University of St Andrews
School of Computer Science

CS4099 Senior Honours Project

Memory-Aware VM Packing and Application to Containerised Workloads

Stavros Michalovits

Supervised by
Dr Ruth Hoffmann

4 April 2025; minor revisions 9 June 2025

Abstract

Virtual Machine (VM) Packing is the problem of minimising the number of required physical machines (hosts) to run a set of virtual machines (guests) when the VMs can share identical memory pages if run on the same host. Minimising the number of hosts in data centres offers substantial benefits for service providers and for society, providing strong isolation, safe execution environments and reduced cost and energy consumption. For these reasons, VMs are widely used to encapsulate workloads, even in modern, containerised data centres.

However, the problem is NP-Hard and consequently a candidate for approximation. Accordingly, we approximate it using existing algorithms, design novel syntheses of them and implement an open, unified library in modern C++. As part of this synthesis, we prove that single-host variant of a related problem, VM Maximisation, can be used to approximate VM Packing with tight theoretical bounds. Additionally, we generate a test dataset of VM Packing instances that match the structural characteristics of containerised workloads, by statically analysing Ubuntu 24.04 images and 98 common Docker images.

We demonstrate significant improvements using our synthesised approaches. For the otherwise best-performing existing algorithm, we achieve an additional 11% reduction in host count relative to a Next Fit baseline. A similarly treated First Fit heuristic produces competitive packings while offering lower empirical runtime than Next Fit—among others.

Declaration

Contents

1. Introduction **6**
- 1.1. Contributions . . . 6
- 1.2. Context Survey . . . 7

2. Representations of VM Packing Instances **10**
- 2.1. General Representation . . . 10
- 2.2. Tree Representation . . . 11
- 2.3. Cluster-Tree Representation . . . 13

3. Direct Approximations **16**
- 3.1. Next Fit . . . 16
- 3.2. First Fit . . . 17
- 3.3. Greedy Placement by Efficiency . . . 19
- 3.4. Greedy Placement by Opportunity-Aware Efficiency . . . 23
- 3.5. Overload and Remove . . . 24
- 3.6. Pre-treatment by Ordering . . . 27
- 3.7. Tree-Based Placement . . . 28
- 3.8. Summary of Direct Approximations . . . 30

4. VM Maximisation-Based Approximations **32**
- 4.1. Exact Reduction of VM Packing to VM Maximisation . . . 33
- 4.2. Approximate Reduction of VM Packing to Single-Host VM Maximisation 36
- 4.3. Greedy Placement by Subset Efficiency . . . 38
- 4.4. Cluster-Tree-Based Dynamic Programming . . . 43
- 4.5. Summary of VM Maximisation-Based Approximations . . . 47

5. Generating Synthetic Microservice Workloads **49**
- 5.1. Modelling a Data Centre and its Workloads . . . 49
- 5.2. Extracting Guest OS Pages . . . 52
- 5.3. Extracting Container Pages . . . 54
- 5.4. Generating General Representations . . . 56
- 5.5. Generating Tree and Cluster-Tree Representations . . . 56
- 5.6. Summary of Methodology: Contributions and Limitations . . . 58

6. Empirical Analysis **61**

7. Notes on Testing and Implementation **66**

8. Evaluation and Critical Appraisal **67**

9. Conclusion **69**

A. Appendix **73**
A.1. List of Container Images 73
A.2. Instructions for Use 73
A.3. Complete Trial Outcomes 74
A.4. Complete Test Instances 74

1. Introduction

Virtual Machine (VM) Packing is the problem of minimising the number of required physical machines (hosts) to run a set of virtual machines (guests) when the guest VMs can share identical memory pages, as long as they are placed on the same host machine [26].

In data centres, guest VMs are typically deployed on host machines to enforce isolation between workloads, provide software with a consistent view of the operating system independent of the physical hardware and facilitate the migration of entire workloads between machines [1, 2, 8]. Minimising the number of physical hosts in a data centre facilitates scaling, increases energy efficiency and lowers infrastructure costs, among other benefits.

VM Packing is related to, but distinct from, the well-studied Bin Packing problem [23]. In Bin Packing, each guest has a weight and all hosts share a common capacity. However, this model fails to capture a key characteristic of real VM deployment: guests on the same host can share identical memory pages, reducing their total memory footprint [28, 9]. In data centres whose hosts support memory deduplication, clever deployment of VMs can exploit this fact to minimise the overall physical machine requirement.

A key theoretical observation is that VM Packing can specialise to Bin Packing, meaning that it is at least as hard. In other words, the two problems are identical whenever there is no sharing to exploit between VMs; in those cases, we just need to solve a standard Bin Packing instance. Because Bin Packing is NP-Hard in the general case [11], it follows that VM Packing is also NP-Hard.

This motivates the study of approximation methods for VM Packing. While many variants of VM Packing have been proposed—for example, constraining CPU capacity, communication distance between guests, or allowing variation in host resources [13, 25, 20]—there is little research comparing approximations for the core, memory-aware variant originally proposed by [26]. In this work, we will focus on this core memory-aware formulation of VM Packing with constant-size hosts. Establishing this common formulation allows approaches to be consolidated and compared.

1.1. Contributions

Many of the insights utilised by other variants of VM Packing can also be applied the core variant. It is our goal to extract these insights, combine them where possible and empirically evaluate them.

We thus contribute some new approaches synthesised from our literature study, incorporating:

- The *tree* and *cluster tree* hierarchical sharing variants proposed in [26], which to our knowledge, subsequent research has neglected;
- Reductions to algorithms for the related problem of VM Maximisation;
- Insights from heuristics for more specialised variants of VM Packing; and
- Pre-treatment of the instances and post-treatment of the packings.

On the level of theory, we prove an exact reduction from VM Packing to VM Maximisation, as well as tight theoretical bounds for its worst-case effectiveness if a merely approximate VM Maximisation algorithm is used. We also demonstrate how a general algorithm by [11] can be specialised in order to reduce VM Maximisation to Single-Host VM Maximisation, a variant in which only one host machine is allowed.

Further, current literature only offers limited empirical results, based on randomised datasets. To our knowledge, no dataset with real-world characteristics has been used to evaluate memory-focused VM Packing approximations since the problem was posed by [26]. Some approximations for related problems with multi-resource constraints derive real-world sharing characteristics by statically analysing the operating system (OS) images used by virtual machines (VMs) [25]. We achieve a more fine-grained and up-to-date static analysis by extracting relevant portions of OS images and incorporating pages from Docker images, which correspond to modern containerised workloads.

Additionally, open implementations do not exist for most current algorithms. As such, our implementation of the algorithms in modern C++ represents a meaningful contribution to future research.

Our synthesised approaches achieve strong results on the synthetic container dataset. For the base algorithm we found most effective, Overload and Remove [12], we reduce packing sizes by an additional 11% compared to the baseline after pre-treating it with a technique inspired by [26]. We also demonstrate that a similarly treated First Fit heuristic produces competitive packings while achieving a 4.9× speedup over Overload and Remove. More generally, the synthesised approaches offer substantial net improvements over the algorithms on which they are based.

1.2. Context Survey

1.2.1. Memory-Aware VM Packing

The memory-aware VM Packing problem was first tackled in [26] with approximation methods based on two hierarchical representations of the problem: trees and cluster trees. The representations themselves constitute approximations, in that they do not capture all possible sharing for the set of VMs that they describe. A detailed analysis is provided in Chapter 2, but for now, it is crucial to note that these representations rely on a hierarchical notion of sharing. That is, they imagine that guests may share pages due to running the same OS and OS version and within that, common software libraries, applications, and so on.

[12] untie the problem from the domain of VM deployment and propose approaches to solving these general instances. In addition to two novel heuristics for VM Packing,

they implement genetic algorithms to promising effectiveness but face difficulty scaling to the larger instances. As these larger instances typify our scenario and offer limited interpretability, they are not directly relevant to this dissertation. Bin-packing problems remain an area of research in genetic algorithms, however [15].

1.2.2. VM Maximisation

[18] addresses the VM Maximisation problem, in which each guest has an associated reward value and the goal is to maximise total profit within a fixed number of hosts. In their work, they fix the number of hosts to 1. We refer to this as the single-host variant of VM Maximisation problem.

An online variant of this problem—wherein guests must be placed in a given order of arrival—has also been studied [19], but this constraint reduces its relevance to our aims. The online variant is also no more relevant to practical deployment scenarios. For instance, large-scale optimising schedulers commonly perform periodic optimisations across an entire machine cluster, rather than introduce delays by computing optimal placements at the time that each scheduling unit (e.g. a VM) arrives [27].

Another approach extends placement decisions to consider multiple guests simultaneously. This has been studied in [16] in the context of hosts with multiple resource constraints. By isolating the memory-sharing-aware core of their single-host algorithm, we can view it as a generalisation of [18]—one that considers constant-size sets of guests for placement at a time.

1.2.3. Multiple Resource Constraints

[13] study this problem with an added CPU constraint, but under the condition that the integer CPU and Memory capacity inputs satisfy (Total CPU) = 2 × (Total Memory). While the authors suggest that such configurations are common among cloud providers, they do not provide empirical evidence for this claim. This assumption implicitly requires a normalisation of resource units (e.g., CPU cores and GB of RAM). Still, equating any fixed ratio of these inherently different resources is arguably artificial. Moreover, the assumption of variable-capacity hosts departs from our purpose of assuming hosts of fixed capacity to focus on the core VM Packing formulation.

In studying a joint optimisation problem between communication distance and memory sharing, [25] contribute a placement heuristic that, in essence, includes a concept of opportunity cost for guests with several possible beneficial placements. Formalising this opportunity cost is discussed in Section 3.4.

1.2.4. Empirical Evaluation of VM Packing

The algorithm in [25] is evaluated on a dataset in which the CPU and memory requirements of guest VMs are approximated by the resources of typical Google Cluster [21] workloads. The memory *sharing* characteristics are approximated by the pairwise OS page-sharing ratios identified by [3]—however, sharing due to common applications is unaccounted for.

The algorithms of [26] were evaluated on memory traces from 31 volunteer machines in a university department and from 20 synthetic workloads representing a broader range of software and VM setups. This analysis does not involve modern containerised workloads. Further, the dataset is only used to evaluate the tree- and cluster-tree-based approximations.

In [12], benchmarking is randomised for controlled guest counts, capacities and distributions of pages (symbols, in their terminology) to VMs (tiles). These instances are not informed by the characteristics of typical VM deployment scenarios—in fact, their research is divorced from the application area of VM deployment.

1.2.5. Empirical Analysis of VM Memory Sharing

Before the formalisation of VM packing, the Memory Buddies project [28] had already published a dataset of memory traces from university department machines. Their work focused on placing and migrating VMs under various business constraints (e.g. fault tolerance), with particular emphasis on memory—though the algorithmic details are only briefly discussed.

Subsequent analysis of these traces remains one of the few empirical foundations for research into memory sharing characteristics [28].

Static, non-runtime analysis of OS images has also been carried out in [3], which is used to evaluate packing algorithms in [25]. However, their approach paginates the entire OS image—effectively assuming that all its contents are relevant to runtime memory sharing.

While other static analysis of OS images has taken the same approach, it aimed at compressing storage size and reducing network transmission delay when they are requested [14]—not at representing the contents of memory at runtime. The overall data in the image is relevant to their purpose, but not to ours or to that of [25].

2. Representations of VM Packing Instances

In this section, we will look at different ways of modelling instances of the VM Packing problem. We begin by defining VM Packing.

Definition 2.1 (VM Packing)**.** Given an integer host capacity C, a set of pages P and a non-empty set of guests $G = \{g_1, \ldots, g_n\}$, where each guest is a set $g \subseteq P$ that satisfies $|g_i| \leq C$, construct a packing

$$H = \{h_1, \ldots, h_m\}$$

of guests into the minimum number of hosts. Each host $h_j \subseteq G$ must satisfy

$$\text{pages}(h_j) = \left| \bigcup_{g_i \in h_j} g_i \right| \leq C.$$

That is, the number of unique pages across all guests assigned to a host guests does not exceed C.

Example 2.2. Consider the guests $g_1 = \{1\}, g_2 = \{2, 3, 5, 8\}, g_3 = \{1\}, g_4 = \{3, 5\}, g_5 = \{6, 8\}$.

- The page set is $P = \{1, 2, 3, 5, 6, 8\}$.
- The size of the host $h = \{g_1, g_2\}$ would be $|\{1, 2, 3, 5, 8\}| = 5$.
- Given capacity $C = 4$, a minimal packing is $H = \{\{g_1, g_3, g_5\}, \{g_2, g_4\}\}$. There is no packing of size 1, as the single host would need to contain all 6 pages.

Pages need not be represented by integers, but we can analyse the behaviour of the algorithms by using integers without loss of generality. In fact, we will later see that it is practically useful to represent memory pages as integers.

The remainder of this chapter will focus on how the same instance of VM Packing can be modelled in different ways, each facilitating different approximations.

2.1. General Representation

Following from Definition 2.1, we only need two pieces of information to fully represent an instance of the problem:

- The host capacity; and
- The set of pages in each guest.

As such, a general instance of the problem is simply:

Definition 2.3 (General Instance of VM Packing)**.** A general instance of VM Packing is the pair (G, C), consisting of a set of guests and a host capacity.

The general instance directly matches the problem statement and captures all information about potential page sharing—that is, it is possible to tell exactly what pages are shared among the guests.

We borrow the wording "general instance" from [26] which, to our knowledge, is also the only study of alternative, non-general representations VM Packing instances.

2.2. Tree Representation

We can exploit features of common deployment scenarios to come up with a representation that organises the guests amenably.

[26] observe that in many scenarios, VMs will share some memory pages due to their OS. They argue that there is a base level of sharing due to OS platform, (e.g. "Linux") and subsequently OS version, (e.g. "Ubuntu 24.04.3"). Any applications running within two different OS platforms, however, are highly unlikely to share memory pages.

Before moving forward, it is important to distinguish between *instruction* and *data* pages in memory. This will strengthen the case for the tree representation given in [26].

- *Instruction* pages contain executable code compiled for a particular architecture and, typically, OS platform. When two VMs run programs that use a common library, there may be overlap in the pages associated with that library. Similarly, when two VMs on the same OS platform run the same program, their instruction pages for that program will be identical.
- *Data* pages, on the other hand, hold the data used by a program. This includes both initialised and uninitialised segments, many of which is modified at runtime. As a result, it is highly unlikely that different VMs will meaningfully share data pages.

Our focus is therefore on instruction pages, which enables the idea of hierarchical sharing. Since guests running on different OS platforms do not meaningfully share instruction pages, [26] suggest we can group them into a tree structure as shown in Figure 2.1.

In this tree, we lift the OS-level instruction pages of each guest into shared internal nodes. Each descendant of such a node inherits all of its associated pages. At the leaves, each guest is associated with the remaining instruction pages: those captured by none of its ancestor nodes.

Additionally, [26] suggest that the tree can be further subdivided—by OS version, then by shared libraries, then by applications, and so on.

However, the case for such fine-grained subdivision is weaker and not explored in detail by the authors. Let us demonstrate by considering the effect of subdividing the *Ubuntu 24* node into *Ubuntu 24.04.1* and *Ubuntu 24.04.3*, and then further by libraries. This scenario is illustrated in Figure 2.2.

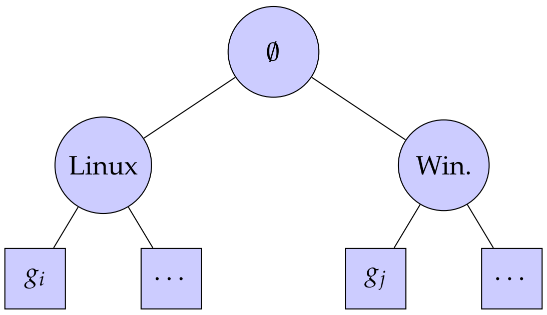

Figure 2.1.: A tree model with one internal layer. The middle layer captures the pages of two OS platforms and the bottom the unique pages of each guest.

In practice, it is common for the same library—e.g., the C++ Boost library—to be used across different Ubuntu versions. Yet, any sharing due to the Boost instruction pages will not be captured across guests under different OS versions in this tree—there could be multiple internal nodes for the same version of Boost, compiled for the same architecture and OS platform. The sharing hierarchy begins to break down for OS versions and library-level nodes. It seems more limited than [26] assume. In Chapter 5, we will also see that it is less applicable to containerised environments.

Nonetheless, the authors report that their tree models capture 67% of possible sharing. This rises to 82% when accounting for the possibility that guests may connect to multiple nodes in the layer above. This allows, for instance, the Boost to be shared within subtrees rooted at Ubuntu 24.04.1 (though still not between the subtrees of Ubuntu 24.04.1 and 24.04.3). We will examine how the cluster-tree model achieves this in the following section.

It is also important to note that these metrics are based on a set of synthetic workloads comprising four desktop-class programs and three server-class programs. Limited information is given on how these are distributed between OS versions. If, in their dataset, certain apps only ever appear under one OS platform and version, this limitation of the model would not be apparent.

Let us now formalise the definition of the tree model.

Definition 2.4 (Tree Instance of VM Packing)**.** A tree instance of VM Packing comprises a set of nodes $\{n_1, ..., n_k\}$ with the added functions:

- pages(n_i), which gives the distinct pages held by the node n_i and shared in all the guests descended from it.
- parent(n_i), which gives the parent node of n_i.

We refer to two nodes with the same parents as siblings, and say that they root sibling subtrees.

