Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH **Open Access**

Constrained optimal grouping of cloud application components

Marta Różańska[1*] and Geir Horn[1*]

Abstract

Cloud applications are built from a set of components often deployed as containers, which can be deployed individually on separate Virtual Machines (VMs) or grouped on a smaller set of VMs. Additionally, the application owner may have inhibition constraints regarding the co-location of components. Finding the best way to deploy an application means finding the best groups of components and the best VMs, and it is not trivial because of the complexity coming from the number of possible options. The problem can be mapped onto may known combinatorial problems as binpacking and knapsack formulations. However, these approaches often assume homogeneus resources and fail to incorporate the inhibition constraints. The main contribution of this paper are firstly a novel formulation of the grouping problem as constrained Coalition Structure Generation (CSG) problem, including the specification of the value function which fulfills the criteria of a Characteristic Function Game (CFG). The CSG problem aims to determine stable and disjoint groups of players collaborating to optimize the joint outcome of the game, and a CFG is a common representation of a CSG, where each group is assigned a value and where the value of the game is the sum of the groups' contributions. Secondly, the Integer-Partition (IP) CSG algorithm has been modified and extended to handle constraints. The proposed approach is evaluated with the extended IP algorithm, and a novel exhaustive search algorithm establishing the optimum grouping for comparison. The evaluation shows that our approach with the modified algorithm evaluates on average significantly less combinations than the CSG state-of-the-art algorithm. The proposed approach is promising for optimized constrained Cloud application management as the modified IP algorithm can optimally solve constrained grouping problems of attainable sizes.

Introduction

Cloud applications are by definition distributed applications consisting of multiple, communicating components often deployed as containers [1]. More and more applications are hybrid Cloud applications meaning that the application components are deployed to different Cloud infrastructures, private infrastructures, and on heterogeneous hardware provided by dedicated computing devices close to the sensors for applications collecting and processing data from sensors. Furthermore, each application component has resource requirements and dependencies on the data location. Also, an application component may depend on a timely data flow with other components or external data sources. Cloud application components can be deployed individually on separate Virtual Machines (VMs) or grouped on a smaller set of Virtual Machines (VMs), "Edge", and "Fog" nodes in the computing continuum [2].

Finding the best way to deploy and manage distributed Cloud applications over multiple Cloud providers is a complex optimization problem [3]. Solving this optimization problem means finding the best application *configuration* so that the application *utility* is maximized by the deployment and subject to the operational constraints set for the application. It is well known that the utility of an application is essential to guide autonomic decisions related to the application configuration or its

*Correspondence:
Marta Różańska
martaroz@ifi.uio.no
Geir Horn
Geir.Horn@mn.uio.no
[1] University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway

Springer Open

© The Author(s) 2024. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

deployment [4], and rational economical agents should maximize their utility [5].

This optimization problem can be decomposed into a two-stage approach. One first decides on the constrained attribute values of the components to be deployed, and then uses these values to *filter* the available VMs selecting only the VMs that can be used to instantiate each component type [3]. The grouping can of benefit be combined with the selection step. The problem is then to *partition* the set of components into a set of non-overlapping *groups* and then for each group (subset) *select* one of the available VMs that gives the highest *utility* for the application owner.

There could be several reasons for grouping to be beneficial. An application with temporal workflow dependencies among the application components will benefit from grouping components from various stages of a workflow path into one execution group. This one execution group can use and re-use a single VM for executing the chain of components one after the other thereby avoid data communication delays. It is particularly important for time-sensitive workflows, where the workflow must be executed as fast as possible, like workflows processing medical data [6]. Secondly, a grouping of application components can be beneficial where a set of components processes data from the same data source. Grouping will then minimize data traffic outside the VM and minimize latency in the data access and the communication between the application components, and hence improve the application performance. It is beneficial for latency-sensitive applications, like streaming or gaming applications, which are becoming more and more popular. Thirdly, deploying only one application component per VM may inherently waste resources. Consider the case where a component requires 5 cores to run, but there are only VMs available with 4 or 8 cores. Hence, only the larger VM can be used, and one will save cost by deploying also a smaller application component requiring only two or three cores to the same VM, if such a component type exists in the application. All these aspects are relevant for the applications deployed in the computing continuum, where the efficient use of resources is crucial [7].

The grouping of application components can be approached in two ways:

1. One may first select the VMs to use, and then deploy the components in an optimal way over the selected machines; or
2. One may first group the components and then select the appropriate VMs providing enough resources to host the various groups of components.

From a datacentre point of view the machines are not virtual, and therefore datacentre management must be based on the first approach, which has then been carried over to Cloud resource management solutions [8], such as Kubernetes[1]. The second approach gives more freedom to the optimization process. It allows for situations where the application will be deployed to many different Cloud providers offering different VM types, or where the available VM types may change over the application lifetime. The second approach will better support application management from the application owner perspective where the number of Cloud provider deployment offers is almost countless. Furthermore, according to the authors' knowledge, the second approach was not investigated before. The second approach is aligned with the modern Cloud application optimization approaches like the *Multi-cloud Execution ware for Large scale Optimised Data Intensive Computing*[2] (MELODIC), which are used to manage Cloud applications deployed in hybrid Cloud settings where hardware accelerators and Cloud platform services are included as application component deployment options [3]. The second approach to the grouping is therefore the main focus in this paper.

The main contribution of this paper is a new approach for solving the constrained grouping problem for Cloud application deployment and optimization. It is a two-stage process where not only the best placement of components into VMs is found, but also the best set of VMs. The grouping problem is then formulated as Coalition Structure Generation (CSG) problem. According to the authors' knowledge, the algorithms from the CSG field have not been used for Cloud application components management before. To solve this grouping problem, the authors propose a novel cost-saving function which allows the use of algorithms defined for Characteristic Function Game (CFG) type of CSG.

Finally, we have extended the CSG Integer-Partition (IP) [9] algorithm to include inhibition constraints. We also present a novel restricted growth function exhaustive search algorithm, which is guaranteed to generate all possible combinations of component groups and provides the baseline for evaluation our CSG IP algorithm . A thorough and realistic evaluation shows that our modified IP outperforms the state-of-the-art vanilla algorithms for a larger number of grouping restrictions among the application components, and for some experiments for all of the tested cases. The number of constructed groupings by the modified IP algorithm is much lower than by the state-of-the-art CSG algorithm, and thousands of times lower than by the exhaustive search.

[1] https://kubernetes.io/
[2] https://melodic.cloud/

It means that the decision on the constrained grouping of Cloud application components can be optimal for applications composed of more components. The constrained grouping gives more optimal deployments in terms of both cost and performance, hence a better overall optimization of Cloud applications.

Since the application component grouping problem can be mapped onto multiple well studied combinatorial optimization problems, this paper starts with a formal definition of the grouping problem and its complexity in "Grouping background" section and gives an overview of relevant approaches and results from the literature. This section makes the paper comprehensive and self-contained, but can be omitted on a first reading as it is only a background for understanding the optimization problem defined in "Cloud application optimization" section and the grouping approach in "Grouping approach" section. "Exhaustive search" section derives our exhaustive search algorithm and provides an analysis of the complexity leading to a method for computing the complexity of a given grouping problem with inhibition constraints among the application components. This section provides the baseline for the evaluation, and may be omitted on a first reading if one is only interested in the evaluation of our approach. "Constraint aware CSG" section provides a detailed exposition of our constraint aware CSG algorithm, and this quite technical section is a reference for anyone interested in replicating our thorough evaluation results presented in "Evaluation" section. Finally, the discussion of the results and the future work are presented "Discussion" and "Conclusion and future work" sections.

Grouping background

In this section, we present the related problems and algorithms for the set partitioning problem that can possibly be used for the constrained optimal grouping of Cloud application components. Also, existing algorithms and approaches to optimize the Cloud application components with inhibition constraints and possible limited VM offering are discussed. We note that the majority of approaches are focused on scheduling or management on the given fixed set of resources or on the given cluster of VMs while the approach presented in this paper aims to decide on both the application components placement and the best VMs to host them. The approach presented in this paper makes the problem more complex, but also more realistic because both decisions must be made to perform the optimal deployment of Cloud applications. We present a few papers we managed to find in this area as well as the Cloud application management approaches we found.

Grouping model

The Cloud application components can be deployed individually on separate VMs or grouped together on a smaller set of VMs. The grouping $\mathbb{G}$ is a partitioning of the set of component types into exhaustive and disjoint groups, $\mathbb{G} = \{\mathbb{G}_1, \ldots, \mathbb{G}_{|\mathbb{G}|}\}$ In other words, for an index set $\mathbb{I} \subset \mathbb{N}^+$ where $\inf(\mathbb{I}) = 1$, the goal is to find a grouping $\mathbb{G} = \left\{\mathbb{G}_1, \ldots, \mathbb{G}_{|\mathbb{I}|}\right\}$ so that $\mathbb{T} = \cup_{i\in\mathbb{I}}\mathbb{G}_i$ with disjoint groups, i.e. $\mathbb{G}_i \cap \mathbb{G}_j = \emptyset$ for $i \neq j$, maximizing the total value of the groups

$$\max V(\mathbb{G}) \tag{1}$$

The least number of groups in the grouping is the single group identical to $\mathbb{T}$, and the most groups that can partition the application component type set $\mathbb{T}$ is the grouping where each group has one single component type as a member. Hence, the number of groups in the partition must fulfil $1 \leq |\mathbb{I}| \leq |\mathbb{T}|$.

Once the requirement attributes of the various application components have been decided, the components can potentially be grouped together on the *best* fitted VMs, if there is at least one VM that is capable to host each group.

The selection of the VM to host the group is defined by the combined attributes of the application components in the group: An allocation of a group to a VM is *feasible* if the combined resource requirements of the assigned attribute values of the components in the group do not exceed the resource capacity of the VM in any resource dimension.

The requirement attributes for components are decided at the *type* level, so it is sufficient to know the component types of the components in a group to decide on the VM capable of hosting the group. Hence, the grouping done at the type level has the consequence that the size of a grouping $|\mathbb{G}|$ must be less or equal the number of types, i.e. $|\mathbb{G}_i| \leq |\mathbb{T}|$. Furthermore, one can note that the different groups possible is the *power set* of the set of component types less the empty set, $\mathbb{G} \in 2^{\mathbb{T}}\backslash\emptyset$, and the number of different groups is therefore one less than the size of the power set: $2^{|\mathbb{T}|} - 1$. It is therefore theoretically possible to enumerate *a priori* all the different groups that can be formed for an application having less than around fifty different component types $\mathbb{T}$.

Worst case complexity

The complexity of finding the optimal grouping depends on the search strategy used, and it is therefore impossible to state the complexity in generality. However, in the worst case, an exhaustive search is needed to evaluate all possible ways to partition the set of application component types as the optimal partition can only be found by assigning a value to each partition and then ranking the values of the partitions.

Let the number of groups in the partition be fixed as $|\mathbb{I}|$, then the number of ways the $|\mathbb{T}|$ application component types can be grouped into $|\mathbb{I}|$ groups is given by the *Stirling number of the second kind* ([10], p. 73-76):

$$\mathscr{S}_2(|\mathbb{T}|, |\mathbb{I}|) = |\mathbb{I}|\mathscr{S}_2(|\mathbb{T}| - 1, |\mathbb{I}|) + \mathscr{S}_2(|\mathbb{T}| - 1, |\mathbb{I}| - 1) = \frac{1}{|\mathbb{I}|!}\sum_{i=0}^{|\mathbb{I}|}(-1)^{|\mathbb{I}|-1}\binom{|\mathbb{I}|}{i} i^{|\mathbb{T}|}$$

where

$$\binom{|\mathbb{I}|}{i} = \frac{|\mathbb{I}|!}{i!(|\mathbb{I}| - i)!} = \left[(|\mathbb{I}| + 1 - i)\binom{|\mathbb{I}|}{i-1}\right] \Big/ |\mathbb{I}| \quad (2)$$

is the binomial coefficient with the recursive form (2) better supporting computation with integer arithmetic. Adding the Stirling number of the second kind together for all the possible number of groups there can be in the partition, or the grouping, gives the total number of partitions one can create from the set $\mathbb{T}$. This is known as the *Bell number* $\mathscr{B}(|\mathbb{T}|)$ from being discussed by E. T. Bell [11]:

$$\mathscr{B}(|\mathbb{T}|) = \sum_{k=1}^{|\mathbb{T}|} \mathscr{S}_2(|\mathbb{T}|, k) \quad (3)$$

$$= \sum_{k=1}^{|\mathbb{T}|-1} \binom{|\mathbb{T}| - 1}{k} \mathscr{B}(k) \quad (4)$$

$$= \frac{1}{e}\sum_{k=1}^{\infty} \frac{k^{|\mathbb{T}|}}{k!} \quad (5)$$

$$= \left\lceil \frac{1}{e}\sum_{k=1}^{2|\mathbb{T}|} \frac{k^{|\mathbb{T}|}}{k!} \right\rceil \quad (6)$$

$$< \left(\frac{0.792 \cdot |\mathbb{T}|}{\ln(|\mathbb{T}| + 1)}\right)^{|\mathbb{T}|} \quad (7)$$

where the recursive formula (4) was proven by Rota [12]; the form (5) was given by Dobinski [13]; the finite series expansion (6) was given by Comtet [14], and the upper bound (7) was recently established by Berend and Tassa [15]. If the size of the application component type set $|\mathbb{T}|$ is not too large, the Bell numbers can be computed[3] with polynomial complexity by combining the recursive binomial coefficient (2) with the recursive formula (4).

It is useful to obtain the Bell number for a particular set of application components to ensure the time feasibility of solving directly the grouping problem by an exhaustive search since the number of possible groupings grows exponentially as the Bell number with the size of the set of application components $|\mathbb{T}|$.

Group Technology

A similar problem to the one described in this paper existed for many years and it was known in the manufacturing industry as Group Technology [16]. It is a manufacturing technique where there are machines and there are parts that should be prepared with the use of these machines. The goal is similar to the grouping of VMs which means to find the set of groups called *families* and a set of parts called *cells* that should be executed together. One can map cells to component groups and families to VMs. However, the machines in Group Technology are physical hence they have limited cardinality whereas the infinite elasticity promised by Cloud providers lifts this limitation. The solutions to these problems such as Production Flow Analysis [17] were firstly designed to be solved manually, without computers. Then some computer techniques were also applied, and the problem was formulated as the Set Partition Problem.

Allocation

Given the first approach mentioned in "Introduction" section to optimization where one first will select the VMs to be used by the application, one may consider the optimized grouping problem as an *allocation* problem [18]. In this case the application component instances are allocated to a predefined subset of the available VMs since the number of available VMs offered by the available Cloud providers make the optimization problem prohibitively large. However, if one would like to optimize the allocation, it will be better to use *bin packing* algorithms.

