamento em orquestradores de contêineres. Baseando-se nestes trabalhos relacionados e no estudo da literatura apresentada no Capítulo 2, foi-se abordado o problema associado ao custo de alocação dos nós e agendamento de *pods* do Kube-scheduler. No Capítulo 4 será apresentada uma modelagem que busca minimizar o valor da solução final.

4. Modelagem

Neste capítulo se detalha o problema a ser resolvido e a modelagem construída para a realização dos experimentos computacionais, trazendo a formulação de PI com suas respectivas restrições.

4.1 Problema

O Kubernetes é um orquestrador de contêineres que ficam em uma mínima unidade de um *cluster* Kubernetes, chamada de *pod*. Os *pods* são responsáveis por escalar os serviços (aplicações) e atender às demandas direcionadas a esses serviços, dentro da capacidade de uso do *cluster*. A capacidade do *cluster* está diretamente relacionada ao tamanho dos recursos das máquinas computacionais alocadas para cada nó funcionar, visto que os nós correspondem às máquinas virtuais ou físicas acopladas a esse orquestrador.

Com base nas definições apresentadas no Capítulo 2, foi construída uma formulação de PI baseada no problema de CFLP [14], problema este que objetiva apontar a localização ideal para a abertura de novas fábricas de maneira a minimizar o custo de atendimento a consumidores. O CFLP guarda certa similaridade com o problema abordado nesta dissertação e, por isso, foi utilizado como base para a modelagem feita.

O problema abordado nesta dissertação envolve um conjunto de $N = \{n_1, n_2, \dots, n_m\}$ nós, associados a um dado *cluster* Kubernetes, e à necessidade de que, a cada *slot* de tempo t , o orquestrador aloque cada um dos $P = \{p_1, p_2, \dots, p_n\}$ *pods*, demandados pelas aplicações em curso, a um desses nós. Essa alocação envolve também decidir quais nós serão ou não abertos, a cada *slot* t , para atender tal demanda. Mais especificamente, deseja-se obter uma alocação que minimize o custo a ela associado, onde o custo de uma alocação é definido pelos seguintes critérios: (i) custo de abertura dos nós utilizados; (ii) custo de alocação de cada *pod* aos nós; e (iii) custo relacionado à quantidade de erros

registrados num dado nó em decorrência dos *pods* que rodaram ou ainda rodam neste nó.

Associado a cada nó existe uma capacidade de recursos (U_i) e associado a cada *pod* também existe uma quantidade de recursos utilizados (u_j). Em nosso problema, as entradas são de números de *pods*, números de nós, as capacidades e uso de recurso de *pods* e nós, o custo de alocação de um novo nó (α), o custo de alocação de *pod* por nó (β), a quantidade de erros em *pods* ($e_j y_{i,j}$), e a punição por erros dos *pod* (γ); e a saída seria então o custo final da alocação, em que se busca o valor ótimo com o resolvidor MIP e os *pods* alocados ao devidos nós, onde todos são utilizados no primeiro e no segundo conjunto de estudos experimentais. Vale ressaltar que no terceiro conjunto de estudos experimentais também há como saída a quantidade de nós realmente utilizados para a alocação dos *pods* demandados. Diferentemente do CFLP padrão, que as entradas são os conjuntos de clientes, instalações, capacidades e custo e a saída são as instalações abertas e o número de clientes alocados.

A descrição das variáveis adotadas na modelagem depende da definição de alguns coeficientes, como conjunto de *pods*, conjunto de nós, capacidade dos recursos e associação entre nó e *pod*, dentre outros que estão apresentados na Tabela 4.1. É importante citar que, nesta modelagem, olha-se somente para um determinado *slot* de tempo t e, por isso, t não aparece como mais um índice das variáveis e coeficientes. Essa dinâmica de acoplagem de *pods* a nós é demonstrada na Figura 4.1, na qual são mostrados diferentes instantes de tempo t e onde cada execução da formulação proposta condiz com a situação encontrada num instante de tempo t por vez. Esses parâmetros foram criados para sintetizar o conceito de orquestração e facilitar suas menções ao longo do trabalho. No primeiro instante de tempo t , cada nó apresenta cinco *pods* agendados. No segundo instante de tempo $t + 1$, o terceiro nó perde dois *pods*. No terceiro instante de tempo $t + 2$, diminui-se novamente a quantidade de *pods* do terceiro nó, reduzindo-os para apenas um. No quarto instante de tempo $t + 3$, pelo fato de não haver mais nenhum *pod* a ser agendado no terceiro nó, não há também mais a necessidade de se manter o terceiro nó dentro do *cluster*, removendo-o da infraestrutura e, consecutivamente, reduzindo o custo operacional relativo a α . Sempre que um *pod* é removido do agendamento de um nó, também se reduz o custo referente a β e qualquer possibilidade de punição em γ , já que o valor depende de haver ou não um *pod* para haver ou não erros (e_j), conforme mostra a Tabela 4.1.

O fundamento dos instantes de tempo apresentados na Figura 4.1 serão esclarecidos no terceiro conjunto de estudos experimentais computacionais apresentados na Seção 5.5, onde dentro do modelo é adotado o conceito de instantes de tempo para os algoritmos a serem comparados.

n	Número total de nós pertencentes a um <i>cluster</i> C
m	Número total de <i>Pods</i> associados a quaisquer nós do <i>cluster</i> C
x_i	Variável de decisão que designa se um nó i está aberto ou não
$y_{i,j}$	Variável de decisão que designa se um <i>Pod</i> j é atendido por um nó i
u_j	Quantidade de recursos utilizados por um <i>Pod</i> j
e_j	Quantidade de erros de um <i>Pod</i> j
U_i	Capacidade de recursos de um nó i
α_i	Custo operacional associado à abertura (uso) do nó i
$\beta_{i,j}$	Custo operacional associado à alocação de um <i>Pod</i> j a um nó i
γ_i	Custo operacional associado ao acúmulo de registros de erro e_j relativos a cada <i>Pod</i> j alocado ao nó i

Tabela 4.1: Descrição das variáveis e coeficientes usados na formulação do problema.

Este problema se assemelha ao *Facility Location Problem*, que é um problema pertencente ao âmbito da Pesquisa Operacional e que se aprofunda no estudo de localização ideal de instalações para a redução de diversos custos, tal como o de transporte, considerando fatores como, por exemplo, evitar a colocação de instalações que lidem com materiais perigosos perto de habitações, ou instalações que sejam próximas àquelas dos concorrentes. A formulação de PI aqui proposta, a ser apresentada detalhadamente na próxima seção, busca redistribuir a infraestrutura do *cluster* para reduzir o consumo de recursos, aumentar sua resiliência e garantir o seu bom aproveitamento.

4.2 Formulação

A modelagem aqui proposta aborda o problema de alocação de *Pods* a nós, no contexto de um *cluster* Kubernetes, que obtém uma composição de valores das variáveis de decisão que leva à minimização dos custos de abertura de nó (α_i), alocação de *Pods* a nós (β_i) e punição por erro cometido por *Pods* alocados a um nó (γ_i). A função objetivo busca alocar *Pods* ao nó i que possuir um menor custo de alocação β_i , assim como evitar a penalização (α_i) pela abertura de novos nós. Além disso, busca-se evitar a alocação de *Pods* a nós que acumulem um maior registro de erros e_j vinculados aos *Pods* j associados a esses nós.

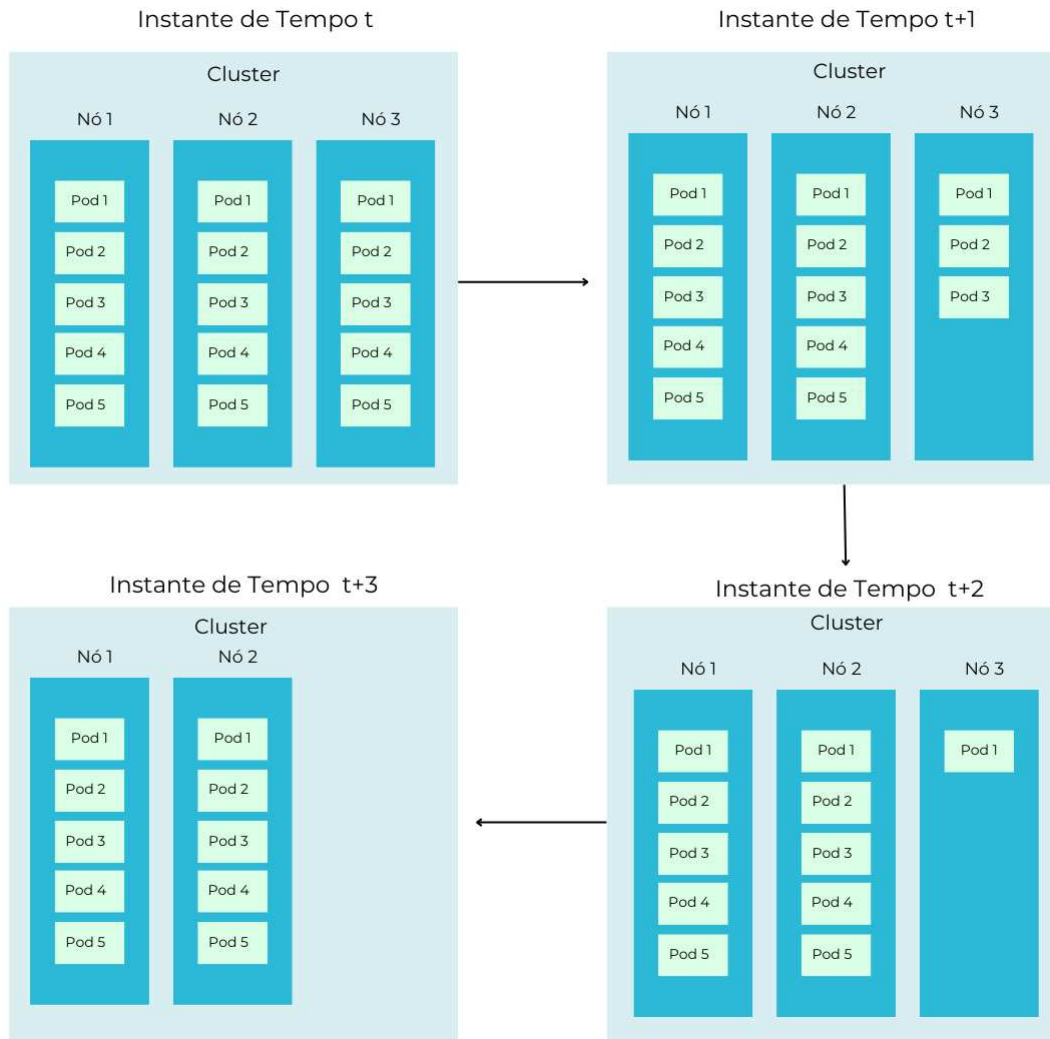


Figura 4.1: Funcionamento de cluster Kubernetes na tentativa de alocar pods a nós em diferentes instantes de tempo

$$\begin{aligned}
 & \min \sum_{i=1}^n \left(\alpha_i x_i + \beta_i \sum_{j=1}^m y_{i,j} + \gamma_i \sum_{j=1}^m e_j y_{i,j} \right) \\
 & \text{sujeito a} \\
 & \sum_{i=1}^n x_i \geq 1 \quad (4.1) \\
 & y_{i,j} \leq x_i \quad i = 1, \dots, n \quad j = 1, \dots, m \quad (4.2) \\
 & \sum_{j=1}^m y_{i,j} \geq 1 \quad i = 1, \dots, n \quad (4.3) \\
 & \sum_{j=1}^m u_j y_{i,j} \leq U_i \quad i = 1, \dots, n \quad (4.4) \\
 & x_i \in \{0, 1\} \quad i = 1, \dots, n \quad (4.5) \\
 & y_{i,j} \in \{0, 1\} \quad i = 1, \dots, n \quad j = 1, \dots, m \quad (4.6)
 \end{aligned}$$

A formulação de minimização de custo obtida é sujeita a um total de seis conjuntos de restrições. A restrição (4.1) define que o número de nós sendo utilizados deve ser de pelo menos um. Por sua vez, a restrição (4.2) afirma que um *pod* j só pode estar alocado a um nó i que esteja aberto. Na restrição (4.3), é garantido que cada *pod* j é alocado a um nó i . Na restrição (4.4), a quantidade de recursos utilizados por todos os *pods* alocados a um nó i não pode ser superior à capacidade de recursos do nó i . Finalmente, as restrições (4.5) e (4.6) informam que x_i e $y_{i,j}$ são variáveis binárias.

Com base na formulação de Programação Inteira proposta neste capítulo, uma avaliação de performance precisa ser realizada, através de experimentos computacionais, para avaliar a sua efetividade. No próximo capítulo, esses experimentos serão apresentados em detalhes.

5. Experimentos Computacionais

Neste capítulo, apresentam-se as avaliações quantitativas de desempenho da solução proposta, por meio de experimentos computacionais. Para tal, foram adotados cenários de uso de nós representando um *cluster* Kubernetes, a fim de se comparar a formulação de Programação Inteira (PI) proposta com uma implementação do algoritmo padrão do Kube-scheduler traduzida para a linguagem Java, de uso habitual do Kubernetes. Assim, nas próximas seções, o ambiente computacional usado, as instâncias utilizadas, os cenários de experimentação, assim como as implementações de ambas as abordagens e as etapas dos experimentos, serão descritos e explicados.

