

Formulation and heuristic for the container image reconstruction problem*

Oscar dos Santos Esgalha

Instituto de Computação - UNICAMP

Av. Albert Einstein, 1251, Cidade Universitária, 13083-852, Campinas, SP, Brasil

oscaresgalha@gmail.com

Celso Cavellucci

Instituto de Computação - UNICAMP

Av. Albert Einstein, 1251, Cidade Universitária, 13083-852, Campinas, SP, Brasil

celsocv@ic.unicamp.br

Fábio Luiz Usberti

Instituto de Computação - UNICAMP

Av. Albert Einstein, 1251, Cidade Universitária, 13083-852, Campinas, SP, Brasil

fusberti@ic.unicamp.br

ABSTRACT

Using container images is a popular technique to share and deploy software by packaging it with all its dependencies. Software deployment with containers was an important enabler for cloud-native computing and changed the way software development teams deliver software. Docker and other tools use content-addressable storage to share identical layers across images and reduce storage usage. However, in practice, some layers can be very similar but not equal, which reduces potential storage savings. In this article, we propose the reconstruction of sets of container images to increase layers similarity and optimize storage usage. We formulate the image reconstruction problem as an integer linear model and propose an off-line constructive heuristic to optimize storage usage within the context of a software development team. We analyze our solution with 4 open-source applications with multiple containers and compare the results with a solver.

KEYWORDS. Container. Docker. Storage.

OR in Telecommunications and Information Systems; Combinatorial Optimization

*This work was supported by FAPESP (2015/11937-9).

1. Introduction

Some events in the 2010s had a great impact on how technology teams package, share, and run their applications. With the release of the first version of Docker in 2013 [Barbier, 2014] building containers to package and run software became more accessible to software development teams. In 2015 the Linux Foundation announced the formation of the Cloud Native Computing Foundation (CNCF) with multiple organizations as founding members [Cloer, 2015], the foundation's goal is to advance the state-of-the-art for building cloud-native applications and services, with cloud-native defined as application or services that are container-packaged. Some years later, in 2022, CNCF declared that cloud-native software development is "the new normal" based on a survey with more than 2000 participants [CNCF, 2023].

A container image contains an application packaged with all its dependencies, and almost any Linux server can run a container based on that image [Noyes, 2013]. A container image is defined by a *manifest* that contains metadata describing the image and a list of layers; each layer is compressed and contains a subset of the files in the image [Zhao et al., 2019]. That is, what determines which files are available in a container at runtime is the list of layers; the image itself doesn't directly determine their files. In practice, image sets present duplicated files in distinct layers, even when those images are crafted by experienced professionals.

In this article, we propose the reconstruction of layers from a set of images to optimize the total storage usage. Given a set of images, their layers are analyzed to determine which files must be available at runtime. With that information new layers are built, trying to minimize the number of duplicate files in distinct layers and therefore the total storage usage. To evaluate our results we chose some open-source projects from *GitHub*^{1,2,3,4} that uses microservices architecture (MSA) and are composed of multiple containers.

We defined this paper objective using the Goal Question Metric [Van Solingen et al., 2002] template as follows: **analyze** heuristics usage to reconstruct container images **for the purpose of** evaluate their performance **with respect to** storage savings **from the viewpoint of** practitioners and researchers **in the context of** a team that develops and operates cloud-native software.

2. Background

Docker became the *de facto* standard to build container images [Reis et al., 2021]. Developers write a specification called *Dockerfile* to create container images with Docker. Figure 1 shows an example of how instructions in a Dockerfile shape the resulting image. Some instructions add new layers, with their specific files changed, and other instructions update the image metadata defined in the *JSON* manifest. Each layer is uniquely identified with a SHA256 digest. The *FROM* instruction inherits layers from a parent image, making it easier to extend existing images.