Bin packing

Bin packing minimizes the number of VMs used for hosting the given set of components, i.e. minimize the bins to pack. There are many algorithms that can be used to approximate the optimal packing [19]. Bansal et al. have derived the currently best algorithm for homogeneous VMs where the aim is to pack the components on as few VMs as possible [20]. To the authors' knowledge, the best algorithm for heterogeneous VMs

[3] The authors have a C++ implementation available on request.

that also allows associating a *cost* to the various bin types available, i.e. the VMs, with the goal of minimising the total cost of the bins is the approximation scheme due to Patt-Shamir and Rawitz [21].

Multidimensional Knapsack Problem

One may alternatively consider the optimization as a knapsack problem [22], or more specifically as a multidimensional knapsack problem given that the components to be grouped have resource requirements in multiple dimensions. There are many heuristics proposed for this kind of problems [23]. However, the problem at hand is known as a Multiple Multidimensional Knapsack Problem (MMKP) since it is necessary to pack all the VMs at the same time. This variant of the knapsack problem has received little attention, but Yi and Cai have proposed a polynomial time approximation [24], and Song et al. have, to our knowledge, developed the only exact algorithm available [25]. The use of MMKP for data centre management has been demonstrated by Camati et al. who evaluated several MMKP heuristics for allocating VMs to homogeneous servers [26]. The problem can be relaxed by considering each VM as an individual knapsack that is assigned component instances, and only one of each type. This corresponds to the Multiple-Choice Multidimensional Knapsack Problem (MCMKP), and the best polynomial time approximation scheme is due to He et al. [27]. The issue with using MCMKP in this way is that there is no global optimization of the goodness of the assignment and trying to maximise all the packings at the same time leads to a Multiobjective Multidimensional Knapsack Problem (MOMKP) [28]. However, only problems with at most three objectives have been considered for which heuristic approaches exist and finding a solution is closely related to the Pareto front of the utility optimization problem [29].

From a Cloud *application* management perspective, it will be desirable to group the components first and then select the VMs that optimizes the global application *utility* which is the second approach mentioned in "Introduction" section. For instance, the MELODIC platform provides utility optimized autonomic cross-Cloud application deployment and management, and acquires the needed application resources from the wide variety of VMs offered by multiple Cloud providers [3]. Hence, the number of possible 'bins' available would render the bin packing infeasible. However, there is an alternative to bin packing as Voß and Lalla-Ruiz have shown that the MCMKP can be reformulated as a *set partitioning problem* [30]. This approach is followed in this paper.

Game theory: Coalition Structure Generation

The set partitioning problem is well studied in the field of game theory where is it known as coalitional games or Coalition Structure Generation (CSG). One of the most common types of CSG is the Characteristic Function Game (CFG). It is a type of CSG where the value of each group is the same independently from other groups in the grouping, and the value of the grouping is the sum of group values. There are many algorithms for the optimal CSG in CFG. A survey provided by Rahwan et al. [31] describes the various approaches in this area and states that algorithms for the optimal CSG in CFG are the fastest in this field. However, for a larger number of component types, it may be not possible to run an optimal algorithm because of its exponential complexity. This was the motivation for anytime CSG in CFG algorithms, such as the Integer-Partition (IP) [9] proposed by Rahwan et al.. They are searching the space in a specific order to provide a solution with the expected quality for a given deadline. Michalak et al. proposed a distributed version of this algorithm [32], so the performance can be improved when running in parallel mode, and then Michalak et al. improved further by adding the optimal dynamic programming aspects with the Optimal Dynamic Programming - Integer-Partition (ODP-IP) [33] algorithm as the result. Currently, the fastest optimal algorithm is called ODP-IP with complexity $O(3^{|\mathbb{T}|})$.

Rahwan et al. defined a constrained coalition formation problem [34]. It can be seen as similar to then problem presented in this paper, but the definition of the problem is different. Firstly, Rahwan et al. considered general constraints on the grouping (i.e. the size of groups in the grouping), which is not directly applicable to the Cloud application components grouping problem defined in "Cloud application optimization" section. Secondly, Rahwan et al. introduced a set of possible groups from which at least one of the groups has to be present in the optimal grouping. This kind of constraint is also not a part of the constrained Cloud application grouping problem. Even though the inhibition constraints were also introduced in that paper, the overall approach was designed to solve a specific problem with various types of constraints that are not applicable to the problem presented in this paper. Therefore, it is not possible to compare our approach with the one of Rahwan et al..

We formulate a constrained grouping of Cloud application components as a coalition formation problem in "Constraint aware CSG" section, and we modify IP algorithm to be able to solve the constrained CSG problem. We use ODP-IP and modified IP in "Evaluation" section to evaluate the approach. Applying these algorithms imposes the CFG requirement on the objective value

function. In other words, this approach is only feasible if a value of grouping is the sum of the individual values of each of the groups that are partitioning the set of components. The CFG also requires that the placement of the components outside the group $\mathbb{G}_i$ does not change the value of this group.

Optimization based approaches

It may be be complicated to assume specific values of each possible grouping, and Ueda et al. proposed to use a generic utility objective function for all possible groups [35]. The utility value of a particular group of components would then be found by solving an optimization problem for finding the best assignment of the requirement attribute values for all the components in the group. Since the groups partition the set of component types, a component type can belong to one and only one of the groups, and the optimization is performed independently for each group and in parallel. Ueda et al. developed an approximation algorithm that considers the incomplete power set with set sizes up to a maximum size, and gives a limit for how close this approximation is to the optimum. This approach can be used as an alternative to the traditional CFG formulation, for cases where formulation of a characteristic function is not natural, or the evaluation of the function is computationally expensive.

Cloud application management

The application component placement problem is often considered a resource management problem [8]. Therefore, it is being solved from the data centre perspective so how to allocate a VM on physical machines. One may consider open source Kubernetes[4] framework as a similar approach. Kubernetes optimizes the placement of application components within the given cluster of nodes. It schedules pods (components) on nodes that are a part of cluster based on the component requirements. The process can look similar, but it is significantly different from the complexity perspective. Kubernetes operates within the given cluster, so it is possible to enumerate and score the best node (VM) to host the component while in the grouping problem, the VMs are unlimited. Even though Kubernetes has a Horizontal Pod Autoscaler[5], it provides only the possibility to scale the cluster according to simple policies, and it does not consider the heterogenous set of VMs.

A more flexible open source framework that operates with Kubernetes is KEDA[6]. It provides event-driven autoscalers which can utilize custom metrics from various sources. However, it still follows the same approach so to firstly create or choose a set of VMs, and then place the components on them, without considering inhibition constraints.

Paraskevoulakou et al. proposed recently the Reinforcement Learning (RL)-based approach for Cloud application component placement [36]. The proposed algorithm is aligned with the approach proposed in this paper. Even though it does not consider inhibition constraints explictely, it will be evaluated and compared to our approach as a part of the future work.

Task scheduling and workflow applications in the Cloud

An algorithm for scheduling the workload and minimizing the makespan has been proposed by Chitgat et al. [37]. One of the algorithm's goals is to increase the utilization of VMs, which is a similar goal as we considered in this paper. However, the algorithm assumes that all VMs are split into three groups of VMs that are arranged based on the processing power, which makes the grouping and scheduling problem simpler. Selvarani et al., proposed another scheduling algorithm that considers both performance, measured as makespan, and cost aspects [38], but it also operates on the given and fixed set of resources.

Another related approach was presented by Nishio et al. for a mobile cloud, where resource-sharing and outsourcing of the work is necessary [39]. This approach could be transformed into a Cloud environment as it is based on utility functions defined for the most important deployment aspects such as latency and power. However, it is not as flexible as the approach presented in this paper because it does not consider inhibition constraints as well as the dynamic VM provisioning.

Multi-Cloud management platforms

According to the authors' knowledge, only MELODIC[7] offers both optimisation of resources needed by Cloud applications, and grouping of application components, and is open source. There are some commercial solutions developed by big tech companies. Even though Google Anthos[8] can be considered one of the most advanced Multi-Cloud and it offers the constrained grouping of Docker containers, it is based on Kubernetes and similar simple autoscaling policies[9]. Furtermore, it does not consider an abstract Cloud application components, and it is not open source. Also, IBM Multicloud Management Platform[10] has wide resources optimisation capabilities,

[4] https://kubernetes.io/

[5] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[6] https://keda.sh/

[7] https://melodic.cloud

[8] https://cloud.google.com/anthos

[9] https://cloud.google.com/anthos/clusters/docs/on-prem/latest/concepts/cluster-autoscaler

[10] https://www.ibm.com/services/cloud/multicloud/management

but it is not able to consider the possibility of co-allocate components under inhibition constraints.

Cloud application optimization

Variability and search space

Modern applications developed for Cloud deployment consist of a set of communicating components [7]. They can be containerized microservices [40]. The application components have different *types* $\mathbb{T} = \{T_1, \ldots, T_{|\mathbb{T}|}\}$.

Each component type $T \in \mathbb{T}$ has a set of attributes $\mathbb{A}_T$. Examples of attributes can be the number of cores the component requires, the amount of memory it requires, or the Cloud providers capable of hosting instances of this component type, or the number of instances necessary to satisfy non-functional application requirements. Each attribute $a_{T,i} \in \mathbb{A}_T$ has a domain $\mathbb{V}_{T,i}$ defining the possible values for this attribute. For instance, a component type may need a certain minimum number of cores to run, and cannot benefit from more than some maximum number of cores.

The variability space $\mathbb{V}_T$ for the component type T is the Cartesian product space of all the domains of its attributes: $\mathbb{V}_T = \mathbb{V}_{T,1} \times \mathbb{V}_{T,2} \times \cdots \times \mathbb{V}_{T,|\mathbb{A}_T|}$. In order to find the best deployment of the application, the optimal application *configuration*, one must assign values to all attributes of all component types from their respective domains. The search space to find the optimal configuration is therefore the application's variability space given as the Cartesian product space of the variability of all the component types of the application: $\mathbb{V} = \mathbb{V}_1 \times \mathbb{V}_2 \times \cdots \times \mathbb{V}_{|\mathbb{T}|}$.

The benefit of finding the optimal configuration at the type level is that even though the search space $\mathbb{V}$ can be large, it is independent of the number of instances of each component type and the number of VMs offered by the usable Cloud providers. Furthermore, constraints among the component type attributes will reduce the effective search space.

Optimization

We consider the set of utility dimensions defined by We et al. with business users [41] and also utility functions presented by Rozanska et al. by surveying MELODIC and MORPHEMIC project business use case application providers [42]. The goal of the deployment optimization is to find the application configuration $\boldsymbol{c}^* \in \mathbb{V}$ that maximizes the application's *utility* for the application owner given a vector of measurements, $\boldsymbol{\theta}(t_k)$, characterizing the application's execution context at the current time t_k [3], and the performance indicators, $\boldsymbol{\psi}(\boldsymbol{c}, \boldsymbol{\theta}(t_k))$, which are the functional dependency between the configuration and the execution context [43]. It must be noted that even though the variability space $\mathbb{V}$ may contain both the number of instances needed for a particular component type and the available Cloud providers as component attributes, it ignores the available VMs. It is often possible that not all combinations of variables from the variability space are *feasible* to be deployed on the available VMs. For instance, one may not find a VM with one core and huge amount of memory. Therefore, we introduce the *feasible space*, $\mathbb{F}\left(\boldsymbol{\theta}(t_k), \hat{\boldsymbol{\psi}}[\boldsymbol{c}, \boldsymbol{\theta}(t_k)]\right)$, which depends on the current execution context $\boldsymbol{\theta}(t_k)$ and performance indicators $\boldsymbol{\psi}(\boldsymbol{c}, \boldsymbol{\theta}(t_k))$. It is assumed that it is possible to capture the utility of the application owner for the deployment of this application as a functional expression, $U(\boldsymbol{c}) : \mathbb{V} \mapsto [0, 1]$, that can be maximized as a standard non-linear programme [44].

$$\boldsymbol{c}^*(t_k) = \underset{\boldsymbol{c} \in \mathbb{F}}{\operatorname{argmax}}\, U\left(\boldsymbol{c}, \boldsymbol{\psi}(\boldsymbol{c}, \boldsymbol{\theta}(t_k)) \middle|\, \boldsymbol{\theta}(t_k)\right) \tag{8}$$

subject to

$$\begin{aligned} \boldsymbol{g}(\boldsymbol{c} \,|\, \boldsymbol{\theta}(t_k)) &\leq \boldsymbol{0} \\ \boldsymbol{h}(\boldsymbol{c} \,|\, \boldsymbol{\theta}(t_k)) &= \boldsymbol{0} \\ \boldsymbol{c} &\in \mathbb{F}\left(\boldsymbol{\theta}(t_k), \hat{\boldsymbol{\psi}}[\boldsymbol{c}, \boldsymbol{\theta}(t_k)]\right) \end{aligned}$$

Combining resource requirement attributes

Combining resource requirement attributes for the component types grouped in a group $\mathbb{G}_i$ means adding them together by some resource function given the feasible configuration $\boldsymbol{c}$ consisting of all attribute value assignments, $a_{T,j}$, for all components types,

$$r_j(\mathbb{G}_i | \boldsymbol{c}) = \sum_{T \in \mathbb{G}_i} a_{T,j} \tag{9}$$

for the attribute dimension j defined for any of the component types that are in the group $\mathbb{G}_i$. This combination must be strictly additive for resources that cannot be shared, like memory. However, for other resources, the requirement can be a peak requirement and then the sum of the combination does not need to equal the sum of the individual requirement attributes, but can be the maximum. Examples of this can be that two components may still work acceptably well if they get less network bandwidth than their combined peak bandwidth requirements, and two components may be able to multitask on a smaller number of cores than the sum of their individually required cores. Therefore, the attribute function for the cores can be the maximum resource requirement attributes of the components in the group:

$$r_j(\mathbb{G}_i | \boldsymbol{c}) = \max_{T \in \mathbb{G}_i} a_{T,j} \tag{10}$$

It should be noted that some attributes may not be available for all component types in a group so in this case the function should consider only the available resource attributes requirements $a_{T,j}$, because the set of combined attributes is the union of the attribute sets of the group types. Consequently, the resource function for all resource requirements attributes in a group is a vector, $\boldsymbol{r}(\mathbb{G}_i|\boldsymbol{c})$.

The requirement attributes of an application component represents the resources needed for maximizing the application utility for the given execution context. A benefit of this approach is that the application optimization will not need to consider a large number of VMs offered by many Cloud providers. The available resources in the Cloud are plentiful and come in many different configurations, and separating the optimization into two steps will allow the optimization problem of each step to be smaller. Given that combinatorial optimization algorithms have exponential complexity, solving two smaller optimization problems may enable the management of applications that would otherwise be intractable [45]. Furthermore, a multi-stage optimization approach allows different *utility* objectives to be optimized in each stage.