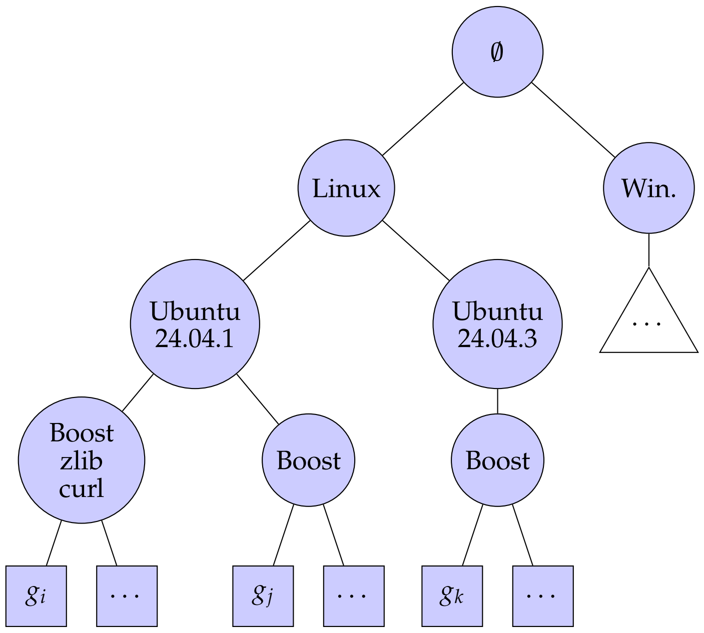

Figure 2.2.: A tree model with three internal layers, capturing shared OS platforms, OS versions and software libraries.

2.3. Cluster-Tree Representation

The cluster-tree representation [26] extends this model by allowing nodes to be linked to multiple parents in the layer above. Each node becomes part of a cluster: fig. 2.3 illustrates that Boost, curl and zlib are now separate nodes in the same cluster. Each cluster has an associated set of child clusters and its nodes are restricted to selecting parent nodes from within those child clusters.

This structure allows more flexible sharing relationships. For instance, the Boost + curl + zlib cluster can be an ancestor of both a guest that uses only Boost and another that uses all three libraries. The model will thus capture shared pages due to Boost even when the guests differ in the rest of their library software.

We also capture more sharing between OS versions with a new node for the pages associated with all Ubuntu versions based on 24.04.

Crucially, however, we still fail to capture a situation where *Boost* should be shared between a guest running Ubuntu 24.04.1 and one running Ubuntu 22.04.3.

Definition 2.5 (Cluster-Tree Instance of VM Packing)**.** A cluster-tree instance of *VM Packing* comprises a set of clusters $\{N_1, ..., N_k\}$ with the added functions:

- nodes(N_i), which gives the nodes contained in the cluster N_i. Each node is defined like a tree instance node, except that its *parent* function is replaced by the function *parents*, which outputs the set of its parent nodes.
- parent(N_i), which gives the parent cluster of N_i.

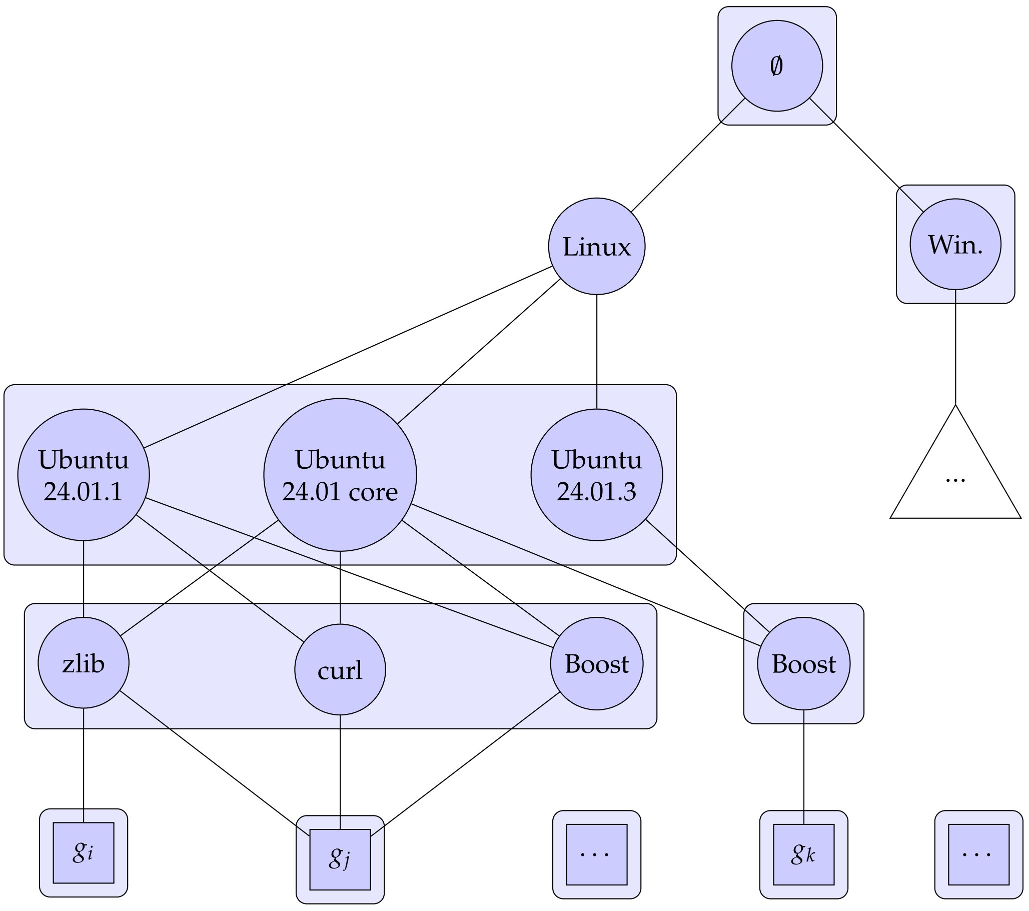

Figure 2.3.: A cluster-tree model with three internal layers, capturing shared OS platforms, versions and software libraries.

Consistent with [26], we refer to the maximum number of nodes held at any cluster as the "width" of the cluster tree. As with regular nodes, we refer to clusters with the same parent as siblings.

3. Direct Approximations

This section focuses on what we refer to, from here on, as *direct* approximations of VM Packing; that is, algorithms that aim directly to approximate VM Packing. Other non-direct approaches that take advantage of reduction to VM Maximisation will be discussed in Chapter 4.

3.1. Next Fit

The first direct approach we consider is the simple heuristic Next Fit, which employs no selection strategy for its choice of host for every guest. It can be helpful to conceptualise it as a form of blind packing. Because it makes no optimising decisions, this heuristic will serve as the baseline for evaluating all other algorithms.

Intuitively, we begin with a single host and attempt to place guests into it one by one. If the current host cannot accommodate a guest, it is closed and we start a new host with that guest.

The pseudocode of Figure 3.1 outlines the operation of Next Fit in more detail. We iterate over the guest set and check whether the latest host in our packing (i.e., the current host) can accommodate the guest. There is no decision to be made: if the guest fits, we must add it to the host. If not, a new host is created and the old one is never considered again.

```
void nextFit(Packing &hosts, const GuestSet &guests, int capacity) {
    for (const Guest &guest : guests) {
        if (hosts.empty() || !accommodates(hosts.back(), guest, capacity)) {
            hosts.emplace_back(); // Create a new host if needed
        }
        hosts.back().addGuest(guest);
    }
}
```

Figure 3.1.: Pseudocode for Next Fit.

We use the helper function Accommodates to determine whether a guest can be placed on a given host. It computes the union of the guest's pages with those already on the host and checks whether the result is within the host capacity (fig. 3.2).

Let us consider an example.

Example 3.1. Consider the guests $g_1 = \{1\}, g_2 = \{2,3,5,8\}, g_3 = \{1\}, g_4 = \{3,5\}, g_5 = \{6,8\}$ and capacity $C = 4$.

- g_1 is placed in a new host: $H = \{h_1\}$, $h_1 = \{g_1\}$ and pages$(h_1) = \{1\}$.

```
bool accommodates(const Host& host, const Guest& guest, int capacity) {
    PageSet &pages = guest.pages();
    for (const Guest &g : host.guests()) {
        pages.insertAll(g.pages());
        if (pages.size() > capacity) {
            // Exit early to reduce unnecessary computation
            return false;
        }
    }
    return true;
}
```

Figure 3.2.: Pseudocode for the *accommodates* helper function.

- g_2 cannot fit in the host, so open a new host: $H = \{h_1, h_2\}$, $h_2 = \{g_2\}$ and pages$(h_2) = \{2, 3, 5, 8\}$.
- g_3 cannot fit in h_2, so open a new host: $H = \{h_1, h_2, h_3\}$, $h_3 = \{g_3\}$ and pages$(h_3) = \{1\}$.
- g_4 can fit in h_3: $H = \{h_1, h_2, h_3\}$, $h_3 = \{g_3\}$ and pages$(h_2) = \{1, 3, 5\}$.
- Finally, g_5 cannot fit in h_3, so open a new host: $H = \{h_1, h_2, h_3, h_4\}$, $h_4 = \{g_5\}$ and pages$(h_4) = \{6, 8\}$.

The packing has size 4. An optimal packing would have size 2, as shown in Example 2.2.

3.1.1. Time Complexity of Next Fit

By virtue of being the simplest heuristic, Next Fit is also the most time-efficient asymptotically.

Theorem 3.2. *Next Fit has worst-case time complexity $O(|G||P|)$.*

Proof. In a naive implementation, for each of the $O(|G|)$ guests, we compute the union of their pages with those on the current host. This union involves $O(|G|)$ guests, each with up to $|P|$ pages. That gives a worst-case time complexity of $O(|G|^2|P|)$.

This is improved if we maintain a cached set representing the union of all pages currently on the host. We update the page set in $O(|P|)$ steps each time a guest is added. Since we add each guest once, the overall time complexity drops to $O(|G||P|)$. □

3.2. First Fit

Next Fit has a clear drawback: it only ever considers the most recently opened host. Even if a guest could fit in an earlier host, that opportunity is never realised. In Example 3.1, we saw that the final guest was only considered for placement in the last host. Could it have fit elsewhere?

First Fit answers this question by considering every existing host for the placement of a guest. It the packs guest in the first host which can accommodate it.

In the pseudocode of Figure 3.3, First Fit considers one guest at a time and scans the current packing for a host that can accommodate it. Once such a host is found, the scan stops with the guest being added to the accommodating host. If no such host is found, we must create a new host.

Let us see if this improves on the packing of Example 3.1.

```
void firstFit(Packing &hosts, const GuestSet &guests, int capacity) {
    for (const Guest &guest : guests) {
        bool placed = false;

        for (Host &host : hosts) {
            if (accommodates(host, guest, capacity)) {
                host.addGuest(guest);
                placed = true;
                break;
            }
        }

        if (!placed) {
            hosts.emplace_back();
            hosts.back().addGuest(guest);
        }
    }
}
```

Figure 3.3.: Pseudocode for First Fit.

Example 3.3. Consider the guests $g_1 = \{1\}, g_2 = \{2,3,5,8\}, g_3 = \{1\}, g_4 = \{3,5\}, g_5 = \{6,8\}$ and capacity $C = 4$.

- g_1 is placed in a new host: $H = \{h_1\}$, $h_1 = \{g_1\}$ and $\text{pages}(h_1) = \{1\}$.
- No host fits g_2, so open a new host: $H = \{h_1, h_2\}$, $h_2 = \{g_2\}$ and $\text{pages}(h_2) = \{2,3,5,8\}$.
- The first host that fits g_3 is h_1: $H = \{h_1, h_2\}$, $h_1 = \{g_1, g_3\}$ and $\text{pages}(h_1) = \{1\}$.
- The first host that fits g_4 is h_1: $H = \{h_1, h_2\}$, $h_1 = \{g_1, g_3, g_4\}$ and $\text{pages}(h_1) = \{1,3,5\}$.
- Finally, no host fits g_5, so open a new host: $H = \{h_1, h_2, h_3\}$, $h_3 = \{g_5\}$ and $\text{pages}(h_3) = \{6,8\}$.

The packing has size 3. An optimal packing would have size 2, as shown in Example 2.2.

This is an improvement over Next Fit, but still suboptimal. What went wrong?

Our placement decisions were not necessarily smart: we could have placed g_4 in h_2 at no capacity cost, since h_2 already contained all the pages of g_4. But First Fit chose h_1 instead, simply picking the first viable option.

3.2.1. Time Complexity of First Fit

Before considering better strategies, let us close this section by considering the time complexity of First Fit.

Theorem 3.4. *First Fit has worst-case time complexity* $O(|G|^2|P|)$.

Proof. For each of the $O(|G|)$ guests, we must scan a number of hosts that is bounded above by the number of guests that have already been placed, i.e., $O(|G|)$.

Therefore, for each of these $O(|G|)$ hosts, we need to check whether it can accommodate the current guest. Using the caching strategy outlined in the proof of Theorem 3.2, this check takes $O(|P|)$ steps per host.

This is $O(|G||P|)$ steps to place each of $O(|G|)$ guests, so the total worst-case time complexity is $O(|G|^2|P|)$. □

3.3. Greedy Placement by Efficiency

Returning to Example 3.3, it would have been useful to evaluate the efficiency of each guest when placed on a particular host. g_4 should not have been placed on h_1 where it occupies additional size, when it could have been placed on h_2 with zero footprint.

[12] study an algorithm to this effect under the name "Best Fusion." For generality, let us call this Greedy Efficiency.

Greedy Efficiency is implemented the same way as First Fit, but instead of the picking the first accommodating host, we pick the most efficient one.

For brevity, we will not re-implement this pseudocode. We will instead focus on the theory behind our efficiency metric: the relative size of a guest g in host h.

Definition 3.5 (Relative Size of g on h)**.** The relative size of a guest g on a host h is defined as

$$\mathrm{RelSize}(g, h) = \sum_{p \in g} \frac{1}{\max\{1, |\{g' \in h \mid p \in g'\}|\}}$$

That is, it is the sum of the reciprocals of the number of guests on h that contain each page in g [12].

In effect, the weight of each page is scaled down from a maximum of 1 based on how frequently it appears among the guests currently in the host. The maximum value of $\mathrm{RelSize}(g, h)$ is $|g|$, achieved when none of the pages are shared. The minimum value is 0, either when g has no pages, or in the limit, when each page appears infinitely often on the host.

It would be sensible to question why we are counting the frequency of a page on a host, instead of using its true contribution to capacity, which is either 0 (page already present) or 1 (not present). This is not answered directly by the authors, but we can build a theoretical case for it.

We necessarily have a fixed number of occurrences of any page p across all guests. Suppose we are evaluating the placement of a guest that has page p onto a host h, where p appears with high frequency. For a sufficiently high frequency of p on h, we can be confident that most of the guests containing p are already on h. When this holds for more pages in g, the case for placing it on h is stronger: we become more confident that better sharing opportunities do not exist elsewhere. By contrast, pages that appear with low frequency should not have the same appeal.

In effect, the relative size metric penalises g based on the opportunity cost of placing it on h without directly computing the missed alternatives.

Taking stock, if we have found a host h' for g with $\mathrm{RelSize}(g, h') < |g|$, then we have found an improvement over a placement that exploits no sharing.

It is nonetheless possible that no such candidate exists, or that no such candidate can accommodate the guest. When that happens, [12] suggest that we force the creation of a new host. This is an interesting theoretical trade-off. As before, let us build an intuitive theoretical case for it.

Suppose we are considering a placement of $g = \{4, 5, 6\}$ on a host with $\mathrm{pages}(h) = \{1, 2, 3\}$ where it exploits no sharing. Consequently, $\mathrm{RelSize}(g, h) = 1$. We have two options: place g into h, resulting in $\mathrm{pages}(h) = \{1, 2, 3, 4, 5, 6\}$; or place g on a new host h' with $\mathrm{pages}(h') = \{4, 5, 6\}$. If we implement the first option and later encounter more guests with pages 4, 5, 6, we will place them in h. If we later encounter guests with pages 1, 2, 3, we will place those, too, in h. But this raises a subtle question: do we want those pages to be on the same host? The answer depends on whether subsequent guests will share pages from both set $\{1, 2, 3\}$ and set $\{4, 5, 6\}$. If not, then it could be that $\{1, 2, 3\}$ and $\{4, 5, 6\}$ would be better off in different hosts. For example, if we encounter many guests with pages both in $\{4, 5, 6\}$ and $\{7, 8, 9\}$, we do not want $\{1, 2, 3\}$ to have polluted h and possibly prevent the placement of $\{7, 8, 9\}$.

Returning to [12], we can view the forced creation of a new host as a tactic for deferring this decision until after all guests have been placed. We force the creation of the new $\mathrm{pages}(h') = \{4, 5, 6\}$ host, and later—once all guests are placed—we will attempt to pull into the original host h the guest that we have put in h'. If there is indeed much commonality between h and h', that sharing may now be realised. If there is little commonality after having placed all guests, then that sharing will not be realised.

In [12], this *decanting post-treatment* identifies improvements for 21% of Greedy Efficiency packings. In our data, the mean improvement was more modest; see Chapter 6.

3.3.1. The Decanting Post-Treatment

The decanting post-treatment considers each pair (h_l, h_r) of hosts in the packing such that h_l lies to the left of h_r. At each step, it partitions the guests in the right host h_r by a given strategy, and then attempts to move (i.e., decant) each partition from h_r to h_l.

On first sight, it might seem unusual that decanting only considers ordered pairs of hosts—but there is a reason for the procedure being left-biased. If we operated both on (h_l, h_r) and on (h_r, h_l) as described, our later movements could undo some of the movements carried out before: we would first decant guests to the left and later to the right. Accordingly, and without loss of generality, we always decant in the left direction.

As argued in [12], three successive decanting steps are needed at increasing levels of granularity.

We will not repeat the full argument here, but the underlying intuition is to begin with more ambitious consolidation and gradually move to finer-grained improvements.

1. **Whole-host partitioning.** First, we attempt to move the entire right host h_r into a left host h_l, treating the set of guests in h_r as a single unit.

2. **Component partitioning.** Next, we identify groups of guests within h_r that share at least one page with another guest in the same group. This can be modelled as an undirected graph: each guest is a node, and an edge connects any two guests that share a page. The connected components of this graph form the partitions, and can be computed by any search algorithm—depth-first search in our implementation.

3. **Individual partitioning.** Finally, we attempt to move each guest in h_r individually into h_l, thereby capturing any remaining sharing opportunities.

Note that decanting would be redundant on First Fit. Decanting moves partitions to the leftmost host that can accommodate them, but First Fit has already packed all guests as such: if it were viable to move a partition $\{g_1, ...\}$ of h_r into h_l, this would necessitate that g_1 alone can be moved into h_l, which cannot be true if the original packing was generated by First Fit.