The following components are present in the lifecycle of container distributed software:

- *Registry*: a service that implements *APIs* responsible for receiving, storing, and distributing image manifests and layers.
- *Builder*: an application that can build a container image (its manifest and its layers) and push the manifest and layers to a container registry.
- *Runtime*: an application that can interpret an image manifest and mount the layers to run the packaged container image. It can pull image manifests and their layers from container registries.

¹<https://github.com/GoogleCloudPlatform/microservices-demo>

²<https://github.com/thingsboard/thingsboard>

³<https://github.com/spring-petclinic/spring-petclinic-microservices>

⁴<https://github.com/travis-ci/travis-ci>

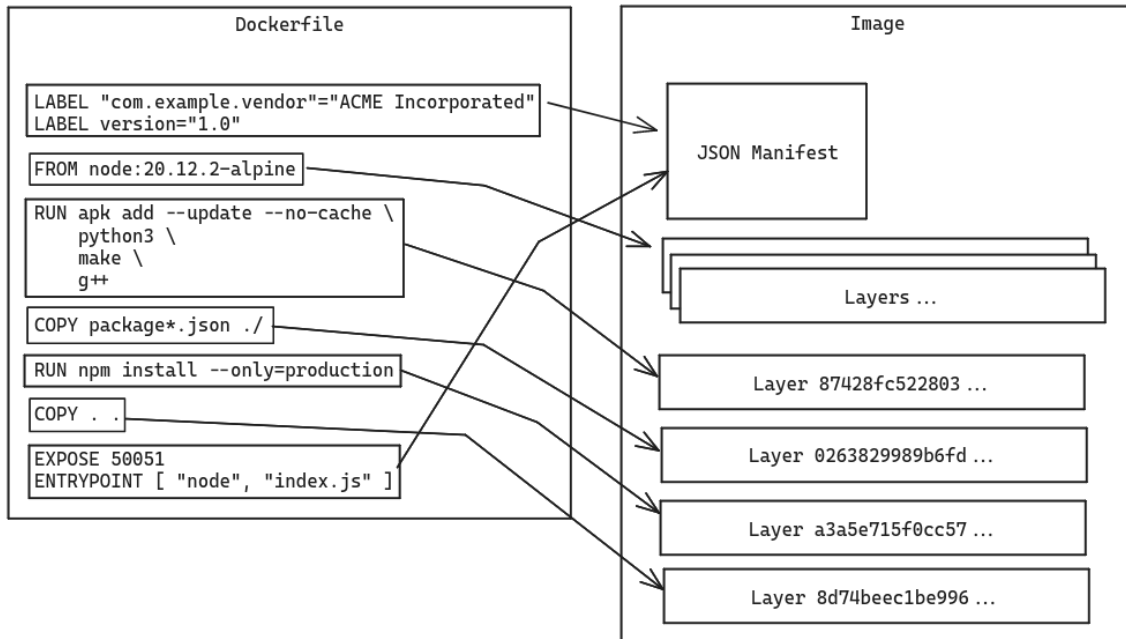


Figure 1: Example of a Dockerfile and its correspondent image.

The implementation of those components follows specifications defined by the *Open Container Initiative*^{5,6,7}. This layer-based organization allows those components to optimize some operations. They use a content-addressable storage scheme to store and transmit a single copy of each unique layer [Skourtis et al., 2019]. Identical layers are efficiently shared across distinct images. That is, a *builder* doesn't push layers that are already present in a *registry*, likewise, a *runtime* doesn't pull layers available in the local storage. A *runtime* usually leverages *union mounting* to mount layers into a single file system tree when starting a container [Zhao et al., 2020]. This allows running multiple containers from the same image without multiplying the storage cost.

In practice, developers build container images independently, hence some layers can share a large number of files but aren't identical and must be stored independently [Skourtis et al., 2019]. Multiple studies identified file redundancy across layers. Skourtis et al. [2019] found 80% duplicate files analysing a 10.000 image set. Li et al. [2022] analyzed the similarity of 2.200 images and found 80% identical files. Zhao et al. [2019] analyzed 167TB of uncompressed images and discovered that layers are rarely shared, they also detected that only 3% of the files were unique.