Inhibition constraints

The application owner can define which component types to be placed on the same VM, and which component types cannot be placed on the same VM. For the components that have to be placed on the same VM, it is possible to create a super-component and treat them as one component in the optimization process. However, the inhibition constraints have to be handled in the grouping step of the optimization process. The inhibition constraints are used to validate the feasible groups. We introduce an index function $I(T|\mathbb{G}) \in |\mathbb{T}|$ for a given grouping $\mathbb{G}$ that takes a component and returns the index of the group that this component is assigned to. Thus, the constraint that states that component T_i and component T_j cannot be in the same group can be defined as

$$I(T_i|\mathbb{G}) \neq I(T_j|\mathbb{G}) \tag{11}$$

Table 1 Component requirements

Component Name	CPU	RAM
A	6	10
B	2	4
C	1	3
D	1	2

Table 2 Available Virtual Machines (VMs). The price is given in € and is an estimated price for weekly use of the VM

VM Name	CPU	RAM	Price
VM_1	1	2	2
VM_2	2	4	4
VM_3	4	8	8
VM_4	8	16	16
VM_5	16	32	32

Table 3 Groups values when cores re-use is allowed (I) and when cores re-use is not possible (II)

Group	I: VM Name	I: Cost	II: VM Name	II: Cost
A	VM_4	16	VM_4	16
B	VM_2	4	VM_2	4
C	VM_2	4	VM_2	4
D	VM_1	2	VM_1	2
AB	VM_4	16	VM_4	16
AC	VM_4	16	VM_4	16
AD	VM_4	16	VM_4	16
BC	VM_3	8	VM_3	8
BD	VM_3	8	VM_3	8
CD	VM_3	8	VM_3	8
ABC	VM_5	32	VM_5	32
ABD	VM_4	16	VM_5	32
ACD	VM_4	16	VM_4	16
BCD	VM_4	16	VM_4	16
ABCD	VM_5	32	VM_5	32

Example

The following example aims to give an intuition why grouping can be beneficial. Assume that there are four components: A, B, C, and D with maximal requirement attribute values described in Table 1. Furthermore, assume that there is a limited set of available VMs presented in Table 2. The available VMs are limited but their parameters are similar to offers available in the majority of Cloud Providers[11].

The prices of possible groups are presented in Table 3. These prices where calculated as by taking the price of the cheapest VM capable of hosting the group. One may consider two cases: when cores re-use is allowed or where it is not possible to use the same core by two or more components. Table 4 presents the prices for all possible groupings for the two cases of core sharing. It can be easily seen that the best grouping for the case with cores sharing is $\{ABD, C\}$ so to have components $\{ABD\}$ in one group on a VM_4 and C on a separate VM_2, or $\{ACD, B\}$ so

[11] https://aws.amazon.com/ec2/pricing/

Table 4 Grouping values when cores re-use is allowed (I) and when it is not possible (II)

Grouping	I: Cost	II: Cost
$\{ABCD\}$	32	32
$\{ABC, D\}$	34	34
$\{ABD, C\}$	20	36
$\{ACD, B\}$	20	20
$\{BCD, A\}$	32	32
$\{AB, CD\}$	34	34
$\{AC, BD\}$	34	34
$\{AD, BC\}$	34	34
$\{AB, C, D\}$	22	22
$\{AC, B, D\}$	22	22
$\{AD, B, C\}$	24	24
$\{BC, A, D\}$	26	26
$\{BD, A, C\}$	28	28
$\{CD, A, B\}$	28	28
$\{A, B, C, D\}$	26	26

to have $\{ACD\}$ in one group on a VM_4 and B on a separate VM_2. The cost in both cases is 20. Furthermore, if a user does not allow for core-sharing, the best grouping is still $\{ACD, B\}$, but $\{ABD, C\}$ has a significantly higher cost, because $\{ABD\}$ has to be hosted on the bigger VM_5.

Grouping approach

This section presents our approach to the grouping problem, defined as a multi-step optimization where the grouping is a separate step.

Price calculation

Cost is one of the main reasons for companies to move applications to the Cloud [46], and the used VMs are deciding the cost. For each of the completely enumerated groups $\mathbb{G}_i$ one must first *filter* the available VMs to select the VMs that are feasible for the combined resource requirements of the group $\boldsymbol{r}(\mathbb{G}_i|\boldsymbol{c})$, and then choose the VM with the lowest cost. We define $P(\mathbb{G}_i)$ as the price of the cheapest VM capable of hosting the components in $\mathbb{G}_i$, so fulfilling the requirements of the group $\boldsymbol{r}(\mathbb{G}_i|\boldsymbol{c})$. Consequently, the price of the grouping, $P(\mathbb{G})$ is the sum of the prices of the groups of the grouping.

Value function

To formulate the problem as a CFG the value function (1) should be calculated as the sum of the group. It allows us to use and extend the algorithms for CSG in CFG which are the fastest class of algorithms solving of CSG problem. The value function for CFG requires that the value of each group should not depend on the other groups in the grouping. The goal of solving the grouping problem is to find the grouping that maximizes the total value of the grouping, i.e. minimizes the total price of deployment.

As a consequence, if the cost of deploying components on separate VMs $P(\{T_i\}) + P(\{T_j\}) + \cdots + P(\{T_n\})$ is higher than the cost of deploying these components on a single VM $P(\{T_i, T_j, \ldots, T_n\})$, then the value of a group with two or more components deployed together $v(\{T_i, T_j, \ldots, T_n\})$ on a single VM is greater than $v(\{T_i\}) + v(\{T_j\}) + \cdots + v(\{T_n\})$, the sum of values of these components deployed separately.

We propose a normalization approach for the characteristic value function for the cost.

Let P^+ be the cost of the most expensive available VM. The *value* of the group $v(\mathbb{G}_i)$ is defined as the savings made compared to the most expensive deployment which is when all components from the group are deployed separately and using the most expensive VM. In other words, one can calculate the most expensive deployment as a multiplication of the most expensive VM and the number of components in the group, and then subtract the actual price of the group.

$$v(\mathbb{G}_i) = |\mathbb{G}_i| \cdot P^+ - P(\mathbb{G}_i) \tag{12}$$

The grouping problem therefore translates into *selecting* the groups that maximize the total savings for all groups in the grouping, which is the sum of the saving values associated with the groups in the grouping. The value of the grouping $V(\mathbb{G})$ is therefore the sum of its group values

$$V(\mathbb{G}) = \sum_{\mathbb{G}_i \in \mathbb{G}} v(\mathbb{G}_i) \tag{13}$$

and the best grouping $\mathbb{G}^*$ should maximize $V(\mathbb{G})$.

We note that the value function is not contradicting with any reasonable utility function for Cloud application optimization [41]. According to the approach proposed in this paper and the resource function (9), deploying two or more components into the single VM is allowed only if the VM fulfills the resource requirements of all components so they all can be hosted without any loss on the performance. Communication latency is the only other utility dimension besides cost that may be influenced by the grouping of Cloud components. When component-based software applications are deployed using VMs, each component may be deployed on a separate VM. This can lead to an increased number of inter-service communications, which can negatively impact latency, as it was shown by Gribaudo et al. [47]. However, the latency dimension can easily be defined in a way to be included in the value function as an additional reward $l(\mathbb{G}_i)$ or a penalty.

$$v(\mathbb{G}_i) = |\mathbb{G}_i| \cdot P^+ - P(\mathbb{G}_i) + l(\mathbb{G}_i) \tag{14}$$

A function that considers both cost and latency should be constructed to have values reflecting the importance of the latency aspect in relation to the cost. The latency function value that considers inter-component communication can naturally be calculated for a group and its value does not depend on the rest of the groups in the grouping. Therefore, a value function with the latency aspect will still be a characteristic function.

Table 5 presents the Example from "Cloud application optimization" section with $V(\mathbb{G})$ calculated using formula (13) for the cases when core re-use is, or is not, possible. The grouping values are consistent with the condition on the relation between group prices and group values. It is easy to notice that the cheapest option also gives the highest value so the proposed function (12) can be used to solve the grouping problem.

Grouping process

We propose two-step optimization, where the resource attribute values for all components are proposed during the first step, and then the grouping of components is combined with the selection of the best VM to host the components in the second step. The proposed approach is presented as a high-level sequence of actions in Fig. 1.

The optimization problem solver solves the Constraint Problem (8) and assigns values to all the requirements attributes a_{T_i} for all $T_i \in \mathbb{T}$. Then, for every feasible group $\mathbb{G}_i$ in terms of inhibition constraints, the cheapest available VM is found. The selection step is performed with the use of resource function $\boldsymbol{r}(\mathbb{G}_i|\boldsymbol{c})$. If there is no VM that is capable of hosting a group, the group is considered *infeasible* and it is deleted from the set of feasible groups. For all the feasible groups, the value function (12) is evaluated and the grouping algorithm proceeds to find optimal grouping $\mathbb{G}$ of groups. One can note that this multi-stage optimization approach allows different *utility* objectives to be optimized in each stage, hence, the CFG requirement that the total grouping value is a sum of the individual values of the component groups will only be necessary for the grouping stage.

We consider the grouping on the component type level, so every instance of the same component must belong to the same group and it is not possible to host two instances of the same component in the same group. This follows from the observation that putting the same component types instances multiple times into the same group is equivalent to setting the resource attributes requirements for this component proportionally higher. Furthermore, if some groups should be deployed in multiple copies, the group assignment of a component type will remain the same.

For some deployment configurations, the cardinalities of each component type in a group may differ. The deployment process will complete the groups as far as there are component instances available to populate each group fully. When another copy of a group cannot be deployed because all the instances for one or more component type(s) in the group have already been deployed, it is necessary to solve the grouping problem excluding the object type(s) whose instances have all been deployed. In this way, the deployment and the grouping successively smaller component type sets alternate until all required instances are deployed.

Table 5 Grouping cost, $P(\mathbb{G})$, and grouping value, $V(\mathbb{G})$, when cores re-use is allowed (I) and when cores re-use is not possible (II)

Grouping $\mathbb{G}$	I:$P(\mathbb{G})$	I:$V(\mathbb{G})$	II:$P(\mathbb{G})$	II:$V(\mathbb{G})$
$\{ABCD\}$	32	96	32	94
$\{ABC, D\}$	34	94	34	94
$\{ABD, C\}$	20	108	36	92
$\{ACD, B\}$	20	108	20	108
$\{BCD, A\}$	32	96	32	96
$\{AB, CD\}$	34	104	34	104
$\{AC, BD\}$	34	104	34	104
$\{AD, BC\}$	34	104	34	104
$\{AB, C, D\}$	22	106	22	106
$\{AC, B, D\}$	22	106	22	106
$\{AD, B, C\}$	24	104	24	104
$\{BC, A, D\}$	26	102	26	102
$\{BD, A, C\}$	28	100	28	100
$\{CD, A, B\}$	28	100	28	100
$\{A, B, C, D\}$	26	102	26	102

Exhaustive search

Variables and constraints

Solving the partitioning problem means assigning a group to every application component type. In other words, assign a value to a group variable $g_k(T_k) \in \mathbb{I}$ for all $T_k \in \mathbb{T}$, and so $1 \leq k \leq |\mathbb{T}|$. This assignment can be done sequentially for each of the types: The first type will always start the first group, $g_1(T_1) = 1$. Then, the second type can either be assigned to the same group as the first type or create a new group. Continuing this way, the next type to be grouped can either join one of the already created groups or start a new group. This leads to the following constraints for $2 \leq k \leq |\mathbb{T}|$

$$1 \leq g_k(T_k) \leq \max \left\{ g_1(T_1), \ldots, g_{k-1}(T_{k-1}) \right\} + 1 \tag{15}$$

Thus, any kind of combinatorial solver can be used to maximize the objective function (13) for the $|\mathbb{T}|$ integral variables $g_k(T_k)$ subject to the set of constraints (15). The vector $\boldsymbol{g}_{|\mathbb{T}|}$ of all the group assignments satisfying (15) and with the initial value $g_1(T_1) = 1$ is known as a

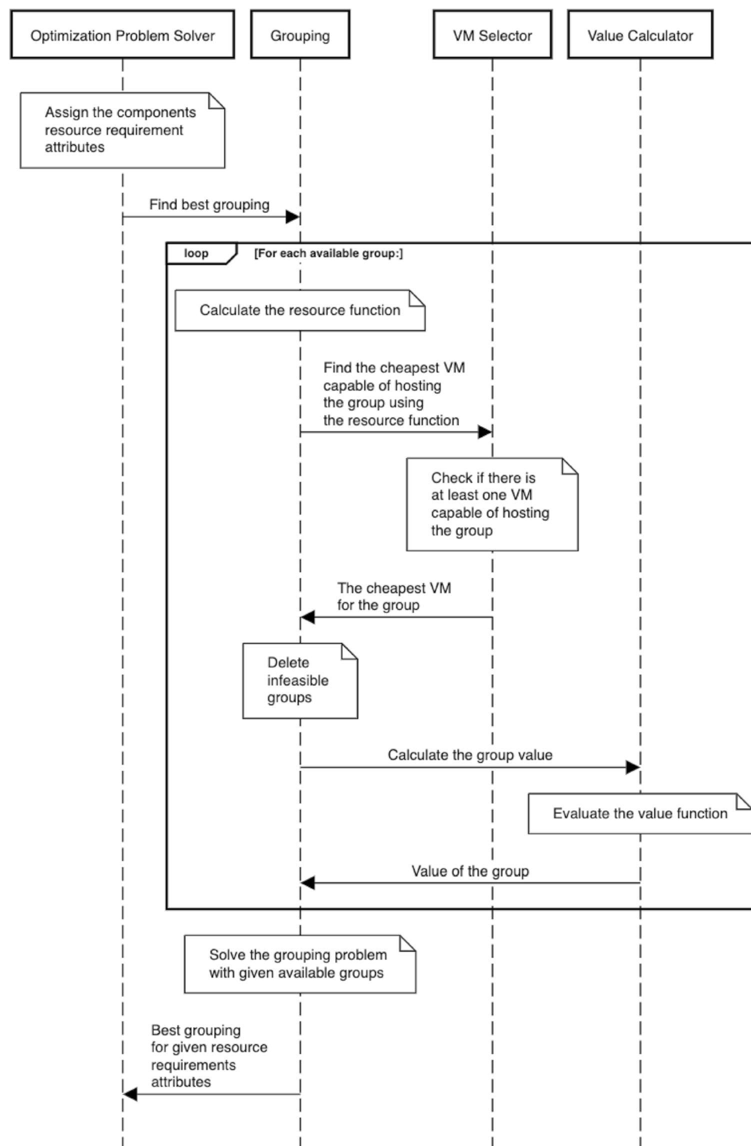

Fig. 1 The high-level schema of the reasoning process for the constrained grouping of Cloud application components

restricted growth function, and Stanton and White have established the bijection between the set of partitions and the set of restricted growth functions ([48], p.18-19). This has allowed efficient algorithms based on Gray codes to list all partitions of a set [49]. However, to our knowledge, there is no algorithm enlisting the partitions under pairwise inhibitions constraints (11), and so a recursive depth first algorithm will be developed in the following.