In the pseudocode of Figure 3.4, we assume that one of three partitioning strategies is provided. From there, decanting individually considers each left host h_r and proceeds as follows.

- For each host h_r to the right of h_l, partition the guest in h_r using the given strategy.

- Iterate over the partitions sequentially, greedily moving each partition that can be accommodated in h_l into it and out of h_r.

- After processing the last h_l, delete any hosts that are now empty.

3.3.2. Time Complexity of Greedy Placement By Efficiency

The Greedy Efficiency algorithm operates in two stages: the packing phase and the decanting phase.

Let us evaluate the complexity of decanting first.

Theorem 3.6. *The decanting post-treatment has worst-case time complexity $O(|G|^4|P|)$.*

Proof. Each of the three decanting phases considers all ordered pairs of hosts, of which there are $O(|G|^2)$ in the worst case.

```
void decantGuests(Packing &hosts, PartitionStrategy partitionGuests) {
    for (Host& leftHost : hosts) {
        for (Host &rightHost : hosts after leftHost) {
            // Partition the right host using the given strategy
            List<GuestSet> partitions = partitionGuests(rightHost.getGuests());

            // Greedily move as many as possible into the left host
            for (const GuestSet &partition : partitions) {
                if (!leftHost.canAccommodate(partition)) {
                    continue;
                }

                for (const Guest &guest : partition) {
                    leftHost.addGuest(guest);
                    rightHost.removeGuest(guest);
                }
            }
        }
    }
    // Clean up the hosts we managed to fully consolidate into others
    hosts.removeIf(Host::isEmpty);
}
```

Figure 3.4.: Pseudocode for the decanting post-treatment.

For each such pair (h_l, h_r), we attempt to move each partition of the guests in h_r to h_l. Checking whether a host can accommodate a partition of size $O(|G|)$ requires computing the union of their pages, which can be done in $O(|G||P|)$ time amortised over all partitions.

Thus, the steps per host pair are:

$$O(|G||P| + \text{cost of partitioning})$$

The worst-case cost of partitioning is incurred in the component-based strategy. We search the host as an undirected graph with $O(|G|)$ nodes, having checked for edges between all $O(|G|^2)$ pairs of guests. Each edge requires computing the intersection of two guests in $O(|P|)$ time, giving a total cost of $O(|G|^2|P|)$ to construct the graph and $O(|G|)$ to visit each node once.

So, the total cost of partitioning is $O(|G|^2|P| + |G||P|) = O(|G|^2|P|)$, and the overall cost per host pair becomes:

$$O(|G||P| + |G|^2|P|) = O(|G|^2|P|)$$

Over $O(|G|^2)$ host pairs and three decanting passes, the total cost is:

$$O(3 \cdot |G|^2 \cdot |G|^2|P|) = O(|G|^4|P|)$$

□

The number steps taken in the core packing stage is comparatively insignificant and thus the overall complexity is dominated by decanting.

Theorem 3.7. *Greedy Efficiency has worst-case time complexity $O(|G|^2|P|)$, assuming no decanting post-treatment.*

Proof. Consider the initial packing phase. For each of $O(|G|)$ guests, we compute the relative size on $O(|G|)$ hosts. By using a caching strategy for the frequencies of each page on a host, computing relative size and placing a guest are both possible in $O(|P|)$ time, giving $O(|G|^2|P|)$ for the placement phase. □

While this algorithm is relatively expensive in theory, it is not among the most expensive in practice—as we will see in Chapter 6. This is because the number of hosts is typically much smaller than the worst-case bound of $O(|G|)$, by constant factors depending on the host capacity and page-sharing potential in the instance. Additionally, algorithms that reduce packing size also benefit by a slower growth in the number of hosts. When the original packing is smaller, the impact of the decanting phase is also lower.

3.4. Greedy Placement by Opportunity-Aware Efficiency

So far, we have seen that the Greedy Efficiency algorithm places each guest on the host where its efficiency is maximised, estimating the cost of missed sharing using only local information. However, we also noted that this packing phase is relatively insignificant in complexity compared to the decanting phase. This gives us some room to improve upon Greedy Efficiency by incorporating more opportunity-cost information.

While this issue has not been directly addressed in the literature, [25] include a factor to this effect in their heuristic, which jointly optimises for memory usage and communication distance with variable-capacity hosts. We define the following heuristic, which behaves equivalently to their formulation when communication distance is omitted and all hosts have fixed capacity.

Definition 3.8 (Opportunity-Aware Efficiency)**.** Given a guest g, a host h, and a set of all hosts H, the opportunity-aware efficiency of placing g on h is defined as

$$\text{OA-Efficiency}(g, h, H) = \frac{|g \cap \text{pages}(h)| + \min_{h' \in H \setminus \{h\}} |g \setminus \text{pages}(h')|}{|g|}$$

That is, a guest's efficiency on a given host is the number of its pages already present on that host, plus the least number of pages it would have to contribute elsewhere—all scaled by the size of the guest.

Our goal is to maximise this metric. In particular, OA-Efficiency(g, h, H) is high when:

- g has relatively many pages on h, i.e., $|g \cap \text{pages}(h)|$ is high; and

- Placing g on any other host would result in a relatively large contribution, i.e., the minimum size increase that another host would incur, $\min_{h' \in H \setminus \{h\}} |g \setminus \text{pages}(h')|$, is high.

We can again apply the decanting treatment from [12].

3.4.1. Time Complexity of Greedy Placement by Opportunity-Aware Efficiency

Theorem 3.9. *Greedy Opportunity-Aware Efficiency has worst-case time complexity $O(|G|^2|P|)$.*

Proof. Consider the initial packing phase. For each of $O(|G|)$ guests, we compute the efficiency on $O(|G|)$ hosts. At each host, we compute the difference of g with the page set of every other host, each in $O(|P|)$ steps if the hosts' page sets are cached. At each host, therefore, we do $O(|G|^2|P|)$ work, which is $O(|G|^3|P|)$ over the whole packing phase.

However, when placing a given guest, we can precompute and cache the sizes of the two smallest page differences it has with the hosts. This allows us to compute the efficiency metric for each host in just $O(|P|)$ time, without a third iteration over all hosts and only $O(1)$ extra space.

The overall complexity of the packing phase therefore drops to $O(|G|^2|P|)$. □

The initial packing phase of Greedy Opportunity-Aware Efficiency has the same time complexity as Greedy Efficiency, namely $O(|G|^2|P|)$, at $O(1)$ asymptotic space cost. In practice, however, computing the additional cache for each guest may incur a significant overhead—it involves an extra pass over all $O(|G|)$ *guests*, which is practically much more expensive than the $O(|G|)$ terms arising from passes over hosts, because guests are more numerous than hosts.

3.5. Overload and Remove

Overload and Remove, introduced by [12], takes a different approach to improving upon their own Greedy Efficiency algorithm. We will not expand on the reasoning behind Overload and Remove much here, as it is well-described by the authors and unlike Greedy Efficiency, does not constitute a building block for any other known algorithms. However, we will highlight how it departs from the approaches discussed so far.

Overload and Remove continues to select the best host based on relative size but allows displacing guests from full hosts in order to accommodate newly discovered beneficial placements.

The key decision under this approach is which guests to remove to make room for the new one.

The metric used to identify the *worst* guests g on a host h is (the maximum of)

$$\frac{\text{RelSize}(g,h)}{|g|}$$

That is, the relative size of the guest on the host, scaled by its actual size to facilitate comparison across guests of different sizes. The same reasoning behind *RelSize* applies as before, but here we normalise the metric to enable comparison.

Since the removed guests need to be rescheduled for packing, we maintain a retry-queue of all guests pending placement. Removed guests are pushed to the back of the queue. While this retry order can only be arbitrary—Overload and Remove is agnostic to guest order, including when retrying placements—it is also sensible to defer reattempting problematic guests until the eligible hosts have changed more meaningfully; hence the first-in-first-out ordering.

At the end of this process, some hosts may remain overfull. In that case, we remove all their guests and reinsert them into the existing packing using a separate algorithm. We use First Fit for this step to avoid coupling more complex heuristics in ways that could obscure their individual empirical performance.

```
void overloadAndRemove(Packing& hosts, GuestSet guests, int capacity) {
    Queue<Guest&> unplaced = guests;
    Map<Guest&, Set<Host&>> attemptedPlacements;

    while (unplaced) {
        Guest guest = unplaced.popFront();

        // Find the host with minimum relSize, as long as we haven't already
        ↪  tried to place this guest there
        Host &bestHost = NULL;
        double bestRelSize = INF;
        for (Host &host : hosts) {
            if (attemptedPlacements[guest].contains(host)) {
                continue;
            }
            double candidateRelSize = relSize(guest, host);
            if (candidateRelSize <= bestRelSize) {
                bestRelSize = candidateRelSize;
                bestHost = host;
            }
        }

        // Create a host if there are none
        if (bestHost == NULL) {
            hosts.emplace_back();
            bestHost = hosts.back();
        }

        // Add to the best host
        bestHost.addGuest(guest);
        attemptedPlacements[guest].insert(bestHost);

        // Bring the host back under capacity by removing the least efficient
        ↪  guests, scaled by their size
        while (bestHost.isOverfull()) {
            Guest &worst = bestHost.guests().minBy(
            candidate -> sizeToRelSizeRatio(candidate, bestHost)
            );
            bestHost.removeGuest(worst);
            unplaced.pushBack(worst);
        }
    }

    // Remove all overfull hosts and re-pack their guests by First Fit
    for (Host &host : hosts) {
        if (host.isOverfull()) {
            continue;
        }
        for (Guest &guest : host.guests()) {
            unplaced.pushBack(guest);
        }
        host.clearGuests();
    }

    // Repack using First Fit
    firstFit(hosts, unplaced, capacity);

    // Optional decanting passes
    decantGuests(hosts, partitionAllGuestsTogether);
    decantGuests(hosts, partitionConnectedGuestsTogether);
    decantGuests(hosts, partitionGuestsIndividually);
}
```

Figure 3.5.: Pseudocode for Overload and Remove.

3.5.1. Time Complexity of Overload and Remove

Theorem 3.10. *The worst-case time complexity of Overload and Remove is*

$$O(|G|^3|P| + T_{pack}(|G|))$$

where $T_{pack}(|G|)$ denotes the maximum number of steps taken in a single pass of the fallback packing heuristic, assuming no decanting post-treatment.

Proof. In the worst case, we try to pack each of $O(|G|)$ guests $O(|G|)$ times—one to eliminate every possible choice of host. With each attempt, we consider $O(|G|)$ hosts where it has not been placed before, giving $O(|G|^3|P|)$ for the packing phase of the algorithm.

Finally, we pack at worst $|G|$ guests using the fallback packing heuristic in $O(T_{\text{pack}}(|G|))$ time. The overall complexity for both stages is $O(|G|^3|P| + T_{\text{pack}}(|G|))$.

□

3.6. Pre-treatment by Ordering

One aspect of the algorithms we have not yet addressed is their dependence on the initial ordering of guests. This opens up another potential avenue for optimisation: to pre-process the input by ordering the guests advantageously.

To our knowledge, this question has not been studied directly in the VM Packing literature, aside from general bin-packing schemes such as First Fit Decreasing (which sorts in descending order of weight). [26] also note that in their use case, First Fit produces better results when they "initially place the two [guests] that share the most pages [in the same host] ... and consecutively pick [the guest] that shares the most with [the most the recently created host]".

This is a useful intuition, as Example 3.11 illustrates.

Example 3.11. Consider a general instance $(\{g_1 = \{1,2,3\}, g_2 = \{3,4,5\}, g_3 = \{2,3,4\}, ...\}, C = 5)$, whose the first three guests are passed to our solvers in that order.

All the algorithms we have seen so far would place g_1 in a host h_1 and then evaluate g_2, assigning it to the same host. g_3, which follows, would be placed in a new host h_2.

However, given no additional information about subsequent guests, placing g_3 with g_1 would be a Pareto improvement on this intermediate packing: it would use the same number of hosts while reducing the load on h_1.

The processing order $g_1, g_3, g_2, ...$ would therefore be strictly better.

A keen-eyed reader may have noticed that we already have two strategies for grouping guests by their sharing characteristics: tree and cluster-tree instances.

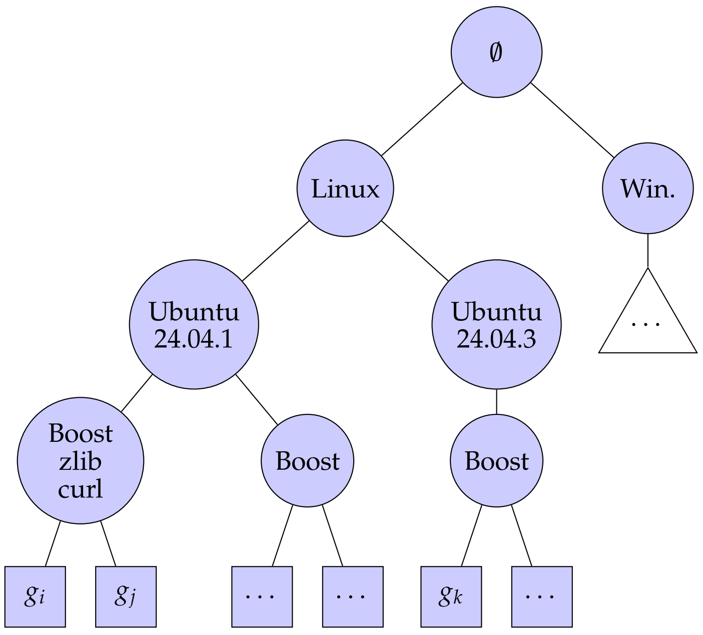

Figure 3.6.: A tree model with three internal layers, capturing shared OS platforms, OS versions and software libraries.

Consider the tree instance Figure 5.4. Sibling leaves share their entire software stack: gi, g_j share Boost, zlib, curl, Ubuntu 24.04.1 and Linux. In a well-constructed tree or cluster tree, as the separation between leaves increases, the difference in their software stacks grows: for example, only Linux is shared between g_i and g_k.

Hence we have a new variant for each algorithm we have seen so far—one that operates on a tree or cluster-tree instance by extracting its leaves in order and feeding them to the general-instance algorithm. Under this tree- or cluster-tree-ordering pre-treatment, a guest such as g_j would be placed just after its sibling g_j.

3.7. Tree-Based Placement

We now turn to the algorithm proposed by [26] for packing tree instances.

As with Overload and Remove, we do not contribute new theory here. Our goal in this section is to explain the underlying intuition for the algorithm, while referring to the original paper for additional theoretical results.

Let us explain the operation of the algorithm in its three phases.

Input: A tree instance

1. At each node, compute a *lower bound* on the number of hosts required to pack all guests in its subtree.
    - [26] suggest computing this bound by allowing guests to be fragmented across multiple hosts. We refer to their work for further details.
    - Observe that if the lower bound at a node is 1, then no guest needed to be fragmented in the lower bound packing.
    - In particular, if the root has a lower bound of 1, then all guests can be packed into a single host: return a single-host packing.
    - While [26] does not discuss other lower-bound heuristics, there is no reason why a tighter bound wouldn't lead to better behaviour. This would be an interesting direction for future research.
2. Identify a node whose children all have lower bounds of 1, but which itself does not.
    - Such a node always exists: e.g., a parent of leaves (each leaf must be individually packable by Definition 2.1). If not, we can move up a level until we find one, eventually reaching the root (which, because of the previous phase, cannot have a lower bound of 1).
    - If there are multiple candidates, tiebreak by choosing the one with the smallest lower bound.
3. Add all guests descended from that node to the current packing and remove the subtree from the instance.

- We use plain First Fit as a baseline for the behaviour of the algorithm in empirical analysis. However, this may leave room for further optimisation: a more advanced heuristic such as Overload and Remove could be used instead—an option we will evaluate empirically.

Repeat: From step (1) on the remaining instance.

The reader may wish to take a moment to convince themselves that Tree-Based Placement functions as a dynamic ordering strategy for the underlying heuristic used to pack the leaves of each subtree. As we have described it here, it is a more general version of a tree-ordered First Fit that dynamically adjusts the guest order by selecting subtrees rather than processing the leaves left-to-right.

Finally, we are free to apply the decanting post-treatment step from [12] as well.

3.7.1. Time Complexity of Tree-Based Placement

[26] specify that this is a polynomial-time algorithm, but let us investigate in greater precision.

First, an intermediate result.

Theorem 3.12. *If $p(x)$ is a polynomial, then $\deg(p(x_1) + \ldots + p(x_n)) = \deg(p(x_1 + \ldots + x_n))$, where $\deg$ gives the degree of a polynomial.*

Follows from the binomial theorem.

Theorem 3.13. *If the underlying packing heuristic is polynomial-time with respect to the number of guests, then the worst-case time complexity of Tree-Based Placement is*

$$O(|G|^2|P| + T_{pack}(|G|),$$

where $T_{pack}(|G'|)$ denotes the maximum number of steps taken in a single pass of the underlying packing heuristic on any general instance (G', C).

Proof. Assume that no internal node in the tree is unused by all guests. Under this assumption, the total number of tree nodes is bounded by $O(|G|)$.

Each iteration of the three-phase loop begins by computing lower bounds for all nodes in a bottom-up traversal. The page set of each node is merged with that of its subtree which it roots. This process takes $O(|G||P|)$ time per iteration, that is, an $O(|P|)$ merge for each of $O(|G|)$ nodes.

As each iteration removes at least one guest, the loop runs at most $O(|G|)$ times, yielding a total cost of $O(|G|^2|P|)$ for the lower-bound and selection phases.

In the final phase of each iteration, we pack a disjoint subset $G_i \subseteq G$ using the underlying heuristic. Since the subsets $G_1, G_2, \ldots$ partition G, we have $\sum_i |G_i| = |G|$. Each packing call takes at most $T_{\text{pack}}(|G_i|)$ steps.