5.1 Ambiente computacional

O ambiente computacional adotado nos experimentos consiste num *notebook* Apple com a seguinte configuração: oito núcleos de CPU Apple M1, 16 GB de memória RAM e sistema operacional MacOS Monterey 12.3.1. As implementações de ambas as abordagens – a solução de PI proposta e o algoritmo padrão do Kube-scheduler, originalmente feito em linguagem de programação Golang¹ – foram feitas em linguagem de programação Java, na sua versão 17.0.4.1 [17]. Para implementar e executar a formulação de PI proposta, descrita no Capítulo 4, foi utilizada a versão 10.0.0 do resolvidor Gurobi a partir de sua API para Java [18].

Os códigos desenvolvidos nesta pesquisa, assim como o resultado dos experimentos, estão disponíveis publicamente em um repositório criado no Github².

¹<https://go.dev>

²<https://github.com/camilla-m/facility-problem>

5.2 Instâncias utilizadas

Com o propósito de validar e avaliar a eficácia da solução proposta, foi gerado aleatoriamente um conjunto de instâncias artificiais considerando os seguintes pares de números de nós e de *pods*: $N = \{10, 20, 50, 100, 200\}$ e $P = \{50, 100, 200, 500, 1000, 5000, 10000\}$, respectivamente. Tratam-se de valores arbitrariamente escolhidos e que são representativos de um cenário de computação em nuvem usando Kubernetes. Esses pares foram usados como entrada para dois algoritmos de escalonamento implementados em Java: um deles simulava o Kube-scheduler padrão, onde os *pods* eram instanciados nos nós de forma aleatória, e o outro implementa a abordagem de escalonamento customizada para o Kube-scheduler, por meio do resolvidor Gurobi³, atendendo à modelagem descrita no Capítulo 4.

Para gerar artificialmente as instâncias de entrada para ambos escalonadores, foram sorteados valores arbitrariamente definidos para os parâmetros U_i , u_j , α_i e β_i , para cada combinação de número de *pods* e número de nós.

A faixa de valores usada para o sorteio do parâmetro U_i , que representa a capacidade de um nó i , teve como limite inferior o número de *pods* m sobre o número de nós n mais um, ou seja, $m/(n+1)$. Já o limite superior foi igual ao número de *pods* multiplicado por dois, ou seja, $2n$. Com relação aos valores sorteados para o parâmetro u_j , que se refere ao uso de recursos de um *pod* j , adotou-se o intervalo entre 1 e 10. Esses valores foram escolhidos arbitrariamente, buscando garantir que cada instância possua ao menos algumas soluções viáveis.

Quanto ao parâmetro α_i , que representa o custo de abertura de um nó i , foi definido como um valor a ser sorteado entre o limite mínimo de 1 e o limite máximo de $4n$, que é o número de nós multiplicado por quatro. O mesmo intervalo se aplica para o sorteio do valor β_i , relacionado ao custo de alocação de um *pod* qualquer a um nó i . Neste caso, os limites superiores foram arbitrados visando um cálculo de custo que seja consistente para a solução do problema de otimização.

Além disso, é importante citar que, tanto para a implementação do Kube-scheduler padrão quanto para a formulação de PI proposta, o cálculo da taxa de penalidade (γ_i) é adicionado ao valor de uma solução caso os *pods* alocados a um determinado nó i apresentem erros, pois tais erros podem causar problemas e impactar o *cluster* como um todo. Os valores para γ_i foram sorteados dentro de um intervalo de 1 a 20, arbitrado de maneira a se obter um cálculo de custo consistente. Os valores de $e_j y_{i,j}$ foram calculados

³<https://www.gurobi.com/>

também arbitrariamente em um intervalo de 1 a 20.

Para a geração das instâncias artificiais, a semente do mecanismo gerador de números pseudoaleatórios correspondeu a 100. Mais especificamente, a semente foi definida no início do processo de geração das instâncias artificiais. O método gerador de números pseudoaleatório utilizado foi o padrão da linguagem Java.

5.3 Implementação do Kube-scheduler

O Algoritmo 2 descreve a lógica de alocação de *pods* a nós realizada pelo Kube-scheduler. A entrada do algoritmo corresponde a uma lista de *pods* pendentes, contendo as suas informações de uso de recursos, e uma lista de nós disponíveis, com as suas respectivas informações de capacidade. Assim, a entrada do algoritmo corresponde a uma lista P de *pods* a serem alocados e uma lista N de nós disponíveis. Inicialmente, na linha 3, o algoritmo inicializa um mapeamento *alocacao* de *pods* para nós, inicialmente vazio, que armazenará a alocação a ser feita. Na linha 4, é criada uma lista *StoredNodes* contendo uma ordenação decrescente dos nós em N, de acordo com algum critério (por exemplo, afinidade do nó ou custo de abertura). Depois disso, para cada *pod* p se busca um nó n que tenha capacidade disponível e suficiente para acomodar o *pod* p. Ou seja, se, mesmo considerando todos os *pods* alocados a n neste momento, esse nó ainda possui capacidade disponível que atenda ao requisito de uso de recursos do *pod* p (linhas 5 - 7). Se essa condição for satisfeita, o *pod* p será alocado ao nó n e essa alocação é refletida no mapeamento *alocacao* (linhas 7 - 10). O algoritmo executa esses passos até que todas os *pods* em P tenham sido considerados e, ao final, na linha 13, retorna a alocação obtida na variável *alocacao*.

Para o algoritmo ser mais fiel à realidade e ao uso do Kube-scheduler com parâmetros de filtragem, foi implementado o uso do conceito de **NodeAffinity**, que é um dos principais filtros do Kube-scheduler e pode simular manualmente o algoritmo customizado, no sentido de permitir que um nó seja rotulado como o mais barato e que haja preferência de alocação de *pods* de uma determinada implantação a esse nó. Foi então provida esta possibilidade de se rotular um nó, realizando-se assim, no algoritmo, uma ordenação dos nós pelo critério prioritário de rotulação, tipicamente refletindo o menor custo de abertura dos nós, tal como expresso pela linha 4. Desta forma, os *pods* são preferencialmente alocados a esses nós de menor custo de abertura, minimizando o custo de soluções obtidas pelo Kube-scheduler padrão e, consequentemente, tornando a comparação mais justa.

Algoritmo 2: Algoritmo do Kube-scheduler

```
1: Entrada: Lista de Pods pendentes P, Lista de nós disponíveis N
2: Saída: Mapeamento de Pods P para nós N
3: Criar um mapeamento vazio de Pods pra nós:  $alocacao \leftarrow \{\}$ 
4: Ordenar nós N na ordem decrescente de acordo com alguns critérios:
    $sortedNodes \leftarrow sort(N)$ 
5: for cada pod  $p \in P$  do
6:   for cada node  $n \in sortedNodes$  do
7:     if n tem capacidade suficiente de servir p then
8:        $alocacao[p] \leftarrow n$ 
9:       Atualizar a capacidade do nó n baseado no uso de recursos pelo pod p
10:    end if
11:  end for
12: end for
13: return  $alocacao$ 
```

5.4 Implementação da Formulação

A implementação da formulação de PI proposta correspondeu à definição da função objetivo e das suas respectivas restrições, apresentadas no Capítulo 4, na linguagem Java, por meio da API do resolvidor Gurobi. Esse resolvidor foi utilizado para encontrar uma solução ótima, ou uma solução viável que esteja dentro do *gap* definido, para cada instância artificial gerada.

No que se refere ao critério de parada para a implementação da formulação proposta usando o resolvidor Gurobi, foram usados múltiplos valores para o parâmetro *MIPGap*, que define o *gap*, isto é, a distância entre os limites inferiores e superiores, calculados na árvore de *Branch-and-Bound* feita pelo resolvidor. Os valores desse parâmetro foram variados, a depender do conjunto de estudos experimentais, para encurtar a quantidade de tempo decorrida até que o resolvidor atinja o critério de parada e termine a sua execução, o que é especialmente crítico para instâncias com maior de quantidade de *Pods*.

5.5 Etapas dos Experimentos

Nesta dissertação, houve três conjuntos de experimentos computacionais, realizados de modo que a metodologia e os resultados fossem aprimorados evolutivamente. No primeiro conjunto, objetivando a elaboração de um artigo publicado no LV Simpósio Brasileiro de Pesquisa Operacional (SBPO 2023) [58], para as instâncias artificiais geradas, não foi considerada a taxa de penalidade por erros (γ_i), que teve seu valor definido como 0. Considerou-se, assim, que não havia erros ocasionados por *Pods* ou nós. Adicional-

mente, também não se fez uso de um mecanismo de filtragem, tal como a rotulação por *NodeAffinity*, na implementação realizada para o Kube-scheduler padrão. Neste primeiro conjunto, como não havia taxa de penalidade, o que diminuía a dificuldade do problema, foi adotado um valor de 0,01 para o parâmetro $MIPGap$, que define o *gap* mínimo a ser atingido para fins de parada.

Por sua vez, no segundo conjunto de estudos experimentais computacionais, realizado para o artigo publicado na *15th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT 2023)* [57], foram considerados ambos os pontos anteriormente suprimidos, a saber: (i) a taxa de penalidade (γ_i), com valores discriminativos, a fim de simular o fato de que os *pods* podem ocasionar erros; e (ii) o uso de um mecanismo de filtragem por meio da rotulação de nós por *NodeAffinity*. Ademais, pelo fato de o problema, ao contemplar a taxa de penalidade, se tornar mais difícil, foi adotado o valor de 0,03 para o parâmetro $MIPGap$.

Para os dois primeiros conjuntos de experimentos, para fins de obtenção da estimativa de tempo de execução dos algoritmos, foram executadas 10 rodadas de cada instância de entrada com a mesma semente de números pseudoaleatórios, sem modificações entre elas, e computado o tempo médio de execução dessas 10 rodadas.

Para o terceiro conjunto de estudos experimentais computacionais, foi adotado o conceito de horizonte de tempo. Inicia-se com uma determinada instância artificial em um instante de tempo t_0 e, em cada novo instante t_{i+1} , essa instância é modificada em relação à instância do instante t_i . Mais especificamente, a modificação feita no instante t_{i+1} corresponde a escolher, ao acaso, adicionar ou remover uma determinada quantidade de *pods*, que varia aleatoriamente no intervalo de 5% a 30%, à quantidade de *pods* contida em t_i . A cada instante de tempo t_i , as abordagens do Kube-scheduler padrão e da formulação de PI são executadas para a instância corrente, e procede-se assim até que o fim do horizonte de tempo T seja atingido.

Para cada execução das abordagens do Kube-scheduler padrão e da formulação de PI em um instante de tempo t_i , são obtidas as seguintes informações: as características da instância corrente em t_i , a solução obtida, o valor dessa solução e o tempo de execução tomado. Dessa maneira, a ideia desse conjunto de estudos experimentais correspondeu a avaliar e comparar o desempenho de ambas as abordagens em um cenário mais realista, à medida que novos *pods* chegam para ser escalonados e outros terminam as suas execuções, ao longo do tempo.

Destaca-se, assim, que nesse conjunto de estudos experimentais houve uma atenção

especial para a quantidade de nós empregados na solução obtida pelas duas abordagens, a fim de se entender quantos nós foram efetivamente abertos para acomodar a quantidade de *pods* que deviam ser escalonados. Assim sendo, nessa perspectiva de análise, torna-se possível comparar as soluções obtidas para além dos seus custos, realçando mais fidedignamente questões relevantes para uma análise focada em um ambiente Kubernetes.

É importante citar ainda que, para o terceiro conjunto de estudos experimentais, os dois algoritmos só são executados uma vez para cada instância corrente em um instante t_i . Ademais, foi adotado o valor de 0,03 para o parâmetro $MIPGap$.

Por fim, cabe citar que foi feita uma tentativa de execução de [16] em 0,00 para o segundo e o terceiro conjunto de estudos experimentais, porém não foi obtido sucesso devido às limitações de *hardware* do ambiente computacional disponível para as execuções dos algoritmos.

6. Resultados

Neste capítulo, apresentam-se os resultados obtidos nos três conjuntos de experimentos computacionais explicados no capítulo anterior, conduzidos com o objetivo de validar a formulação apresentada nesta dissertação e avaliar o seu desempenho em relação à solução padrão provida pelo orquestrador do Kubernetes. O propósito destes experimentos e da avaliação de seus resultados, de maneira sintética, corresponde a responder à seguinte pergunta de pesquisa: qual estratégia de alocação permite ao *cluster* Kubernetes atender à demanda existente de *pods* utilizando o menor número de nós e incorrendo em uma solução de baixo custo.

6.1 Resultados do primeiro conjunto de estudos experimentais computacionais

Conforme descrito na Seção 5.5, o primeiro conjunto de estudos experimentais computacionais corresponde àquele que não utilizou o mecanismo de filtragem na implementação produzida para o Kube-scheduler padrão, assim como não fez uso da taxa de penalidade em casos de erros de *pods* ou nós. A Tabela 6.1 apresenta, para cada combinação de número de nós e de número de *pods* testada, os valores médios obtidos, em 10 rodadas de execução, para o custo da solução e seu tempo de execução em milissegundos, tanto para a implementação do Kube-scheduler padrão (KS) quanto para a formulação de Programação Inteira (PI) proposta. Adicionalmente, é exibida na última coluna a razão entre o custo médio da solução obtida pela implementação do *Kube-scheduler* padrão e o custo médio da solução via formulação de PI.