3. Related work

There are some studies exploring how to leverage file redundancy across layers for potential storage savings. These studies try to solve the problem from distinct perspectives. Zhao et al. [2020], for instance, proposed an alternative architecture for registries that perform file deduplication on the fly. Some studies, however, already explored image reconstruction.

Skourtis et al. [2019] explored reconstructing images for the first time with the goal of optimizing storage costs. The authors considered the storage, network traffic, and operational cost to run images with multiple layers as optimization goals. The proposed algorithm is greedy and

⁵<https://github.com/opencontainers/image-spec>

⁶<https://github.com/opencontainers/distribution-spec>

⁷<https://github.com/opencontainers/runtime-spec>

ignores existing layers: it begins with an empty set of layers and, for each file, iterates through images that require it, picking a layer generated in previous iterations or creating a new layer with that file. Future works referenced this heuristic as *Greedy Offline Image Reconstruction Algorithm* (GOIRA) and can use it as a baseline to compare other image reconstruction techniques. The authors analyzed the 100 most popular *Docker Hub*⁸ images. In the analysis, they found that eliminating all redundancy could save more than three times the storage requirement for those images, and their proposed solution obtained 2.3 times savings. The study however doesn't explicitly validate if the reconstructed images were consistent with the original images, that is, if the new images work like the originals; it also doesn't reveal the required computing time to achieve the 2.3 times storage savings.

Li et al. [2022] created the *Similarity-Aware Online Image Reconstruction Algorithm* (SOIRA). The authors noticed that images within the same repository present more similarity. A repository usually stores distinct versions of the same software. One can expect that few changes can exist between two versions of the same software and that they are more similar than two distinct applications. To confirm this, the study computed the Jaccard index between layers for the same repository (intra-repository) and from distinct repositories (inter-repository). The results confirmed that there is greater similarity between images from the same repository. With this observation, they proposed an online solution: SOIRA reorganizes only the last layers of a new image, trying to maximize the re-utilization of layers available in the repository, generated in previous builds of the same software. The study reported 10% storage savings with their solution, against 1.3% savings from GOIRA. This study reported that for an image set with 1.250 images, the algorithm ran in less than 0.05 seconds more than 80% of the times. However, there are no details about the computing environment where those tests were performed.

Jiang [2023] also made a study that develops an image reconstruction algorithm. Their solution is called *Dynamic Programming-Based Offline Image Reconstruction Algorithm* (DPOIRA). It uses dynamic programming to reorganize images of a single repository with the goal of minimizing the access time for a random file within the running container. In other words, it reorganizes image layers from distinct versions of the same software to achieve faster disk read operations without compromising storage in the registry. This is possible because using union mounting adds some overhead to searching a file across distinct layers. In contrast, this overhead wouldn't exist in a regular file system tree. In the problem formulation, they introduce the storage cost as a constraint $S(L) \leq M$ (the storage cost S from the repository L must be lower than M). Nevertheless, the study reported that DPOIRA achieved 12.6% storage savings in a synthetic dataset, and 7.4% storage savings in a real software repository. There are no details about the computation time required to achieve those results.

The objective of this paper is to discover if a team that develops and operates cloud-native software could optimize image reconstruction to reduce their costs with container image storage. Since a team can generate multiple image versions every day, an acceptable solution must run in a couple of seconds or minutes, not hours. Another important constraint is that the resulting reconstructed images must be consistent with the original and respect runtime and registry restrictions: a container image can have a maximum of 125 layers⁹. Since our goal is to build a solution for software development teams, that solution must find good solutions for image sets with dozens of images.

