Clustered Kernel Search

-

Group project for the course *Optimization Algorithms*

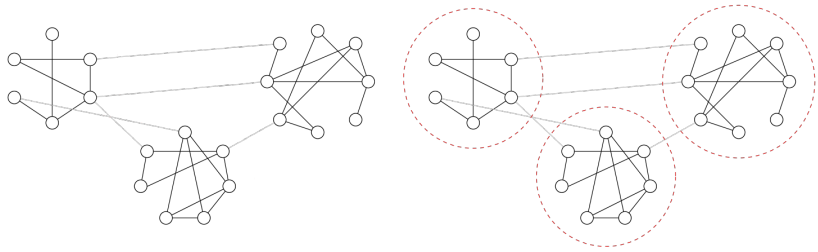
Matteo Salvalai (715827)
Michele Dusi (717462)
Pietro Venturini (715166)

A.Y. 2019/2020

Introduction

Kernel Search[1] is a heuristic framework that can be used to solve mixed integer linear programming problems. One fundamental step of this heuristic is the grouping of the variables that don't belong to the kernel set (i.e. the out-of-basis variables of the LP relaxation optimal solution) within buckets, that will later be used to solve smaller IP sub-problems. A standard approach to buckets construction is to order the out-of-basis variables according to their reduced costs, then grouping them into buckets of fixed size.

In this project we tried to exploit an innovative approach to the buckets construction by considering a graph whose nodes represent the variables in question, while the edges represent the correlation between the variables. A clustering algorithm is then used to find *communities* of nodes into the graph, gathering vertices into groups such that there is a higher density of edges within groups than between them. The identified clusters can then be used to build the buckets for the kernel search algorithm. That's the nature of the *Clustered Kernel Search*.



(a) (b)

Figure 1: A simple illustration that shows the result of applying a community detection algorithm to an undirected graph.

[1] Enrico Angelelli, Renata Mansini, and M. Grazia Speranza. "Kernel search: A general heuristic for the multi-dimensional knapsack problem". In: *Computers and Operations Research* 37.11 (2010). Metaheuristics for Logistics and Vehicle Routing, pp. 2017–2026. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2010.02.002.

Chapter 1

The algorithm

In this chapter we present the main ideas behind the proposed algorithm, describing the procedures composing the *Clustered Kernel Search* and their possible variants.

1.1 Idea of *Clustered Kernel Search*

The initial idea was to approach *Kernel Search* (*KS*) by focusing on improving buckets construction by exploiting known and hidden relations between the variables of the mathematical model.

In order to improve the performances and the quality of the solution, we searched for a relation with the following characteristics:

- easy to compute and identify;
- meaningful for the *KS* algorithm.

We chose to consider two variables as *related* whenever they appear in the same constraint(s). Eventually, this relation can be weighted according to the number of constraints shared by the two variables.

With this criterion, we build a graph in which each node represent a variable of the problem and each edge linking two nodes represent the appearance of the two corresponding variables in the same model constraint (more information in section 1.3.3).

Then, a clustering algorithm groups the vertices with a higher density of connections between them. The resulting clusters can form the buckets directly, or they can be combined further according to strategies defined in section 1.3.2.

Many MILP problems involves thousands of variables and constraints, therefore the resulting graph can be quite large. For this reason we implemented a clustering algorithm designed to handle large size graphs, which has been proposed by Clauset, Newman and Moore to find communities in very large

networks[1].

Note that the buckets construction isn't the only process that can benefit from the clustering-computed knowledge. Our efforts focused on that specific procedure, but we think it's possible to improve other aspects of the method by the same approach.

1.2 Basic definitions

Before proceeding, we define some useful concepts.

- **Item**: a variable of the model, with a value and a reduced cost associated to it.
- **Kernel**: the set of basis variables in the solution of the LP relaxation.
- **Bucket**: a set of variables that are not inside the Kernel.
- **Cluster**: a subset of nodes (that represent items) of a graph. In *CNM*[2] clusters are also referred as *communities*.
- **Modularity**: a property of a graph; it measures when a division is good, in the sense that there are many edges within clusters and only a few between them. This concept is formally explained in definition 1.3.5.
- **Degree** of a vertex: the number of incident edges upon that vertex.

1.3 Algorithms description

In this section we present the procedures composing the *Clustered Kernel Search* algorithm and we comment the most interesting parts of the pseudo-code listings.

The *CKS* algorithm follows the fundamental steps of *Kernel Search*, as shown in Algorithm 1. The object of our interest, the buckets construction, is reported at line 8 and described in detail in Algorithm 5.

Algorithm 2 presents the initialization of items, while algorithms 3 and 4 show how the kernel items can be selected.