A fim de se obter um comparativo visual entre os valores de custo das soluções obtidas por cada abordagem, foram gerados os gráficos contidos nas Figuras 6.1, 6.2, 6.3, 6.4 e 6.5, que mostram o custo da solução obtida em função do número de *pods*, para cada abordagem. Cada figura é referente a uma das seis diferentes quantidades de nós de cada instância, podendo ser entendido como a quantidade de nós de um *cluster* do Kubernetes.

É importante notar que, no eixo x desses gráficos, a evolução do número de *Pods* de cada instância de teste segue uma escala aproximadamente exponencial: 10, 50, 100, 200, 500, 1000, 5000 e 10000 *Pods*.

Pods	Nós	Custo (KS)	Tempo (ms) (KS)	Custo (PI)	Tempo (ms) (PI)	Razão de custo
10	10	399	< 1	250	< 1	1.60
10	20	446	< 1	158	< 1	2.82
10	50	1244	< 1	356	10	3.49
10	100	3854	< 1	586	5	6.57
10	200	7667	< 1	678	9	11.30
50	10	1495	< 1	573	12	2.61
50	20	2016	< 1	1195	7	1.69
50	50	4228	< 1	873	8	4.84
50	100	13578	< 1	1640	45	8.28
50	200	27938	< 1	1964	31	14.22
100	10	2407	< 1	1155	1	2.08
100	20	3954	< 1	1270	5	3.11
100	50	15050	< 1	1530	18	9.83
100	100	19329	< 1	1718	52	11.25
100	200	56409	< 1	1539	94	36.65
200	10	5342	< 1	2653	3	2.01
200	20	7230	< 1	2348	5	3.07
200	50	8709	< 1	1039	43	8.38
200	100	40981	< 1	3641	123	11.25
200	200	88353	< 1	1912	234	46.20
500	10	9776	< 1	3383	8	2.88
500	20	20560	< 1	5428	15	3.78
500	50	58032	< 1	6361	112	9.12
500	100	96226	< 1	11634	54	8.27
500	200	121867	< 1	5568	815	21.88
1000	10	28037	< 1	4354	66	6.43
1000	20	48455	< 1	3319	38	14.59
1000	50	125517	< 1	17289	56	7.26
1000	100	196462	< 1	21298	1160	9.22
1000	200	498364	< 1	20331	3220	24.51
5000	10	131090	14	60671	102	2.16
5000	20	188764	11	60130	843	3.14
5000	50	511526	11	53105	310	9.63
5000	100	731992	11	28751	609	25.45
5000	200	2113569	12	70832	1367	29.83
10000	10	175960	48	94909	166	1.85
10000	20	480883	49	137247	395	3.50
10000	50	895451	45	90058	8413	9.94
10000	100	1728710	52	77527	11418	22.29
10000	200	2356808	50	63702	3038	36.99

Tabela 6.1: Resultados obtidos pela implementação do Kube-scheduler padrão e pela formulação de PI para as instâncias artificiais geradas.

Verifica-se, para este primeiro conjunto de estudos experimentais computacionais, que a formulação de PI proposta leva a soluções de melhor qualidade, ou seja, de muito menor

custo, do que aquelas providas pelo Kube-scheduler padrão. Claramente, a medida que o número de nós do *cluster* aumenta, maior é a diferença entre o custo médio das soluções, chegando a duas ordens de grandeza de diferença em favor da solução proposta para o caso da instância de teste com 200 nós e 1000 *Pods*, conforme denotado pela última linha da Tabela 6.1.

Através da última coluna da Tabela 6.1, pode-se verificar também que a razão de custo médio entre as soluções tende a aumentar à medida que a quantidade de nós das instâncias aumenta, considerando-se uma mesma quantidade de *Pods* para serem alocados. Essa razão de custo médio pode ser compreendida como o ganho obtido pela solução baseada em formulação de PI em relação à solução padrão usando Kube-scheduler. De modo geral, pode-se então afirmar que, quanto maior é o número de combinações para alocações de *Pods* a nós, menos otimizada é a solução escolhida pelo Kube-scheduler em relação à solução escolhida pela formulação de PI, o que se reflete na tendência crescente de diferença entre seus custos.

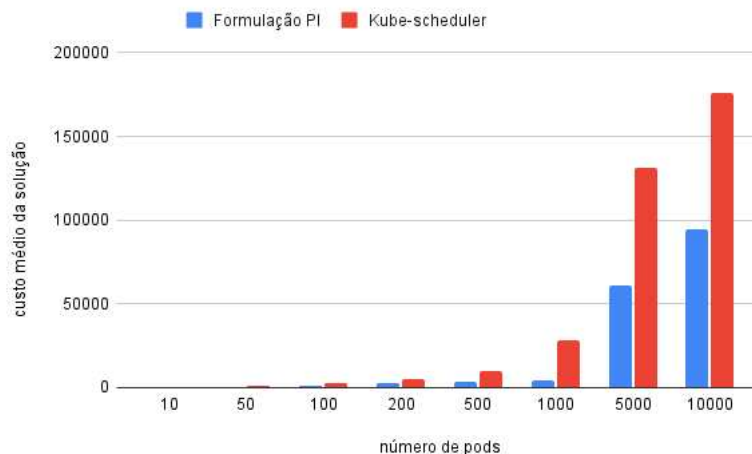


Figura 6.1: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 10 nós no primeiro conjunto de estudos experimentais computacionais.

As Figuras 6.6, 6.7, 6.8, 6.9, 6.10, 6.11, 6.12 e 6.13 exibem, em escala logarítmica, a média do tempo de execução, para 10 rodadas, de ambas as abordagens para as instâncias artificiais geradas. Neste caso, optou-se agrupar as medidas em função do número de *Pods* das instâncias de teste. Assim sendo, cada figura mostra a variação do tempo médio de execução em função da quantidade de nós, ou tamanho do *cluster*, de cada instância. A evolução da quantidade de nós se deu também de forma aproximadamente exponencial: 10, 20, 50, 100 e 200 nós. Vale notar também que, para os casos onde o tempo médio de execução foi inferior a 1 ms, nenhum valor foi plotado nesses gráficos.

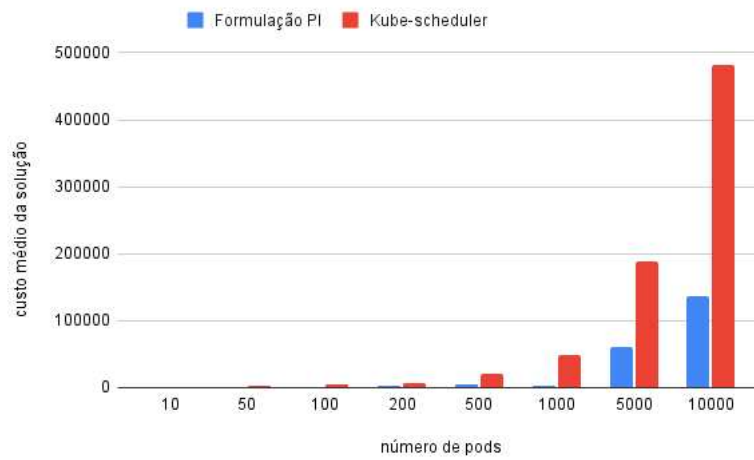


Figura 6.2: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 20 nós no primeiro conjunto de estudos experimentais computacionais.

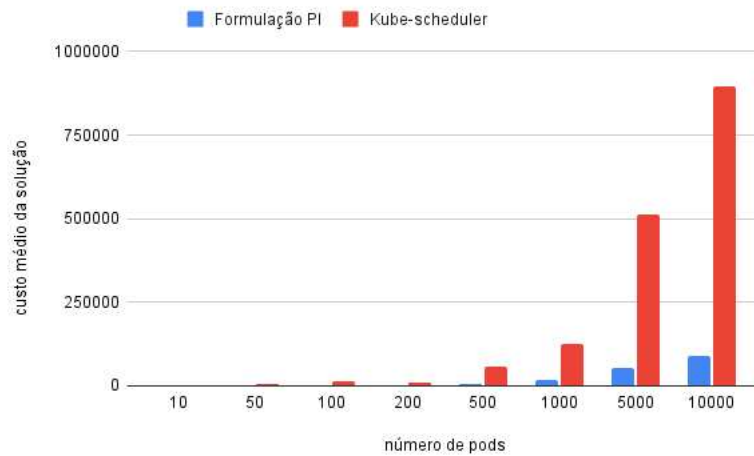


Figura 6.3: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 50 nós no primeiro conjunto de estudos experimentais computacionais.

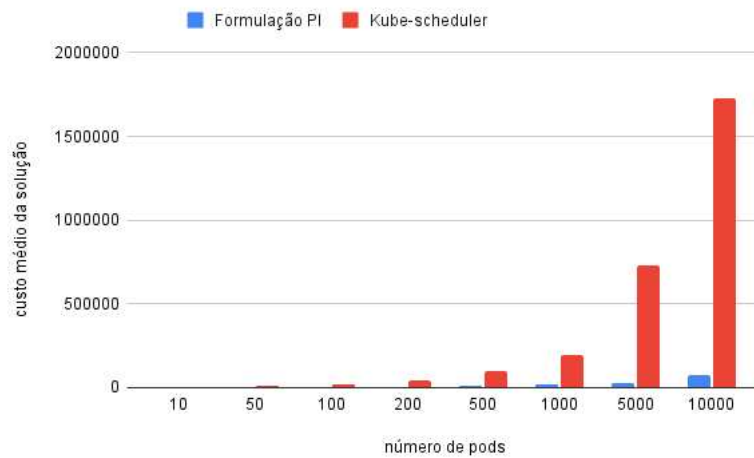


Figura 6.4: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 100 nós no primeiro conjunto de estudos experimentais computacionais.

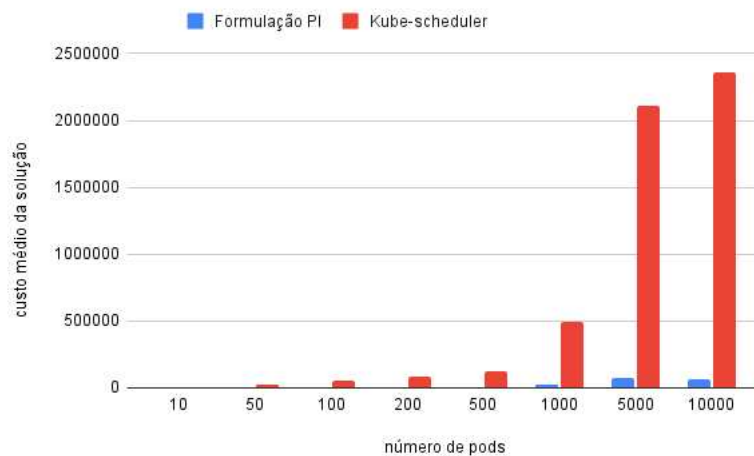


Figura 6.5: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 200 nós no primeiro conjunto de estudos experimentais computacionais.

É possível perceber que a formulação proposta toma uma quantidade de tempo consideravelmente maior do que o Kube-scheduler padrão para realizar o escalonamento de *pods*, sobretudo para instâncias contendo números maiores de nós e de *pods*. Isso se explica pelo fato de o resolvidor MIP, ao executar a formulação de PI, levar um determinado tempo até atingir o critério de parada definido, a depender do tamanho e da dificuldade da instância. Já o algoritmo do Kube-scheduler aloca aleatoriamente um *pod* para um nó disponível sem levar em consideração questões de custo e ainda o *NodeAffinity*, conforme explicado na Seção 5.3, o que certamente exige menos recursos computacionais.

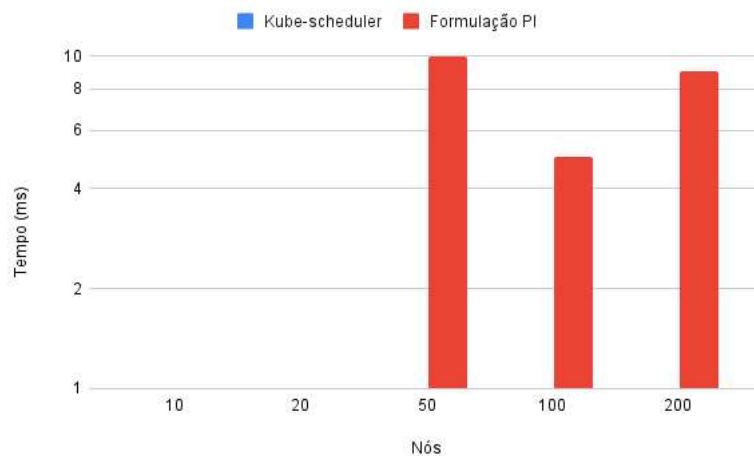


Figura 6.6: Tempo de execução da formulação de PI e do Kube-scheduler para 10 *pods* alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.

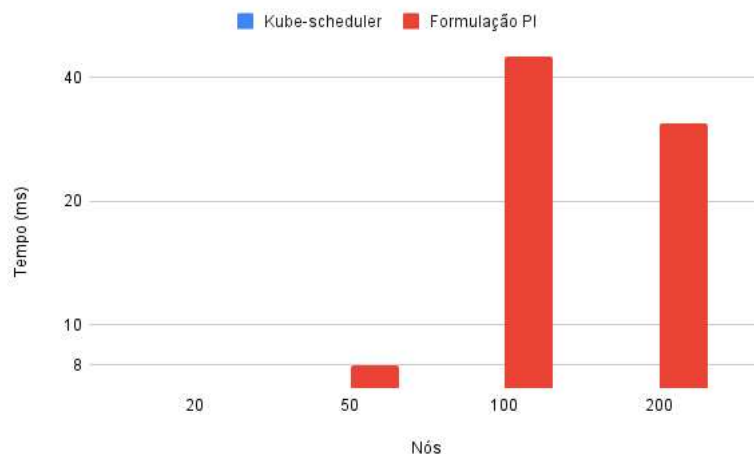


Figura 6.7: Tempo de execução da formulação de PI e do Kube-scheduler para 50 *pods* alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.