Recursive search under inhibition constraints

The inhibition constraint (11) translates directly to a constraint on the group indices for the two application component types: $g_i(T_i) \neq g_j(T_j)$. Thus, despite the negative conclusion regarding complexity in "Cloud application optimization" section, it can be possible to search larger application component type sets exhaustively provided that there are sufficient inhibition constraints available to limit the search to a space of tractable size.

An algorithm can be devised by considering the constraint (15) as defining the *domain* $\mathbb{g}_k \subseteq \mathbb{I}$ of values of the constrained group variable $g_k(T_k)$. Thus, $g_1(T_1) \in \mathbb{g}_1 = \{1\}$, $g_2(T_2) \in \mathbb{g}_2 = \{1, 2\}$, and so forth. Let $\boldsymbol{g}_m = [g_1(T_1), \ldots, g_m(T_m)]^T$ be the vector of the partial group variable value assignments with $m < |\mathbb{T}|$, then one has in general that $g_{m+1}(T_{m+1}) \in \mathbb{g}_{m+1} = \left\{1, \ldots, 1 + \sup\left(\boldsymbol{g}_m\right)\right\}$, i.e. the upper bound of the group variable domain equal the upper bound of (15). A recursive depth-first search readily

follows by assigning values to the group variables $g_k(T_k)$ one by one, and for each assignment select the values of the next group variable from its domain one by one. The search will recursively descend a path through the search tree until the full grouping $\mathbb{G}$ is decided by the group value assignments $\boldsymbol{g}_{|\mathbb{T}|}$ for all the application component types $\mathbb{T}$ allowing the evaluation of the objective function (13).

The inhibition constraints are simply removing options from the value domains of the group variables based on the decisions taken by the group variables already fixed. As an example, consider the constraint $g_i(T_i) \neq g_j(T_j)$ with $i < j$ and, from the sequential assignment of the group variables, $g_i(T_i)$ will have a value when the values for $g_j(T_j)$ will be assigned. The domain for $g_j(T_j)$ can then be reduced by removing the group index assigned to $g_i(T_i)$ leaving the domain as $\left\{1, \ldots, 1 + \sup\left(\boldsymbol{g}_{j-1}\right)\right\} \setminus \left\{g_i(T_i)\right\}$. The depth-first recursion is given in Algorithm 1 and this algorithm is started from the second type, $i = 2$ with $\boldsymbol{g}_1 = [1]^T$ since $g_1(T_1) = 1$ always.

Algorithm 1 ExSearch

input : A set of application component types $\mathbb{T}$
A value function $v(\mathbb{G}_i) \mapsto \mathbb{R}^+$ with $\mathbb{G}_i \subseteq \mathbb{T}$
Exclusions $\mathbb{E} = \left\{\mathbb{E}_2, \ldots, \mathbb{E}_{|\mathbb{T}|}\right\}$ where
$\mathbb{E}_i = \{j \in \mathbb{N}^+ : 1 \leq j < i, g_i(T_i) \neq g_j(T_j)\}$
Group variable assignments $\boldsymbol{g}_{i-1}$
output: Optimal grouping $\mathbb{G}^*$ maximizing (1)
The total value of $\mathbb{G}^*$

```
1  g_i = {1, ..., 1 + sup(g_{i-1})}            // Unconstrained domain
2  d_i = g_i \ {g_j(T_j) : j ∈ E_i}            // Constrained domain
3  G* = ∅                                      // Optimal grouping
4  V* = 0                                      // Total value of G*
5  foreach g_i(T_i) ∈ d_i do                   // Scan domain
6      g_i = [g_{i-1}^T, g_i(T_i)]^T           // Group indices
7      if i < |T| then                         // Do depth search
8          {G, V} = ExSearch(T, v, E, g_i)
9      else                                    // Full depth i = |T|
10         G = ∅                               // Grouping for g_i
11         V = 0                               // Total value for g_i
12         for m = 1 to sup(g_i) do            // Process each group
13             G_m = {T_k ∈ T : g_k(T_k) = m}
14             V = V + v(G_m)
15             G = {G, G_m}
16     if V* < V then                          // Store best values
17         G* = G
18         V* = V
19 return {G*, V*}
```

Grouping complexity with inhibition constraints

It is obvious that if there are no constraints, the number of groupings evaluated by the exhaustive search will equal the Bell number, $\mathscr{B}(|\mathbb{T}|)$, as defined in "Worst case complexity" section. The constraints will reduce the size of the search space, but how beneficial the constraints will be depends on how the constraints are distributed among the application component types. Consider for instance the situation where there are $|\mathbb{T}| - 1$ constraints and one component type is involved with all constraints forcing it to be allocated as a singleton, while the other component types can be freely grouped. In this case the number of groupings to search will be reduced to $\mathscr{B}(|\mathbb{T}| - 1)$. However, if the same number of constraints are distributed over almost all the component types, the search space is not reduced that much because there are still plenty of legal groups that can be included in the grouping.

To compute the search space complexity, it is necessary to consider the constraint graph with the component types as vertices and the constraints defined as edges. The complete graph, $K_{|\mathbb{T}|}$, will have $|\mathbb{T}|(|\mathbb{T}| - 1)/2$ edges, and so this is an upper bound on the number of inhibition constraints a given component set $\mathbb{T}$ can have. It also follows that for a problem with m constraints, the number of ways they can be chosen is given by the binomial coefficient,

$$\binom{|\mathbb{T}|(|\mathbb{T}| - 1)/2}{m} \tag{16}$$

This confirms that the same number of constraints can give many different reductions in the search space size.

An *independent partition* of the vertex set of a graph is a partition whose subsets do not contain adjacent vertices. This fits well with the constraint graph where the inhibition constraints define adjacent vertices, and two adjacent vertices should not be in the same subset, i.e., group. The number of independent partitions of a graph, $\mathscr{G}$, is know as the Bell number of the graph, $\mathscr{B}(\mathscr{G})$ [50]. Since the adjacent vertices have to go to different subsets, there is a clear relation to graph coloring. Berceanu established the Bell number of a graph as a weighted sum of the Bell numbers of the possible subset sizes weighted by the coefficients χ_i of the chromatic polynomial of the graph, $\chi_{\mathscr{G}}(k)$, giving the number of ways the graph $\mathscr{G}$ can be coloured with k colors [51]:

$$\mathscr{B}(\mathscr{G}) = \sum_{i=0}^{|\mathbb{T}|} \chi_i \mathscr{B}(i) \tag{17}$$

$$= \frac{1}{e} \sum_{i=0}^{\infty} \frac{\chi_{\mathscr{G}}(i)}{i!} \tag{18}$$

where the last expression was proven by Kereskényi-Balogh and Nyul [50]. Pemmaraju and Skiena have given a recursive algorithm for computing the needed chromatic polynomial [52].

This means that even though there is no simple formula for computing the effect of a number of inhibition constraints on the search space size of the exhaustive search, the search space size can be computed for a particular set of inhibition constraints. This will be done for the inhibition constraints of the "Evaluation" section.

Constraint aware CSG

Grouping using CSG algorithms

The set partitioning problem can be considered as a Characteristic Function Game problem, as discussed in "Game theory: Coalition Structure Generation" section. A CFG consists of a set of component types and a value function, typically called the characteristic function, v, that assigns a real number to every group $\mathbb{G}_i \subseteq \mathbb{T}$ representing the group's value [31]. The characteristic function is therefore a mapping $\nu : 2^{|\mathbb{T}|} \mapsto \mathbb{R}$. It is normally assumed that the value of the empty group is zero, $\nu(\emptyset) = 0$. The value function (12) fulfils the criteria to be a characteristic function. We have modified the CSG IP-algorithm by Rahwan et al. [9] to handle constraints, and the resulting algorithm is presented in this section.

Constraints

According to the description in "Cloud Application Optimization" section, there are two aspects that can be seen as constraints from the CSG point of view: The first one is related to the inhibition constraints (11). Inhibition constraints are used to validate the possible groups during the initialization phase which means that they allow the algorithm to prune groups that are not legal. The second aspect is connected with the deployment feasibility of the group. Resource requirement attributes $\boldsymbol{r}(\mathbb{G}_i \mid \boldsymbol{c})$ are used during assessing feasibility of the group $\mathbb{G}_i$ to determine if there is available VM to host this group.

For some CSG algorithms, like Optimal Dynamic Programming – Integer-Partition (ODPIP) [33], the assumption that all groups are feasible is needed, because ODP-IP uses dynamic programming. In this case, during the validation of constraints in the initialization phase, the negative infinity or zero value can be used to code the infeasibility of some groups. The fact that infeasible groups are not deleted obviously influences the performance of the algorithm. It might even be possible that the fact that one group gives a negative infinity value does not exclude it from being chosen for the grouping so the constraints have to be validated when the proposed grouping is constructed. An algorithm that can work without having all groups feasible should be faster and, most importantly, correct. This is the case of IP algorithm and our modifications presented in this paper.

The introduction of infeasible groups does not influence the correctness of the IP algorithm because there is no assumption in the original IP algorithm that all groups have to be feasible. The search space representation is based on the possible integer partitions and the recursive search of dynamic group lists. The approach used in the IP algorithm is based on pruning the unpromising integer partitions according to pre-calculated statistics. Constraints simply create more unpromising integer partitions which can be pruned in the same way. Consequently, the fact that some groupings are not feasible is not the reason for not finding the best grouping provided that at least one feasible grouping exists. In the grouping of Cloud application components, at least one feasible grouping always exists because every component can be deployed on a single separate VM.

Complexity

The currently best known CSG Algorithm is the ODP-IP algorithm and it achieves the worst complexity $O(3^{|\mathbb{T}|})$. The original IP algorithm has the worst complexity $O(|\mathbb{T}|^{|\mathbb{T}|})$, but it does not need to consider all groups even if they are infeasible, which can be a reason for it to show better performance when solving the grouping of Cloud application components problem. It is hard to asses the expected complexity as it heavily depends on the number of constraints and available VMs (see "Grouping complexity with inhibition constraints" section for more details). Both ODP-IP and IP do not store a significant amount of data. There is a need to store the input data, which is estimated by the number of groups, $2^{|\mathbb{T}|}$ and all integer partitions of $|\mathbb{T}|$. The number of integer partitions grows polynomially with the size of the component set; and currently the best grouping. Therefore, the space complexity is not significant compared to the computational complexity.

Preprocessing: groups and bounds

The main recursion of the IP-algorithm is based on the Integer Partition of the number of components $|\mathbb{T}|$ to be grouped. The parts in an integer partition define the length of the groups in the grouping, i.e., how many components each group contains. The groups are therefore represented for this algorithm with two indices, $\mathbb{G}_{s,i}$, where the s is the size of the group, $s = |\mathbb{G}_{s,i}|$ and i is an index. The pre-processing steps for our modified IP algorithm are presented in Algorithm 2.

The first step of the algorithm in line 1 is to define the set $\mathbb{E}$ containing inhibition index pairs based on the inhibition constraints (11). Then the set $\mathbb{S}$ of all possible groups in the power set of the application component types are generated in line 2 retaining only the groups that do not contain two inhibited component types. Recall that this algorithm runs after the first step of the optimization process, see Fig. 1, when there is a feasible configuration, $\boldsymbol{c} \in \mathbb{F}$, assigning values to all requirement attributes for all the components in $\mathbb{T}$. The next step in line 3 is therefore to validate that there is a VM capable of hosting the group using a VM selection function on the combined resources required by the group, $\mathrm{VM}(\boldsymbol{r}(\mathbb{G}_{s,i} \mid \boldsymbol{c}))$, and filter out the groups for which there is no corresponding VM. The maximum value bound for each group size is computed next. It is also necessary to list all the integer partitions of the number $|\mathbb{T}|$, and then the

absolute best possible grouping value, V^+ is found for the partition whose sum of maximum group values by size is the largest possible. It should be noted that it may not actually be possible to achieve the upper bound grouping value V^+ because one or more of the best valued groups of different sizes may contain the same component type. These quantities will later be used to bound the search process.

Algorithm 2 Preprocessing

input : The set of application component types, $\mathbb{T}$
A feasible configuration, $\boldsymbol{c}$
output: The set of feasible groups, $\mathbb{S}$
Maximum group values by group size, $\boldsymbol{v}^+$
Grouping value upper bound, V^+
All integer partitions of $|\mathbb{T}|$, $\mathbb{P}$

// Define the exclusion set of inhibition index pairs
1 $\mathbb{E} = \{\{i,j\} \in \mathbb{N}^{|\mathbb{T}|\times|\mathbb{T}|} : I(T_i \mid \mathbb{G}) \neq I(T_j \mid \mathbb{G})\}$
// Collect the groups not containing inhibited pairs of component types
2 $\mathbb{S} = \{\mathbb{G}_{s,\alpha} \in 2^{|\mathbb{T}|}\backslash\emptyset : \langle T_i, T_j\rangle \in \mathbb{G}_{s,\alpha}, \{i,j\} \notin \mathbb{E}\}$
// Keep only the groups that can be deployed
3 $\mathbb{S} = \{\mathbb{G}_{s,\alpha} \in \mathbb{S} : \mathrm{VM}(\boldsymbol{r}(\mathbb{G}_{s,\alpha} \mid \boldsymbol{c})) \neq \emptyset\}$
// Calculate the upper bound on the value of each group size
4 **for** $s = 1$ **to** $|\mathbb{T}|$ **do**
5 $\quad v_s^+ = \max\{v(\mathbb{G}_{L,\alpha}) : \mathbb{G}_{L,\alpha} \in \mathbb{S}, L = s\}$
// List all integer partitions of $|\mathbb{T}|$
6 $\mathbb{P} = \{\boldsymbol{p} : p_i \in \{1,\ldots,|\mathbb{T}|\} \subset \mathbb{N}^+, \sum p_i = |\mathbb{T}|\}$
// Calculate the upper bound on the grouping value
7 $V^+ = \max_{\boldsymbol{p}\in\mathbb{P}} \sum_{s\in\boldsymbol{p}} v_s^+$
8 **return** $\{\mathbb{S}, \boldsymbol{v}^+, V^+, \mathbb{P}\}$

Recursion on an integer partition

An integer partition is a vector of integral numbers whose sum of elements equal the number to be partitioned,

$$\boldsymbol{p} = \{p_i \in \mathbb{N}_+ : 1 \le p_i \le |\mathbb{T}|, \sum p_i = |\mathbb{T}|, p_1 \le \cdots \le p_{|\boldsymbol{p}|}\}$$

Note that the parts of the partition is assumed to be in increasing order. The IP-algorithm constructs a grouping whose groups have sizes according to the elements of the integer partition, and with as many groups as there are parts in the integer partition, $\mathbb{G} = \{\mathbb{G}_{p_1,1}, \ldots, \mathbb{G}_{p_R,R}, \ldots, \mathbb{G}_{p_{|\boldsymbol{p}|},|\boldsymbol{p}|}\}$ with all $\mathbb{G}_{p_R,R} \in \mathbb{S}$ from Algorithm 2. The grouping is constructed via a simple recursion starting from the first taking a feasible group of size p_1, and then take a group of size p_2, and so forth.