Finally, Algorithm 6 and Algorithm 7 explain in details how each phase of buckets construction is executed.

[1] Aaron Clauset, M. Newman, and Cristopher Moore. "Finding community structure in very large networks". In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 70 (Jan. 2005), p. 066111. DOI: 10.1103/PhysRevE.70.066111.

[2] Clauset, Newman, and Moore, "Finding community structure in very large networks".

Algorithm 1 *Clustered Kernel Search*

procedure CLUSTERED KERNEL SEARCH($\mathcal{M}$)
input
    $\mathcal{M} \leftarrow$ the model of the problem
begin
    Let $\mathcal{I} \leftarrow$ BUILD ITEMS($\mathcal{M}$) the list of items
    Sort $\mathcal{I}$ by Reduced Costs
    Let $\mathcal{K} \leftarrow$ BUILD KERNEL($\mathcal{I}$, kernel_size) the list of items into Kernel
    Let $\mathcal{J} \leftarrow \mathcal{I} \setminus \mathcal{K}$ the set of out-of-kernel items
    Let $\mathcal{B} \leftarrow$ BUILD BUCKETS($\mathcal{J}$, $\mathcal{K}$, buckets_size)
    SOLVE KERNEL($\mathcal{K}$)
    SOLVE BUCKETS($\mathcal{B}$)
end procedure

Algorithm 2 *Build Items*

procedure BUILD ITEMS($\mathcal{M}$)
input
    $\mathcal{M}$ = the model of the problem
output
    $\mathcal{I}$ = the list of items in the problem
begin
    Solve the LP-relaxation of $\mathcal{M}$
    Extract from the solution the set of variables $\{v_1, v_2, ...v_3\}$
    Initialize $\mathcal{I} \leftarrow \emptyset$
    for all variable v **do**
        Build an item $i = (v,\ \textbf{value}(v),\ \textbf{rc}(v))$
        $\mathcal{I} \leftarrow \mathcal{I} \cup \{i\}$
    end for
    return $\mathcal{I}$
end procedure

1.3.1 Kernel construction

As in *Kernel Search*, the construction of the Kernel is a fundamental step of the algorithm. We decided to adopt the same two strategies defined in the original *KS* algorithm. Both can be used freely, even though the second is probably the optimal one.

Kernel Builder Positive

The idea is to insert into the Kernel only items with positive values. However, this method does not assure the predefined Kernel size is respected.

Algorithm 3 *Build Kernel Positive*

procedure BUILD KERNEL($\mathcal{I}$, ks)
input
    $\mathcal{I}$: the list of items in the problem
    ks : the predefined size of Kernel $\mathcal{K}$
output
    $\mathcal{K}$: the Kernel
begin
    Initialize $\mathcal{K} \leftarrow \emptyset$
    for all $i \in \mathcal{I}$ **do**
        if value$(i) > 0$ **then**
            $\mathcal{K} \leftarrow \mathcal{K} \cup \{i\}$
        end if
    end for
    return $\mathcal{K}$
end procedure

Kernel Builder Percentage

The idea is to insert into Kernel a fixed percentage of items, based on the predefined size and preferring items with higher reduced cost.

Algorithm 4 *Build Kernel Percentage*

procedure BUILD KERNEL($\mathcal{I}$, ks)
input
    $\mathcal{I}$: the list of items in the problem
    ks : the predefined size of Kernel $\mathcal{K}$
output
    $\mathcal{K}$: the Kernel
begin
    Initialize $\mathcal{K} \leftarrow \emptyset$
    SORT($\mathcal{I}$) according to the reduced costs of the items
    $\mathcal{K} \leftarrow$ TAKE($\mathcal{I}$, ks), take the first (ks) elements
    return $\mathcal{K}$
end procedure

1.3.2 Buckets Construction

As shown in Algorithm 5, this procedure is split into distinct phases: the graph building, the graph clustering and the buckets composition. The first two are presented in sections 1.3.3 and 1.3.4 respectively; we'll now focus on the different possibilities for the last one.

We defined essentially four strategies, besides the default construction offered by the *Kernel Search*:

Simple clusters

This strategy takes the clusters set $\mathcal{C}$ and returns a set of buckets $\mathcal{B}$ where each bucket corresponds exactly to a cluster of items.

In the next chapters, we'll refer to this strategy as SIM.

Mixed clusters

Each bucket of the resulting set $\mathcal{B}$ is composed by mixing the content of the clusters in $\mathcal{C}$. The cluster items are distributed one per bucket, starting from the most promising ones. Each bucket is filled until its maximum size is reached.

We'll denote this strategy with the acronym MIX.