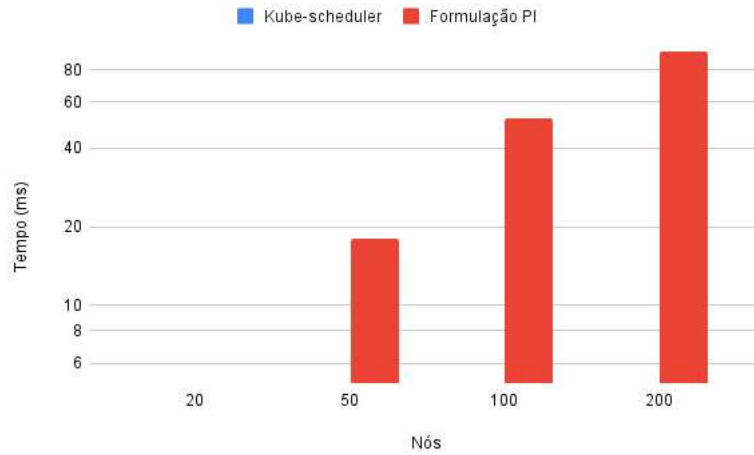


Figura 6.8: Tempo de execução da formulação de PI e do Kube-scheduler para 100 *pods* alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.

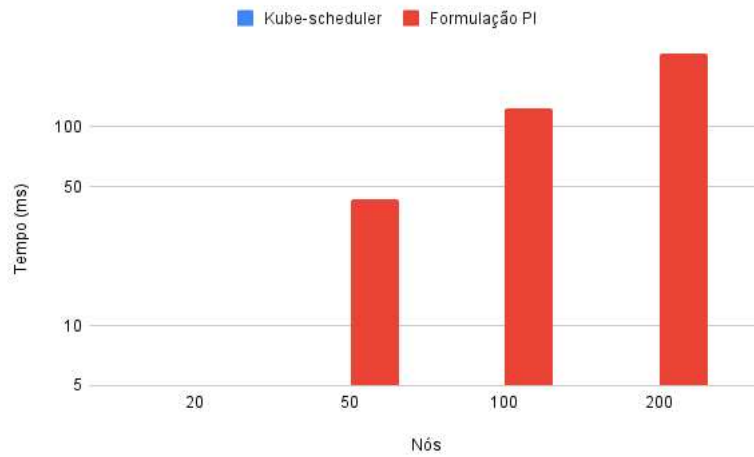


Figura 6.9: Tempo de execução da formulação de PI e do Kube-scheduler para 200 *pods* alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.

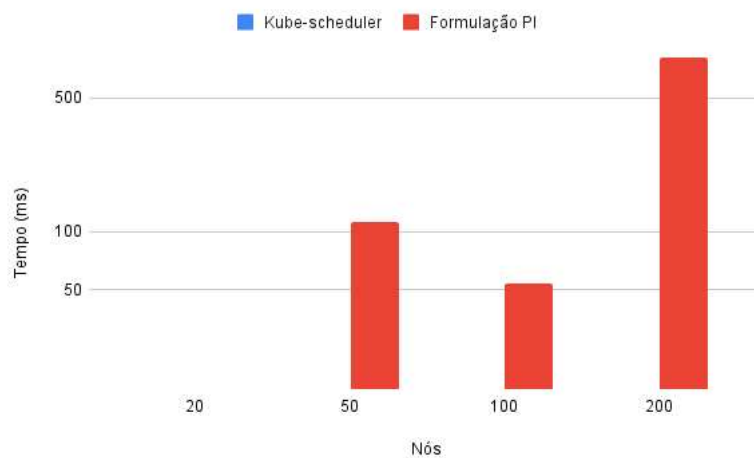


Figura 6.10: Tempo de execução da formulação de PI e do Kube-scheduler para 500 *pods* alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.

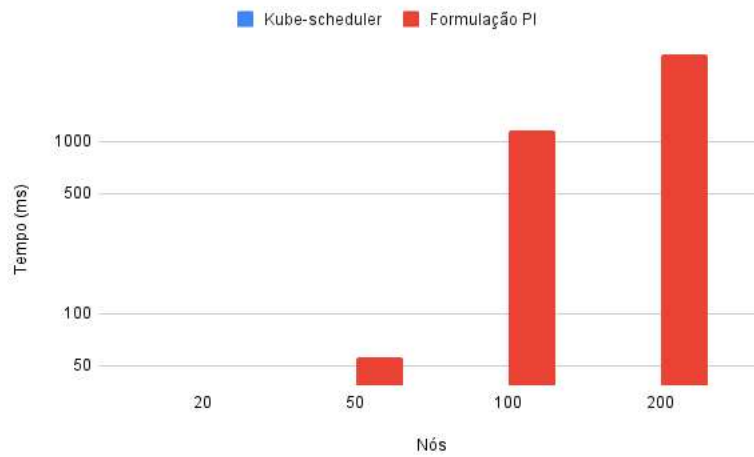


Figura 6.11: Tempo de execução da formulação de PI e do Kube-scheduler para 1000 *pods* alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.

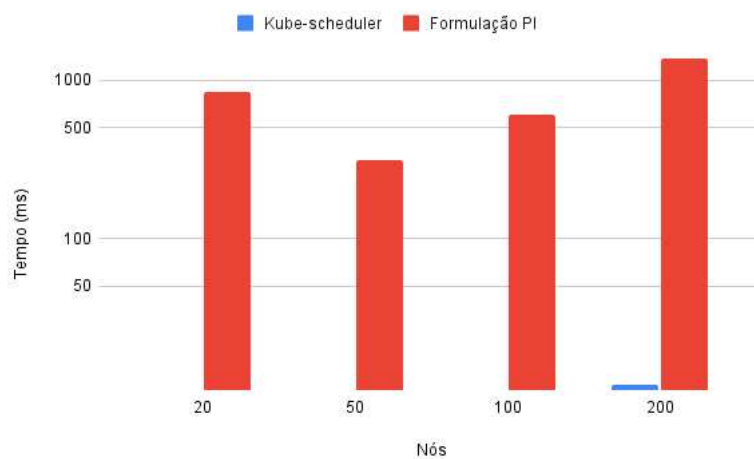


Figura 6.12: Tempo de execução da formulação de PI e do Kube-scheduler para 5000 *pods* alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.

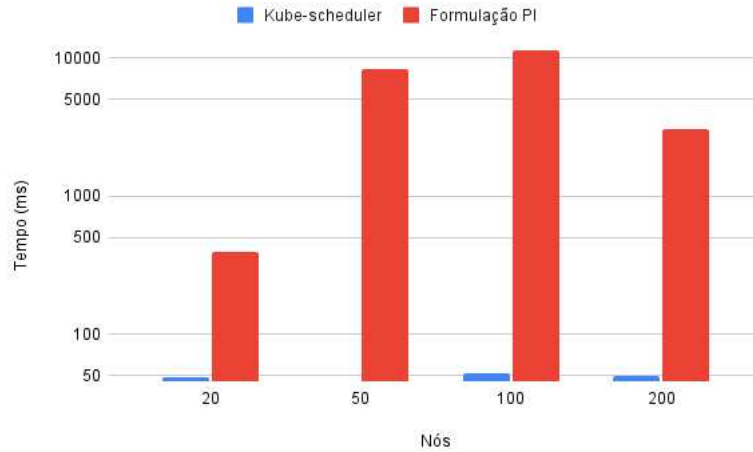


Figura 6.13: Tempo de execução da formulação de PI e do Kube-scheduler para 10000 *pods* alocados a diferentes números de nós, no primeiro conjunto de estudos experimentais.

6.2 Resultados do segundo conjunto de estudos experimentais computacionais

A Tabela 6.2 apresenta os resultados obtidos no segundo conjunto de estudos experimentais computacionais, agora com os filtros de **NodeAffinity**, implementados na solução padrão do Kube-scheduler, e uma taxa de penalidade γ_i , diferente de zero, incorporada na solução por Programação Inteira. Cada instância também foi executada 10 vezes e os resultados apresentados para custo e tempo de execução de cada abordagem correspondem aos valores médios obtidos para as respectivas métricas. A Tabela 6.2 também apresenta a razão de custo médio entre a solução usando Kube-scheduler (KS) e a que faz uso de Programação Inteira (PI).

As Figuras 6.14, 6.15, 6.16, 6.17 e 6.18 apresentam gráficos de comparação entre os custos médios das soluções obtidas pela implementação do Kube-scheduler e pela formulação de Programação Inteira, seguindo os mesmos critérios usados para a apresentação dessa métrica no conjunto de estudos experimentais anterior: custo médio em função do número de *pods* a serem alocados, escala aproximadamente exponencial para a variação desse número, cada figura abrangendo uma certa quantidade de nós compondo o *cluster*.

Percebe-se o mesmo comportamento em relação ao que foi verificado no primeiro conjunto de estudos experimentais computacionais, demonstrando a diferença discrepante entre o custo médio da solução via PI e o da solução padrão do Kube-scheduler, com razões de custo por exemplo que variam até 46.20%.

Da mesma forma que observado para o primeiro conjunto de estudos experimentais, verifica-se que o custo médio da solução via Kube-scheduler torna-se gradativamente

muito maior que o da solução via PI, à medida que o número de combinações possíveis entre alocações de *Pods* a nós aumenta, chegando a por exemplo até 11418(ms).

Pods	Nós	Custo (KS)	Tempo (ms) (KS)	Custo (PI)	Tempo (ms) (PI)	Razão de custo
10	10	1061	< 1	185	< 1	5.74
10	20	1334	< 1	233	1	5.73
10	50	2060	1	415	3	4.96
10	100	3489	< 1	749	5	4.66
10	200	6478	< 1	768	11	8.43
50	10	3920	< 1	823	5	4.76
50	20	5111	< 1	432	13	11.83
50	50	9109	< 1	1246	14	7.31
50	100	16909	< 1	767	17	22.05
50	200	12254	< 1	1570	48	7.81
100	10	8563	< 1	1141	3	7.50
100	20	10356	< 1	1048	8	9.88
100	50	14444	< 1	355	16	40.69
100	100	27551	< 1	1106	57	24.91
100	200	48494	< 1	2536	62	19.12
200	10	16922	< 1	2370	7	7.14
200	20	18653	< 1	6212	5	3.00
200	50	43370	< 1	2926	66	14.82
200	100	42889	< 1	2818	96	15.22
200	200	65331	< 1	3265	179	20.01
500	10	54774	< 1	3275	28	16.72
500	20	35414	< 1	8852	16	4.00
500	50	113140	< 1	10793	28	10.48
500	100	125193	< 1	4567	236	27.41
500	200	205573	< 1	3551	515	57.89
1000	10	95368	< 1	7960	22	11.98
1000	20	110123	< 1	5563	136	19.80
1000	50	150831	< 1	16087	54	9.38
1000	100	170350	< 1	10233	102	16.65
1000	200	448345	< 1	8656	2389	51.80
5000	10	531743	13	27217	103	19.54
5000	20	499999	12	89309	161	5.60
5000	50	765229	11	130610	272	5.86
5000	100	1109917	18	49063	623	22.62
5000	200	1698940	11	39542	1396	42.97
10000	10	562464	50	92471	661	6.08
10000	20	1101196	46	81446	362	13.52
10000	50	1548091	48	208608	693	7.42
10000	100	3471036	49	108704	1295	31.93
10000	200	4730309	49	142533	2924	33.19

Tabela 6.2: Resultados obtidos pela implementação da formulação de Programação Inteira para as instâncias artificiais geradas no segundo conjunto de estudos experimentais computacionais.

Também com relação ao primeiro conjunto de estudos experimentais, verifica-se o mesmo padrão de comportamento quanto ao tempo médio de execução, em milissegun-

dos, para ambas as soluções, conforme mostram as Figuras 6.19, 6.20, 6.21, 6.22, 6.23, 6.24, 6.25 e 6.26. Ou seja, a solução baseada em formulação de PI possui um tempo de execução muito maior que o da implementação do Kube-scheduler, dada a diferença de complexidade computacional existente para ambas as soluções. Aqui também se optou por exibir esses tempos médios em função do número de nós do *cluster*, com cada figura apresentando essa evolução para a uma mesma quantidade de *pods* a serem alocados.

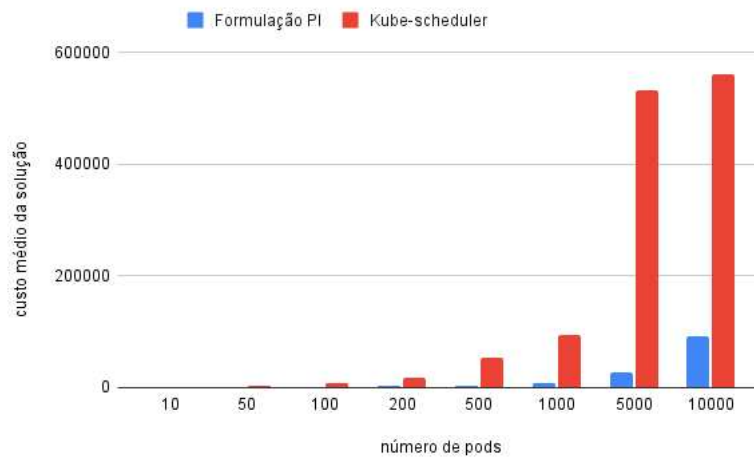


Figura 6.14: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 10 nós no segundo conjunto de estudos experimentais computacionais.