We consider two cases:

- Case 1. Suppose $T_{\text{pack}}(|G_i|)$ is exactly polynomial. Then by Theorem 3.12,

  $$\deg\left(\sum_i T_{\text{pack}}(|G_i|)\right) = \deg(T_{\text{pack}}(|G|)$$

  Consequently,

  $$\sum_i T_{\text{pack}}(|G_i|) \in O(T_{\text{pack}}(|G|))$$

- Case 2. Suppose $T_{\text{pack}}(|G_i|)$ is sub-polynomial. Then the growth of each term $T_{\text{pack}}(|G_i|)$ is asymptotically no greater than in Case 1, and we again have

  $$\sum_i T_{\text{pack}}(|G_i|) \in O(T_{\text{pack}}(|G|)),$$

Thus, the total cost of the packing phase over all iterations is $O(T_{\text{pack}}(|G|))$.

Combining all three phases over the whole guest set, the total complexity of Tree-Based Placement is

$$O(|G|^2|P| + T_{\text{pack}}(|G|)).$$

□

3.8. Summary of Direct Approximations

Table 3.1 compares the runtime complexity of our direct approximations.

Approximation	**Order?**	**Decant?**	**Complexity**
NEXT FIT	✗	✗	$O(\|G\|\|P\|)$
... *with eligible treatments applied*	✓	✓	$O(\|G\|^4\|P\|)$
FIRST FIT	✗	*N/A*	$O(\|G\|^2\|P\|)$
... *with eligible treatments applied*	✓	*N/A*	$O(\|G\|^2\|P\|)$
GREEDY EFFICIENCY	✗	✗	$O(\|G\|^2\|P\|)$
... *with eligible treatments applied*	✓	✓	$O(\|G\|^4\|P\|)$
GREEDY OPP.-AWARE EFFICIENCY	✗	✗	$O(\|G\|^2\|P\|)$
... *with eligible treatments applied*	✓	✓	$O(\|G\|^4\|P\|)$
OVERLOAD AND REMOVE	✗	✗	$O(\|G\|^3\|P\| + T_{\text{pack}}(\|G\|))$
... *with eligible treatments applied*	✓	✓	$O(\|G\|^4\|P\| + T_{\text{pack}}(\|G\|))$
TREE-BASED PLACEMENT	*N/A*	✗	$O(\|G\|^2\|P\| + T_{\text{pack}}(\|G\|))$
... *with eligible treatments applied*	*N/A*	✓[1]	$O(\|G\|^4\|P\| + T_{\text{pack}}(\|G\|))$

Table 3.1.: Summary of approximations, treatments applied and their worst-case time complexities. The middle columns show the eligible treatments for each approximation algorithm.

4. VM Maximisation-Based Approximations

A related problem to VM Packing is VM Maximisation [26]. In fact, as we will prove in Section 4.1, a general instance of VM Packing is reducible to one of VM Maximisation—that is, we can convert a general instance of VM Packing to an equivalent VM Maximisation instance. Using an algorithm for VM Maximisation, we may solve this instance and convert the result back into a solution to VM Packing. Algorithms developed for VM Maximisation can thus form another part of our toolbox of approximations. Figure 4.1 shows this strategy diagrammatically.

First, let us state VM Maximisation.

Definition 4.1 (VM Maximisation)**.** Given a number M of hosts, each with integer capacity C, a set of pages P, positive rewards $R = \{r_1, \ldots, r_n\}$ and guests $G = \{g_1, \ldots, g_n\}$, where each guest $g_i \subseteq P$ satisfies $|g_i| \leq C$, construct a packing

$$H = \{h_1, \ldots, h_M\}$$

of some guests $G' \subseteq G$ such that each host $h_j \subseteq G'$ satisfies

$$\left| \bigcup_{g_i \in h_j} g_i \right| \leq C.$$

That is, the number of unique pages across all guests assigned to a host does not exceed C. The goal is to maximise the total reward generated from G':

$$\sum_{g_i \in G'} r_i$$

Definition 4.2 (Single-Host VM Maximisation)**.** An instance of *Single-Host VM Maximisation* is any instance of *VM Maximisation* where $M = 1$.

We will not motivate the problem statement in practice. For us, it serves as a means to the end of developing VM Packing approximations. In general, though, it can be useful to service provider to choose a subset of all guests based on profitability and sacrifice some others.

This section will first define a reduction from VM Packing to VM Maximisation. Then, we will see can approximate general VM Maximisation using algorithms for Single-Host VM Maximisation. Finally, we will use these results to approximate VM Packing by two Single Host VM Maximisation methods:

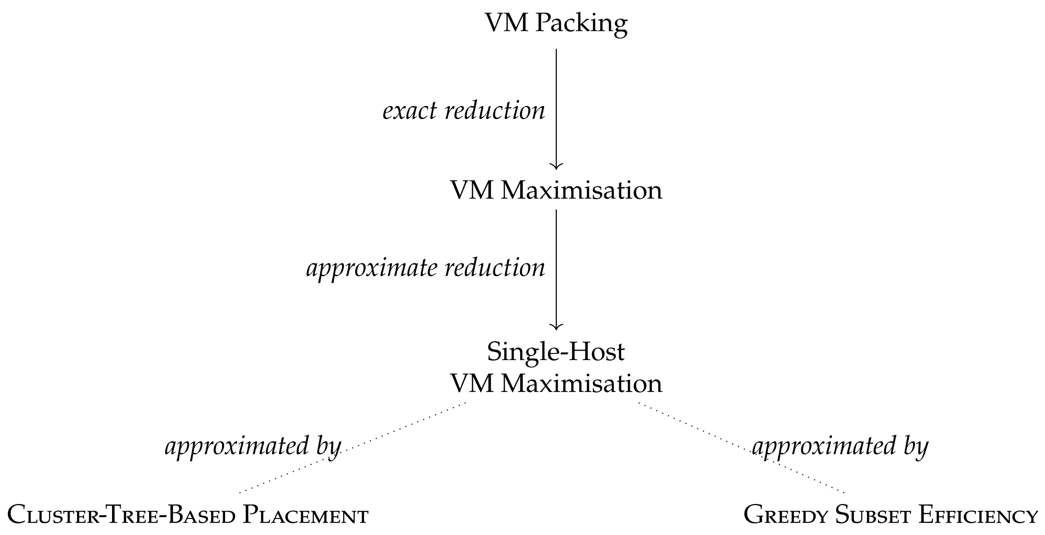

Figure 4.1.: Outline of the reduction strategy for approximating VM Packing by Single-Host VM Maximisation.

- A greedy family of methods that consider subsets of mutually efficient guests for placement at a time; and
- A cluster-tree-based dynamic programming approximation.

4.1. Exact Reduction of VM Packing to VM Maximisation

The first step in building our toolbox of VM Maximisation-based approximations is to show that the two problems are equivalent; that is, VM Packing can be reduced to VM Maximisation.

Theorem 4.3. *There exists an exact polynomial-time reduction of VM Packing to VM Maximisation.*

Proof. We begin with an arbitrary general instance (G, C) of VM Packing. Our aim is to construct an instance (G, C, R, M) of VM Maximisation by augmenting it with an appropriate reward set R and number of hosts M.

We begin by fixing $R_{\text{ones}} = \{1 \mid g \in G\}$.

Suppose M_{opt} is the minimum value of M for which solving the VM Maximisation instance $(G, C, R_{\text{ones}}, M_{\text{opt}})$ yields a total reward of $|G|$. Such an M_{opt} always exists: for example, if we set $M = |G|$, placing each guest in its own host achieves a reward of $\sum_{r \in R_{\text{ones}}} r = |G|$.

Our main proposition is that a packing of G into M_{opt} hosts is the optimal solution to VM Packing. To show this, suppose for contradiction that there exists a packing of G into $M < M_{\text{opt}}$ hosts. VM Maximisation could achieve reward $|G|$ with fewer than M_{opt} hosts, which would be a contradiction.

Now, as M_{opt} lies in the interval $[1, |G|]$, we can perform a linear search over all $M' \in [1, |G|]$ and solve the VM Maximisation instance $(G, C, R_{\text{ones}}, M')$ at each step. The first M' for which we obtain a reward of $|G|$ is M_{opt}, and the packing which achieves it an optimal solution to the VM Packing instance (G, C).

With regard to time complexity, if each of the $O(|G|)$ instances of VM Maximisation can be solved in at most polynomially many steps $T_{\text{max}}(|G|)$, then the reduction takes $O(|G| \cdot T_{\text{max}}(|G|))$ steps.

Therefore, we have given a polynomial-time reduction from VM Packing to VM Maximisation. □

We can slightly improve on the time complexity of this reduction.

Theorem 4.4. *The worst-case time complexity of reducing VM Packing to VM Maximisation is*

$$O(\lg |G| \cdot T_{max}(|G|))$$

if the VM Maximisation algorithm operates in at most $T_{max}(|G|)$ steps.

Proof. We will proceed by the method in Theorem 4.3 but search for M_{opt} using binary search on $[1, |G|]$.

Observe that any value to the left of M_{opt} will produce an incomplete packing (with reward less than $|G|$) and any value to its right a full packing (with reward $|G|$). When the midpoint of the current interval cannot produce a full packing, we recur to the values after the midpoint. When it *can* produce a full packing, we keep note of the packing and recur to the values before the midpoint. We return the latest value of M found that can produce a correct packing.

Since binary search performs $O(\log |G|)$ iterations and each VM Maximisation instance is solved in at most $T_{\text{max}}(|G|)$ steps, the overall time complexity is $O(\log |G| \cdot T_{\text{max}}(|G|))$. □

We now have a method for converting any exact algorithm for VM Maximisation to an exact algorithm for VM Packing. However, VM Maximisation were solvable by an exact algorithm in polynomial time, then VM Packing—and consequently Bin Packing—would be polynomial-time, which is impossible unless $P = NP$. It is prudent to instead focus on approximating VM Maximisation and investigating how these algorithms hold as proxies for approximating VM Packing.

For the purposes of this dissertation, we are interested in the empirical effectiveness of the approximations. As such, we have not provided theoretical bounds for our algorithms with respect to the optimal packing size. The reductions will be an exception to this rule: we want to show that given a VM Maximisation algorithm with provably good approximation bounds, the reduction will preserve the existence of bounds on the worst-case result.

4.1.1. Tight Bound Preservation in the Exact Reduction of VM Packing to Approximate VM Maximisation

Our reduction indeed holds to reasonable bounds for approximations of VM Maximisation.

Theorem 4.5. *VM Packing can be approximated to a factor of*

$$O\left(-\frac{\lg|G|}{\lg(1-\alpha)}\right)$$

using any algorithm that can approximate VM Maximisation to a factor of $O(\alpha)$.

Proof. We will build a proof by considering the number of superfluous guests that an approximate reduction generates compared to an exact one.

Let the true optimal number of hosts for a given VM Packing instance to be M_{opt}. For the equivalent VM Maximisation instance $(G, C, R_{\text{ones}}, M_{\text{opt}})$ considered in the reduction, the maximum reward that can be generated is $|G|$. Our goal is to determine how many hosts are created by an α-approximate VM Maximisation algorithm in terms of the optimal number M_{opt}.

An α-approximation of VM Maximisation estimates the optimal reward of $|G|$ to bounds $\alpha|G|$. The number of guests $G^{(1)}$ that remain is exactly equal to the reward that the approximation could not realise:

$$\begin{aligned}|G^{(1)}| &= |G| - \alpha|G| \\ &= |G|(1-\alpha)\end{aligned}$$

It remains to determine how many more hosts are needed to accommodate these leftover guests.

An optimal VM Packing algorithm can pack $G^{(1)}$ in at most M_{opt} additional hosts, as it is able to pack its superset G in M_{opt} hosts. By the same reasoning as before, the α-approximation would leave $|G^{(2)}| = |G^{(1)}|(1-\alpha)$ remaining guests. More generally, the number of remaining guests remaining after packing nM_{opt} hosts is

$$\begin{aligned}|G^{(n)}| &= |G^{(n-1)}|(1-\alpha) \\ |G^{(0)}| &= |G|\end{aligned}$$

Thus $|G^{(n)}| = |G|(1-\alpha)^n$ and we are looking for n such that the number of hosts remaining is $|G^{(n)}| < 1$. In particular, for such an n we have

$$\begin{aligned}|G^{(n)}| < 1 &\Leftrightarrow |G|(1-\alpha)^n < 1 \\ &\Leftrightarrow n > -\frac{\lg|G|}{\lg(1-\alpha)}\end{aligned}$$

Therefore, zero guests remain at

$$n = \left\lceil -\frac{\lg|G|}{\lg(1-\alpha)} \right\rceil + 1$$

And thus the maximum number of hosts needed is

$$nM_{\text{opt}} = \left\lceil -\frac{\lg|G|}{\lg(1-\alpha)} + 1 \right\rceil M_{\text{opt}}$$

Proving that the VM Packing approximation has a worst-case bound of

$$O\left(-\frac{\lg|G|}{\lg(1-\alpha)}\right)$$

□

4.2. Approximate Reduction of VM Packing to Single-Host VM Maximisation

4.2.1. Tight Bound Preservation in the Approximate Reduction of VM Packing to Single-Host VM Maximisation

Before we describe the operation this approximate reduction method, let us motivate why we apply it to VM Packing by stating its theoretical bounds.

Theorem 4.6. *VM Maximisation can be approximated to a factor of*

$$O\left(\frac{\alpha}{\alpha+1} - \varepsilon\right)$$

for any desired $\varepsilon > 0$ using any algorithm that can approximate Single-Host VM Maximisation to a factor of $O(\alpha)$.

Theorem 4.6 follows directly from a general result proven for all *Separable Assignment Problems* in [11]—that is, for problems that involve precisely a fixed set of bins (i.e., the M hosts in our case); a fixed set of items (i.e., the set of guests G); a value for each item (i.e., the set of rewards R); a packing constraint for each bin (i.e., that $|\bigcup_{g_i \in h_j} g_i| \leq C$); and the goal of packing items to maximise the total value across all bins.

Theorem 4.7. *VM Packing can be approximated to a factor of*

$$O\left(-\frac{\lg|G|}{\lg|1+\varepsilon-\frac{\alpha}{\alpha+1}|}\right)$$

for any desired $\varepsilon > 0$ using any algorithm that can approximate the optimal reward of Single-Host VM Maximisation to a factor of $O(\alpha)$.

Follows directly from Theorems 4.5 and 4.6.

4.2.2. Operation of the Approximate Reduction of VM Packing to Single-Host VM Maximisation

We now turn to how the reduction operates. We will begin by looking at the operation of the intermediate reduction of VM Maximisation to Single-Host Maximisation, as given in its general version in [11]. We will first restate their method with no modifications (other than using our VM Maximisation terminology). In certain steps, the reader may notice that the method can be simplified to match the particular characteristics of VM Maximisation. We will show how to exploit this immediately thereafter.

Input: A set of guests G, capacity C, host count M, α-approximate single-host maximiser and parameter ε, which adjusts the theoretical bounds of the approximation.

1. Start with an empty set of M hosts.
2. Fix $n_{\text{iterations}} = \frac{M}{\alpha} \lg \frac{\alpha}{\varepsilon(\alpha+1)}$.
3. For $i = 1, \ldots, n_{\text{iterations}}$:
   a) Consider each host h individually.
   b) Define the marginal reward of each guest as:

$$\text{marginal}(g_i) = r_i - \begin{cases} \text{the reward } r_i \text{ generated by } g_i \text{ on } h' & \text{if } g_i \in h' \text{ and } h' \neq h \\ 0 & \text{otherwise} \end{cases}$$

   c) Compute the overall marginal reward for the host h by using the maximiser to pack guests into it by their marginal rewards.
   d) Define the improvement for the host h to be the difference between its overall marginal reward and its current total reward.
   e) Implement the repacking for the host with the greatest improvement. The host is cleared and all guests in the new packing moved to it.

We begin our simplifications with step 3(b). Guests' rewards are same irrespective of the host on which they are placed, so a simple but equivalent definition of marginal reward is:

$$\text{marginal}(g_i) = \begin{cases} 0 & \text{if } g_i \in h' \text{ and } h' \neq h \\ r_i & \text{otherwise} \end{cases}$$

Observe that there is zero marginal reward for moving a guest that is already placed on a host to the current host. As it is *never* beneficial for the single-host maximiser to accommodate a guest with zero reward, the algorithm effectively packs a distinct subset host at each step. Therefore, the reduction is semantically equivalent to the following process.

Input: A set of guests G, capacity C, host count M, α-approximate single-host maximiser and parameter ε, which adjusts the theoretical bounds of the approximation.

1. Start with an empty set of M hosts.

2. Fix $n_{\text{iterations}} = \frac{M}{\alpha} \lg \frac{\alpha}{\varepsilon(\alpha+1)}$.
3. For $i = 1, \ldots, n_{\text{iterations}}$:
   a) Pack one of the empty hosts using the single-host maximisation algorithm.
   b) Remove the packed guests from G.

By Theorem 4.3, we must perform a search for the value $M = M_{\text{opt}}$ in order to obtain a solution to VM Packing. This search can be carried out either using the general binary search method described in Section 4.1, or—in the case of reduction to Single-Host VM Maximisation—by simply allowing the intermediate reduction process to use the maximum number of hosts $|G|$ and discarding any that remain empty at the end. This approach is valid because increasing the number of hosts past M_{opt} does not affect the packing: each host is packed once and independently of the others, because guests are never moved between hosts, as we just showed. In that case, we can simplify the number of iterations in the reduction to just $n_{\text{iterations}} = |G|$ without affecting the outcome.

Theorem 4.8. *Reducing VM Packing to Single-Host VM Maximisation has worst-case time complexity*

$$O(|G| \cdot T_{single}(|G|))$$

if the Single-Host VM Maximisation algorithm operates in at most $T_{single}(|G|)$ steps.

Proof. For each of $O(|G|)$ hosts, we run the single-host maximisation algorithm exactly once. Each invocation is invoked on $O(|G|)$ guests, so if its worst-case time complexity is $T_{\text{single}}(|G|)$, the overall complexity is

$$O(|G| \cdot T_{\text{single}}(|G|))$$

□

4.3. Greedy Placement by Subset Efficiency

It remains to be seen which single-host maximisation algorithms can be used as part of this reduction. In this section, we will explore a family of greedy algorithms that select guests based on an efficiency metric which balances sharing potential and reward. Algorithms in this family appear in both [16] and [18], so we generalise their common strategy.