⁸<https://hub.docker.com/>

⁹https://github.com/moby/moby/blob/v26.1.0/layer/layer_store.go#L24-L29

4. Problem formulation

Given a set of images $i \in \mathbb{N}$ ($i = 1, \dots, N_i$), where N_i is the number of images, each containing an image layer set and their required files. With each file's absolute path and their content digest we can build a set with the unique files $f \in \mathbb{N}$ ($f = 1, \dots, N_f$) where N_f is the number of unique files for that image set. We also have $w_f \in \mathbb{N}$ where w_f is the size in bytes of the file f . With the image set and the unique files set, we can define $a_{if} \in \mathbb{B}$ that represents if a file f is present 1 or absent 0 from the image i .

Our goal is to build new layers and group them towards the reconstructed images minimizing the total storage cost of those new layers. The decision variable $x_{fli} \in \mathbb{B}$ indicates if a file f is present in the image i through the layer l , $y_{li} \in \mathbb{B}$ selects if the layer l is present in the image i , $z_{fl} \in \mathbb{B}$ determines if the file f is present in the layer l , and $n_{fi} \in \mathbb{Z}^+$ is the number of layers that contains file f in the image i . The problem is defined as the following integer linear model:

$$Z = \min \sum_{f=1}^{N_f} \sum_{l=1}^{N_l} z_{fl} \cdot w_f \quad (1)$$

$$\sum_{l=1}^{N_l} x_{fli} = n_{fi} \cdot a_{fi} \quad \forall f, \forall i \quad (2)$$

$$n_{fi} \geq a_{fi} \quad \forall f, \forall i \quad (3)$$

$$y_{li} \geq x_{fli} \quad \forall f, \forall l, \forall i \quad (4)$$

$$z_{fl} \geq x_{fli} \quad \forall f, \forall l, \forall i \quad (5)$$

$$y_{li} + z_{fl} \leq 1 + a_{fi} \quad \forall f, \forall l, \forall i \quad (6)$$

$$\sum_{l=1}^{N_l} y_{li} \leq 125 \quad \forall i \quad (7)$$

The objective function (1) minimizes the total layers storage cost. Since the layers are persisted on disk only once, it doesn't matter how many images use the same layer. Files present in more than one layer increase that cost, which represents the redundant file issues observed in practice. Constraint (2) represents the file, layer, and image distribution: the input variable a_{fi} ensures that x_{fli} only distributes the files that each image needs; the decision variable n_{fi} gives the possibility to duplicate a file across layers which might be undesirable for the objective but necessary for the solution feasibility. Constraint (3) states that if an image i has a file f , then file f must appear in the new image i at least once. Constraint (4) states that if a file f is present in the image i through the layer l , then the layer l must be present in the image i . Likewise, constraint (5) states that if the file f is present in the image i through the layer l , then the file f must be present in the layer l . Constraint (6) ensures that a layer l can be present in an image i and the file f can be present in the same layer l only if $a_{fi} = 1$, that is if the image i contains the file f . Finally, constraint (7) represents the runtime restriction where images can have at most 125 layers. The number of variables and the number of constraints in the model are both bounded by $O(N_f \cdot N_l \cdot N_i)$. Despite the container image reconstruction problem having features of packing

problems, it is still an open question whether this problem admits a polynomial solution or if it is NP-hard.

There is a visual representation of the problem input and output in Figure 2. In this example, the input at Figure 2a is a set of two images. Image 1 has the files F1, F2, F3, F4, F5, F6, and F7 distributed across 3 layers. The first layer of Image 1, for instance, contains F1, F2, and F3. Image 2 has the files F1, F2, F3, F4, F8, and F9 distributed across 3 layers. Note that despite both images having files from F1 up to F4, there are no identical layers across both images, therefore those files are stored twice. The output in Figure 2b illustrates the reconstructed image set with an optimal arrangement of layers. Both Image 1 and Image 2 still contain the same set of files from the original ones, but the file arrangement across layers has changed. The first layer from both images contains all the common files. In this setting, files F1 up to F4 are stored only once, because they are contained in an identical layer shared between both images. The second layer of both images contains their unique files and are also stored only once. In practice, a set with up to 10 images has tens of thousands of files with varying degrees of similarity. Some examples will be presented in Section 6.