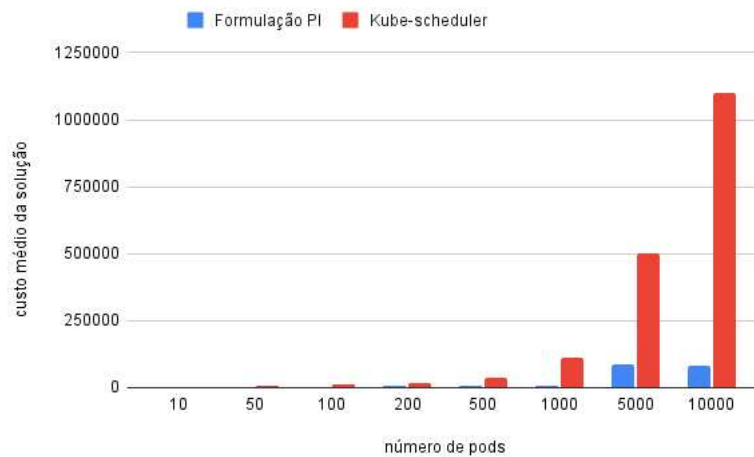


Figura 6.15: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 20 nós no segundo conjunto de estudos experimentais computacionais.

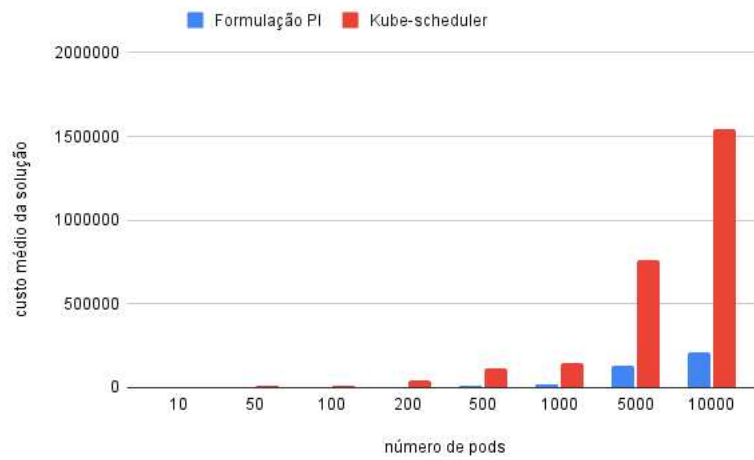


Figura 6.16: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 50 nós no segundo conjunto de estudos experimentais computacionais.

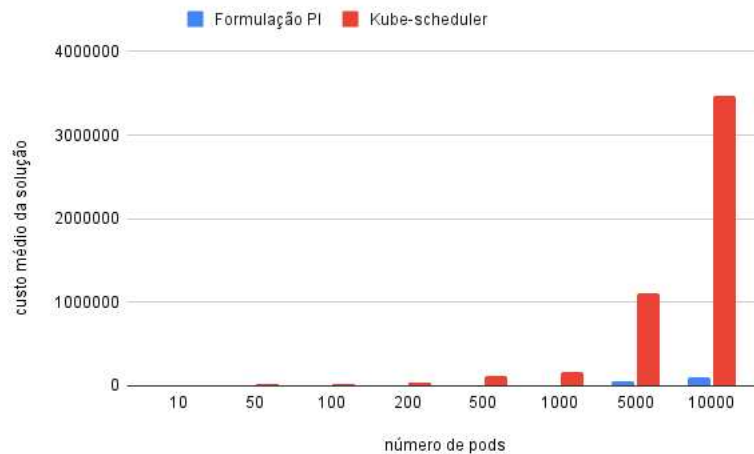


Figura 6.17: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 100 nós no segundo conjunto de estudos experimentais computacionais.

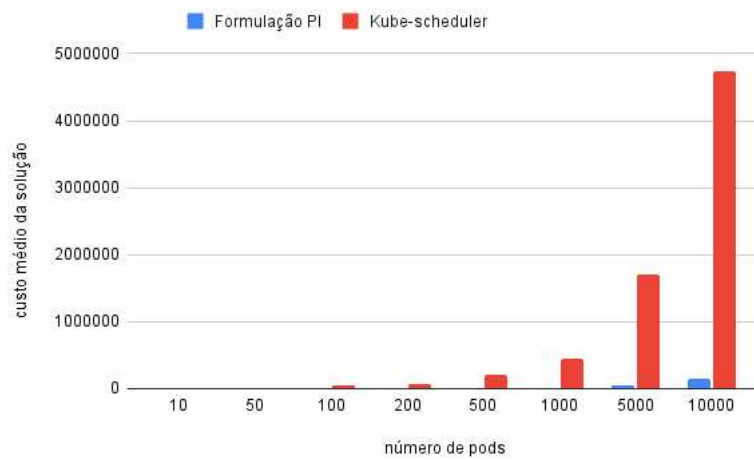


Figura 6.18: Custo médio da solução obtida pela formulação de PI e pelo algoritmo padrão do Kube-scheduler para 200 nós no segundo conjunto de estudos experimentais computacionais.

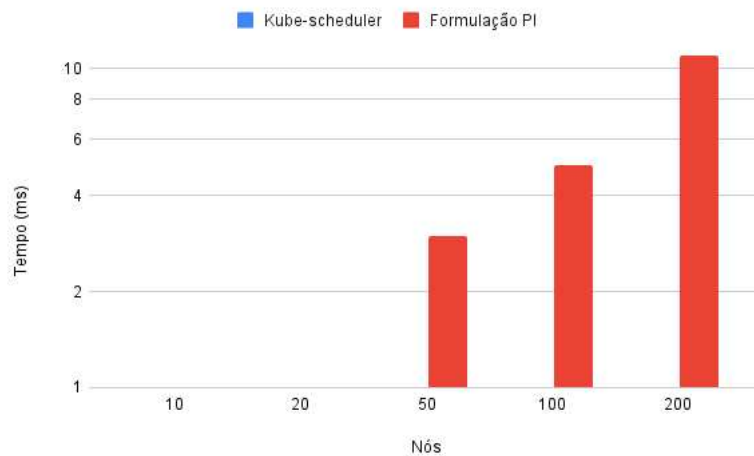


Figura 6.19: Tempo de execução da formulação de PI e do Kube-scheduler para 10 *pods* alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.

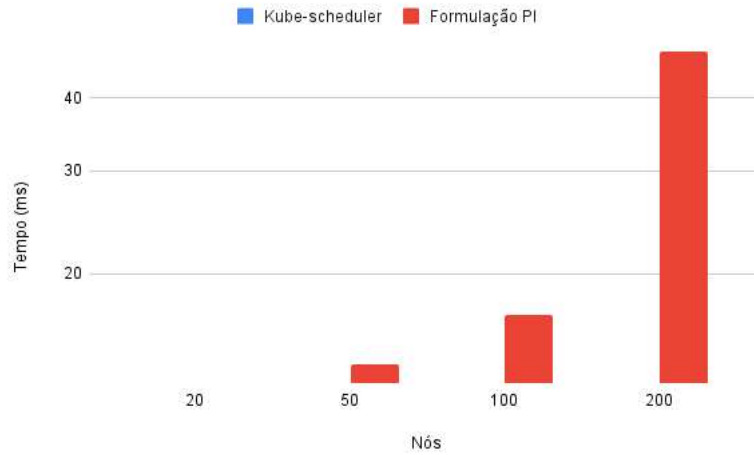


Figura 6.20: Tempo de execução da formulação de PI e do Kube-scheduler para 50 *pods* alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.

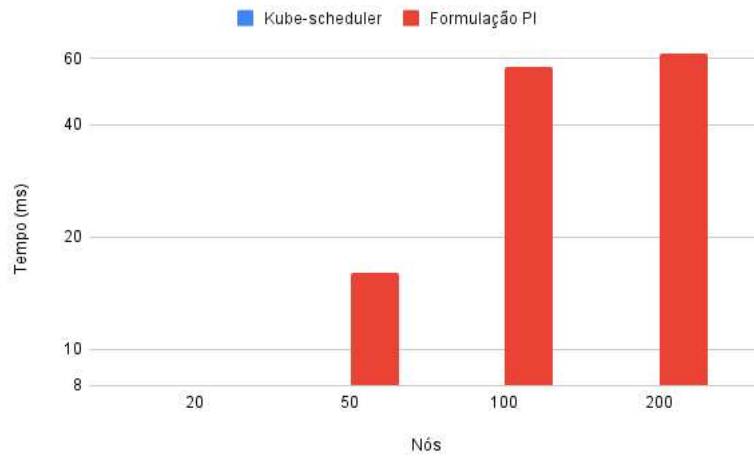


Figura 6.21: Tempo de execução da formulação de PI e do Kube-scheduler para 100 *pods* alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.

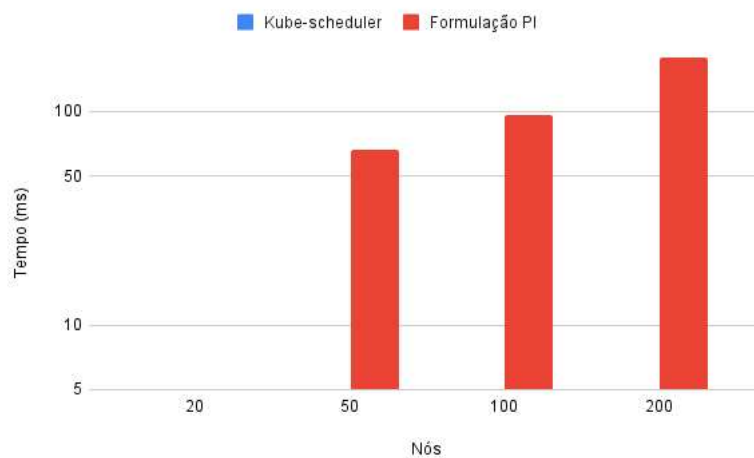


Figura 6.22: Tempo de execução da formulação de PI e do Kube-scheduler para 200 *pods* alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.

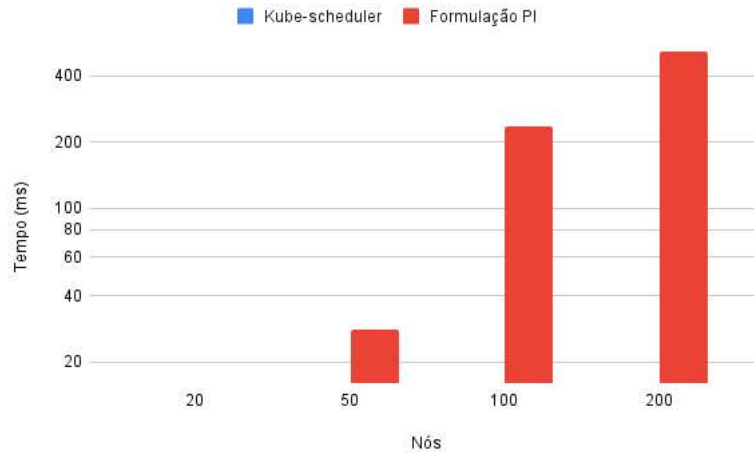


Figura 6.23: Tempo de execução da formulação de PI e do Kube-scheduler para 500 *pods* alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.

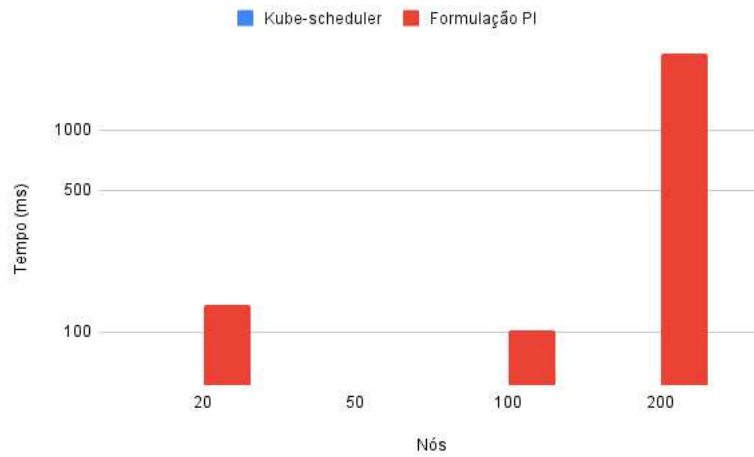


Figura 6.24: Tempo de execução da formulação de PI e do Kube-scheduler para 1000 *pods* alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.