For each recursion level $R = 1, \ldots, |\boldsymbol{p}|$ one must ensure that the group taken with size p_R has no common component types with the already taken groups. This is done by considering the reduced set of components for the grouping at recursion level R, that is $\mathbb{T}_R = \mathbb{T}\backslash\left(\bigcup_{i=1}^{R} \mathbb{G}_{p_i,i}\right)$, so by deleting the groups that contain already used component types, with $\mathbb{T}_1 = \mathbb{T}$. Let $\mathbb{C}_R(|\mathbb{T}_R|, p_R)$ be the $\binom{|\mathbb{T}_R|}{p_R}$ combinations or k-subsets $\mathbb{c}_{R,k}$ consisting of the unique integers $m_{R,k,j} \in \{1,\ldots,|\mathbb{T}_R|\} \subset \mathbb{N}^+$ taking exactly p_R integers at the time. Each index combination set is taken to be enumerated and ordered, $\mathbb{c}_{R,k} = \{m_{R,k,1}, m_{R,k,2}, \ldots, m_{R,k,p_R}\}$, with $m_{R,k,1} < m_{R,k,2} < \cdots < m_{R,k,p_R}$.

The recursion at level R will then form successively the groups $\mathbb{G}_{p_R,R} = \{T_i \in \mathbb{T}_R : i \in \mathbb{c}_{R,k}\}$ for $k = 1, \ldots, \binom{|\mathbb{T}_R|}{p_R}$. For each group it will invoke the next recursion level $R+1$ with the new set of component types to be grouped being the current set of component types less the ones taken by the current group, $\mathbb{T}_{R+1} = \mathbb{T}_R\backslash\mathbb{G}_{p_R,R}$. At the deepest recursion level, $R = |\boldsymbol{p}|$ the full grouping will be known, and the value of the full grouping can be computed as the sum of the values of its constituting groups according to Eq. (13). This basic recursion is given in Algorithm 3, which also includes some enhancements to be discussed next.

Algorithm 3 Search recursively an integer partition (IPRec)

input : The component types that are yet to be grouped, $\mathbb{T}_R$
The best grouping known, $\mathbb{G}^+$
The grouping being created by this recursion, $\tilde{\mathbb{G}}$
The integer partition of $|\mathbb{T}|$ that is being searched, $\boldsymbol{p}$
The recursion depth, R
The first pivot element of the $R-1$ combination, $m_{R-1,k,1}$
Maximum group values by group size, $\boldsymbol{v}^+$
The set of feasible groups, $\mathbb{S}$
The upper bound value of the best grouping, V^+
The bound within optimality for a solution to be acceptable, $\beta^* \ge 1$
output: The best complete grouping known, $\mathbb{G}^+$

// Establish the lower bound on the pivot elements of the index combinations at this recursion level R
1 **if** $(R > 1) \wedge (p_R = p_{R-1})$ **then** // Restrict the search if the previous part in the integer partition equals this part
2 $\quad m^- = m_{R-1,k,1}$ // The first components of $\mathbb{T}_R$ have been searched and can be skipped
3 **else**
4 $\quad m^- = 1$ // Start the search from the first component in $\mathbb{T}_R$
// Establish the upper bound on the pivot elements of the index combinations at this recursion level R
5 **if** $(R > 1) \wedge (p_R = p_{R-1})$ **then** // Restrict the search further if also the next part of the partition equals this part
6 $\quad m^+ = |\mathbb{T}_R| + 1 - 2p_R$ // Skip the last components of $\mathbb{T}_R$ as they will be searched in the recursion
7 **else**
8 $\quad m^+ = |\mathbb{T}_R|$ // Search all components in $\mathbb{T}_R$
// Cycle over the possible index combinations from the set $\{1,\ldots,|\mathbb{T}_R|\}$ of size p_R
9 **foreach** $\mathbb{c}_{R,k} \in \{\mathbb{C}_R(|\mathbb{T}_R|, p_R) : m^- \le m_{R,k,1} \le m^+\}$ **do** // Filter combinations based on the pivot index
10 $\quad \mathbb{G}_{p_R,R} = \{T_i \in \mathbb{T}_R : i \in \mathbb{c}_{R,k}\}$ // Form the group by the component type indices in $\mathbb{c}_{R,k}$
11 $\quad$ **if** $\mathbb{G}_{p_R,R} \in \mathbb{S}$ **then** // Feasible group?
12 $\quad\quad \tilde{\mathbb{G}} = \tilde{\mathbb{G}} \cup \mathbb{G}_{p_R,R}$ // Add the group to the grouping under construction
13 $\quad\quad$ **if** $\left(R = |\boldsymbol{p}|\right) \wedge \left(V(\mathbb{G}^+) < V(\tilde{\mathbb{G}})\right)$ **then** // Complete grouping with best value?
14 $\quad\quad\quad \mathbb{G}^+ = \tilde{\mathbb{G}}$ // Accept new best grouping
15 $\quad\quad\quad$ **if** $\left(V^+/V(\mathbb{G}^+) \le \beta^*\right) \vee \left(V(\mathbb{G}^+) = \sum_{s\in\boldsymbol{p}} v_s^+\right)$ **then** // Close to optimal or optimal grouping?
16 $\quad\quad\quad\quad$ **return** $\mathbb{G}^+$ // Terminate the search
17 $\quad\quad$ **else if** $V(\mathbb{G}^+) < \left[\sum_{\mathbb{G}_{s,i}\in\tilde{\mathbb{G}}} v(\mathbb{G}_{s,i})\right] + \left[\sum_{s\in\{p_{r+1},\ldots,p_{|\boldsymbol{p}|}\}} v_s^+\right]$ **then** // Potential for a better grouping?
18 $\quad\quad\quad \mathbb{G}^+ = \mathtt{IPRec}(\mathbb{T}_R\backslash\mathbb{G}_{p_R,R}, \mathbb{G}^+, \tilde{\mathbb{G}}, \boldsymbol{p}, R+1, m_{R,k,1}, \boldsymbol{v}^+, \mathbb{S}, V^+, \beta^*)$
// Descend recursion tree for more groups
19 **return** $\mathbb{G}^+$

Aborting the recursion

There are two main reasons for terminating early the recursion. One is related to the feasibility of the found

group with respect to the inhibition constraints and the group's combined resource requirements. This is handled in the line 11 of Algorithm 3.

The second termination criteria is related to the value of the grouping. The grouping is complete if this recursion level adds the last group of the grouping, and this new grouping has a higher value than the currently best group. This condition is tested in line 13. Still, for the grouping to be optimal for the integer partition $\boldsymbol{p}$ it must be within a certain distance from the theoretical optimal value, or it must be the best possible grouping for this integer partition. This is tested in line 15. If the grouping is not accepted as optimal, it will remain as the baseline group when searching further index combinations, and can eventually be returned as the best grouping when all index combinations have been searched.

Given the value of the groups in the partially constructed grouping, the new groups to add in the following recursion steps should allow the completed grouping to have a value higher than the currently best grouping. This can be assessed by the maximum group values $\boldsymbol{v}^+$ computed in Algorithm 2. Since the size of the groups to be added by further recursion is given by the integer partition, $\boldsymbol{p}$, the corresponding maximum group values can be taken from $\boldsymbol{v}^+$, then the sum of these maximum values is the maximum value that can be added to the value of the current grouping. Thus, if the grouping value plus the maximum value that can be added is still less than the value of the best grouping, there is no need to do further recursions. This is tested in line 17.

The fact that the number of groupings possible grows like the Bell number, $\mathscr{B}(|\mathbb{T}|)$ of (3), makes it impossible to search all groupings if the number of component types is large. Sandholm et al. proved that the ratio of the best solution to the optimal solution is bounded, so $V^+/V(\mathbb{G}^+) \leq \beta^*$, where $\beta^* \geq 1$ is the bound on this ratio depending on the number of groupings searched [53]. This user defined bound is used in Algorithm 3 line 15 to abort the recursion if the found grouping is acceptably close to the optimal value. It should be noted that setting β^* too close to unity will inevitably cause all the groupings to be evaluated and the returned $\mathbb{G}^+$ to be the optimal grouping.

Equal integer partition parts

Rahwan et al. realized that the recursive search for the best grouping based on the integer partition can be improved for the case when there are multiple equal parts in the integer partition [9]. Assuming that the index combinations are generated in lexicographical order, the first combinations $\mathbb{c}_{R,\alpha}$ for $\alpha = 1, \ldots$ will all have one as the first element, $m_{R,\alpha,1} = 1$. These combinations will then be followed by another block of combinations having two as the first element, $m_{R,\beta,1} = 2$, and so forth. For each combination, a complete grouping may be formed by the following recursion if the groups are feasible and give a better value for the grouping. This means that if $p_{R+1} = p_R$, then one will generate groups of the same size as in the previous recursion level, and if $m_{R,\beta,1} = 2$ one knows that all groups of size p_R involving the first element of $\mathbb{T}_R$ have already been constructed and searched. One also knows that since $\mathbb{T}_{R+1} \backslash \mathbb{G}_{p_R,R}$ the set of objects to be grouped at recursion level $R+1$ will contain the first element of $\mathbb{T}_R$ since $m_{R,\beta,1} = 2$ so the grouping at recursion level R can start from the second element of $\mathbb{T}_R$ since the group $\mathbb{G}_{p_{R+1},R+1} = \{(T_R)_2, (T_R)_1, \ldots\}$ should never be generated since it is equivalent to the already tried group $\mathbb{G}_{p_R,R} = \{(T_R)_1, (T_R)_2, \ldots\}$ assuming that the rest of the components in the two groups are the same. This means that one can, at recursion level $R+1$, ignore the first element of $\mathbb{T}_{R+1}$ if $m_{R,\beta,1} = 2$. It is therefore sufficient to consider only the last elements of $\mathbb{T}_{R+1}$, and by extending the above argument one may establish the lower bound on the first element of the index combination sets searched at the next recursion level, and Rawhan et al. proved the following relation ([9], Appendix E)

$$m_{R+1,\beta,1} \geq m_{R,\alpha,1} \tag{19}$$

The notation is adopted to the notation of this paper and the combination index k was generalized to α and β to indicate that this index is only valid for each recursion level and the result is independent of the rank of the combinations at each level.

More surprisingly, it is also possible to establish an upper bound on the pivot element, $m_{R,k,1}$ of an index combination. The number of component types grouped at recursion level R is $|\mathbb{T}_R|$, and all the groups at this level has length p_R. This means that at the next recursion level, $R+1$, there are $|\mathbb{T}_{R+1}| = |\mathbb{T}_R| - p_R$ component types to be grouped. The component types in a group are supposed to be in lexicographical order, and so the last group of p_{R+1} component types will start at component type index $|\mathbb{T}_{R+1}| - (p_{R+1} - 1)$ in the ordered set $\mathbb{T}_{R+1}$. This means that the largest pivot element of any index combination at recursion level $R+1$ is $\max m_{R+1,\beta,1} = |\mathbb{T}_{R+1}| - (p_{R+1} - 1)$. Note that by assumption $p_{R+1} = p_R$, and so by (19)

$$\begin{aligned}
\max m_{R+1,\beta,1} &= |\mathbb{T}_{R+1}| - (p_{R+1} - 1) \\
&= |\mathbb{T}_{R+1}| - (p_R - 1) \\
&= |\mathbb{T}_R| - p_R - (p_R - 1) \\
&= |\mathbb{T}_R| + 1 - 2p_R \geq m_{R,\alpha,1} \geq m_{R-1,k,1}
\end{aligned} \tag{20}$$

The upper and lower bound on the pivot element of the index combination is defined in Algorithm 3 in the lines 1-8.

Constraint aware IP-algorithm

Given that the recursion will only be continued in Algorithm 3 line 17 if the following groups have the potential to make the value of the grouping larger than the currently best group, it will be beneficial to search the groupings whose group sizes are given by an integer partition of $|\mathbb{T}|$, according to the order of the largest theoretical grouping value based on the upper bounds on each of the group values calculated in Algorithm 2 line 5. This is the main loop of the Algorithm 4 in line 3 scanning the sorted set $\mathbb{P}$ of all possible integer partitions of $|\mathbb{T}|$. The groups whose lengths are given by an integer partition will be recursively searched only if the partition has the potential to return a larger grouping value than the currently best grouping.

Algorithm 4 Constraint aware integer partition search

```
input  : The set of application component types, 𝕋
         A feasible configuration, c
         The bound for optimality of a solution, β* ≥ 1
output: The optimal grouping, 𝔾*
   // Compute the feasible groups, bounds, and the integer partitions
1  {𝕊, v⁺, V⁺, ℙ} = Preprocessing(𝕋, c)
2  𝔾⁺ = ∅
   // Search the groupings of the integer partitions ℙ of the number of component types |𝕋| in promising order
3  foreach p ∈ { p_i ∈ ℙ : Σ_{s∈p_i} v_s⁺ ≥ Σ_{s∈p_{i+1}} v_s⁺ } do
4     if V(𝔾⁺) < Σ_{s∈p} v_s⁺ then                         // Promising partition?
5        𝔾̃ = IPRec(𝕋, 𝔾⁺, p, 1, 1, v⁺, 𝕊, V⁺, β*)
6        if V(𝔾⁺) < V(𝔾̃) then
            // Record the new best grouping
7           𝔾⁺ = 𝔾̃
            // Terminate search if value is close enough to optimal
8           if V⁺/V(𝔾⁺) ≤ β* then
9              return 𝔾⁺
10 return 𝔾⁺
```

Evaluation

The optimization algorithm must be able to handle constraints and generate valid groupings that satisfy these constraints. Efficient handling of constraints can significantly reduce the search space, making it easier to find an optimal solution in a reasonable time. Therefore, the ability to handle grouping constraints is crucial and it is the main focus of this evaluation.

Methodology and setting

The experiments were conducted on various sizes of applications in terms of the number of component types and requirements attributes representing different configurations $\boldsymbol{c}$. All tested configurations are presented in Table 7. The experiments were conducted on two sets of available Node Candidates presented in Tables 2 and Table 6. The details of the setup and the summary of the results are presented in Table 8. All tests were performed assuming that sharing cores and ram sharing is not allowed to make the problem more difficult. Furthermore, various numbers of constraints were used to

Table 6 Available Virtual Machines (VMs) (bigger set)

VM Name	CPU	RAM	Price
VM_1	1	1	1
VM_2	1	2	2
VM_3	2	2	3
VM_4	2	4	4
VM_5	4	4	6
VM_6	4	8	8
VM_7	8	8	12
VM_8	8	16	16
VM_9	16	16	24
VM_{10}	16	32	32
VM_{11}	32	32	48
VM_{12}	32	64	64

assess the performance, starting from zero constraints and up to the maximum possible number of constraints, calculated as $(|\mathbb{T}| \cdot (|\mathbb{T}| - 1))/2$. There are a maximum of 20 possible constraints for an application with seven components, 44 for an application with ten components, and 104 for an application with fifteen component types.