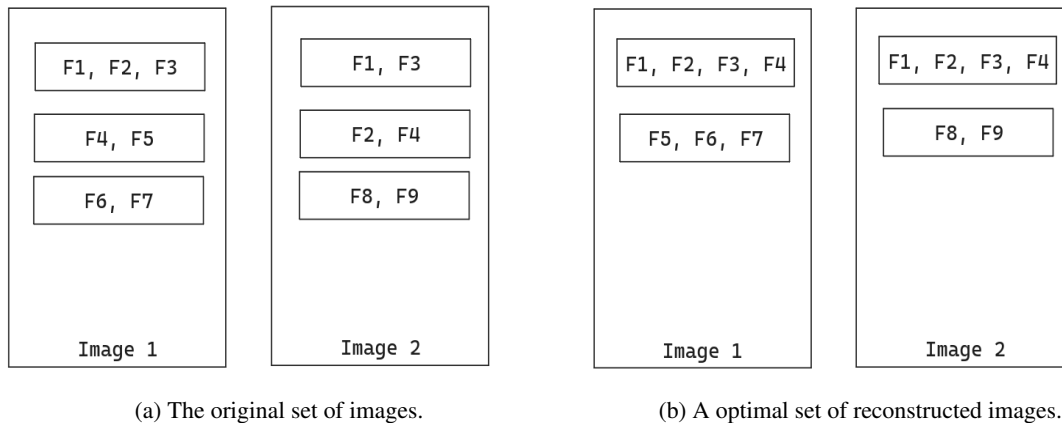


Figure 2: The input and output for the image reconstruction problem.

5. Constructive heuristic for image reconstruction

We propose a constructive heuristic to build new layers and optimize storage usage to solve the image reconstruction problem. The pseudocode is shown in Algorithm 1. The input is similar to the one of the integer linear model: a_{fi} is a matrix that determines if a file f must be present in the image i , w_f an array with file sizes in bytes, and N_f that represents the unique files count. The output x_{ilf} is a matrix that maps if layer l is present at the image i and which files f are present at each layer l .

The heuristic is divided into two main phases: the first builds layers with unique files for each image (up to line 7); the second phase iterates through each file and decides if it should use a layer created in a previous iteration or create a new layer. Before iterating through the files, they are sorted by frequency and file size (line 9), that is, files with greater deduplication potential and larger storage size are evaluated first. The decision between reusing a previous layer or creating a new one is as follows: if there is a layer with the exact same images that require the new file, that layer is re-utilized, otherwise a new layer is created (lines 13 up to 17). If a new layer is required, the maximum layers per image restriction is evaluated to build consistent images (lines 19 to 22).

To illustrate the heuristics usage, take the example instance from Figure 2 and assume all files have the same 16 bytes of size. Note that since those images don't share any identical layer,

Algorithm 1 Constructive Heuristic

Input a_{fi}, w_f, N_f

Output x_{ilf}

```
1: procedure CONSTRUCTIVE HEURISTIC
2:
3:    $x_{ilf} \leftarrow \{\emptyset\}$ 
4:
5:   for each image  $i$  do
6:      $UniqueFor \leftarrow$  subset of  $f$  with files that are unique for  $i$ 
7:      $x_i \leftarrow$  a layer  $l$  with  $UniqueFor$ 
8:
9:   sort  $f$  in descending order by frequency in  $a_{if}$ , resolve ties with  $w_f$ 
10:  for each file  $f$  do
11:    if  $f$  is present in only one image then
12:      continue
13:     $InterestedImages \leftarrow$  subset of  $i$  with images that require  $f$ 
14:    if a layer  $l$  associated to the exact same images as  $InterestedImages$  exists then
15:       $x_{il} \leftarrow f$  ▷ Adds the  $f$  file at the layer  $l$  of each image  $i$ 
16:    else
17:       $NewLayer \leftarrow$  a new layer  $l$  with  $f$ 
18:      for each image  $i$  of  $InterestedImages$  do
19:        if image  $i$  has less then 125 layers then
20:           $x_i \leftarrow NewLayer$ 
21:        else
22:          Add  $f$  to the layer with unique files of the image  $i$ 
```