Unlike the efficiency metrics discussed so far, this approach considers entire sets of guests for placement at once. In particular, it begins with an empty host and repeatedly selects the subset of unplaced guests that together have the highest efficiency-adjusted value for placement.

Let us define the efficiency metric before proceeding with a worked example.

Definition 4.9. Given a host h for which reward should be maximised, a set of guests

$G = \{g_1, ..., g_n\}$ not on the host and rewards $R = \{r_1, .., r_n\}$, the efficiency of G on h is

$$\text{Subset-Efficiency}(G, h) = \frac{\sum_{g_i \in G} r_i}{\max\left\{1, \left|\bigcup_{g_i \in G} g_i \setminus \text{pages}(h)\right|\right\}}$$

That is, the efficiency of a set of guests on a host is their total reward scaled by the number of pages that they contribute to the host.

Intuitively, this introduces information about other unplaced guests to each placement decision. With a subset size of k, we are considering with each guest not individually, but in the context of another $k-1$ guests which can be placed along with it. This is illustrated in Example 4.10.

Example 4.10. Consider the guests $g_1 = \{1, 2, 3\}, g_2 = \{4, 5, 6\}, g_3 = \{4, 5, 6\}$, all with rewards equal to 1, a host h which currently holds $\text{pages}(h) = \{1, 7\}$ and a capacity $C = 6$.

Observe that we can either place g_1 alone (for a reward of 1) or g_2, g_3 together (for a reward of 2) on h, but not both. Let us see what happens when we compute efficiency by groups instead of individually.

- *Individual Choice* (Subset Size $k = 1$). g_1 has the least memory footprint on h and captures the same reward as each other individual guest; its efficiency is $\frac{1}{|\{2,3\}|}$. Therefore, we place g_1 onto h, which now cannot accommodate more guests. We have generated an additional reward of 1.
- *Group Choice* (Subset Size $k = 2$). We can now consider $\{g_2, g_3\}$ *together*, with a subset efficiency of $\frac{1+1}{|\{4,5,6\}|} = \frac{2}{3}$. We place g_2 and g_3 on h for a reward of 2.

Increasing the subset size has allowed us to guide placement decisions with information about other unplaced guests.

However, we also note that there are cases where fixing the subset size could be detrimental. For instance, g_1 of Example 4.10 could have offered a high reward of 3, but as it cannot be placed jointly with any other guest, it would have overlooked in our evaluation of subsets of exactly size 2. One way to mitigate this concern is to always look at subsets of size $k = 1$ *or* 2, allowing choices such as the singleton $\{g_1\}$. This would be an explicit and correct fix, at the cost of evaluating more subsets.

An alternative view posits that we have given a contrived counterexample: normally, g_1 would be considered for placement as part of the larger sets, but we constructed a case in which g_1 is pathological enough to make this is impossible. Further, if no size-k subsets are small enough to pack, we could update the subset size to $k-1$ and retry. It is generally better to prioritise size-k subsets even if it risks individual high-reward, high-size guests being overlooked—unless we know this to be a common occurrence in our input data. In our reduction, all rewards are made equal to 1.

This latter perspective is not merely practical but also validated by theory: a larger subset size always tightens the worst-case approximation bounds. This is formally shown in [16] for multi-resource constraints and therefore also applies to the special case of a single resource—memory—that concerns us. We will focus on this approach, which is in line with current literature, and leave the other to future work.

We began this section by stating that to our knowledge, two algorithms of this family exist in the literature. In particular, the approximation in [18] is semantically identical to the algorithm we have described here with subset size $k = 1$. [16] let k be a general input parameter. They focus on problems with multiple resource constraints, so we obtain our algorithm by isolating the memory-related aspects of their approach.

To summarise alongside the pseudocode of Figure 4.2, we begin with a subset size of k and the empty host h. We compute the efficiency of each size-k subset of the unplaced guests that can be accommodated in h and implement the choice with highest efficiency. If there exists *no* size-k subset that can be accommodated, then k is permanently reduced. If no subset is found for any $k > 0$, then no more guests are viable for packing, thus the algorithm terminates.

```
void greedySubsetEfficiency(Host &host, const Map<Guest&, int> &rewards, int k)
{
    while (!rewards.empty()) {
        Optional<GuestSet&> bestGuests = findMostEfficientSubset(rewards, host, k);
        // Reduce the subset size until we find a subset that can be accommodated
        while (bestGuests.isNothing() && --k > 0) {
            bestGuests = findMostEfficientSubset(unplaced, host, k);
        }
        // If there is no viable guest, the algorithm can terminate
        if (bestGuests.isNothing()) {
            break;
        }

        for (const Guest &guest : bestGuests) {
            rewards.erase(guest);
            host.addGuest(guest);
        }
    }
}
```

Figure 4.2.: Pseudocode for Greedy Subset Efficiency.

4.3.1. Time Complexity of Greedy Placement by Efficiency

In order to investigate the impact of the parameter k, it is important to consider time complexity with respect to both k and G . As we will see in Chapter 6, there is substantial runtime difference between $k = 1$ and $k = 2$ in practice. In theory, k can only be as large as $|G|$—but reasoning about it independently of $|G|$ will provide more accurate insight into runtime performance.

Theorem 4.11. *Greedy Subset Efficiency has worst-case time complexity*

$$O\left(\binom{|G|}{\min\{k, |G|/2\}} k^2 |P|\right)$$

letting the input parameter k vary and using $\binom{|G|}{r}$ to denote the number of size-r subsets of the set G.

Proof. Greedy Subset Efficiency proceeds by successive iterations over a number of subsets of G.

Over the runtime of the algorithm, suppose that we process m_1 subsets of size 1, m_2 of size 2 and so on, up to m_k of size k. In particular, the total number of subsets that we consider across all iterations is

$$\begin{aligned} & m_1\binom{|G|}{1} + m_2\binom{|G|}{2} + \ldots + m_k\binom{|G|}{k} \\ &= \sum_{r=1}^{k} m_r \binom{|G|}{r} \end{aligned} \tag{4.1}$$

There can also only be one iteration per subset size r for which r guests are not extracted from G. Therefore, $m_r - 1$ iterations for subset r will place r guests.

$$\begin{aligned} 1 \cdot (m_1 - 1) + 2 \cdot (m_2 - 1) + \ldots + k \cdot (m_k - 1) &\leq |G| \\ \Rightarrow 1 \cdot m_1 + 2 \cdot m_2 + \ldots + k \cdot m_k &\leq |G| + k^2 \\ \Leftrightarrow \sum_{r=1}^{k} r \cdot m_r &\leq |G| + k^2 \end{aligned} \tag{4.2}$$

Further, all m_r are nonzero because each subset size is considered at least once, so

$$\prod_{r=1}^{k} m_r \neq 0 \tag{4.3}$$

We now turn to question of finding worst-case values for the sequence of m_r; we want to maximise the sum in eq. (4.1) subject to the constraints in eq. (4.2) and eq. (4.3). Specifically, we will seek the value $r = r_{\text{worst}}$ which maximises $f(r) = \frac{1}{r}\binom{|G|}{r}$; that is, the single term for which we obtain the highest value compared to its r factor in eq. (4.2).

Then, all other $r \neq r_{\text{worst}}$ are exactly 1 and a maximum overall sum is obtained.

This value of r must be less than $\frac{|G|}{2}$, as for any $r > \frac{|G|}{2}$ we could obtain a higher $f(|G| - r)$:

$$f(r) = \frac{1}{r} \cdot \binom{|G|}{r} \leq \frac{1}{|G| - r} \cdot \binom{|G|}{r} = \frac{1}{|G| - r} \cdot \binom{|G|}{|G| - r} = f(|G| - r)$$

But as $r < \frac{|G|}{2}$ means $f(r)$ is monotonically increasing, we need only look for the first r such that

$$\begin{aligned}
f(r) > f(r+1) &\Leftrightarrow \frac{1}{r} \cdot \binom{|G|}{r} > \frac{1}{r+1} \cdot \binom{|G|}{r+1} \\
&\Leftrightarrow 1 > \frac{r}{r+1} \cdot \frac{\binom{|G|}{r+1}}{\binom{|G|}{r}} \\
&\Leftrightarrow 1 > \frac{r}{r+1} \cdot \frac{\frac{|G|!}{(r+1)!(|G|-r-1)!}}{\frac{|G|!}{r!(|G|-r)!}} \\
&\Leftrightarrow 1 > \frac{r}{r+1} \cdot \frac{|G| - r}{r+1} \\
&\Leftrightarrow 2r^2 + (2 - |G|)r + 1 > 0 \\
&\Leftrightarrow r > \frac{|G| - 2 + \sqrt{|G|^2 - 4|G| - 4}}{4}
\end{aligned}$$

Therefore $r_{\text{worst}} = \left\lceil \frac{|G|-2+\sqrt{|G|^2-4|G|-4}}{4} \right\rceil$. The corresponding m_{worst} is obtained by setting all other m_r to 1 in eq. (4.2) and solving for the equality to obtain an upper bound on the pathological case:

$$\begin{aligned}
&(1 \cdot 1 + 2 \cdot 1 + \ldots + r_{\text{worst}} m_{\text{worst}} + \ldots + k \cdot 1) \leq |G| + k^2 \\
&\Rightarrow r_{\text{worst}} m_{\text{worst}} + k^2 \leq |G| + k^2 \\
&\Leftrightarrow m_{\text{worst}} \leq \frac{|G|}{r_{\text{worst}}} \\
&\Rightarrow m_{\text{worst}} = \left\lceil \frac{4|G|}{|G| - 2 + \sqrt{|G|^2 - 4|G| - 4}} \right\rceil \approx 2 \quad \text{for large enough } |G|
\end{aligned}$$

The sum of eq. (4.1) is therefore:

$$\begin{aligned}
&\left[1\binom{|G|}{1}+1\binom{|G|}{2}+\cdots+1\binom{|G|}{m_{\text{worst}}}+\cdots+1\binom{|G|}{k}\right]+(m_{\text{worst}}-1)\binom{|G|}{m_{\text{worst}}}\\
&\leq k\binom{|G|}{\min\{k,\lceil |G|/2\rceil\}}+m_{\text{worst}}\binom{|G|}{m_{\text{worst}}}\\
&\leq (k+2)\binom{|G|}{\min\{k,\lceil |G|/2\rceil\}} \quad \text{for large enough } |G|
\end{aligned}$$

To complete the proof, for each subset we consider at most k guests to check all their pages that are on a host, in at most $O(k|P|)$ time. Therefore, we have an overall time complexity of

$$O\left(\binom{|G|}{\min\{k,|G|/2\}}k^2|P|\right)$$

□

Corollary 4.12. *Greedy Subset Efficiency, when $k \leq \frac{|G|}{2}$, has worst-case time complexity*

$$O\left(\binom{|G|}{k}k^2|P|\right)$$

4.4. Cluster-Tree-Based Dynamic Programming

The final algorithm that we will explore maximises reward on the host using dynamic programming on the cluster-tree model.

Dynamic Programming (DP) is an algorithmic technique that recursively breaks a problem into subproblems, solves each subproblem once and uses the results to construct an overall solution [5]. If smaller subproblems can be combined to solve larger ones, we may begin with the smallest and build up to a solution incrementally.

In our case, the easily solvable subproblem is that of packing the minimal subtrees consisting of just a leaf cluster each, containing just a guest. Considering each leaf subtree individually, reward is maximised by simply packing it on the host.

From there, we need a way to combine the solutions for the *small* subtrees rooted at multiple sibling clusters, obtaining a solution for the *large* subtree rooted at their parent cluster. We may have to make a decision: if the host cannot accommodate the entire *large* subtree, some of its children will have to be discarded—the guests that form their leaves will be left out of the packing. We obtain a solution for the entire tree by working upwards from the leaves.

Let us describe the DP recurrence relation from [26] before working through an example.

Define:

$$\text{cap}[N, S, j, r] = \begin{array}{l}\text{Least host capacity required to achieve reward} \geq r \\ \text{from the subtree at cluster } N \text{ by packing some of the} \\ \text{guests descended from a subset } S \text{ of the nodes of} \\ \text{cluster } N \text{ and the first } j \text{ children of } N.\end{array}$$

The best viable solution for the subtree rooted at N allows guests to be picked from all its children and constrains the capacity to C; that is, the $\text{cap}[N, S, |\text{children}(N)|, r] \leq C$ with highest r.

Initialise:

$$\text{cap}[N, S, 0, r] = \begin{cases} \left|\bigcup_{n \in S} \text{pages}(n)\right| & \text{if } S \text{ contains guests with reward} \geq r \\ \infty & \text{otherwise} \end{cases}$$

In this phase, we initialise the entries at which zero ($j = 0$) children are available for every cluster N. Still, the zero reward initialises $\text{cap}[N, S, 0, 0] = \left|\bigcup_{n \in S} \text{pages}(n)\right|$ for all N and S. All other entries are ∞ except at the leaf clusters, which achieve rewards when their nodes are directly selected, no matter the value of j.

Merge:

$$\text{cap}[N, S, j, r] = \min_{0 \leq r' \leq r} \left\{ \text{cap}[N, S, j{-}1, r{-}r'] + \min_{\substack{S' \subseteq \text{nodes}(N_j) \\ \text{parents}(S') \subseteq S}} \{\text{cap}[N_j, S', |\text{children}(N_j)|, r']\} \right\}$$

where N_j denotes the j^{th} child of N. For each $j = 0, ..., |\text{children}(N)|$ in order, we check if we can achieve every possible reward value r. In particular, we check if we can obtain all possible $r' < r$ from the j^{th} child of N_j, picking the S which requires the least capacity. The remaining reward $r - r'$ must come from the previous children $N_0, ..., N_{\text{j-1}}$, i.e., the term $\text{cap}[N, S, j{-}1, r{-}r']$. We sum the contributions to capacity to obtain an upper bound for the total cost.

Example 4.13. Consider the cluster of Figure 4.3 and capacity $C = 6$.

We will highlight a selection of *cap* entries that illustrate how the final result is built. The reader may find it a useful exercise to work out additional entries of *cap*.

After the initialisation phase:

- $\text{cap}[N_1, \{n_1, n_2\}, 0, 3] = 3$, since we can pack g_1, g_2 for reward ≥ 3 at capacity 3.
- $\text{cap}[N_1, \{n_1, n_2\}, 0, 2] = 3$, since we can achieve reward ≥ 2 by the same packing.
- $\text{cap}[N_i, S, 0, 0] = 0$ for all N_i, S, since reward ≥ 0 can be achieved with any choice of nodes at zero capacity.

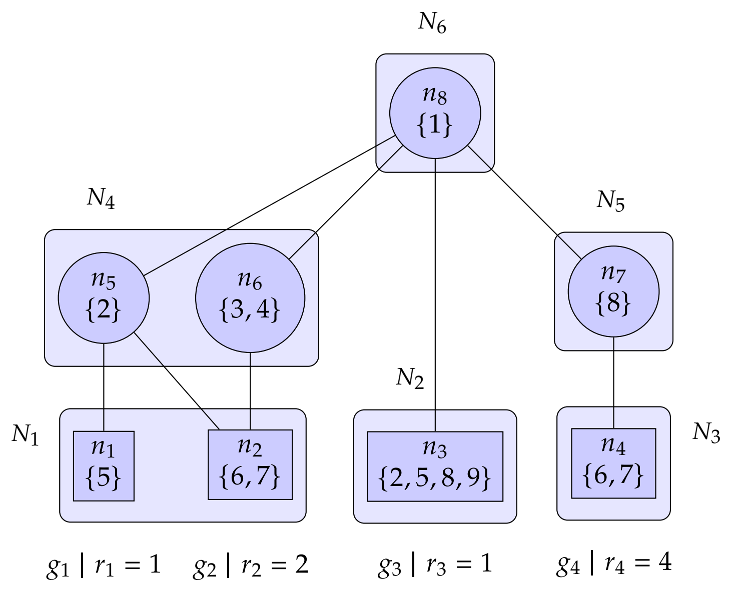

Figure 4.3.: A generic cluster tree. Clusters and nodes are numbered bottom-up in the order that CLUSTER-TREE-BASED DP traverses them. This cluster tree misses some sharing: n_3 contains pages that are also in other subtrees.

- $\text{cap}[N_4, \{n_5\}, 0, 1] = \infty$, since it is impossible to achieve *any* profit using n_5 and ignoring the child cluster N_1.

We proceed upwards, merging the results for N_1 into results for N_4.

- $\text{cap}[N_4, \{\}, 0, 0] = 0$ from the initialisation phase before.
- $\text{cap}[N_4, \{\}, 0, 1] = \infty$ from the initialisation phase before.
- $\text{cap}[N_4, \{\}, 1, 1] = \infty$ since even if we allow access to N_1, the empty node selection from N_4 disables all choices of nodes from N_1. Therefore, the reward of 1 is generated by taking $r = 1$ from $\text{cap}[N_4, \{\}, 0, 1] = \infty$ and $r' = 0$ from the 1st child N_1.
- $\text{cap}[N_4, \{n_5\}, 1, 1] = 2$ since we can now access both the nodes n_1, n_2. Minimum capacity is obtained by taking reward $r = 0$ at cost $\text{cap}[N_4, \{n_5\}, 1, 0] = 1$ and selecting n_1 from the child cluster to render the remaining reward $r' = 1$ at cost $\text{cap}[N_1, \{n_1\}, 0, 1] = 1$.
- $\text{cap}[N_4, \{n_5\}, 1, 2] = 3$ by similarly selecting just n_2 from N_1.
- $\text{cap}[N_4, \{n_4, n_5\}, 1, 3] = 6$ by selecting both n_1 and n_2 from N_1.

By the same process as N_5, we get:

- $\text{cap}[N_5, \{n_7\}, 1, 4] = 2$, via the single path to the guest g_4.

Now at N_1:

- $\text{cap}[N_1, \{n_8\}, 3, 5] = 5$ by taking:

  $r = 1$ from the first two children, N_4 and N_2. We take $\text{cap}[N_4, \{n_5\}, 1, 1] = 2$, which corresponds to selecting n_5 and n_1.

  $r' = 1$ from the third child, N_5, by the entry $\text{cap}[N_5, \{n_7\}, 1, 4] = 2$.