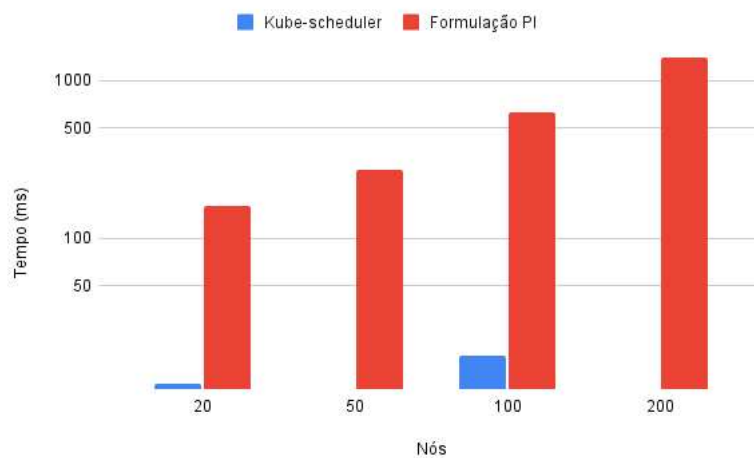


Figura 6.25: Tempo de execução da formulação de PI e do Kube-scheduler para 5000 *pods* alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.

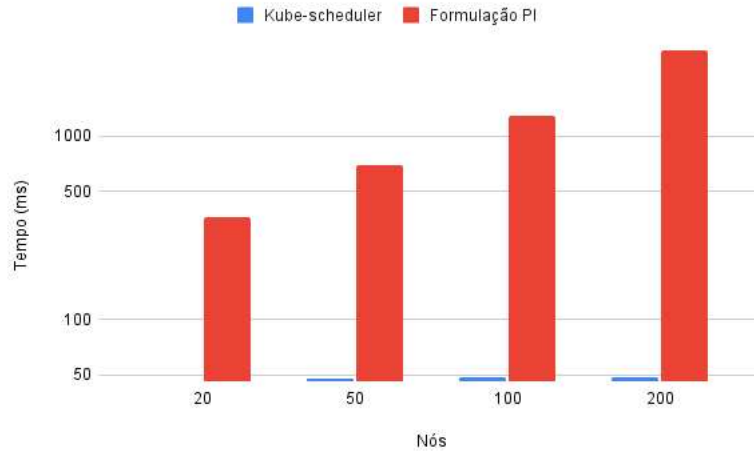


Figura 6.26: Tempo de execução da formulação de PI e do Kube-scheduler para 1000 *pods* alocados a diferentes números de nós, no segundo conjunto de estudos experimentais.

6.3 Resultados do terceiro conjunto de estudos experimentais computacionais

O terceiro conjunto de estudos experimentais faz uso do conceito de horizonte de tempo, conforme apresentado na Seção 5.5. Em cada experimento deste conjunto, foram fixados o número inicial de *pods* a serem alocados e superdimensionou-se a capacidade de nós do *cluster*, de maneira a evitar que um *pod* não possa ser alocado por falta de nós com recursos suficientes disponíveis. O limite T para o horizonte de tempo foi fixado em 50 instantes (*slots*) de tempo.

6.3.1 Resultados para uma alocação inicial de 5000 *pods*

A Tabela 6.3.1 apresenta, para cada instante de tempo, o número de *pods* alocados, o número de nós que foram abertos em cada solução para viabilizar essa alocação de *pods*, o custo de cada solução, o tempo de execução de cada solução e a razão de custo entre as soluções (KS sobre PI). Neste experimento, a quantidade inicial de *pods* foi fixada em 5000 a capacidade do *cluster* foi superdimensionada em 100 nós.

Instantes	Pods	Nós (KS)	Custo (KS)	Tempo (ms) (KS)	Nós (PI)	Custo (PI)	Tempo (ms) (PI)	Razão
0	5000	8	1729642	2	4	60413	338	28.63
1	5817	10	1906366	4	7	76160	459	25.03
2	7342	12	2211833	6	12	125793	1375	17.58
3	5820	10	1913300	8	7	76013	453	25.17
4	5292	8	1796252	9	4	65058	354	27.61
5	6579	10	2064578	10	9	98475	502	20.97
6	6177	10	1982946	12	8	86457	464	22.94
7	4537	6	1610931	13	4	52570	280	30.64
8	4968	8	1727455	14	4	59764	292	28.90

9	4078	6	1444546	14	4	44868	289	32.20
10	4490	6	1589770	15	4	51466	275	30.89
11	3728	6	1319970	16	4	39062	257	33.79
12	2861	5	996177	16	3	26851	191	37.10
13	3264	6	1157759	17	3	32359	212	35.78
14	3010	6	1064622	17	3	28942	207	36.78
15	3749	6	1325656	17	4	39510	260	33.55
16	3948	6	1394970	18	4	42774	285	32.61
17	3416	6	1204626	18	3	34454	211	34.96
18	2595	5	869604	19	2	23729	176	36.65
19	2789	5	963366	19	3	25890	199	37.21
20	2613	5	874000	19	2	23858	167	36.63
21	2085	4	641440	19	2	18089	145	35.46
22	1708	4	534356	19	1	14048	88	38.04
23	1265	4	415689	20	1	10504	69	39.57
24	1350	4	439164	20	1	11184	80	39.27
25	1177	4	390867	20	1	9800	68	39.88
26	949	4	327647	20	1	7976	53	41.08
27	863	4	303086	20	1	7288	46	41.59
28	984	4	336704	20	1	8256	55	40.78
29	813	4	288333	20	1	6888	45	41.86
30	880	4	306645	20	1	7424	50	41.30
31	1096	4	366726	20	1	9152	61	40.07
32	1013	4	343533	20	1	8488	55	40.47
33	784	4	281305	20	1	6656	42	42.26
34	923	4	319681	20	1	7768	49	41.15
35	1178	4	389872	20	1	9808	62	39.75
36	1405	4	452935	20	1	11624	84	38.97
37	1703	4	535783	20	1	14008	88	38.25
38	2045	4	630565	21	2	17754	147	35.52
39	1802	4	560509	21	1	14800	95	37.87
40	2022	4	620953	21	2	17530	147	35.42
41	1655	4	520273	21	1	13624	82	38.19
42	1779	4	554620	21	1	14616	96	37.95
43	1609	4	507102	21	1	13256	82	38.25
44	1893	4	585516	21	2	16198	132	36.15
45	2340	5	746082	22	2	21039	156	35.46
46	2574	5	860754	22	2	23601	170	36.47
47	1814	4	564850	22	2	15306	120	36.90
48	1707	4	534380	22	1	14040	89	38.06
49	1938	4	598685	22	2	16623	134	36.02

Tabela 6.3.1: Resultados obtidos para as instâncias geradas no terceiro conjunto de estudos experimentais computacionais para uma quantidade inicial de 5000 *pods*.

O gráfico da Figura 6.27 mostra a variação do custo das soluções ao longo dos instantes de tempo. No eixo x, optou-se por indicar a demanda de *pods* a serem alocados no respectivo *slot* de tempo. Para a representação dos valores dos custos computacionais, adotou-se uma escala logarítmica. Verifica-se que o custo da solução por formulação de PI apresenta um custo significativamente menor que o da solução usando Kube-scheduler.

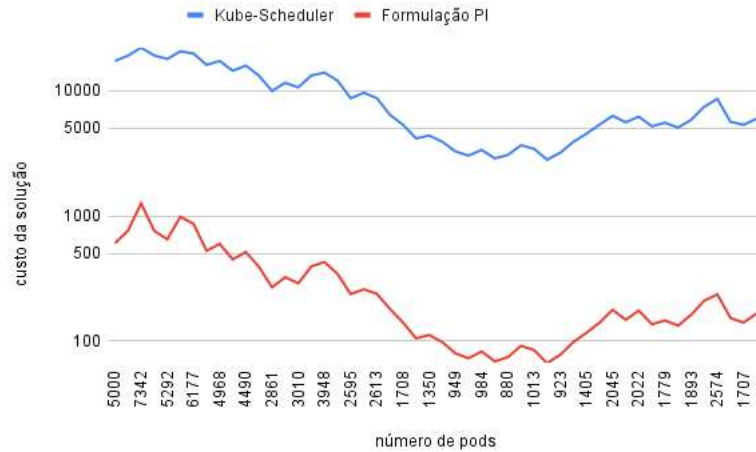


Figura 6.27: Comparativo entre o custo das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI no terceiro conjunto de estudos experimentais computacionais, iniciados com 5000 *pods*.

A Figura 6.28 compara os tempos computacionais de execução para cada implementação ao longo dos instantes de tempo. Como era de se esperar, o tempo de execução para a solução baseada em PI é muito superior que para o Kube-scheduler e com muito mais variabilidade, principalmente na fase inicial. Contudo, à medida que houve uma redução do número de *pods* a serem alocados ao longo dos instantes de tempo, devido à aleatoriedade na geração desta demanda, percebe-se uma queda significativa do tempo computacional para a solução via PI. Tal se deve à complexidade de tempo desta solução em função do número de combinações a serem verificadas.

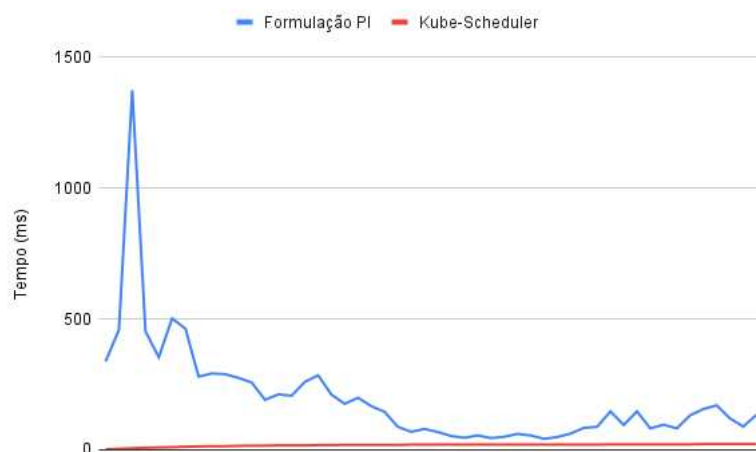


Figura 6.28: Comparativo entre o tempo de execução das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI no terceiro conjunto de estudos experimentais computacionais, iniciados com 5000 *pods*.

Com relação ao número de nós abertos por cada solução, verifica-se pelo gráfico da Figura 6.29 que esses valores são bastante próximos. Entretanto, a solução baseada em

PI sempre utiliza um número de nós igual ou, na maioria das vezes, menor que o número de nós utilizados pelo Kube-scheduler.



Figura 6.29: Comparativo entre a abertura de nós das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI para determinada quantidade de nós no terceiro conjunto de estudos experimentais computacionais, iniciados com 5000 *pods*.

6.3.2 Resultados para uma alocação inicial de 10000 *pods*

A Tabela 6.3.2 apresenta, para cada instante de tempo, o número de *pods* alocados, o número de nós que foram abertos em cada solução para viabilizar essa alocação de *pods*, o custo de cada solução, o tempo de execução de cada solução e a razão de custo entre as soluções (KS sobre PI). Neste experimento, o número inicial de *pods* foi duplicado, correspondendo a 10000 *pods*, e a capacidade do *cluster* foi superdimensionada em 200 nós.

Instantes	Pods	Nós (KS)	Custo (KS)	Tempo (ms) (KS)	Nós (PI)	Custo (PI)	Tempo (ms) (PI)	Razão
0	10000	6	5760289	4	6	162151	1752	35.52
1	11039	7	6189631	8	6	188696	1931	32.80
2	12681	8	6674270	14	6	230476	1861	28.96
3	15399	10	8460188	21	11	311838	3755	27.13
4	17429	11	8881422	30	10	379898	5099	23.38
5	18464	11	9100892	40	10	413074	4387	22.03
6	14050	8	7586722	47	7	269081	24032	28.19
7	16194	11	8621851	54	9	339982	5670	25.36
8	20148	12	10500000	66	11	476889	3942	22.02
9	15561	10	8458879	74	8	314893	4838	26.86
10	16461	11	8645287	82	9	345487	4683	25.02
11	17934	11	8959012	92	10	396488	3536	22.60
12	22585	13	12300000	107	13	587347	8601	20.94
13	19794	12	10200000	120	11	464720	81259	21.95
14	24659	15	13600000	140	13	683923	4861	19.89
15	28130	17	14900000	163	14	863925	6660	17.25

16	23717	14	13100000	181	13	638397	5700	20.52
17	29745	18	15600000	207	16	957948	7303	16.28
18	25762	16	14300000	227	14	738612	63375	19.36
19	20674	12	10900000	240	11	496930	3824	21.93
20	17457	11	8879466	250	10	379286	39382	23.41
21	19402	12	9875577	262	11	448093	38416	22.04
22	24840	15	13700000	280	13	690113	4782	19.85
23	17697	11	8940852	291	10	386136	3888	23.15
24	20507	12	10800000	302	11	490615	3703	22.01
25	22788	13	12500000	316	13	590343	5423	21.17
26	21552	13	11600000	329	13	531547	25614	21.82
27	26938	16	14500000	349	14	801642	5860	18.09
28	22878	13	12500000	365	13	596308	44479	20.96
29	24753	15	13600000	382	13	686261	4839	19.82
30	17639	11	8928079	391	10	384071	3193	23.25
31	15113	8	8282788	399	7	298934	2648	27.71
32	12876	8	6777241	404	6	234104	4930	28.95
33	11442	7	6286048	409	6	197766	2407	31.79
34	9359	6	5342957	412	4	146149	1622	36.56
35	6620	4	3610498	414	3	86840	1105	41.58
36	5619	2	3318707	415	2	67009	825	49.53
37	6648	4	3638469	417	3	87016	1043	41.81
38	5261	2	3201854	418	3	58750	806	54.50
39	5948	2	3419570	420	3	72877	929	46.92
40	5518	2	3290725	421	3	63585	871	51.75
41	4614	2	3002752	422	3	45918	663	65.39
42	5557	2	3301995	423	3	64782	896	50.97
43	6413	3	3546128	425	3	82179	968	43.15
44	5404	2	3248290	426	3	61614	833	52.72
45	4221	2	2884063	427	2	37653	572	76.60
46	5446	2	3271890	428	3	62263	863	52.55
47	6972	4	3922626	430	3	93944	1204	41.75
48	6186	3	3489655	431	3	77501	1069	45.03
49	4621	2	3024220	432	3	45631	659	66.28

Tabela 6.3.2: Resultados obtidos para as instâncias geradas no terceiro conjunto de estudos experimentais computacionais para uma quantidade inicial de 10000 *Pods*.