the total storage size is the sum of each file in bytes: 112 bytes (7 files) for Image 1 and 96 bytes (6 files) for Image 2, thus 208 bytes in total.

First, we begin with an empty x_{ilf} (line 3). In the first phase, for each image i we can check the input variable a_{fi} and verify if a given file f is present only in that image i , that subset with unique files creates a unique layer for each image (lines 5 to 7). In our example from Figure 2a, we can detect that files F5, F6, and F7 are present only in Image 1, thus we can create a layer that contains those exact files and are present only in Image 1. You can see those layers with unique files for each image in 2b. The same process is repeated for files F8 and F9 in Image 2.

Now we sort files by their frequency in a_{fi} and resolve ties with w_f (line 9), in our example frequency for F1, F2, F3, and F4 is 2, because those files are present both in Image 1 and Image 2. The other file frequency is 1. Now take F1 as an example: it is present in 2 images, therefore we skip the lines 11-12 conditional. Checking a_{fi} we compute a list of images that need F1, in this case both Image 1 and Image 2 (line 13). There is no layer shared only between Image 1 and Image 2 yet, therefore we go to the else clause (lines 16-17). We create a new layer with F1 and associate that layer in Image 1 and Image 2 (lines 17-20). When we analyze files F2, F3, and F4, there is a layer shared only between Image 1 and Image 2 (created in the iteration with F1), therefore it enters the conditional from line 14 and updates that layer (line 15). The other files (F5, F6, F7, F8, and F9) are present only in one image and were already inserted in an exclusive layer, in that second loop they are simply skipped because of the conditional from lines 11-12.

The result is the same from Figure 2b, now we have a layer with 4 files that is identical in both images. That shared layer has 64 bytes, and the unique layers for Image 1 with 3 files and Image 2 with 2 files have respectively 48 bytes and 32 bytes, therefore the new reconstructed images weigh 144 bytes in total, against 208 bytes from the original configuration.

6. Computational Experiments

To evaluate the heuristic performance the following aspects were considered: running time and storage savings. The running time is important because the objective is to make a solution that is practical for daily usage in a team that generates multiple images per day, thus, it should run in a few seconds or minutes at most. Another constraint related to the daily usage goal is the required hardware to run the solution: it shouldn't require too much computational resources to run, otherwise, the storage savings won't be worth it.

To ensure that the results for each instance are comparable, all the computing experiments were executed at the same PC with an 11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz 8 cores processor, 16 GB of RAM and 4 GB of swap memory. We also state those configurations for the purpose of reproducibility.

Using Dockerfile present at some repositories from GitHub we build a set with 14 test instances from real container images. Two of the selected repositories are demonstrations, reference architectures, or examples of a framework usage.^{10,11} The other repositories contain real applications developed with microservices and available as open-source projects.^{12,13} The 14 test instances were created combining some of those projects container images, only intra-project combinations were made. These instances are listed in Table 1. The first column is the instance number, the Images column states the total images evaluated, the Total Files column states the total number of files counting duplicates across distinct images, Unique Files states the total number of unique files for

¹⁰<https://github.com/GoogleCloudPlatform/microservices-demo>

¹¹<https://github.com/spring-petclinic/spring-petclinic-microservices>

¹²<https://github.com/thingsboard/thingsboard>

¹³<https://github.com/travis-ci/travis-ci>

Table 1: Test instances.