  We must also add the capacity cost of the base entry for N_1, that is $\text{cap}[N_1, \{n_8\}, 0, 0] = 1$. In total, we require a minimum capacity of $2 + 2 + 1 = 5$.

- $\text{cap}[N_1, \{n_8\}, 3, 6] = 9$ by taking:

  $r = 3$ from the first two children, N_4 and N_2, taking $\text{cap}[N_4, \{n_4, n_5\}, 1, 3] = 6$.

  $r = 4$ from the third child as before.

  Overall, we can achieve reward $7 \geq 6$ by requiring a minimum capacity of $6 + 2 + 1 = 9$.

Looking for the maximum achievable reward at N_1 when considering all $j = 3$ of its children and not exceeding a capacity of 6, we find total reward 5 from $\text{cap}[N_1, \{n_8\}, 3, 5] = 5 \leq 6$. Assuming we kept track of the guests taken each merge, we would see that this result corresponds to the packing $\{g_1, g_2, g_4\}$ on the single host.

This example has shown how Cluster-Tree-Based DP evaluates sharing by assuming that the capacity requirements of all sibling subtrees at a merge step can be added together. In other words, it assumes that no pages are duplicated between different subtrees.

But this is was not true in Example 4.13, resulting in some unrealised reward: g_3 could have been packed alongside $\{g_1, g_2, g_4\}$ at an additional cost of 1 page, which can be accommodated. Our model, however, simply adds the capacity requirement for the subtrees of N_2, N_4 and N_5, assuming that they hold completely distinct pages, thus overestimating the capacity requirement as much as the size of n_3. Cluster-Tree-Based DP can only be as accurate as the underlying cluster-tree model.

4.4.1. Time Complexity of Cluster-Tree-Based DP

As with Greedy Subset Efficiency, we will inspect the time complexity of Cluster-Tree-Based DP in relation to the parameters of the cluster tree, specifically its width and the number of children of each cluster. This will give us a clear view of how runtime can vary as we tune these parameters.

Theorem 4.14. *Cluster-Tree-Based DP has worst-case time complexity $O(\frac{4^w}{w} d \cdot r_{all}^2 |G|)$ where r_{all} is the reward of packing all guests, w the maximum number of nodes in any cluster and d the maximum number of children for any cluster.*

Proof. The algorithm solves a number of subproblems and writes their results to unique entries cap$[N, S, j, r]$. We will separately consider how many such entries—and thus subproblems—exist and the number of steps taken to solve each.

First, the number of entries in *cap* is $O\left(\frac{|G|}{w} \cdot 2^w \cdot d \cdot r_{all}\right)$, because there are at most:

- $O(|G|/w)$ choices of cluster N;
- $O(2^w)$ choices of node set S;
- $O(d)$ choices of number of children j; and
- $O(r_{\text{all}})$ choices of attempted reward r.

Now, each entry requires $O(2^w \cdot r_{\text{all}})$ steps to solve the corresponding subproblem, because we evaluate at most:

- $O(2^w)$ choices of node set S' in cluster N_j; and
- r_{all} choices of the complementary reward r' we are attempting.

Therefore, we have an overall time complexity of $O\left(\frac{4^w}{w} d \cdot r_{\text{all}}^2 |G|\right)$. □

Corollary 4.15. *Cluster-Tree-Based DP has worst-case time complexity* $O\left(\frac{4^w}{w} d|G|^3\right)$ *when all rewards are equal to 1, w denotes the maximum number of nodes in any cluster and d denotes the maximum number of children for any cluster.*

4.5. Summary of VM Maximisation-Based Approximations

Taking stock, we have established two polynomial-time reductions:

- An exact reduction from VM Packing to VM Maximisation.
- An approximate reduction from VM Maximisation to Single-Host VM Maximisation.

Both reductions preserve tight approximation bounds, with the first reduction being exact when the VM Maximisation algorithm itself is exact. We use approximate reduction from VM Maximisation to Single-Host VM Maximisation as an intermediate step to reducing VM Packing to Single-Host VM Packing, achieving:

- Worst-case time complexity of

  $$O(|G| \cdot T_{\text{single}}(|G|))$$

  where $T_{\text{single}}(|G|)$ is the worst-case time complexity of the single-host maximisation algorithm; and

- Approximation bounds of

$$O\left(-\frac{\lg|G|}{\lg|1+\varepsilon-\frac{\alpha}{\alpha+1}|}\right)$$

where α is the approximation bound of the single-host maximisation algorithm.

Table 4.1 compares the performance of our two single-host algorithms when combined with the two reductions.

Approximation	**Order?**	**Decant?**	**Complexity**
Greedy Subset Efficiency	*N/A*	✗	$O\left(\binom{\lvert G\rvert}{k}k^2\lvert G\rvert\lvert P\rvert\right)$ [1]
... *with eligible treatments applied*	*N/A*	✓	$O\left(\lvert G\rvert^4\lvert P\rvert+\binom{\lvert G\rvert}{k}k^2\lvert G\rvert\lvert P\rvert\right)$
Cluster-Tree-Based DP	*N/A*	✗	$O\left((4^w/w)d\lvert G\rvert^4\right)$
... *with eligible treatments applied*	*N/A*	✓	$O\left(\lvert G\rvert^4\lvert P\rvert+(4^w/w)d\lvert G\rvert^4\right)$

Table 4.1.: Summary of VM Maximisation-based approximations, treatments applied and their worst-case time complexities. Ordering is not applicable to either single-host algorithm, thus not relevant to the overall approximations.

5. Generating Synthetic Microservice Workloads

We now turn to the empirical part of our work. This chapter will describe our methodology for generating a dataset comprising VMs that run containerised workloads.

Although containers are often deployed directly on bare metal to minimise virtualisation overhead [22], multi-tenant data centres—such as those operated by AWS and Google Cloud—commonly run user containers inside VMs to enforce stronger security boundaries than containers alone provide [1, 8]. Additionally, VMs provide abstraction over hardware such that resources can be independently partitioned, migrated and scaled without affecting users' view of the system.

Prior research has explored online algorithms for minimising the number of resource-heterogeneous VMs needed to pack incoming containers [17]. Our focus is coarser-grained: we investigate how to place those VMs onto physical hosts to maximise memory page sharing.

In particular, the chapter that follows will justify our choices for modelling the data centre: host memory capacities, guest VM configurations and guest operating systems. We will describe how we generated a dataset of Docker containers and grouped them into VMs. We will explain how to statically approximate memory sharing between different OS and Docker images. Finally, we will show how to represent this sharing using general instances, tree instances and cluster-tree instances.

5.1. Modelling a Data Centre and its Workloads

Modelling our test data to be generalisable to every possible data centres is infeasible. There is great heterogeneity between data centre configurations [24, 27]. Further, as evidenced by the past focus of VM packing research on memory sharing between applications running directly in virtual machines [3, 26], trends can shift rapidly. Modern, containerised workloads may already exhibit different memory-sharing behaviour—e.g., container images often share base layers and higher layers partially override them.

Instead, we are interested in a qualitative approach that captures the structural features of real workloads, such as:

- The relative number of pages that are shareable;
- When a page is shared, the proportion of different guests that share it; and
- The relative size of guests and hosts.

This represents a meaningful increase in applicability compared to randomised datasets. Further, it allows the first direct comparison of general (e.g. Overload and Remove) and purpose-specific approximation algorithms (e.g. Tree-Based Placement).

5.1.1. Choice of Operating Systems

Containers allow users of a data centre to define their applications' view of the system. As a result, user deployments are OS-agnostic. This enables data centre maintainers to standardise OS platform across their VMs, aiming at uniformity and ease of maintenance. In our model data centre, we will assume that this platform is Ubuntu Server. In particular, we will look at long-term maintenance (LTS) releases of Ubuntu Server, which dominate non-LTS releases in cloud environments and account for 95% of Ubuntu installations [4].

On the other hand, homogeneity in OS versioning is rarely achievable for several reasons. Data centre operators regularly upgrade VM operating systems as bug fixes and security patches are released. Notable examples include urgent patches for the Meltdown and Spectre exploits in CPU out-of-order and speculative execution [2]. However, OS upgrades are not applied to all VMs simultaneously, as doing so would risk downtime should the upgrade cause a failure or need to be rolled back. For instance, $N + 1$ redundancy is standard practice: the upgrade system must run redundant instances of each VM to protect service availability while initialising a new, upgraded VM [2]. In our model, we assume a uniform distribution of three minor versions of the latest Ubuntu LTS release present in the data centre: 24.04.1, 24.04.3, and 24.04.6. This is a generous assumption intended to capture some heterogeneity. The real number of different OS versions will depend on the upgrade policy set by the data centre maintainer.

5.1.2. Choice of Container Images

Docker Hub [6] offers a source for common container images. In particular, Docker official images are a subset of images on Docker Hub that are reviewed and selected as high-quality, secure and widely useful [7]: these represent common workloads, from scientific computing and messaging middleware to OS distributions, language runtimes and databases. The week starting 10 February 2025, we pulled the `:latest` tag (or the latest stable release if absent) of each `x86_64` official image with contribution activity in the last two weeks (to filter out out-of-date or deprecated images), numbering 98 images for a dataset broadly representative of modern workloads. The full list is available in Appendix A.1.

The question remains of how to distribute them among VMs. Let us consider two possible approaches.

- To uniformly distribute images among VMs.
- To scale the probability of a given image appearing by its relative number of recent pulls on Docker Hub; this data is publicly available for official images.

The latter approach would be a poor way of representing the distribution of containers in a data centre. A Docker image can have a high pull count not because it often deployed directly, but because it serves as a common base layer for other containers.

In our dataset, the five most frequently pulled images in the week beginning 17 March 2025 were `alpine`, `redis`, `postges`, `python` and `node`, each with 10-15 million pulls. Conversely, our least popular image `api-firewall` has received little more than 60 thousand pulls in its three-year lifetime and roughly 100 in the last week. Pull count alone would suggest that `api-firewall` sees $150,000\times$ fewer deployments than `alpine`.

But this would be a misleading conclusion. First, `alpine` typically sees deployment not as a standalone container, but rather as a base layer in larger containers. It would be incorrect to assume that our data centre contains a container that appears $150,000\times$ more often than another, each time with the exact same code. It is not necessarily untrue that large multi-tenant data centres will see orders of magnitude more `redis` containers than many standalone specialised applications, but it would be inaccurate to quantify this difference using Docker Hub pull counts.

Further complicating matters, a few factors can distort pull counts. `api-firewall` is an example of an image that is hosted under different popular repositories, aside from the official library for which we have direct pull data. Additionally, specific variants of an image may reuse another image from our list, so a pull of that particular variant will also cause a pull of the base image. Because pull counts are cumulative over *all* variants of an image, there is no way to identify this double-counting and apply a correction.

Filtering out base images is also not straightforward. Consider, for example, `python`. It is entirely possible that a bare `python` container is used as a testing environment as part of a CI/CD pipeline, but also as a base layer for a specialised microservice. Both activities are reflected in the pull count, but they are not the same image—they might share some code, but certainly not all.

Our second option is to uniformly distribute the 98 images across VMs. Multiple appearances of the same image simulate multiple instances of a container, as would occur in a real data centre. Some containers may share a common base layer, e.g., several could be based on a particular version of `alpine`, but the number of instances of `alpine` will not be inflated as a result. What we lose with a flat distribution is the possibility that some general-purpose containers—such as `redis`–can appear often in a data centre: all containers are equally represented and treated as specialised applications.

To summarise, we have been able to capture the following features of a real dataset.

- ✓ Representative workload types
- ✓ Representative container sizes
- ✓ Multiple appearance of the same containers
- ✓ Shared base layers
- ✗ Representative ratio of base images to application containers
- ✗ Frequent appearance of particular container types

While the dataset does not reflect the distribution of container instance counts we may see in practice, it does capture many structural memory-sharing characteristics that matter to our purposes.

5.1.3. Choice of Host Capacity

Our hosts will comprise hosts with 64 GB of memory, which modest for hosts in general-purpose data centres but typical in energy-efficient data centres. For instance, Meta aim to exclusively use 64 GB machines [27]. Conversely, Amazon Web Services currently offer 256 GB for its middle-range, general-purpose *bare-metal instances* that which occupy an entire physical machine [24].

Because workload memory requirements vary based on the data flowing through them, we fix each VM to host 10 containers. This number should reflect three facts about VM deployment:

- Small VMs incur disproportionate overhead;
- Large VMs enforce less isolation between containers;
- VMs can only be as large as the underlying physical machine; and
- Even if we fix the dimensions of a VM, the number of containers within it will vary depending on the size of the containers and the data flowing through them at runtime.

Different business scenarios will balance these trade-offs differently, allowing us some freedom to make a choice that is a practical for our purposes. We conservatively estimate 10 containers in each guest VM. Naturally, multiple containers of the same image can appear in a guest.

With 98 distinct container images, using fewer containers per VM allows us to model scenarios with more guests without oversaturating the dataset with appearances of the same containers.

We will further assume, conservatively, that we want up to 5% of the memory of each VM (3.2 GB) to be taken up by instructions rather than data: this is our capacity. While this can also vary widely by workload, it is preferable to err towards a conservative threshold because we are likely overestimating the runtime footprint of OSs (though we cannot quantify this overestimation).

5.2. Extracting Guest OS Pages

Our goal is to statically analyse OS images to extract a set of instruction pages associated with them. As we noted in Chapter 2, page sharing is principally due to shared code, not to shared data.

An approach taken in related work has been to compute a hash over page-sized segments of different operating system images [3, 25]. We can achieve a finer-grained approach while remaining consistent with this methodology. Specifically, we may observe instruction pages associated with the following OS components at runtime:

- The kernel core (e.g. the scheduler);
- Additional kernel modules (e.g. device drivers);
- System processes (e.g. the SSH daemon);
- Bundled shared libraries (e.g. the C standard library)
- Bundled utilities (e.g. `ls`)

Of course, a different proportion of the instructions in each category will be relevant at runtime: much more of the kernel core is likely to be used compared to the array of bundled utilities. As such, it is important to keep in mind that we are almost certainly overestimating the runtime footprint of the OS. Moreover, guests may use different subsets of OS pages, so we might also overestimate sharing—though this effect is likely small given many guests will rely on common functionality like networking, threading and inter-process communication.

Nonetheless, extracting executable OS components represents an improvement over current approaches. To that end, we:

- Unpack Ubuntu 24.04.1, 24.04.3, and 24.04.6 images;
- Unsquash their filesystems;
- Identify all executables, kernel modules and shared libraries;
- Dump their executable segments in raw format using `objcopy`;
- Partition them into 4KB (page-sized) chunks;
- Hash each chunk with SHA-256 to represent the page as a unique integer;
- Remove duplicate chunks, as they be deduplicated *within* each VM by the hypervisor at runtime; and
- Truncate SHA-256 hashes to 64 bits to facilitate later processing. (This causes no hash collisions across the pages of the three operating systems.)

The results are summarised in Table 5.1. We see a roughly similar memory footprint across the three versions, as well as a roughly similar proportion of their pages being eligible for deduplication. Underscoring the benefit of memory deduplication, note that the version with the lowest total footprint after deduplication is also the largest by total pages.

From here, we can compute the possible sharing due to the OS for each pair of Ubuntu versions, summarised in Figure 5.1. Sharing between two page sets is their Jaccard similarity: the proportion of pages in their union that appear in both sets. This is an intuitive metric and consistent with the literature [3, 25].

Ubuntu Version	Components	Distinct / Total Pages	Footprint After Deduplication
20.04.1	~8,350	76,291 / 191,276 (39.9%)	305 MB
20.04.3	~8,400	66,173 / 204,133 (32.4%)	265 MB
20.04.6	~8,500	71,001 / 184,548 (38.5%)	284 MB

Table 5.1.: Summary of our OS instruction page analysis.

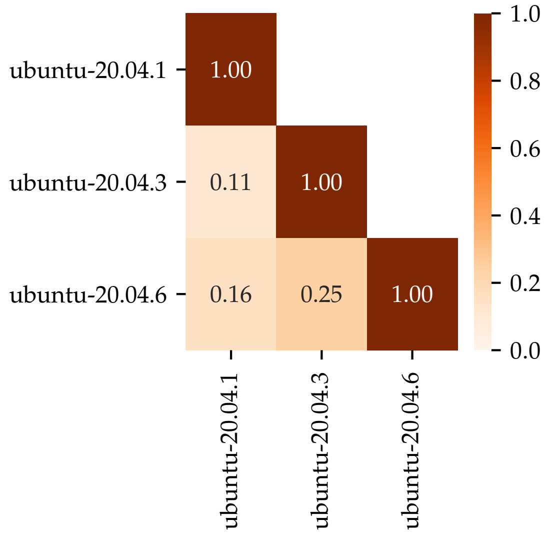

Figure 5.1.: Pairwise Jaccard similarities of the page sets of different Ubuntu 24 versions.

The whole-image analysis of [3] identifies 26% sharing between close but distinct Ubuntu releases: 11.04 and 11.10. We report 13–25% sharing for the same Ubuntu release with different minors. 6.0% of OS pages are shared between all three of our Ubuntu versions.

5.3. Extracting Container Pages

We can analyse Docker images by the same methodology. As we can see in Figure 5.2, pairs images tend to experience either substantial sharing or none at all. The mean sharing potential between the 98 images is 0.0936 (SD = 0.204), showing significant variance.

As we have discussed, instructions might be shared between images due to a common base layer. For instance, the most sharing we saw between different images is for `erlang` and `elixir`, with a Jaccard similarity of ~1.0. Elixir is a variant of the Erlang language and runs in Erlang's BEAM environment [10] and as such, the `elixir` container is based on `erlang` with relatively insignificant differences.