O gráfico da Figura 6.30 mostra que, neste segundo cenário do terceiro conjunto de estudos experimentais, o padrão de comportamento verificado no primeiro cenário se repete, com notável eficiência da formulação PI em relação ao Kube-scheduler padrão em termos de custo de alocação ao longo das instâncias de tempo. O eixo x indica a demanda de alocação de *Pods* a cada *slot* de tempo.

A exemplo do que foi verificado no primeiro cenário, iniciado com uma demanda de 5000 *Pods* a serem alocados, a Figura 6.31 mostra que, neste cenário com alocação inicial de 10000 *Pods*, o mesmo comportamento se repete para o tempo de execução de ambas as soluções. Ou seja, o tempo de execução para a solução baseada em PI é muito superior

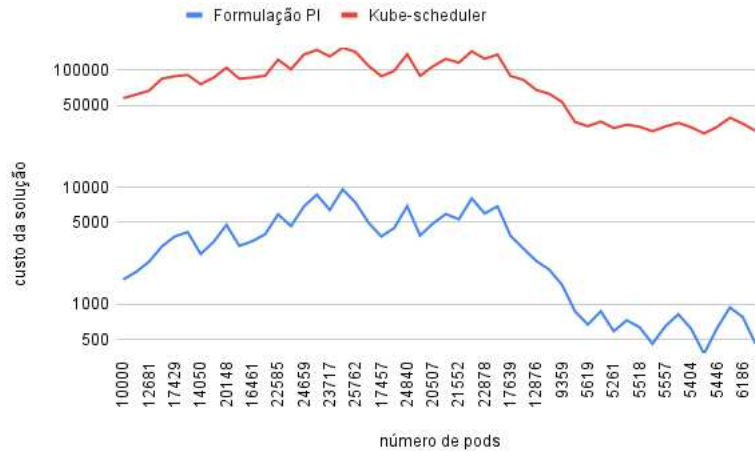


Figura 6.30: Análise comparativa entre o custo das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI para determinada quantidade de nós no terceiro conjunto de estudos experimentais computacionais, iniciados com 10000 *pods*.

e com muito maior variabilidade que a do Kube-scheduler, sendo também muito sensível com relação ao número de *pods* a serem alocados ao longo do tempo.

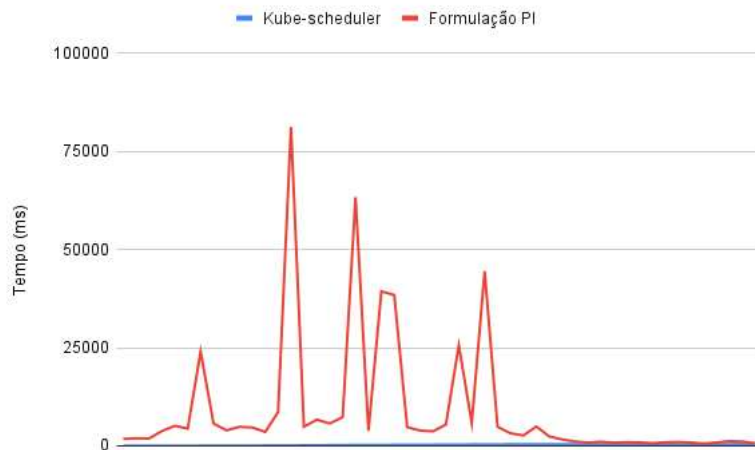


Figura 6.31: Comparativo entre o tempo de execução das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI no terceiro conjunto de estudos experimentais computacionais, iniciados com 10000 *pods*.

A Figura 6.32 apresenta um comparativo entre o número de nós abertos por cada solução ao longo dos instantes de tempo, conforme as respectivas demandas de alocação de *pods*. Verifica-se que ambas soluções possuem curvas muito próximas, com uma ligeira tendência para a solução baseada em PI abrir um número um pouco menor de nós em relação ao Kube-scheduler. Entretanto, diferente do que ocorreu no cenário anterior, desta vez houve casos onde o Kube-scheduler abriu um nó a menos que a solução de PI, como por exemplo para os instantes de tempo 38 a 42, em que esta diferença fica mais explícita.

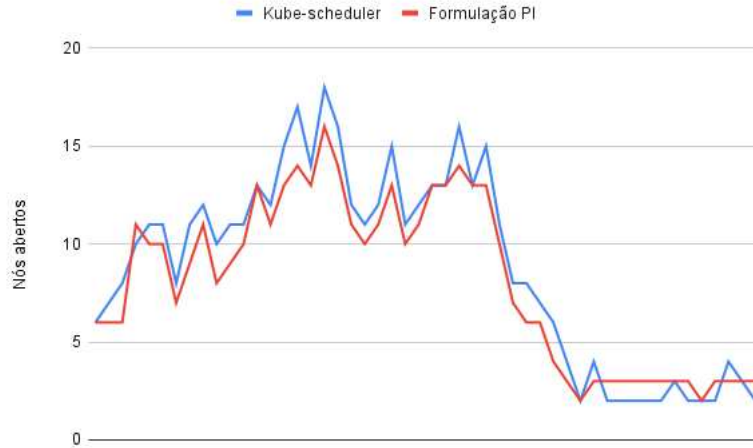


Figura 6.32: Comparativo entre a abertura de nós das soluções dos algoritmos padrão do Kube-scheduler e de formulação PI para determinada quantidade de nós no terceiro conjunto de estudos experimentais computacionais, iniciados com 10000 *pods*.

6.4 Discussão dos resultados

O objetivo dos experimentos conduzidos foi o de validar a formulação proposta tendo como base de comparação o escalonador padrão do Kubernetes, o Kube-scheduler. Para fins desta comparação, usamos como métrica o ganho, em termos de custo computacional, refletido no valor da solução, da nossa proposta contra a do escalonador padrão e também o tempo de processamento gasto em ambas as propostas. Além destas comparações já feitas, a fim de se contemplar um cenário mais realista, realizou-se também a comparação de ambas as abordagens considerando o uso de recursos da infraestrutura ao longo de um dado horizonte de tempo.

Para os dois primeiros conjuntos de experimentos, é possível perceber que, para algumas instâncias com o mesmo número de *pods*, o custo da solução obtida pode aumentar quando se aumenta o número de nós, diferentemente do que seria esperado. Isso acontece devido ao sorteio aleatório que é feito para a geração das instâncias artificiais, que pode chegar a configurações, por exemplo, em que os nós possuem capacidades baixas com custos de abertura e de alocação altos. Isso naturalmente pode levar a instâncias com mais nós que tenham soluções associadas a valores mais altos.

Em relação ao custo computacional, expresso no valor das soluções obtidas pelos dois algoritmos para as instâncias artificiais geradas, tem-se uma diferença média de aproximadamente uma ordem de grandeza obtida pela solução proposta nos três conjuntos de experimentos. Tal fato implica, sob um ponto de vista de ordem prática, em um ganho em termos de utilização da infraestrutura física, ganho esse trazido pela nossa proposta. Isso

permite, portanto, o melhor gerenciamento dos recursos computacionais disponíveis.

Por outro lado, deve-se destacar o fato de que a alocação de *pods* a nós é uma atividade considerada crítica, e que, portanto, deve ser realizada dentro de um intervalo de tempo limite imposto pelo cenário de uso. Nesse sentido, fica evidente que a solução proposta possui um tempo de execução muito maior que a do escalonador padrão, em função do método usado para se buscar uma solução de boa qualidade para o problema. Isso depende do ambiente computacional usado, do método de solução empregado e do tamanho das entradas do problema.

Pode-se observar que, em todos os cenários avaliados, os valores das soluções obtidas pela formulação de PI proposta são melhores do que aqueles das soluções obtidas com o algoritmo Kube-scheduler. Portanto, nestes conjuntos de experimentos com instâncias geradas artificialmente, não foi possível encontrar um cenário específico em que a formulação obtivesse resultados piores que o Kube-scheduler no que se refere ao custo da solução ou ao uso de recursos de infraestrutura. Isso provavelmente se deve ao fato de que a proposta desta dissertação se originou ao trabalhar o problema de alocação de *pods* a nós como um problema de Otimização Combinatória, buscando minimizar um custo total de alocação que foi composto de custos operacionais definidos, enquanto o algoritmo Kube-scheduler, embora disponha de alguns parâmetros de configuração como o **NodeAffinity**, tem um baixo grau de preocupação com a otimização de custos.

Comparando os valores contidos nas Tabelas 6.1 e 6.2, percebe-se uma tendência de aumento na razão de custo médio conforme aumenta o número de nós do *cluster*. Já com relação aos valores das Tabelas 6.3.1 e 6.3.2, onde o *cluster* tem tamanho fixo, verifica-se que os maiores ganhos em termos de razão de custo médio ocorrem para os instantes de tempo nos quais a demanda de alocação de *pods* é menor.

Ainda em relação às Tabelas 6.3.1 e 6.3.2, percebe-se um comportamento comum aos demais conjuntos de experimentos, no qual, mesmo a solução por PI sendo muito mais barata, ela trabalha com um número de nós abertos próximo ao da solução Kube-scheduler padrão, sendo na grande maioria das vezes menor. Isso comprova que a solução proposta, baseada numa formulação de PI, consegue encontrar um cenário próximo ao ótimo que quase sempre possui menos nós abertos para acomodar a demanda de alocação de *pods*, se comparado com os resultados fornecidos pelo algoritmo padrão do Kube-scheduler.

Finalmente, retomando a pergunta de pesquisa elencada no início deste capítulo, conforme evidenciado nos resultados obtidos nos três conjuntos de experimentos conduzidos, pode-se concluir com segurança que a formulação de PI proposta corresponde à estraté-

gia que permite ao ambiente Kubernetes atender à demanda corrente de *pods* com uma tendência de uso de um menor número de nós, e obtendo uma solução cujo custo é consideravelmente menor do que aquela provida pelo Kube-scheduler padrão, o que se traduz em menor uso de recursos de infraestrutura.

7. Conclusão

Neste capítulo, se trazem as considerações finais do trabalho com um breve resumo de todo seu decorrer, desde sua introdução até a elaboração da modelagem e seus experimentos computacionais, além das ideias para trabalhos futuros a serem feitos com base nesta dissertação.

7.1 Considerações finais

Conforme abordado no Capítulo 1, o orquestrador de contêineres Kubernetes traz em sua arquitetura um agendador de *pods*, denominado Kube-scheduler, responsável por alocar os contêineres associados a um dado *pod* nas máquinas virtuais, ou físicas, que fazem parte do chamado *cluster* do Kubernetes. O termo *pod* se refere à implantação de uma instância de aplicação para rodar no *cluster*. Nesse contexto, o Kube-scheduler, por padrão, não considera questões relacionadas com o uso de recursos de infraestrutura, contemplando essa possibilidade apenas por meio de mecanismos de filtragem e priorização que devem ser executados manualmente pelos administradores através de rotulagens pré-existentes na configuração padrão do agendador.

Nesse sentido, esta dissertação propôs um método baseado em uma formulação de Programação Inteira (PI) cujo objetivo é otimizar o custo operacional de alocação de *pods* a nós em um *cluster* Kubernetes, sendo que o termo "nó" é usado para indicar uma máquina física ou virtual de um *cluster* do Kubernetes, sendo assim, parte de uma arquitetura de computação em nuvem. A proposta leva em consideração um conjunto de características dos nós e dos *pods*, tais como a capacidade computacional e/ou de memória dos nós, a quantidade de recursos (de processamento e/ou de memória) usados pelos *pods*, o custo de alocação de um *pod* a um nó, o custo de abertura de um nó, e uma penalidade a ser aplicada em caso de verificação de erros nos *pods* alocados a um nó, dentre outras. A fim de definir o custo de uma solução, foram definidos três coeficientes

na função objetivo: (i) α_i , que se refere ao custo de abertura de um nó i ; (ii) β_i , que concerne ao custo de alocação de um *pod* qualquer a um nó i ; e (iii) γ_i , que representa uma razão de custo aplicada caso os *pods* alocados a um nó i apresentem erros. Além disso, a formulação contém ainda seis restrições que permitem a modelagem adequada do problema abordado.

Assim sendo, a formulação proposta foi comparada com uma implementação do escalonador padrão do Kubernetes em relação ao custo das soluções obtidas e ao tempo de processamento. Foram conduzidos três conjuntos de experimentos computacionais que possuíram a seguinte composição: (i) o primeiro conjunto de estudos experimentais não contemplou o uso da taxa de penalidade por erros, tampouco utilizou o mecanismo de filtragem no algoritmo do Kube-scheduler padrão; (ii) já o segundo conjunto de estudos experimentais contemplou estes dois pontos, de modo que aplicou a taxa de penalidade de erros e utilizou o mecanismo de filtragem e priorização de nós no algoritmo do Kube-scheduler; e (iii) no terceiro conjunto de estudos experimentais, foi considerado o uso de recursos de infraestrutura ao longo de um horizonte de tempo, representando um cenário mais realista, e focando-se no comparativo da quantidade de nós abertos utilizados para o agendamento de *pods* por ambas as soluções.