No	Images	Total Files	Unique Files	Total Size (Bytes)	Required Size (Bytes)
1	2	2307	2250	341790487	341481921
2	2	10966	10624	500370111	499814398
3	3	26017	12738	735282934	349316843
4	2	48385	24748	487437058	251770321
5	11	94252	49876	2083763917	1300244422
6	6	30130	5104	5359010191	1497998644
7	9	67380	31105	6860021470	2039456399
8	2	11295	5698	915026599	473628734
9	3	17201	5777	1483317591	495305285
10	7	39957	5952	3335393497	528773597
11	2	37647	26657	1225696617	771257716
12	3	50426	29499	1513689416	846470173
13	4	74127	40147	2238349918	1096644189
14	5	74464	40354	2291504220	1149601993

that image set, Total Size is the total uncompressed size in bytes used by all files including duplicates, and the Required Size is how much uncompressed storage would be required if all duplication can be removed. Note that the Required Size represents the lower bound, but it may be infeasible due to the problem constraints.

Instances 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, and 14 combine multiple images that use the same programming language and present great potential for storage savings. Instance 5 uses images with distinct programming languages and is representative of potential savings for a complete heterogeneous application. Instances 1 and 2 combine images with distinct programming languages, where the potential savings are smaller.

We implemented the model presented at Section 4 to run with the *Gurobi* solver¹⁴ as a baseline. Using a solver for this problem didn't prove practical. It was capable of finding the optimal solution for instance 1 in 2 minutes. The equipment used for testing was unable to run the solver for other instances, from instance 2 upwards all the available RAM was not enough to find any solution, that is more than 20 GB of RAM is required to find a solution for a set with 2 images and 5104 unique files.

Tables 2 and 3 present the results achieved with the constructive heuristic presented in Section 5. The constructive heuristic was programmed with Rust 1.79¹⁵. In Table 2, the first column is the instance number, column Z^* states the solution found with the heuristic, column Lower Bound states the lower bound value if all file redundancy were eliminated (which may be infeasible), column Savings presents the reduced storage percentage achieved with the heuristic and, column Lower Bound Savings presents how much storage can be saved if all file redundancy is eliminated. Table 3 displays the execution time in milliseconds for each instance. A column with the unique file count is also present to show that execution time increases moderately with the unique file count.

The results available in Tables 2 and 3 show that using the proposed constructive heuristic is a viable and practical solution for daily usage. It was able to reduce storage usage by 24% for an

¹⁴<https://www.gurobi.com/>

¹⁵<https://www.rust-lang.org/>

Table 2: Solutions for the constructive heuristic.

Instance	Z^* (Bytes)	Lower Bound (Bytes)	Savings	Lower Bound Savings
1	341481921	341481921	0.09%	0.09%
2	499814398	499814398	0.11%	0.11%
3	479407146	349316843	34.79%	52.49%
4	251770321	251770321	48.34%	48.34%
5	1579680317	1300244422	24.19%	37.60%
6	2270039259	1497998644	57.64%	72.60%
7	2812678158	2039456399	58.99%	70.27%
8	473628734	473628734	48.23%	48.23%
9	495305285	495305285	66.60%	66.60%
10	2331037057	528773597	30.11%	84.14%
11	771257716	771257716	37.07%	37.07%
12	1030199894	846470173	31.94%	44.07%
13	1743472196	1096644189	22.10%	51.00%
14	1796626498	1149601993	21.59%	49.83%

Table 3: Execution time for the constructive heuristic.

Instance	Unique Files	Execution time (ms)
1	2250	2.39
2	10624	16.17
3	12738	40.12
4	24748	126.10
5	49876	265.81
6	5104	21.69
7	31105	122.58
8	5698	16.51
9	5777	19.81
10	5952	31.71
11	26657	76.95
12	29499	111.80
13	40147	158.47
14	40354	155.00

application composed of 11 distinct container images with tens of thousands of files in less than 300 milliseconds. It presented potential savings of 58% storage usage for other applications composed of 9 microservices in less than 200 milliseconds. Do note, however, that the values presented in the tables are the uncompressed files size and they are stored as compressed layers. The percentages presented were computed using the uncompressed size and analyzed the extracted files for each image, not the original layers size, therefore actual savings might be lower in practice.