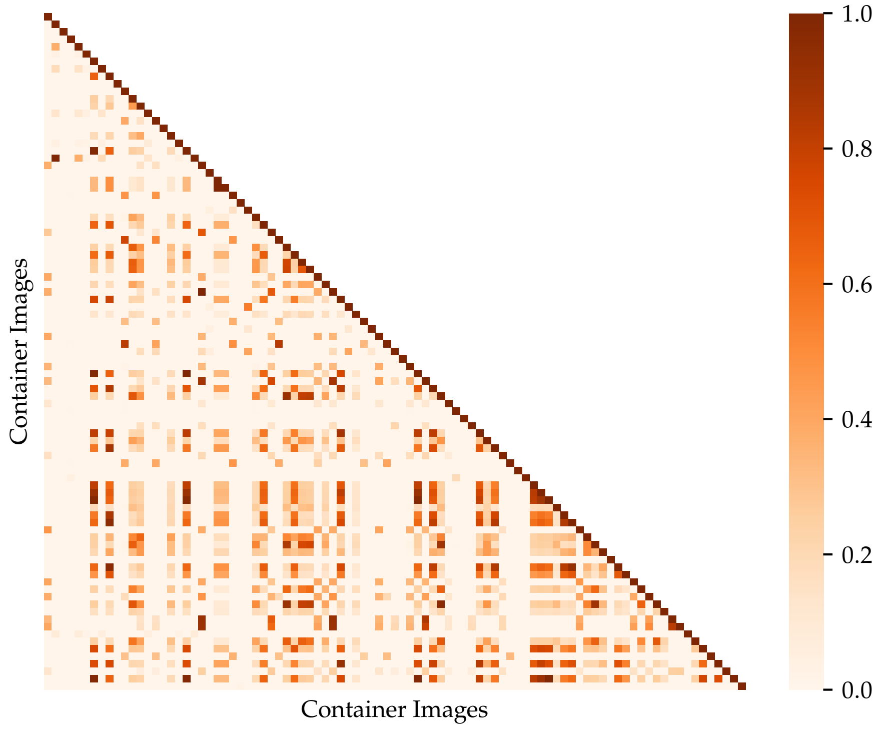

Figure 5.2.: Pairwise Jaccard similarities of the page sets of different containers.

5.4. Generating General Representations

We now have all the tools required to generate general representations of VM Packing instances.

Our host memory size is 64 GB, with a 5% relative capacity for instruction pages that translates to 3.2 GB. We fix the guest number to a value [2, 25, 50, 75, 100, 125, 150, 175, 200] and randomly select one OS page set and 10 container page sets. Figure 5.3 demonstrates that the average guest consumes about 1 GB of runtime memory for its instruction pages compared to the 3.2 GB allotment.

Note that with as few as 25 guests, we are including 250 containers in the instance, so there must be duplicate containers because there are just 98 distinct images. With 200 guests, there are exactly 2000 containers and the probability of some image not being included is $\sim 10^{-9}$. In the larger instances, it is also likely that we include containers for all images.

In order to allow the algorithms to scale to 200 guests in feasibly measurable runtime, we must reduce the size of the page sets while not modifying the sharing properties of our dataset. Therefore, we:

- Compute union of all page sets, OSs and containers together;
- Sample a proportion p of this union; and
- Remove any pages that are not in the sample from the individual page sets.

Sampling with $p = 0.01$, mean sharing between containers is 0.0960 (SD = 0.208), which is very close to 0.0936 (SD = 0.204) before sampling. We were able to reduce the sampling down to $p = 0.004$ while maintaining mean sharing of 0.102 (SD = 0.212). Although slightly more inaccurate, we prefer the $p = 0.004$ sample to allow us to obtain results from more expensive algorithms at the higher guest counts. This negligibly impacts generalisability compared to other limitations of our and of related datasets; for instance, the reliance on static analysis of OS images instead of runtime profiling [3].

Another way we can achieve slightly leaner but equivalent instances is to remove all pages shared between the three Ubuntu versions; these pages will have to be packed in every host anyway, so we can reduce the capacity by their count in advance.

Finally, we generate 10 instances for each guest count.

5.5. Generating Tree and Cluster-Tree Representations

From here, we must turn these instances into tree and cluster-tree representations by deciding how to structure their internal nodes.

The first two layers of the hierarchy can be used as given by [26]. The first is the OS platform. This can be the root, as we have only one OS platform. Conceptually, the root contains all pages shared between Ubuntu versions, though as outlined in Section 5.4, we have removed them and reduced the capacity to obtain leaner instances. The second is the OS version, with three nodes for the pages unique to 24.04.1, 24.04.3 and 24.04.6.

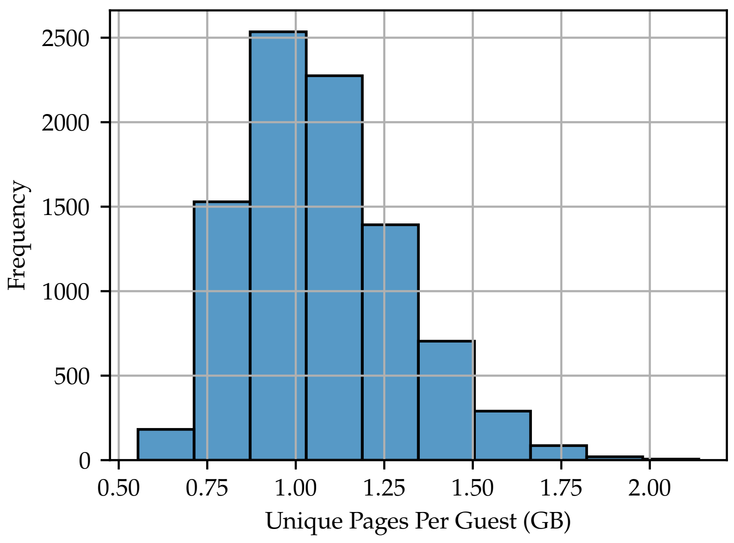

Figure 5.3.: Distribution of the number of unique pages in each guest.

Finally, we must capture sharing due to Docker containers in a third layer. Each OS version receives one node for each of the containers—there is no hierarchy here.

The upshot is that only sharing due to one container can be captured for every guest; one node must be picked from that layer. In order to maximise sharing, we always select the largest (by number of pages) such node when inserting a guest. Of course, this will also mean that larger containers will root larger subtrees. Among instances of 200 guests, the number of containers out of the 98 that have at least one guest in their subtrees is around 25 in our data.

The importance of picking the largest container in a guest is underscored by Figure 5.5, which shows that picking the largest container accounts for 30% (SD = 7.8%) of pages in each guest, on average—significantly more than picking a container at random.

Looking at Figure 5.5, one might be tempted to assign container *pairs* to each node in the final layer. In fact, this was an approach we considered This would increase storage to $O(|G|^2)$ nodes, but account for up to 12% of pages in the container layer on average. However:

- This strategy would have to be adjusted ad hoc according to the number of containers per guest and the global distribution of container sizes. With enough containers per guest, container *triples* might be needed and space complexity rapidly deteriorates.

- More critically, sharing would be captured at a granularity of container pairs. Therefore, unless two guests had the same two largest containers, they would not be added to the same subtree and the instance would miss all container-based sharing between them.

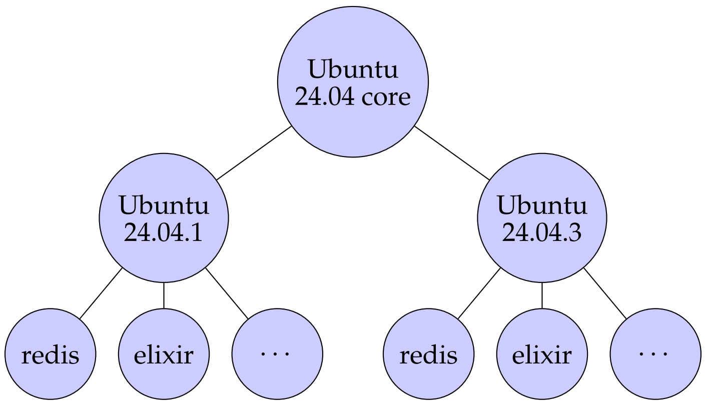

Figure 5.4.: An example tree instance with two internal layers, capturing shared OS versions and containers. Guests (leaves) are omitted.

We therefore take the simpler approach of assigning one container per node.

For cluster trees, there is also a question of how to cluster the nodes in this last layer. Recall that by adding nodes to a cluster, we allow nodes in the cluster's children to have one or more parents in that cluster. For the container layer, that allows the nodes of the layer below—the guests—to descend from several containers from the same cluster. Two parameters need to be tuned: the number of nodes per cluster (or *cluster width*) and strategy we use to decide which nodes to cluster together.

In Section 4.4, we showed that the time complexity of Cluster-Tree-Based DP would be proportional to 4^w if we were to vary the cluster width w. As this cost explodes quickly, we considered cluster widths 2, 3 and 4.

Similarly, we considered two heuristics:

- Randomly cluster containers in groups of size w; and
- Cluster the largest nodes with the smallest ones, so that if $w = 4$, the two largest nodes will be grouped together with the two smallest. The idea is to prevent smaller nodes from receiving no descendants by grouping them with larger nodes.

We evaluated all combinations of cluster tree parameters by computing the proportion of each guest's pages that were captured by its ancestor nodes in the tree. We found the greatest average proportion of pages captured to be $w = 4$ and no different between a random and a largest-smallest grouping strategy. Before removing the pages shared by all Ubuntu versions, the proportion pages captured with random grouping is 58% (SD = 9.0%) and with largest-smallest 58% (SD = 8.8%).

The tree model captures just 43% (SD = 8.8%) of pages in its internal nodes.

5.6. Summary of Methodology: Contributions and Limitations

Our methodology is able to capture much structural detail related to VM Packing instances in containerised environments. In particular:

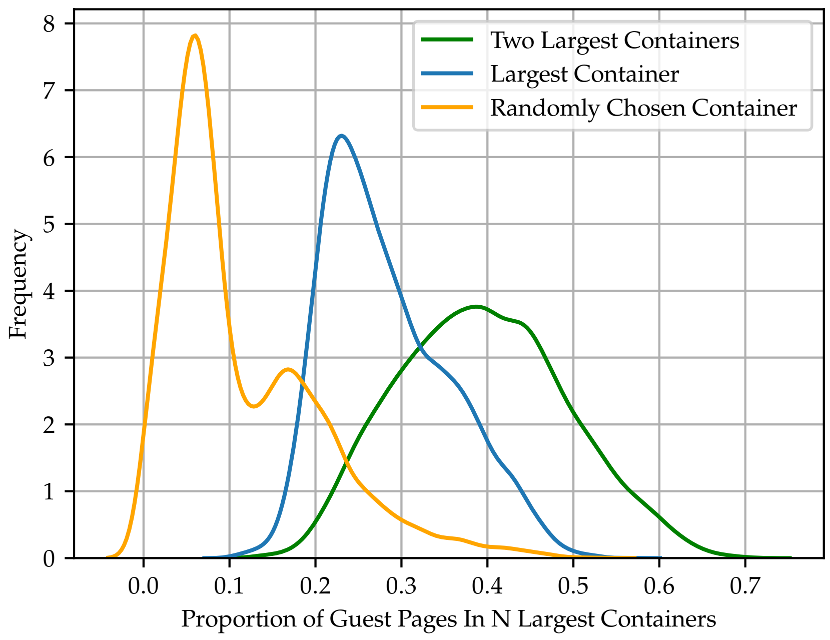

Figure 5.5.: Kernel density estimate of the ratio of the pages of each guest that are also shared by its one or two largest containers. Baseline is randomly chosen container, in orange colour.

- We extract OS memory pages with finer-grained analysis than considering the entire OS image.
- We use a dataset of 98 standard Docker containers to explore the sharing characteristics of containerised workloads.
- We demonstrate how VM Packing instances can be accurately sampled to facilitate comparisons at greater scale.

However, no static analysis of memory characteristics can fully approximate the sharing that occurs at runtime. In particular, depending on the runtime behaviour of containers:

- Only a subset of the instruction pages we have extracted may be used—especially for larger containers that offer more features to developers.
- Even identical images may in practice be divided into different working sets of instruction pages.
- Depending on the space occupied in the guest by data, our assumption of a 5% instruction-page capacity as a proportion of total memory might not hold.

As such, future work will involve runtime memory tracing, similar to the small-scale dataset of [26], in order investigate the extent to which our assumptions—consistent with current literature [3, 25]—hold.

6. Empirical Analysis

We configured each approximation algorithm in three variants: the base algorithm; the base with a decanting post-treatment (where applicable); and the base in cluster tree order with a decanting post-treatment.

We vary the guest count to [2, 25, 50, 75, 100, 125, 150, 175, 200] and generate 10 general instances for each guest count using the process described in Chapter 5.

The tree and cluster-tree instances are assumed to be provided directly to the approximation algorithms. The time taken to construct these instances is not benchmarked; we focus on the time required to solve them.

For each instance, we compare the performance of all algorithms (in terms of the number of hosts per packing) to the baseline of Next Fit, which blindly packs guests without making any optimising decisions.

All algorithms achieve lower average packing sizes (across all guest counts) than the baseline, with the exception of Cluster-Tree-Based DP, which increases packing size by 57% on average, and by 53% on the largest, 200-guest instances. Recall that Cluster-Tree-Based DP packs the largest possible subset of guests into a host based on the sharing captured at the internal nodes of the cluster tree. It then creates a new host and repeats the process. But as the cluster tree only captures 62% of pages—and consequently only a fraction of sharing—it overestimates the size that each guest set occupies on a host. We observe this effect in how many guests are decanted: on the largest, 200-guest instances, decanting reduces the cluster tree packing size by 73% on average. We thus exclude the pathological base variant of Cluster-Tree-Based DP from subsequent discussion and only consider it in conjunction with decanting.

Let us first focus on the largest instances in our dataset—the ten with a guest count of 200. Differences in runtime and packing efficiency between algorithms are most pronounced on these larger, more difficult instances.

The most effective algorithm is Overload and Remove with cluster-tree ordering and decanting. This fully treated variant achieves a mean packing size reduction of 67% (SD = 5.6%). It matches or reduces the best packing given by another algorithm in all 200-guest instances.

Overload and Remove is also the most effective base algorithm, producing an average 56% (SD = 4.6%) improvement over the baseline. The impact of decanting alone is marginal with respect to mean performance and to variance 57% (SD = 4.4%). Instead, it is the combination of cluster-tree ordering and decanting that achieves the best packing efficiency improvement of 67%. This comes with an average runtime that is 364% (SD = 73%) greater than that of Next Fit.

The fastest heuristic was the cluster-tree-ordered variant of First Fit, with a slight average reduction in runtime over the baseline: -5.7% (SD = 3.2%). As the First

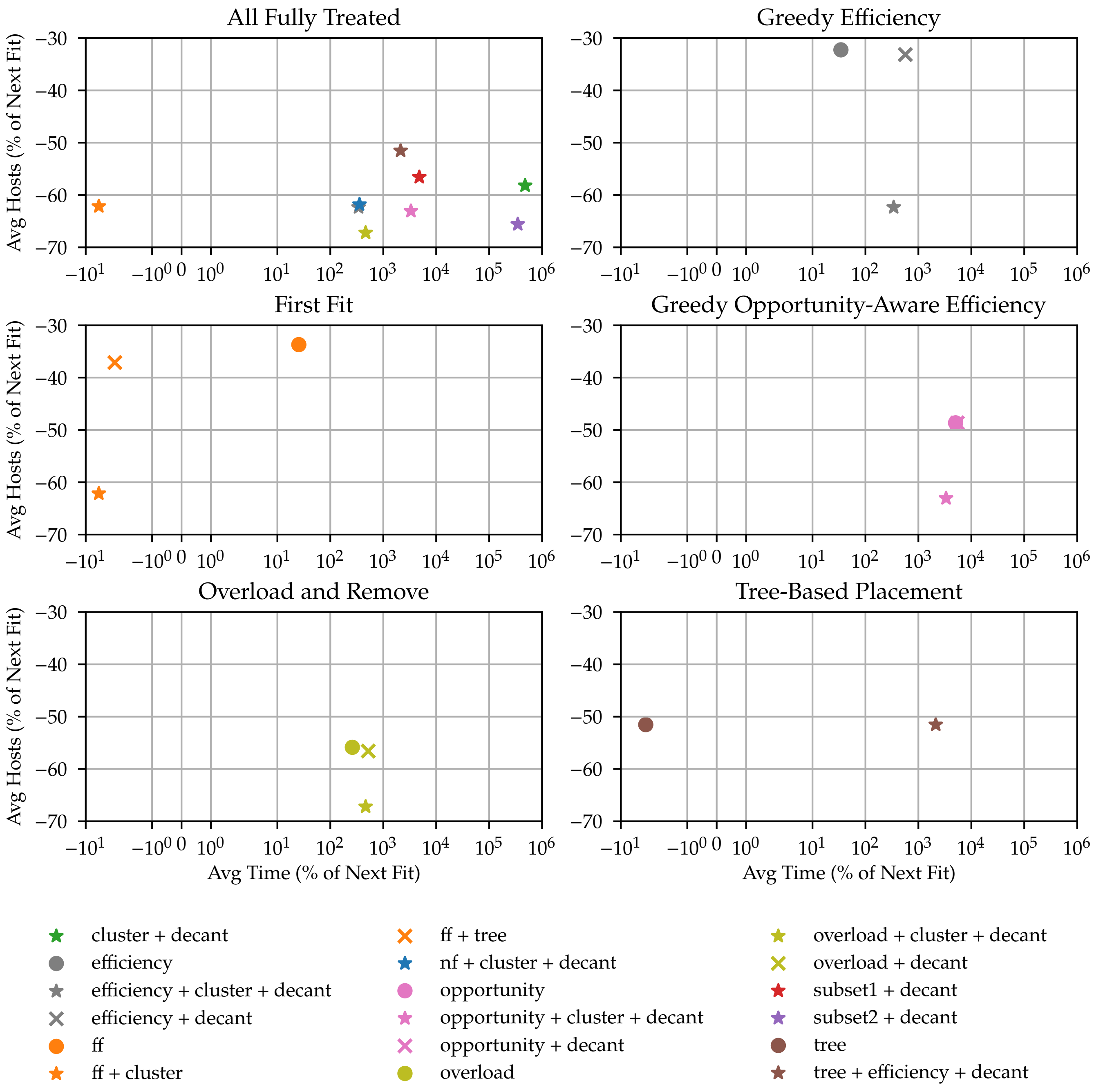

Figure 6.1.: Hosts per packing vs time (both as % of Next Fit for each instance, averaged over all 200-guest instances). Bottom-left is better.
In the legend coding, *+cluster* signifies the addition of a cluster-tree-ordering pre-treatment and *+decant* of the decanting post-treatment. *ff* is First Fit, *nf* is Next Fit and *subset1/2* denote Greedy Subset Efficiency with initial subset sizes 1 and 2, respectively. ⋆ marks the fully-treated variant of each algorithm.