Three algorithms were used to solve the grouping problem:

1. the modified IP algorithm presented in "Constraint aware CSG" section as Algorithm 4,
2. the state-of-the-art Optimal Dynamic Programming – Integer-Partition (ODP-IP) [33] algorithm,
3. the exhaustive search of Algorithm 1 treated as the baseline and a complexity marker.

The focus of the evaluation was on assessing the performance in terms of handling constraints and the unavailability of certain groups. To assess the performance and account for the inhibition constraints, the algorithms were tested on different numbers of randomly generated inhibition constraints. To ensure a broader scope of cases, 100 sets of randomly generated constraints for each possible number of constraints were evaluated for experiments with seven and ten components, and 10 sets of randomly generated constraints were evaluated for the experiments with a 15-component application. The constraints were generated by first creating a list of all possible pairs of numbers from zero to $|\mathbb{T}|$, then shuffling the pairs in a random order using Java method *Collections.shuffle*[12], and after that picking the first k pairs, where k is the desired number of constraints. This approach with various sets of randomly generated constraints allows us to obtain

[12] https://docs.oracle.com/javase/6/docs/api/java/util/Collections.html

Table 7 Tested configurations $\boldsymbol{c}$ with various component requirements for applications with seven ($\boldsymbol{c}_1$), ten ($\boldsymbol{c}_2$ and $\boldsymbol{c}_3$), and fifteen ($\boldsymbol{c}_4$) component types

	7 components: $\boldsymbol{c}_1$		10 components: $\boldsymbol{c}_2$		10 components: $\boldsymbol{c}_3$		15 components: $\boldsymbol{c}_4$	
Component Name	**CPU**	**RAM**	**CPU**	**RAM**	**CPU**	**RAM**	**CPU**	**RAM**
A	1	4	6	10	1	1	6	10
B	2	2	2	4	1	2	2	4
C	2	8	1	3	1	2	1	3
D	3	6	1	2	1	3	1	2
E	3	12	3	2	2	2	3	2
F	4	8	7	18	2	3	7	18
G	6	12	1	2	2	4	1	2
H	-	-	12	20	3	3	12	20
I	-	-	6	14	3	4	6	14
J	-	-	2	2	3	6	2	2
K	-	-	-	-	-	-	3	18
L	-	-	-	-	-	-	1	1
M	-	-	-	-	-	-	12	24
N	-	-	-	-	-	-	16	14
O	-	-	-	-	-	-	2	8

a better understanding of the impact of inhibition constraints on the available groups and the resulting optimal grouping. Therefore, the results presented in this section are based on a comprehensive evaluation of multiple constraint sets, rather than a single set, which increases the confidence in the findings.

To account for the second factor that influences the number of feasible groups, the algorithms were tested on different application configurations $\boldsymbol{c}$, presented in Table 7. Figure 2 shows how the inhibition constraints limit the available groups. It presents a box-and-whisker plot for all possible constraints where each vertical line is for a given number of constraints, starting from zero constraints. For zero constraints, the only infeasibility comes from the limited VM offers. For more constraints, the number of available groups decreases eventually down to $|\mathbb{T}|$ groups so only singletons are available for the maximum possible number of constraints.

Figure 2a presents the available groups for the same configuration $\boldsymbol{c}_1$ of the seven-component application calculated for a smaller set of available VMs presented in Table 2 (red boxes), and a larger set of available VMs presented in Table 6 (orange boxes). The difference in the feasibility of the groups comes only from the available Node Candidates. For the case with zero constraints, all groups are feasible for the larger set of Node Candidates and only 72% of the groups are feasible for the smaller set of Node Candidates. However, when some constraints are introduced, the available Node Candidates have a much lower impact on the number of available groups.

Figure 2b presents the available groups for two configurations of the ten-component application, $\boldsymbol{c}_2, \boldsymbol{c}_3$, and the set of Node Candidates from Table 2. For 10 components, there are $2^{10} = 1024$ groups if there is always a VM capable of hosting the group. The first configuration, $\boldsymbol{c}_2$, marked by red boxes, involved various resource requirement attributes to imitate bigger and smaller components and it aims to be more realistic. For this configuration, for zero constraints, there are around 320 available groups. It means that almost two-thirds of the groups are not available only because of the VM resource limitation and no inhibition constraints. The second configuration, $\boldsymbol{c}_3$, marked by orange boxes, involved smaller components. It represents a case when resource constraints on available VMs are not an important factor and almost all groups are available when no inhibition constraints are applied. This example highlights the impact of resource requirements attributes. Also, one can note that only one inhibition constraint reduces the number of available groups by 30%.

Best grouping value

As for the value function, we used (12) that was presented in "Grouping approach" section. The available groups and hence the best grouping are influenced by the resource attribute requirements, available offers, and inhibition constraints. The available offers and resource attribute requirements do not change for the first experiment case, so only the introduction of inhibition constraints limits the available groups and changes the optimal grouping.

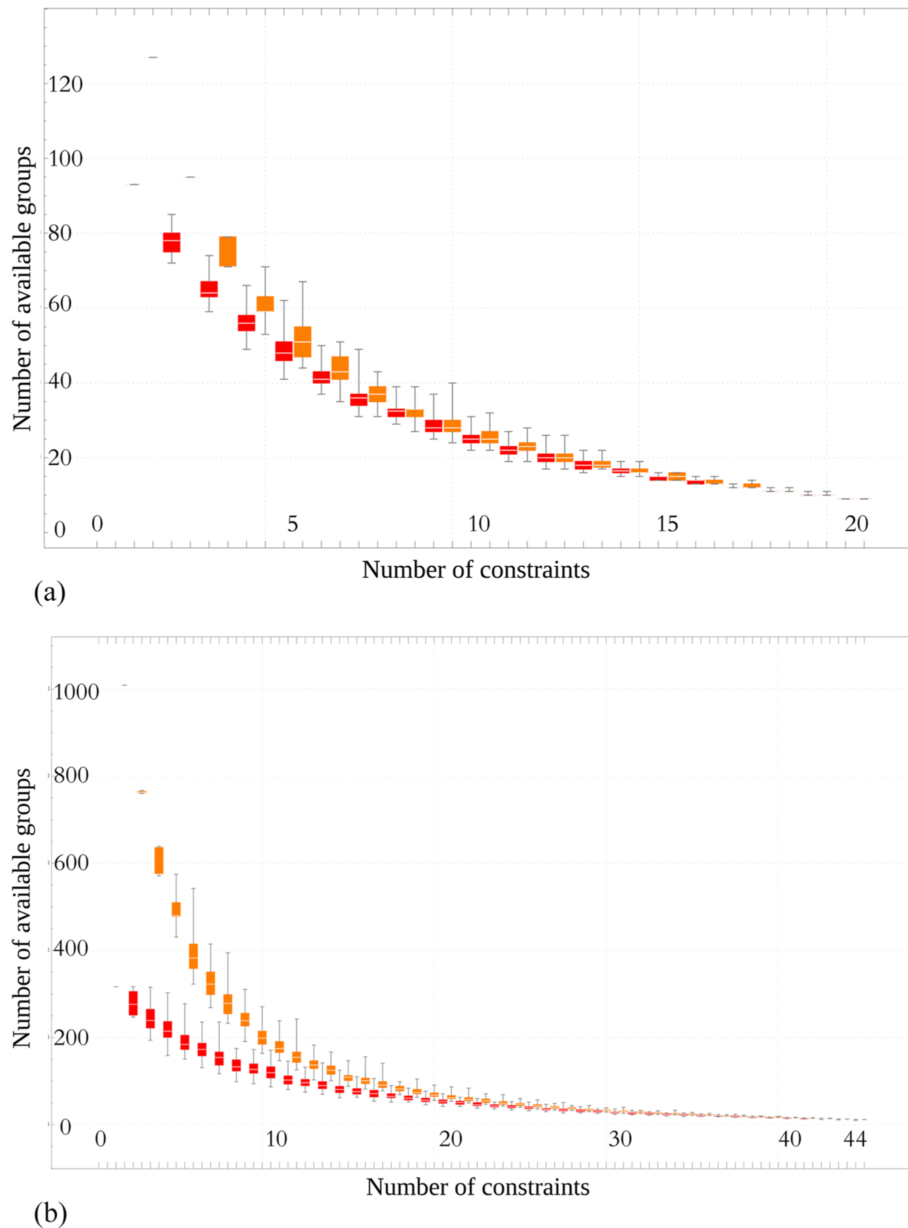

Fig. 2 The total number of available groups for the various numbers of randomly generated constraints for two sets of available Node Candidates for a seven-component application (**a**) for the same configuration $\boldsymbol{c}_1$ of a seven-component application for a smaller (red boxes) and larger (orange boxes) set of available Node Candidates. The infeasibility of the groups comes only from the limited offers, and for the smaller set of available Node Candidates, the number of feasible groups is lower; and for two different configurations of a ten-component application (**b**) for two configurations of a ten-component application: configuration $\boldsymbol{c}_2$ (red) and $\boldsymbol{c}_3$ (orange). The difference in the infeasibility of the groups comes from the resource requirement attributes, which are smaller for $\boldsymbol{c}_3$, so there are more feasible groups for this configuration. Each vertical line is for a given number of constraints, starting from zero up to the maximum number of possible constraints

Figure 3 shows the grouping value, calculated as (13), for the configuration $\boldsymbol{c}_2$ of a ten-component application, for cases with zero inhibition constraints up to the maximum available number of inhibition constraints. It is a box-and-whisker plot where one box represents all cases with the same number of constraints. For zero constraints, there are plenty of available groupings, resulting in the highest possible grouping value of 238. As the number of constraints increased, the grouping value decreased. This is a natural consequence of some groups becoming infeasible, leading to limited potential cost savings. It should be noted that the optimal grouping without inhibition

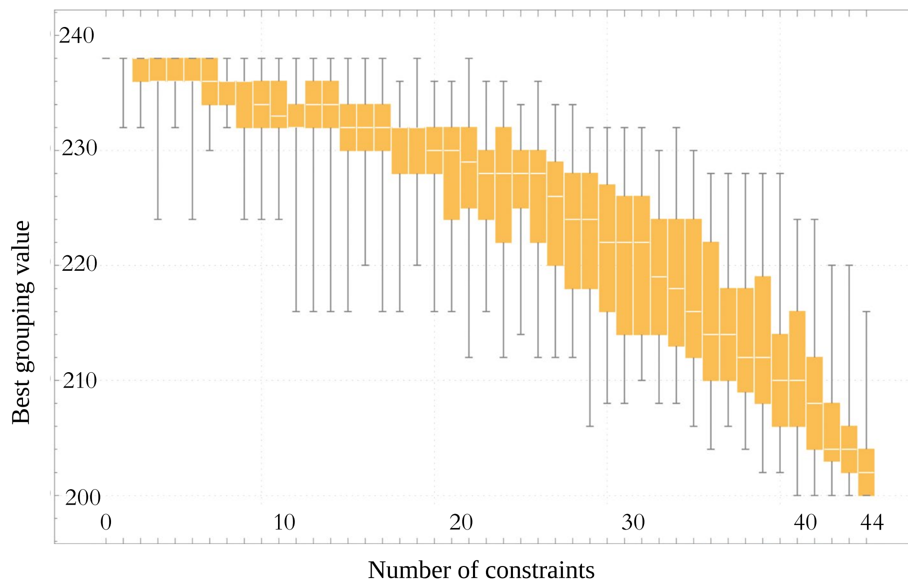

Fig. 3 The utility value of the best grouping for the configuration $\boldsymbol{c}_2$, for the randomly generated constraints from zero to 44. The more constraints are introduced, the value of the best grouping is decreased

constraints gives 19% higher value than the deployment of all components into individual VMs, which highlights the benefits of grouping. The improvement of the utility achieved by the optimal grouping is presented in Table 8 in the last column of the summary of experiments results.

Experiments

For all algorithms, different environments were used to perform experiments. In particular:

- the modified IP algorithm was implemented as a part of the MORPHEMIC project[13],
- ODP-IP algorithm [33] with the implementation provided by authors on Github[14]. The algorithm expects to have all groups available so we encoded infeasible groups as groups with *zero* value,
- and exhaustive search algorithm presented in "Exhaustive search" section which was run on the Mathematica implementation provided by the authors of this paper.

To objectively measure the algorithms' performance across various implementation environments, a specific metric was utilized: the number of evaluated grouping propositions. It is important to note that this metric is applicable even when different implementation environments are used. If an algorithm needs to enumerate all possible groupings, the value of the counter should be equivalent to the Bell number. This is the case for the exhaustive search algorithm with no VM limitations applied.

Due to the space limit, we present only two Figures with detailed results, and we summarize the overall results in Table 8. We performed seven experiments on various resource requirements, number of components, sets of available VMs, and all possible numbers of inhibition constraints. In each experiment, there were 10 or 100 randomly generated sets of constraints for each possible number of constraints, which resulted in a range from 14000 to 45000 test cases conducted for one experiment. We note that both CSG algorithms were able to find the optimal grouping for all cases. The column *Best IP* in Table 8 was calculated by counting the number of groupings constructed for each case by all algorithms, then taking the mean and comparing the means of all three algorithms. The percentage indicates for how many cases the modified IP algorithm constructed the lowest number of groupings from all three algorithms. It can be seen that the modified IP algorithm constructed a lower number of groupings at least for 58% cases, but for experiments on a seven-component application, the modified IP algorithm was always outperforming the state-of-the-art ODP-IP algorithm and the exhaustive search algorithm.

Values of the *IP speedup* column was calculated by taking means of the numbers of constructed groupings for each number of constraints and taking the ratio between the modified IP algorithm and ODP-IP algorithm. For all experiments, the modified IP algorithm on average created at least 9.0 fewer groupings, but for the biggest

[13] https://gitlab.ow2.org/melodic/grouping/-/tree/main
[14] https://github.com/trahwan/ODP-IP

Table 8 The summary of conducted experiments. There were seven experiments performed on various configurations and available VMs. Results present for how many cases the modified IP algorithm performed better than the state-of-the-art ODP-IP algorithm (Best IP), and how much faster the modified IP algorithm was on average (IP speedup). The last column (Utility improvement) presents the benefits of grouping as the percentage difference between the best-achieved utility compared to the situation where each component is hosted on a separate VM

Experiment setup					Results		
ID	$\lvert\mathbb{T}\rvert$	c	VMs Table	Repetitions	Best IP	IP speedup	Utility improvement
1	7	c_1	2	100	100%	11.58	7.5%
2	7	c_1	6	100	100%	9.00	2.9%
3	10	c_2	2	100	78%	74.18	19%
4	10	c_2	6	100	71%	27.84	6.5%
5	10	c_3	2	100	73%	24.24	2.9%
6	10	c_3	6	100	58%	17.63	0.8%
7	15	c_4	2	10	64%	3641.22	2.9%

experiment, the modified IP algorithm was on average a couple of thousands of times faster than ODP-IP algorithm. Finally, the *Utility improvement* column presents the benefits of the grouping itself. For all experiments, the optimal grouping gives higher utility than the deployment of each component on a single VM. This improvement, calculated in percentage, is presented in the last column of Table 8.