7. Conclusion

In this work, we investigated the container image reconstruction problem as an integer linear problem. A constructive heuristic was proposed to solve it. A set of 14 instances was used, based on real container images, with some of them originating from real-world applications available as open-source projects. We have shown that using an integer linear model solver is not practical for daily usage to solve instances of this problem due to the computational resources required. The results achieved by the proposed constructive heuristic show potential savings from 20% up to 50% of storage usage. The proposed constructive heuristic is fast enough for daily usage because the running time increases moderately with the unique file count. The biggest instance with almost 50000 files was solved in under 300 milliseconds. In future work, the solution built with a constructive heuristic can be used within a meta-heuristic to improve potential savings. The proposed solution can be packaged as a tool and made available to practitioners.

8. Acknowledgment

This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* - Brasil (CAPES) - Finance Code 001, and CNPq (311907/2023-7).

References

- Barbier, J. (2014). It's here: Docker 1.0. <https://web.archive.org/web/20220518024454/https://www.docker.com/blog/its-here-docker-1-0/>. Access: 2024-04-25.
- Cloer, J. (2015). New cloud native computing foundation to drive alignment among container technologies. <https://www.cncf.io/announcements/2015/06/21/new-cloud-native-computing-foundation-to-drive-alignment-among-container-tec>. Access: 2024-04-25.
- CNCF (2023). Cloud native computing foundation annual survey 2022. <https://www.cncf.io/reports/cncf-annual-survey-2022/>. Access: 2024-04-25.
- Jiang, Q. (2023). Improving performance of docker instance via image reconstruction. In Hsu, C.-H., Xu, M., Cao, H., Baghban, H., and Shawkat Ali, A. B. M., editors, *Big Data Intelligence and Computing*, p. 511–522, Singapore. Springer Nature Singapore. ISBN 978-981-99-2233-8.
- Li, S., Zhou, A., Ma, X., Xu, M., and Wang, S. (2022). Commutativity-guaranteed docker image reconstruction towards effective layer sharing. In *Proceedings of the ACM Web Conference 2022, WWW '22*, p. 3358–3366, New York, NY, USA. Association for Computing Machinery. ISBN 9781450390965. URL <https://doi.org/10.1145/3485447.3512154>.
- Noyes, K. (2013). Docker: A “shipping container” for linux code. <https://www.linux.com/news/docker-shipping-container-linux-code/>. Access: 2024-04-25.

- Reis, D., Piedade, B., Correia, F. F., Dias, J. P., and Aguiar, A. (2021). Developing docker and docker-compose specifications: A developersâ survey. *Ieee Access*, 10:2318–2329.
- Skourtis, D., Rupprecht, L., Tarasov, V., and Megiddo, N. (2019). Carving perfect layers out of docker images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA. USENIX Association. URL <https://www.usenix.org/conference/hotcloud19/presentation/skourtis>.
- Van Solingen, R., Basili, V., Caldiera, G., and Rombach, H. D. (2002). Goal question metric (gqm) approach. *Encyclopedia of software engineering*.
- Zhao, N., Albahar, H., Abraham, S., Chen, K., Tarasov, V., Skourtis, D., Rupprecht, L., Anwar, A., and Butt, A. R. (2020). DupHunter: Flexible High-Performance deduplication for docker registries. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, p. 769–783. USENIX Association. ISBN 978-1-939133-14-4. URL <https://www.usenix.org/conference/atc20/presentation/zhao>.
- Zhao, N., Tarasov, V., Albahar, H., Anwar, A., Rupprecht, L., Skourtis, D., Warke, A. S., Mohamed, M., and Butt, A. R. (2019). Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, p. 1–10.