Os resultados obtidos validaram a aplicação da proposta nos cenários de uso estudados, visto que essa foi capaz de levar a soluções cujos valores de custo a elas associado é, em média, uma ordem de grandeza menor do que os valores associados às soluções obtidas pelo escalonador padrão do Kubernetes, o Kube-scheduler. De interessante a partir da análise conduzida, está o fato de que se percebeu que o custo da solução através da formulação de PI proposta é menor mesmo que a quantidade de nós utilizados tanto na solução provida pelo algoritmo do Kube-scheduler quanto pela formulação de PI sejam similares, ressaltando assim que a formulação de PI proposta é bem-sucedida em escolher nós associados a custos operacionais mais baixos.

Entretanto, destaca-se que, apesar do custo da solução provida pela formulação de PI ser consideravelmente menor do que o custo da solução obtida pelo algoritmo do Kube-scheduler, tal formulação tende a tomar tempos de processamento elevados, devido ao fato de que o resolvidor MIP aplica o algoritmo de *Branch-and-Bound* até que se chegue a uma solução próxima da ótima – isto é, que esteja dentro do *gap* definido. A depender da dificuldade e do tamanho da instância, isso pode tomar uma quantidade de tempo excessiva. Como o escalonamento de *pods* é uma atividade crítica, que pode exigir resolução dentro de um intervalo de tempo limite restrito, essa é uma questão que deve ser levada em consideração na escolha de qual algoritmo de alocação de *pods* deve ser usado em um ambiente real, principalmente se a necessidade operacional associada ao ambiente

real, de nuvem, consistir em escalar aplicações que exigem uma disponibilidade imediata.

7.2 Trabalhos futuros

Como trabalhos futuros, deseja-se realizar experimentos em nuvem com um cenário real de uso, a partir da modificação do algoritmo do escalonador padrão do Kubernetes, utilizando-se a formulação proposta. Para tal, vislumbra-se que o caminho seja implantar o resolvidor MIP utilizado dentro do próprio ambiente do Kubernetes operando em nuvem, para que seja possível executar o escalonador proposto. Busca-se ainda trabalhar em uma proposta que incorpore o conceito de filas de *pods* pendentes na estratégia de horizonte de tempo adotada, a fim de contemplar os *pods* que porventura não conseguirem sua alocação em um dado instante de tempo.

O trabalho apresentado não leva em consideração o custo de migrar um *pod* de um nó para outro, permitindo que nos cenários dos experimentos aplicados o modelo acabe por realocar os *pods* desnecessariamente quando o modelo é aplicado para cada instante de tempo t_i . Espera-se que em cenários futuros possa-se alocar apenas a demanda de *pods* necessária que ainda não possui vínculo com um nó, evitando assim remanejamentos dentro dos nós desnecessários, que impactam consecutivamente o custo final da solução.

Os recursos de nós e *pods* foram representados de maneira unidimensional, enquanto são, por sua vez multidimensionais (e.g., processamento, memória, armazenamento, comunicação). Seria o caso então realizar experimentos computacionais levando em conta essas múltiplas dimensões, o que o tornaria, sem dúvida, ainda mais interessante.

Também deve-se adotar uma proposta heurística ou meta-heurística para a solução do problema, comum em problemas $\mathcal{NP}$.

Por fim, espera-se também trabalhar com outros limiares de *gap* para o algoritmo de *Branch-and-Bound*, definido por meio de um parâmetro do resolvidor MIP, que possibilitem que se chegue a soluções de boa qualidade para a formulação de PI. Em outras palavras, soluções nas quais os custos associados ainda sejam muito abaixo daqueles obtidos para as soluções por meio do uso do Kube-scheduler. Contudo, almeja-se que esses outros limiares de *gap* também possibilitem uma redução significativa do tempo computacional necessário para gerar as soluções através da formulação de PI.

Referências Bibliográficas

- [1] Zhang, J., Chen, S., Wang, X. & Zhu, Y. Dynamic Edge Server Reservation for Connected Vehicles with Deep Reinforcement Learning. *IEEE INFOCOM 2021 - IEEE Conference On Computer Communications*. pp. 1-10 (2021)
- [2] Cornujols, G., Nemhauser, G. & Wolsey, L. The uncapacitated facility location problem. (Cornell University Operations Research.
- [3] Fischetti, M., Ljubi, I. & Sinnl, M. Redesigning Benders decomposition for large-scale facility location. *Management Science*. **63** pp. 2146-2162 (2017)
- [4] Cornuéjols, G., Nemhauser, G. & Wolsey, L. Discrete Location Theory. (John Wiley & Sons,1990)
- [5] Kuehn, A. & Hamburger, M. heuristic program for locating warehouses. *Management Science*. **9** pp. 643-666 (1963)
- [6] Dhall, S. & Liu, C. On a real-time scheduling problem. *Operations Research*. **26** pp. 127-140 (1978)
- [7] Walker, R. & Chaudhuri, S. Introduction to the scheduling problem. *IEEE Design & Test Of Computers*. **12** pp. 60-69 (1995)
- [8] Fischetti, M., Ljubi, I. & Sinnl, M. Benders decomposition without separability: A computational study for capacitated facility location problems. *European Journal Of Operational Research*. **253** pp. 557-569 (2016), <https://www.sciencedirect.com/science/article/pii/S0377221716301126>
- [9] Li, W., Lemieux, Y., Gao, J., Zhao, Z. & Han, Y. State of the Art, and Future Research Opportunities. *2019 IEEE International Conference On Service-Oriented System Engineering (SOSE)*. pp. 122-1225 (2019)

- [10] Botez, R., Costa-Requena, J., Ivanciu, I., Strautiu, V. & Dobrota, V. SDN-Based Network Slicing Mechanism for a Scalable 4G/5G Core Network: A Kubernetes Approach. *Sensors*. **21** (2021), <https://www.mdpi.com/1424-8220/21/11/3773>
- [11] Wojciechowski, L., Opasiak, K., Latusek, J., Wereski, M., Morales, V., Kim, T. & Hong, M. Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh. *IEEE INFOCOM 2021 - IEEE Conference On Computer Communications*. pp. 1-9 (2021)
- [12] Ren, Y., Wen, P., Liu, H., Zheng, Z., Chen, Y., Huang, P. & Li, H. Secure Two Factor Authentication Using Active Sound Sensing. *IEEE INFOCOM 2021 - IEEE Conference On Computer Communications*. pp. 1-10 (2021)
- [13] Zhong, Z. & Buyya, R. Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources. *ACM Trans. Internet Technol.* **20**, <https://doi.org/10.1145/3378447>
- [14] Liu, B. Theory And Practice Of Uncertain Programming. *Pp.* pp. 157-165 (2009), <https://doi.org/10.1007/978-3-540-89484-1>
- [15] Business logic | Wikipedia, 2022, http://en.wikipedia.org/wiki/Business_logic_layer
- [16] MIPGap | Gurobi, 2023, <https://www.gurobi.com/documentation/9.5/refman/mipgap2.html>
- [17] Downloads | Java, 2023, <https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>
- [18] Documentation | Gurobi Optimization, 2023, https://www.gurobi.com/documentation/current/refman/java_api_overview.html
- [19] Horizontal Pod Autoscaling | Kubernetes, 2023, <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>
- [20] Como o DevOps afeta o trabalho do CIO | Mandic, 2023, <https://blog.mandic.com.br/artigos/como-o-devops-afeta-o-trabalho-do-cio/>
- [21] Expert, 2023, https://stack.desenvolvedor.expert/appendix/docker/container_vm.html
- [22] Components | Kubernetes, 2023, <https://kubernetes.io/docs/concepts/overview/components/>
- [23] Scheduling Framework | Kubernetes, 2023, <https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/>

- [24] Ahmad, I., AlFailakawi, M., AlMutawa, A. & Als Salman, L. assessment. *Journal Of King Saud University - Computer And Information Sciences*. **34** pp. 3934-3947 (2022), <https://www.sciencedirect.com/science/article/pii/S1319157821000562>
- [25] Sierksma, G. & Zwols, Y. Linear and integer optimization: theory and practice. (CRC Press.
- [26] Zhang, D., Yan, B., Feng, Z., Zhang, C. & Wang, Y. Container oriented job scheduling using linear programming model. *2017 3rd International Conference On Information Management (ICIM)*. pp. 174-180 (2017)
- [27] Zhou, R., Li, Z. & Wu, C. Scheduling frameworks for cloud container services. *IEEE/acm Transactions On Networking*. **26** pp. 436-450 (2018)
- [28] Wan, X., Guan, X., Wang, T., Bai, G. & Choi, B. Application deployment using Microservice and Docker containers: Framework and optimization. *Journal Of Network And Computer Applications*. **119** pp. 97-109 (2018)
- [29] Kaur, K., Garg, S., Kaddoum, G., Ahmed, S. & Atiquzzaman, M. Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem. *IEEE Internet Of Things Journal*. **7** pp. 4228-4237 (2019)
- [30] Gong, C., Liu, J., Zhang, Q., Chen, H. & Gong, Z. The Characteristics of Cloud Computing. *2010 39th International Conference On Parallel Processing Workshops*. pp. 275-279 (2010)
- [31] Pizza como serviço? – Entenda os modelos de contratação de Nuvem | 4Future, 2023, <https://4future.com.br/index.php/2021/10/27/pizza-como-servico-entenda-os-modelos-de-contratacao-de-nuvem/>
- [32] Graesser, L. & Keng, W. Foundations of deep reinforcement learning. (Addison-Wesley Professional.
- [33] Wang, W., Rashid, A. & Chuang, H. Toward the trend of cloud computing. *Journal Of Electronic Commerce Research*. **12**, 238 (2011)
- [34] Xue, C. & Xin, F. Benefits and challenges of the adoption of cloud computing in business. *International Journal On Cloud Computing: Services And Architecture*. **6**, 01-15 (2016)
- [35] Ali, M., Khan, S. & Vasilakos, A. Security in cloud computing: Opportunities and challenges. *Information Sciences*. **305** pp. 357-383 (2015)

- [36] Tuli, S., Tuli, S., Tuli, R. & Gill, S. Predicting the growth and trend of COVID-19 pandemic using machine learning and cloud computing. *Internet Of Things*. **11** pp. 100222 (2020)
- [37] Dasgupta, S. & Papadimitriou, C. Algorithms. (2006)
- [38] kube-scheduler | Kubernetes, 2023, <https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/>
- [39] Gurobi, 2023, https://gurobi.github.io/modeling-examples/facility_location/facility_location.html
- [40] Resource Management for Pods and Containers | Kubernetes, 2023, <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>
- [41] Wikipedia - Branch and Bound, 2023, https://en.wikipedia.org/wiki/Branch_and_bound
- [42] What is a Container? | Docker, 2023, <https://www.docker.com/resources/what-container>
- [43] Kubernetes, 2023, <https://kubernetes.io/>
- [44] Github - camilla-m/facility-problem, 2023, <https://github.com/camilla-m/facility-problem>.
- [45] Facility location problems — Mathematical Optimization: Solving Problems using Gurobi and Python, 2023, <https://scipbook.readthedocs.io/en/latest/flp.html>
- [46] PaaS ou IaaS para uso empresarial: As principais diferenças e semelhanças entre eles | Brazuca Cloud, 2023, <https://brazucacloud.com.br/sem-categoria/paas-ou-iaas/>
- [47] SBPO - Simpósio Brasileiro de Pesquisa Operacional, 2023, <https://www.sobrapo.org.br/sbpo-simposio-brasileiro-de-pesquisa-operacional>
- [48] ICUMT 2023, 2023, <https://icumt.info/2023/>
- [49] Service Mesh - o que é e para que serve uma rede service mesh?, 2023, <https://www.redhat.com/pt-br/topics/microservices/what-is-a-service-mesh>
- [50] Papadimitriou, C. & Steiglitz, K. Combinatorial Optimization: Algorithms and Complexity. (Prentice Hall,1982)
- [51] Wikipedia - Gurobi Optimizer, https://en.wikipedia.org/wiki/Gurobi_Optimizer

- [52] Cornuéjols, G., Nemhauser, G. & Wolsey, L. The uncapacitated facility location problem. (Cornell University Operations Research,1983)
- [53] Garey, M. & Johnson, D. Computers and Intractability; A Guide to the Theory of NP-Completeness. (W. H. Freeman & Co.,1990)
- [54] Lawler, E. A Note on the Complexity of the Warehouse Location Problem. *Management Science*. **22**, 802-805 (1976)
- [55] Farahani, R. & Hekmatfar, M. Facility location: concepts, models, algorithms and case studies. (Springer Science & Business Media,2009)
- [56] Cormen, T., Leiserson, C., Rivest, R. & Stein, C. Introduction to algorithms. (MIT press,2022)
- [57] Martins, C., De Souza Moura, P., De Lucena, S. & Vieira Campos, C. Container Scheduling in Kubernetes Clusters Using a Mixed Integer Linear Programming Approach. *2023 15th International Congress On Ultra Modern Telecommunications And Control Systems And Workshops (ICUMT)*. pp. 140-145 (2023)
- [58] Martins, C. Escalonamento de Containers em Clusters Kubernetes Usando uma Abordagem de Programação Linear Inteira Mista. *Anais Do Simpósio Brasileiro De Pesquisa Operacional*. (2024)