Fit has higher worst-case time complexity than Next Fit, this empirical gap may be attributable to the cluster-tree-ordering making pathological features less common in practice. Achieving a lower host count alone implies reduced overhead. In support of this explanation, we observe that untreated First Fit indeed requires higher runtime than Next Fit.

Considering its speed, cluster-tree-ordered First Fit achieves a competitive average improvement of 62% (SD = 4.8%). Again, we observe a significant benefit from cluster-tree ordering: without it, First Fit achieves only 34% (SD = 5.1%) better packings than the baseline.

Further, tree-ordered First Fit falls short of the cluster-tree-ordered variant. We observed this pattern more generally when comparing tree- and cluster-tree-ordered variants of the same base algorithm, though those trial outcomes are omitted here for brevity. The clustering scheme not only groups the guests by their largest container but also by up to three more within the same cluster. As such, it tends to improve on tree-ordering—and, as in the case of First Fit, by a significant margin.

Greedy Opportunity-Aware Efficiency modestly improves on Greedy Efficiency at a pronounced runtime cost,[1] as expected by the time-complexity analysis of Chapter 3.

Let us now consider performance across different guest counts. Figure 6.2 shows the mean improvement achieved by each algorithm.

At two guests per instance, either both guests fit on the same host or they do not. All algorithms are able to detect when they do and thus produce a minimal packing. However, as instance difficulty increases, the differences between algorithms become more pronounced and the fully-treated Overload and Remove dominates. At a much higher runtime, Greedy Subset Efficiency with initial subset size 2 achieves comparable effectiveness.

We also observe that all algorithms scale consistently across guest counts. [12] report that Greedy Efficiency scales more consistently than Overload and Remove as instance difficulty increases and recommend using the latter on a per-case basis. Although we do not discuss the concept of instance difficulty as defined by [12] in this dissertation, note that that under their definition, difficulty grows with the guest count if the global page set is fixed—as is the case for us. As guest count grows, we do not observe a similar effect regardless of whether an ordering pre-treatment is used.

To summarise, our results indicate that on the containerised-workload dataset:

- The decanting post-treatment of [12] can significantly improve the performance of less effective base algorithms, such as Cluster-Tree-Based DP, from notably underperforming the baseline to improving on it by 58%.
- The decanting post-treatment can modestly improve the performance of more effective base algorithms; e.g., both Overload and Remove and Greedy Efficiency improve over the baseline a net 1% more when post-treated.
- The cluster-tree-ordering pre-treatment can significantly improve the effectiveness

[1]This is for the standard implementation, which omits the caching trick described in Section 3.4.

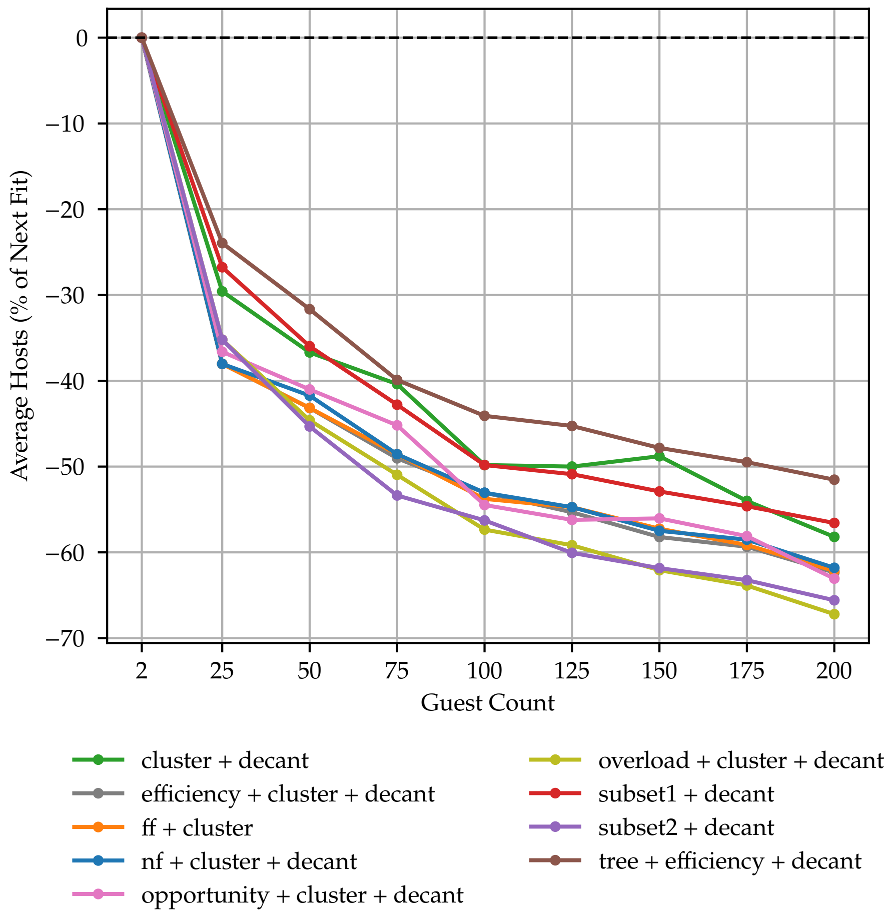

Figure 6.2.: Average host in packing (relative to Next Fit) vs guest count.

of all algorithms. First Fit is a lightweight yet relatively effective option when so pre-treated, achieving the lowest runtime and a 5% increase in packing size from the most effective algorithm.

- The cluster-tree-ordering pre-treatment can optimise the processing order of guests to improve runtime.
- Base algorithms derived by reduction to VM Maximisation can be effective, as in the case of the second-most effective Greedy Subset Efficiency [16, 18] with an initial subset size of 2.
- Compensating for opportunity-unawareness in Greedy Efficiency only marginally improves packing size but increases runtime by a factor of ~150×.
- Tree- and cluster tree-specific algorithms are relatively ineffective given the lower approximating power of tree-based representations on the microservice dataset, in which memory sharing is less hierarchical than assumed in [26]. We capture at most 58% of pages in the internal nodes of the cluster-tree model and 43% in those of the tree model.

7. Notes on Testing and Implementation

The library of approximation algorithms has been developed in C++20 and made publicly available at `https://github.com/smich42/vmpacking`. We hope to continue developing these algorithms in line with new advancements in the literature.

The software was designed with composability in mind. Treatments and algorithms operate on intermediate packings such that they can be combined and validated independently.

Validation is complicated by the fact that the algorithms are intended to produce approximate, not exact solutions. There is no general way of confirming that an approximate algorithm has operated correctly just by inspecting its input and its output. Instead, we took the following validation steps.

- For Overload and Remove and Greedy Efficiency, we used the publicly available randomly generated data provided by [12] and compared against the stated results for each instance.
- We stress-tested all algorithms using large randomly generated instances with the Python scripts under the `scripts` directory. It is important to note that scripts do not guarantee statistical properties of the output test data.
- We introduced assertions throughout our code to verify invariants—e.g., that all nodes in a cluster tree must have nodes in the same parent cluster.
- Manually confirmed expected outputs on a few small and manually traced instances during development.
- Verified that each tree and cluster tree is exactly equivalent to the corresponding general instance, by traversing the trees upwards to obtain the page set for each guest and comparing it to the general instance.

To generate the dataset, we implemented our methodology in interactive Python Jupyter Notebooks. These notebooks are included in the attached code so that the dataset may be reproduced for the current latest Docker images.

8. Evaluation and Critical Appraisal

At the outset of the project, we set the following objectives.

- **Primary Objectives**
  - Implement existing algorithms for VM Packing.
  - Benchmark them on instances approximating real-life server workloads.
  - Compare their solutions to exact algorithms.
- **Secondary Objectives**
  - Provide open, reference implementations.
  - Explore reduction to VM Maximisation.
- **Tertiary Objective**
  - Design worst-case instances to benchmark the algorithms.

We have met all these objectives, albeit with some modifications. Rather than comparing against exact algorithms, we have adopted the baseline of NEXT FIT, which blindly packs instances without optimisation. This allowed us to scale our approximations to higher runtimes without depending on an expensive, exact algorithm as the baseline. We have also not benchmarked our algorithms on worst-case instances. Instead, we designed such pathological instances to formulate proofs of worst-case time complexity, while focusing our empirical analysis on the real-world dataset, confronting a more substantial literature gap.

In particular, we have researched core VM Packing approximations, along with some for related variants that we consolidated into GREEDY OPPORTUNITY-AWARE EFFICIENCY and GREEDY SUBSET EFFICIENCY. We have considered three ways of structuring VM Packing instances, offering renewed discussion of the tree- and cluster-tree models.

Additionally, we have proven an exact reduction to VM Maximisation, as well as that reduction via VM Maximisation to Single-Host VM Maximisation preserves tight approximation bounds.

We have contextualised these results in a practical analysis of containerised data centre environments. While we capture many structural characteristics of such workloads, our results are based on static analysis of operating system and container images. Further, owing to the diversity of data centre configurations, our assumptions regarding host capacity and guest sizes cannot apply to all real-world scenarios. Our main contribution in this respect is an improved methodology for generating such a dataset, for achieving leaner instances while preserving their sharing characteristics and for structuring this data as general, tree and cluster-tree instances. We questioned the suitability of the latter two models to these less hierarchically structured memory-sharing scenarios.

This methodology allowed us to reliably test up to 200-guest instances. Our synthesised approaches improved on even the most effective base algorithms, with cluster-tree-ordering reducing both the packing size and runtime of the algorithms to which it was applied.

With experience now gained in VM packing and the related literature, we can confidently state that this dissertation, along with the accompanying implementation, constitutes a practical and informative companion to those beginning to study the problem.

9. Conclusion

We have investigated the core memory-aware variant of VM Packing, a variant of the NP-Hard Bin Packing problem. Our contribution consists of synthesised approximations, theoretically verified reduction to VM Maximisation, an open-source implementation and a methodology for constructing real-world test datasets for containerised workloads.

The most effective existing approach in the literature was Overload and Remove of [12], which we improved by a cluster-tree-ordering pre-treatment inspired by [26], cutting packing sizes by another 11% compared to the baseline, to 67%. Additionally, First Fit achieved 62% fewer hosts relative to the baseline without increasing runtime.

We found that algorithms based on VM Maximisation, such as Greedy Subset Efficiency ($k = 2$), can offer high accuracy. However, they suffer from high runtime. Our theoretical analysis reveals exponential factors with respect to pre-set parameters (e.g. the initial subset size k). Since this poor runtime is not dominated by the cost of reduction itself, a promising direction for future work is to devise more efficient single-host VM Maximisation approximations.

In practice, the effectiveness of First Fit with the cluster-tree-ordering treatment makes it a competitive candidate for large-scale, performance-critical scheduling decisions in data centres. Therefore, another direction for future work would be to explore how these scheduling decisions can be practically implemented—especially accounting for the kinds of data that operators can to capture about the containers running on their systems.

Finally, the VM Packing literature would benefit from empirical datasets based on runtime memory profiling of real workloads, accounting for potential differences between the instruction pages which are practically used by each container. This opens another avenue for extending this dissertation.

Bibliography

[1] Tim Allclair and Maya Kaczorowski. *Exploring container security: Isolation at different layers of the Kubernetes stack | Google Cloud Blog — cloud.google.com*. https://cloud.google.com/blog/products/gcp/exploring-container-security-isolation-at-different-layers-of-the-kubernetes-stack. May 2018.

[2] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The datacenter as a computer: Designing warehouse-scale machines*. Springer Nature, 2019, pp. 40–42.

[3] Sobir Bazarbayev et al. "Content-based scheduling of virtual machines (VMs) in the cloud". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE. 2013, pp. 93–101.

[4] Canonical. *Ubuntu release cycle | Ubuntu — ubuntu.com*. https://ubuntu.com/about/release-cycle. 2025.

[5] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022, pp. 362–416.

[6] Docker. *Docker Hub Container Image Library — hub.docker.com*. https://hub.docker.com/. 2025.

[7] Docker. *Docker Official Images — docs.docker.com*. https://docs.docker.com/docker-hub/repos/manage/trusted-content/official-images/. 2025.

[8] Amazon ECS. *Fully Managed Container Solution – Amazon Elastic Container Service (Amazon ECS) - Amazon Web Services — aws.amazon.com*. https://aws.amazon.com/ecs/. [Accessed 25-03-2025].

[9] Izik Eidus and Hugh Dickins. *Kernel Samepage Merging; The Linux Kernel documentation — docs.kernel.org*. https://docs.kernel.org/admin-guide/mm/ksm.html. Nov. 2009.

[10] Elixir. *The Elixir programming language — elixir-lang.org*. https://elixir-lang.org/. 2025.

[11] Lisa Fleischer et al. "Tight approximation algorithms for maximum general assignment problems". In: *SODA*. Vol. 6. Citeseer. 2006, pp. 611–620.

[12] Aristide Grange, Imed Kacem, and Sébastien Martin. "Algorithms for the bin packing problem with overlapping items". In: *Computers & Industrial Engineering* 115 (2018), pp. 331–341.

[13] Lifeng Guo, Changhong Lu, and Guanlin Wu. "Approximation algorithms for a virtual machine allocation problem with finite types". In: *Information Processing Letters* 180 (2023), p. 106339.

[14] KR Jayaram et al. “An empirical analysis of similarity in virtual machine images”. In: *Proceedings of the Middleware 2011 Industry Track Workshop*. 2011, pp. 1–6.

[15] Mohamed Amine Kaaouache and Sadok Bouamama. “Solving bin packing problem with a hybrid genetic algorithm for VM placement in cloud”. In: *Procedia Computer Science* 60 (2015), pp. 1061–1069.

[16] Huixi Li et al. “Page-sharing-based virtual machine packing with multi-resource constraints to reduce network traffic in migration for clouds”. In: *Future Generation Computer Systems* 96 (2019), pp. 462–471.

[17] Matteo Nardelli, Christoph Hochreiner, and Stefan Schulte. “Elastic provisioning of virtual machines for container deployment”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. 2017, pp. 5–10.

[18] Safraz Rampersaud and Daniel Grosu. “An approximation algorithm for sharing-aware virtual machine revenue maximization”. In: *IEEE Transactions on Services Computing* 14.1 (2017), pp. 1–15.

[19] Safraz Rampersaud and Daniel Grosu. “Sharing-aware online algorithms for virtual machine packing in cloud environments”. In: *2015 IEEE 8th International Conference on Cloud Computing*. IEEE. 2015, pp. 718–725.

[20] Safraz Rampersaud and Daniel Grosu. “Sharing-aware online virtual machine packing in heterogeneous resource clouds”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.7 (2016), pp. 2046–2059.

[21] Charles Reiss, John Wilkes, and Joseph L Hellerstein. “Google cluster-usage traces: format+ schema”. In: *Google Inc., White Paper* 1 (2011), pp. 1–14.

[22] Tasneem Salah et al. “Performance comparison between container-based and VM-based services”. In: *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE. 2017, pp. 185–190.

[23] Steven S Seiden. “On the online bin packing problem”. In: *Journal of the ACM (JACM)* 49.5 (2002), pp. 640–671.

[24] Amazon Web Services. *Cloud Compute Instances – Amazon EC2 Instance Types – AWS — aws.amazon.com*. https://aws.amazon.com/ec2/instance-types/. 2025.

[25] Jianbo Shao and Junbin Liang. “Joint Optimization of Memory Sharing and Communication Distance for Virtual Machine Instantiation in Cloudlet Networks”. In: *Electronics* 12.20 (2023), p. 4205.

[26] Michael Sindelar, Ramesh K Sitaraman, and Prashant Shenoy. “Sharing-aware algorithms for virtual machine colocation”. In: *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. 2011, pp. 367–378.

[27] Chunqiang Tang et al. “Twine: A unified cluster management system for shared infrastructure”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 787–803.

[28] Timothy Wood et al. “Memory buddies: exploiting page sharing for smart colocation in virtualized data centers”. In: *ACM SIGOPS Operating Systems Review* 43.3 (2009), pp. 27–36.

A. Appendix

A.1. List of Container Images

clojure
openjdk
joomla
oraclelinux:9
elixir
gradle
docker
erlang
xwiki
tomcat
convertigo
mongo
websphere-liberty
geonetwork
bash
golang
postgres
drupal
redmine
odoo
liquibase
crate
maven
flink
aerospike:ee-8.0.0.3
rabbitmq
nginx
hylang
caddy
ruby
nextcloud
matomo
friendica
wordpress
composer
phpmyadmin
postfixadmin
python
redis
swift
spiped
registry
traefik
php
fluentd
ghost
open-liberty
yourls
tomee
amazoncorretto
mediawiki
mysql
nats
unit
jetty
ibm-semeru-runtimes:open-8u442-b06-jre-noble
monica
node
mariadb
znc
telegraf
bonita
rust
groovy
rakudo-star
varnish
arangodb
haproxy
influxdb
memcached
haxe
eclipse-temurin
httpd
api-firewall
eggdrop
amazonlinux
irssi
kapacitor
eclipse-mosquitto
chronograf
sapmachine
kong
julia
alpine
backdrop
neo4j
buildpack-deps
pypy
mageia
dart
jruby
logstash:8.17.2
elasticsearch:8.17.2
kibana:8.17.2
cassandra
swipl
sonarqube
satosa

Version is `:latest` on the week starting 10 February 2025 (except where specified, due the `:latest` tag being sometimes undefined—in those cases we identified the most recent stable release).

A.2. Instructions for Use

Please consult `README.md` in the attached solution. Alternatively, visit `https://github.com/smich42/vmpacking/blob/main/README.md`.

A.3. Complete Trial Outcomes

Please consult `results.json` in the attached solution. Each entry is labelled with an instance identifier and guest count, and includes the runtime and host count produced by each algorithm.

A.4. Complete Test Instances

Please consult `instances.json` for the allocation of operating systems and container images to instances.