Figure 4 shows the results of the experiment (3) and Fig. 5 shows the results of the experiment (5), where two different configurations of a ten-component application were tested. Figures show the number of groupings checked during the search for the best grouping for all evaluated algorithms and problems with from zero to 44 randomly generated constraints. It is a box and whisker plot where the blue boxes are results of ODP-IP algorithm, red boxes are results of the modified IP algorithm, and green results are results of the exhaustive search algorithm. It should be noted that the number of groupings checked is presented on a logarithmic scale. It is important to note that the exhaustive search algorithm evaluates significantly more groupings than both the modified IP and ODP-IP for cases with the lower number of constraints, and for the fifteen-component application, obtaining the result took prohibitively long. This leads to the conclusion that presenting the grouping problem as

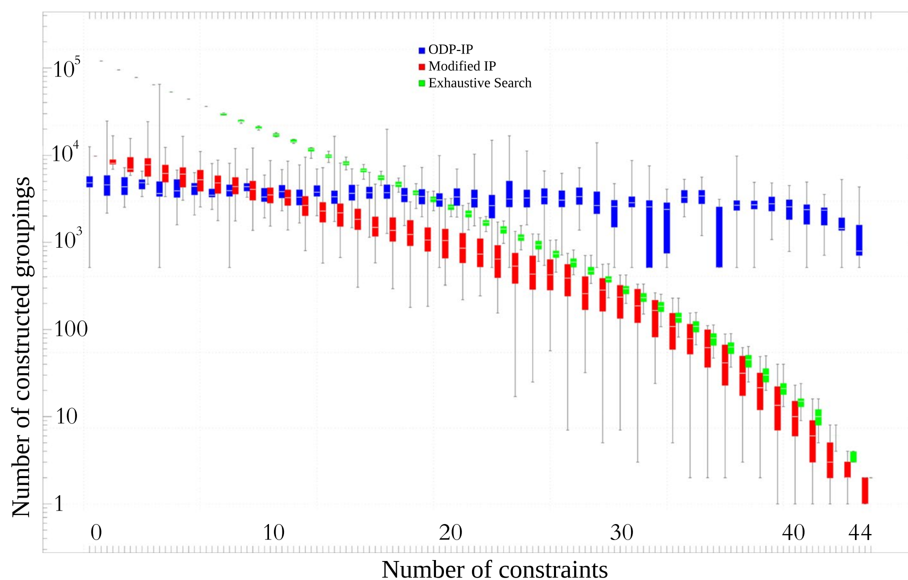

Fig. 4 The number of groupings evaluated by ODP-IP, modified IP, and the exhaustive search algorithms for the experiment (3), where there are many infeasible groups due to limited VM offers, with the increasing number of randomly generated grouping constraints, starting from zero up to 44 constraints. The number of groupings created is presented on a logarithmic scale

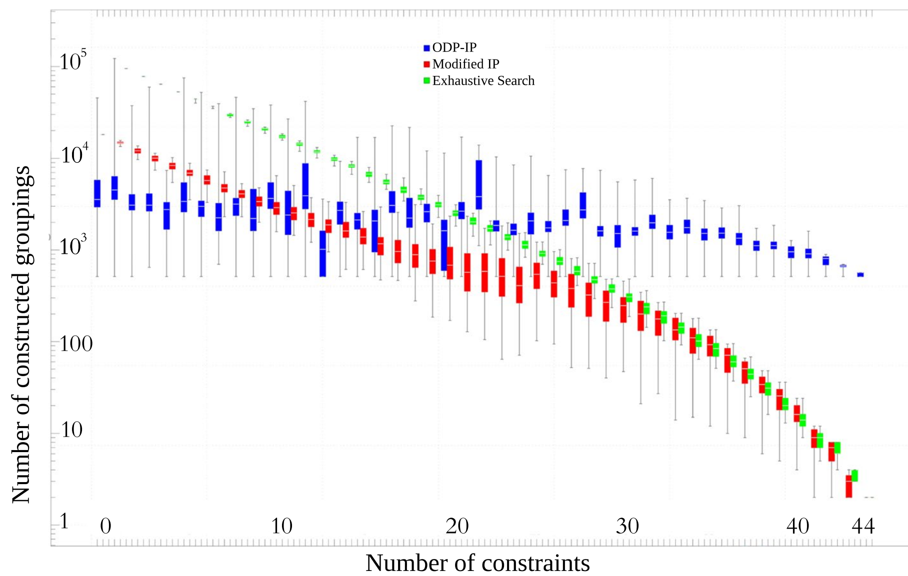

Fig. 5 The number of groupings evaluated by ODP-IP, modified IP, and the exhaustive search algorithms for the experiment (5), where almost all groups are feasible, with the increasing number of randomly generated grouping constraints, starting from zero up to 44 constraints. The number of groupings created is presented on a logarithmic scale

a CSG problem and using the best algorithms from this field can yield good results and make the grouping problem solvable for a larger number of components. Also, the ODP-IP algorithm has a bigger variance for larger problems in terms of the number of available groups, like for the experiment presented in Fig. 5. It makes it less predictable in terms of the final complexity.

The results indicate that the ODP-IP algorithm may be faster for cases with a small number of constraints. It means that it may be better to use ODP-IP for almost unconstrained cases, which confirms the fact that it is a state-of-the-art algorithm. However, for cases with 10 or more constraints, the modified IP evaluates on average a lower number of groupings, making it more efficient as the number of constraints increases. For the extreme case, when each component has to be placed on a separate VM, the modified IP algorithm evaluates only this one possible grouping while ODP-IP algorithm evaluates more than 500 possible groupings, and for fifteen-component applications, ODP-IP algorithm evaluates more than 16000 possible groupings while only one grouping is actually feasible.

Discussion

The results of experiments presented in "Evaluation" section lead us to the conclusion that the proposed approach for solving the constrained grouping problem is efficient and is able to find the best Cloud application component placement. Furthermore, the presented modifications of IP algorithm lead to better performance of this algorithm comparing to the ODP-IP algorithm. Even though the ODP-IP algorithm can be up to eight times more efficient for cases with a small number of constraints and hence many available groups, the modified IP algorithm constructs fewer groupings for the majority of the cases. One can note that for more complex experiments, the superiority of the modified IP algorithm is lower, but still the modified IP algorithm evaluated up to 3000 less groupings than the ODP-IP algorithm. Another interesting result is the observation that even one constraint can reduce the number of available groups by 30%. This observation can suggest to the application owner an easy way to limit the complexity of optimization of the Cloud application.

It is important to note the novel value function for cost minimization introduced in this paper. It is not trivial how to define a value function over a group that is a characteristic function, and that will consider cost minimization and for which the value for the grouping should be maximized. One can think about the simplest function, the negated price $-P(\mathbb{G}_i)$. However, not all algorithms allow for having negative group values. Many algorithms can handle negative values in the theory, but the existing implementations are assumed to work only on positive values. Therefore, the introduced value function can be seen as a significant contribution. We did not consider the performance or latency consequences of the grouping since it heavily depends on the application characteristics

and communication network while cost minimization is the common goal of all Cloud deployments.

One must be aware that the complexity of the constrained optimization problem is NP-complete in general so adding the additional step of solving the grouping problem just increases the overall complexity. However, the two-step optimization approach presented in this paper keeps the overall complexity small enough to be able to solve the problems for a reasonable number of components. The experiments presented in this paper were conducted for applications with seven, ten, and fifteen component types, but we estimate that it should be possible to solve this problem for an application with up to fifty component types, which is more than the number of component types for a typical Cloud application[15].

This paper has not considered the outer optimization to assign the optimal resource requirements for the application components. The presented approach expects that the resource requirements attributes are given as input, and that they are correct and accurate. In this work, authors designed the process of finding the best VMs for Cloud application deployment to be integrated with the MORPHEMIC platform. MORPHEMIC aims on finding the best resource configuration for Cloud application components under varying *execution context* [3], so this limitation is mitigated by this integration.

Conclusion and future work

This paper has proposed a novel approach for solving the constrained grouping problem for Cloud application components deployment and optimization. The approach involved a two-step optimization process, where the first step involved finding the best resource requirements for the application component types, and the second step involved finding the best possible set of VMs and grouping the components to minimize the overall cost. This approach is the first representation of the Cloud application components grouping problem as CSG problem.

Furthermore, a novel cost saving value function was introduced. It is used for representing the cost minimization goal using only positive values that can be summed up to calculate the overall grouping value (13). This function enables the cost benefits from the grouping of Cloud application components, and it does not contradict a utility function used to find the application components' optimal resource requirements. The value function satisfies the criteria of being a characteristic function, allowing the CSG algorithms to solve the grouping problem. Finally, this paper developed a modified IP algorithm that is able to handle collocation constraints. The modified IP algorithm was evaluated and it outperformed the state-of-the-art algorithm by dozens of times for most of the cases. The proposed approach was able to find the best component grouping efficiently, Overall, these results demonstrate the effectiveness of the modified IP algorithm in solving the Cloud application optimization problem, particularly for cases with a large number of inhibition constraints where it can be even thousand times more efficient in terms of constructed groupings than state-of-the-art ODP-IP algorithm.

The presented approach is being implemented as a part of the MORPHEMIC Cloud application management and optimization platform, which will allow for further evaluation and experimentation with real-world business applications, which may help assessing the signifance of limitations related to the complexity of the grouping problem.

The limited sets of VMs used in the experiments serve the purpose of assessing the usability of this approach to solve the grouping problems in the Cloud continuum, which is one of the open resource optimization problems, as it was stated by Bittencourt et al. [7]. Authors will be integrating the presented approach with the NebulOus platform[16], which is a meta-operating system for applications deployed in the Cloud continuum [54]. Further experiments are planned to be conducted after this integration, involving users of the platform.

Furthermore, there are many ideas about how to progress the research in the area of grouping. For instance, a constrained CSG problem can possibly be seen as a combinatorial auction, where it is possible to bid on the combinations of items, i.e., the groups $\mathbb{G}_i$, to find the best package, $\mathbb{G}$. A combinatorial auction can be solved in polynomial time in terms of the number of feasible groups [55], and so if the number of feasible component groups is smaller than around 10% of all possible groups, the combinatorial auction algorithms may have better complexity than CSG algorithms. It might be possible to develop a hybrid meta-algorithm that will decide which algorithm should be used to solve a particular grouping problem: ODP-IP, modified IP, or possibly also a combinatorial auction algorithm.

To conclude, the proposed approach is promising for optimized constrained Cloud application management, and future research and development in this area may

[15] https://www.jrebel.com/blog/2021-microservices-developer-report

[16] https://nebulouscloud.eu/

further improve the efficiency and effectiveness of Cloud applications.

Appendix A Summary of the notation

Table 9 provides the summary of the notation used in the paper.

Table 9 The summary of the notation used in the paper, including the notation used in the modified IP algorithm

Symbol	Description
$\mathbb{T}$	The set of application components.
$\mathbb{T}_i$	A component.
$\lvert\mathbb{T}\rvert$	The number of components.
$\mathbb{G}_i$	A group.
$\mathbb{G}$	A grouping.
$\mathbb{G}^*$	An optimal grouping.
$\mathbb{A}_T$	The set of component's T resource requirements attributes.
$a_{T,i} \in \mathbb{A}_T$	A resource requirements attribute for component T.
$\mathbb{V}_{T,i}$	A domain of attribute $a_{T,i}$.
$\mathbb{V}_T$	The variability space for component T.
$\mathbb{V}$	The configuration search space.
$\boldsymbol{c} \in \mathbb{V}$	An application configuration.
$\boldsymbol{c}^* \in \mathbb{V}$	The optimal configuration.
t_k	The point in time.
$\boldsymbol{\theta}(t_k)$	An execution context vector.
$\mathbb{F}\left(\boldsymbol{\theta}(t_k), \hat{\boldsymbol{\psi}}[\boldsymbol{c}, \boldsymbol{\theta}(t_k)]\right)$	A feasible space.
$U\left(\boldsymbol{c}, \boldsymbol{\psi}(\boldsymbol{c}, \boldsymbol{\theta}(t_k)) \middle\vert\, \boldsymbol{\theta}(t_k)\right)$	The utility function.
g_k	A group variable (exhaustive search algorithm).
$I(T\vert\mathbb{G})$	An index function.
$\mathbb{E}$	The set of exclusions.
$r(\mathbb{G}_i\vert\boldsymbol{c})$	A resource function.
$VM(r(\mathbb{G}_i\vert\boldsymbol{c}))$	The VM capable of hosting the group $\mathbb{G}_i$.
$P(\mathbb{G})$	The price of the grouping $\mathbb{G}$.
$P(\mathbb{G}_i)$	The price of the cheapest VM capable of hosting the group $\mathbb{G}_i$.
p^+	The cost of the most expensive available VM.
$\lvert\mathbb{G}_i\rvert$	The cardinality of $\mathbb{G}_i$.
$v(\mathbb{G}_i)$	The value of $\mathbb{G}_i$.
$\mathbb{S}$	The set of groups.
$V(\mathbb{G})$	The value of $\mathbb{G}$.
V^*	The upper bound on $V(\mathbb{G}^*)$.
V^-	The lower bound on $V(\mathbb{G}^*)$.
$\mathbb{G}^+$	The best grouping found so far.
β^*	The bound within which any solution is acceptable.
$\boldsymbol{v}^+$	A vector of maximum group values by group size.
$\bar{\boldsymbol{v}}$	A vector of average group values by group size.
$\boldsymbol{p}$	An integer partition of $\lvert\mathbb{T}\rvert$.
R	A recursion depth of search in modified IP algorithm.
$\mathbb{T}_R$	The component types that are yet to be grouped.
$m_{R-1,k,1}$	The first pivot element of $R-1$ combination.
$\mathbb{C}_R(\lvert\mathbb{T}_R\rvert, p_R)$	The binom combinations of the unique integers $m_{R,k,j} \in \{1, \ldots, \lvert\mathbb{T}_R\rvert\} \subset \mathbb{N}^+$.

Acknowledgements
This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871643 MORPHEMIC *Modelling and Orchestrating heterogeneous Resources and Polymorphic applications for Holistic Execution and adaptation of Models In the Cloud* and from the EU's Horizon research and innovation programme under grant agreement No. 101070516 NebulOuS project.

Authors' contributions
The main manuscript was a collaborative joint effort between MR and GH. MR took overall responsibility for the paper and the implementation and the experiments. GH supervised the work, made a specific contribution to "Exhaustive search" section, and the presentation of algorithms, and the description of the modified IP algorithm. Both authors reviewed the manuscript.

Funding
Open access funding provided by University of Oslo (incl Oslo University Hospital) This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871643 MORPHEMIC *Modelling and Orchestrating heterogeneous Resources and Polymorphic applications for Holistic Execution and adaptation of Models In the Cloud* and from the EU's Horizon research and innovation programme under grant agreement No. 101070516 NebulOuS project.

Availability of data and materials
The implementation of the modified Integer-Partition (IP) algorithm and the data used for experiments can be found in the GitLab repository (https://gitlab.ow2.org/melodic/grouping/-/tree/main). For additional data supporting the results presented in this publication, interested readers may request it from the corresponding author.

Declarations

Ethics approval and consent to participate
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 23 June 2023 Accepted: 12 April 2024
Published online: 10 May 2024

References

1. Marinescu DC (2022) Cloud Computing: Theory and Practice. Morgan Kaufmann, p 674. Google-Books-ID: XOBWEAAAQBAJ. ISBN: 978-0-323-91047-7
2. Apostolou D, Verginadis Y, Mentzas G (2021) In the fog: Application deployment for the cloud continuum. In: 2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA), IEEE, pp 1–7
3. Horn G, Skrzypek P (2018) MELODIC: Utility based cross cloud deployment optimisation. In: Proceedings of the 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA), IEEE Computer Society, Krakow, pp 360–367. https://doi.org/10.1109/WAINA.2018.00112
4. Kephart JO, Das R (2007) Achieving self-management via utility functions. IEEE Internet Comput 11(1):40–48. https://doi.org/10.1109/MIC.2007.2
5. Onozaki T (2018) Nonlinearity, Bounded Rationality, and Heterogeneity. Springer, Tokyo. https://doi.org/10.1007/978-4-431-54971-0
6. Ahmad Z, Jehangiri AI, Ala'anzy MA, Othman M, Latip R, Zaman SKU, Umar AI (2021) Scientific workflows management and scheduling in cloud computing: Taxonomy, prospects, and challenges. IEEE Access 9:53491–53508. https://doi.org/10.1109/ACCESS.2021.3070785. IEEE Access, ISSN: 2169-3536
7. Bittencourt L, Immich R, Sakellariou R, Fonseca N, Madeira E, Curado M, Villas L, DaSilva L, Lee C, Rana O (2018) The internet of things, fog and cloud continuum: Integration and challenges. Internet Things 3-4:134-155. https://doi.org/10.1016/j.iot.2018.09.005. ISSN: 25426605.
8. Manvi SS, Krishna Shyam G (2014) Resource management for infrastructure as a service (IaaS) in cloud computing: a survey. J Netw Comput Appl 41:424–440. https://doi.org/10.1016/j.jnca.2013.10.004. ISSN: 1084-8045
9. Rahwan T, Ramchurn SD, Jennings NR, Giovannucci A (2009) An anytime algorithm for optimal coalition structure generation.J Artif Intell Res 34:521–567. https://doi.org/10.1613/jair.2695. ISSN: 1076-9757
10. Stanley RP (2011) Enumerative Combinatorics, Cambridge Studies in Advanced Mathematics, vol 1, 2nd edn. Cambridge University Press, Cambridge, p 640. https://doi.org/10.1017/CBO9781139058520. ISBN: 978-1-107-01542-5
11. Bell ET (1938) The iterated exponential integers. Ann Math 39(3):539–557. https://doi.org/10.2307/1968633. ISSN: 0003-486X
12. Rota G-C (1964) The number of partitions of a set. Am Math Mon 71(5):498–504. https://doi.org/10.2307/2312585. ISSN: 0002-9890
13. Dobiński G (1877) Summirung der reihe sum ($n^m/n!$) für $m = 1, 2, 3, 4, 5, \ldots$ Grunert Arch Math Phys 6:333–336
14. Comtet L (1974) Advanced Combinatorics: The Art of Finite and Infinite Expansions. D. Reidel, Dordrecht. ISBN: 90-277-0380-9.
15. Berend D, Tassa T (2010) Improved bounds on bell numbers and on moments of sums of random variables. Probab Math Stat-Pol 30(2):185–205. ISSN: 0208-4147
16. King JR, Nakornchai V (1982) Machine-component group formation in group technology: review and extension. Int J Prod Res 20(2):117–133. https://doi.org/10.1080/00207548208947754. Taylor & Francis
17. Burbidge JL (1985) Production flow analysis. In: Bullinger HJ, Warnecke HJ (eds) Toward the Factory of the Future, Springer, Berlin, pp 34–42. https://doi.org/10.1007/978-3-642-82580-4_7. ISBN: 978-3-642-82580-4
18. Pentico DW (2007) Assignment problems: A golden anniversary survey. Eur J Oper Res 176(2):774–793. https://doi.org/10.1016/j.ejor.2005.09.014. ISSN: 0377-2217
19. Christensen HI, Khan A, Pokutta S, Tetali P (2017) Approximation and online algorithms for multidimensional bin packing: A survey. Comput Sci Rev. 24:63–79. https://doi.org/10.1016/j.cosrev.2016.12.001. ISSN: 1574-0137
20. Bansal N, Eliás M, Khan A (2016) Improved approximation for vector bin packing. In: Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'15), Society for Industrial and Applied Mathematics, Arlington, pp 1561–1579. https://doi.org/10.1137/1.9781611974331.ch106
21. Patt-Shamir B, Rawitz D (2012) Vector bin packing with multiple-choice. Discret Appl Math 160(10):1591–1600. https://doi.org/10.1016/j.dam.2012.02.020. ISSN: 0166-218X
22. Kellerer H, Pferschy U, Pisinger D (2004) Knapsack Problems. Springer, p 566. ISBN: 3-540-40286-1
23. Laabadi S, Naimi M, El Amri H, Achchab B (2018) The 0/1 multidimensional knapsack problem and its variants: A survey of practical models and heuristic approaches. Am J Oper Res 8(5):365–439. https://doi.org/10.4236/ajor.2018.85023
24. Yi C, Cai J (2014) Combinatorial spectrum auction with multiple heterogeneous sellers in cognitive radio networks. In: Proceedings of the IEEE International Conference on Communications (ICC), ISSN: 1938-1883. IEEE, Sydney, pp 1626–1631. https://doi.org/10.1109/ICC.2014.6883555. ISBN: 978-1-4799-2003-7
25. Song Y, Zhang C, Fang Y (2008) Multiple multidimensional knapsack problem and its applications in cognitive radio networks. In: Proceedings of the 2008 IEEE Military Communications Conference (MILCOM), ISSN: 2155-7586. San Diego, pp 1–7. https://doi.org/10.1109/MILCOM.2008.4753629
26. Camati RS, Calsavara A, Lima Jr L (2014) Solving the virtual machine placement problem as a multiple multidimensional knapsack problem. In: Proceedings of the The Thirteenth International Conference on Networks (ICN'14), IARIA, Nice, pp 253–260. ISBN: 978-1-61208-318-6
27. He C, Leung JY-T, Lee K, Pinedo ML (2016) An improved binary search algorithm for the multiple-choice knapsack problem. RAIRO Oper Res 50(4):995–1001. https://doi.org/10.1051/ro/2015061. ISSN: 0399-0559, 1290-3868
28. Lust T, Teghem J (2012) The multiobjective multidimensional knapsack problem: a survey and a new approach. Int Trans Oper Res 19(4):495–520. https://doi.org/10.1111/j.1475-3995.2011.00840.x. ISSN: 1475-3995
29. Horn G, Różańska M (2019) Affine scalarization of two-dimensional utility using the pareto front. In: Proceedings of the IEEE International Conference on Autonomic Computing (ICAC 2019), ISSN: 2474-0756, 2474-0764. IEEE, Umeå, pp 147–156. https://doi.org/10.1109/ICAC.2019.00026
30. Voß S, Lalla-Ruiz E (2016) A set partitioning reformulation for the multiple-choice multidimensional knapsack problem. Eng Optim 48(5):831–850. https://doi.org/10.1080/0305215X.2015.1062094. ISSN: 0305-215X
31. Rahwan T, Michalak TP, Wooldridge M, Jennings NR (2015) Coalition structure generation: A survey. Artif Intell 229:139–174. https://doi.org/10.1016/j.artint.2015.08.004. ISSN: 0004-3702
32. Michalak T, Sroka J, Rahwan T, Wooldridge M, McBurney P, Jennings NR (2010) A distributed algorithm for anytime coalition structure generation. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1, AAMAS '10. International Foundation for Autonomous Agents and Multiagent Systems, Toronto, pp 1007–1014. ISBN: 978-0-9826571-1-9
33. Michalak T, Rahwan T, Elkind E, Wooldridge M, Jennings NR (2016) A hybrid exact algorithm for complete set partitioning. Artif Intell 230:14–50. https://doi.org/10.1016/j.artint.2015.09.006. ISSN: 0004-3702
34. Rahwan T, Michalak T, Elkind E, Faliszewski P, Sroka J, Wooldridge M, Jennings N (2011) Constrained coalition formation. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 25, no. 1. pp 719–725. https://doi.org/10.1609/aaai.v25i1.7888. ISSN: 2374-3468
35. Ueda S, Iwasaki A, Yokoo M, Silaghi MC, Hirayama K, Matsui T (2010) Coalition structure generation based on distributed constraint optimization. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence and the 22nd Innovative Applications of Artificial Intelligence Conference (AAAI-10 / IAAI-10), vol 1. Atlanta, pp 197–203. ISBN: 978-1-57735-464-2
36. Paraskevoulakou E, Tom-Ata JDT, Symvoulidis C, Kyriazis D (2024) Enhancing cloud-based application component placement with ai-driven operations. In: 2024 IEEE 14th Annual Computing and Communication Workshop and Conference (CCWC), pp 0687–0694. https://doi.org/10.1109/CCWC60891.2024.10427694
37. Chitgar N, Jazayeriy H, Rabiei M (2019) Improving Cloud Computing Performance Using Task Scheduling Method Based on VMs Grouping. In: 2019 27th Iranian Conference on Electrical Engineering (ICEE), iSSN: 2642-9527. pp 2095–2099. https://doi.org/10.1109/IranianCEE.2019.8786391
38. Selvarani S, Sadhasivam GS (2010) Improved cost-based algorithm for task scheduling in cloud computing. In: 2010 IEEE International Conference on Computational Intelligence and Computing Research, pp 1–5. https://doi.org/10.1109/ICCIC.2010.5705847
39. Nishio T, Shinkuma R, Takahashi T, Mandayam NB (2013) Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud. In: Proceedings of the first international workshop on Mobile cloud computing & networking, ser. MobileCloud '13. Association for

Computing Machinery, New York, pp 19–26. https://doi.org/10.1145/2492348.2492354. ISBN: 978-1-4503-2206-5
40. Santos J, Wauters T, Volckaert B, De Turck F (2021) Towards end-to-end resource provisioning in fog computing over low power wide area networks. J Netw Comput Appl 175:102915. https://doi.org/10.1016/j.jnca.2020.102915. ISSN: 1084-8045
41. Wu C, Buyya R, Ramamohanarao K (2020) Modeling cloud business customers' utility functions. Futur Gener Comput Syst 105:737–753. https://doi.org/10.1016/j.future.2019.12.044. ISSN: 0167-739X
42. Różańska M, Kritikos K, Marchel J, Folga D, Horn G (2023) Utility function creator for cloud application optimization. In: Barolli L (ed) Advanced Information Networking and Applications, Lecture Notes in Networks and Systems. Springer International Publishing, Cham, pp 619–630. https://doi.org/10.1007/978-3-031-28694-0_58. ISBN: 978-3-031-28694-0
43. Różańska M, Horn G (2022) Proactive autonomic cloud application management. In: Proceedings of the 15th IEEE/ACM International Conference on Utility and Cloud Computing (UCC2022). IEEE/ACM, Vancouver, pp 102–111. https://doi.org/10.1109/UCC56403.2022.00021. ISBN: 978-1-66546-087-3
44. Luenberger DG, Ye Y (2008) Linear and Nonlinear Programming, 3rd edn. Springer. ISBN: 978-0-387-74502-2
45. Korte B, Vygen J (2018) Combinatorial Optimization: Theory and Algorithms, Algorithms and Combinatorics, vol 21, 6th edn. Springer, Berlin Heidelberg, p 627. ISBN: 978-3-540-71843-7
46. Soewito B, Gaol FL, Abdurachman E (2022) A systematic literature review: Risk analysis in cloud migration. J King Saud Univ Comput Inf Sci 34(6):3111–3120. https://doi.org/10.1016/j.jksuci.2021.01.008. ISSN: 1319-1578
47. Gribaudo M, Iacono M, Manini D (2017) Performance evaluation of massively distributed microservices based applications. In: ECMS 2017 Proceedings edited by Zita Zoltay Paprika, Péter Horák, Kata Váradi, Péter Tamás Zwierczyk, Ágnes Vidovics-Dancs, János Péter Rádics, ECMS, pp 598–604. https://doi.org/10.7148/2017-0598. ISBN: 978-0-9932440-4-9
48. Stanton D, White D (1986) Constructive Combinatorics (Undergraduate Texts in Mathematics). In: Gehring FW, Halmos PR (eds), 1st edn. Springer, New York. https://doi.org/10.1007/978-1-4612-4968-9. ISBN: 978-1-4612-4968-9
49. Mansour T, Nassar G (2008) Gray codes, loopless algorithm and partitions. J Math Model Algoritm 7(3):291–310. https://doi.org/10.1007/s10852-008-9086-9. ISSN: 1572-9214
50. Kereskényi-Balogh Z, Nyul G (2014) Stirling numbers of the second kind and bell numbers for graphs. Aust J Comb 58(2):264–274. ISSN: 1034-4942
51. Berceanu C (2001) Chromatic polynomials and k-trees. Demonstratio Math 34(4):743–748. https://doi.org/10.1515/dema-2001-0402. ISSN: 2391-4661
52. Pemmaraju S, Skiena S (2003) Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9781139164849. ISBN: 978-0-521-12146-0
53. Sandholm T, Larson K, Andersson M, Shehory O, Tohmé F (1999) Coalition structure generation with worst case guarantees. Artif Intell 111(1):209–238. https://doi.org/10.1016/S0004-3702(99)00036-3. ISSN: 0004-3702
54. Verginadis Y, Sarros CA, de Los Mozos MR, Veloudis S, Piliszek R, Kourtellis N, Horn G (2023) NebulOuS: A meta-operating system with cloud continuum brokerage capabilities. In: 2023 Eighth International Conference on Fog and Mobile Edge Computing (FMEC), pp 254–261. https://doi.org/10.1109/FMEC59375.2023.10306090
55. de Vries S, Vohra RV (2003) Combinatorial auctions: a survey. INFORMS J Comput 15(3):284–309. https://doi.org/10.1287/ijoc.15.3.284.16077. INFORMS, ISSN: 1091-9856

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.