Aggregated clusters with balanced sizes

Regardless of the convenience of each item, the buckets in $\mathcal{B}$ are composed by the aggregation of the clusters in $\mathcal{C}$; in this way, each bucket can be the union of one or more different clusters. The aggregation of clusters into a bucket continues until the bucket reaches the maximum size.

We'll refer to this strategy as AGG.

Simple clusters with privileged elements

This strategy is similar to the first one SIM. Each resulting bucket in $\mathcal{B}$ corresponds to a cluster in $\mathcal{C}$, with a relevant difference: every bucket is expanded with the most promising items not already placed in the Kernel. In this way, the buckets overlap by those items.

In the next chapters, we'll refer to this strategy as PRI.

1.3.3 Graph building and model interpretation

In order to extract information about variables and their relations, we had to abstract the mathematical model in a structure better suited for analysis; a graph was the natural choice.

The graph we aimed to build should clearly depict the *correlation* between two items, in a way that it would be easy to handle. We formalize this concept in definition 1.3.1.

Algorithm 5 *Build Buckets*

procedure BUILD BUCKETS($\mathcal{J}$, $\mathcal{K}$, bs)
input
    $\mathcal{J}$: the list of items outside the kernel, or $\mathcal{I} \setminus \mathcal{K}$
    $\mathcal{K}$: the list of items inside the kernel $\mathcal{K}$
    bs : the predefined size of the buckets in $\mathcal{B}$
output
    $\mathcal{B}$: a list of buckets containing the items in $\mathcal{J}$
begin
    $\mathcal{G} \leftarrow$ BUILD GRAPH($\mathcal{J}$)
    $\mathcal{C} \leftarrow$ CLAUSET NEWMANN MOORE($\mathcal{G}$) ▷ Clustering procedure
    Build the buckets list $\mathcal{B}$ from clusters $\mathcal{C}$ according to the selected strategy.
    return $\mathcal{B}$
end procedure

Algorithm 6 *Build Graph*

procedure BUILD GRAPH($\mathcal{J}$, $\mathcal{S}$)
input
    $\mathcal{J} = \mathcal{I} \setminus \mathcal{K}$: the list of items outside the Kernel
    $\mathcal{S}$: the list of constraints of the problem model $\mathcal{M}$
output
    $\mathcal{G}$: a graph representing the relations between the variables
begin
    $\mathcal{G} \leftarrow (V, E)$, where $V \leftarrow \emptyset$, $E \leftarrow \emptyset$
    for all constraint $c \in \mathcal{S}$ **do**
        for all variable $v_i \in c$ **do**
            $V \leftarrow V \cup \{v_i\}$ ▷ Create a node associated with the variable v_i
            for variable $v_j \in c$ **do**
                $w \leftarrow$ **corr**(v_i, v_j) ▷ Compute the weight of the edge
                $E \leftarrow E \cup (v_i, v_j, w)$ ▷ Insert the edge between the nodes
            end for
        end for
    end for
    GRAPH PRUNING($\mathcal{G}$) ▷ (Optional. See section 1.3.3)
    return $\mathcal{G}$
end procedure

Definition 1.3.1 (Relations graph)**.** Let $\mathcal{J}$ be the set of the out-of-kernel items, and $\mathcal{S}$ be the set of constraints of the problem. Let $\mathbf{corr}(\cdot,\cdot)\colon \mathcal{J} \times \mathcal{J} \mapsto \mathbb{N}$ be a function measuring the correlation between two variables. The *relations graph* $\mathcal{G} = (V, E)$ is a weighted undirected graph where:

- $V = \mathcal{I}$ is the set of nodes;
- $E = \{(v_1, v_2, w) \mid v_1, v_2 \in \mathcal{J} \wedge w = \mathbf{corr}(v_1, v_2)\}$ is the set of weighted edges.

We assume that a zero-weight edge (if $\mathbf{corr}(v_1, v_2) = 0$) is equivalent to a missing edge between the nodes of items v_1 and v_2.

With the previous definition, the *relations graph* gives a structured representation of the influences between items, plus it can also work with various correlation functions. In our experience, the idea of correlation we chose to represent is based on the presence of specific variables into specific constraints, but in general it's not limited to that. That's also why the set of constraints $\mathcal{S}$ appears in the definition.

We tested *CKS* algorithm with three different correlation measures. All of them are symmetric relations (i.e. $\forall\, x, y\ \mathbf{corr}(x, y) = \mathbf{corr}(y, x)$), therefore the resulting graph is always undirected.

We present the three measures in the next paragraphs. In the final testing phase, we opted for the second and third strategies, since their implementation was the most time and space efficient.

Simple unweighted correlation

If two variables appear together at least once in a constraint, their respective nodes are connected.

Definition 1.3.2 (Simple correlation)**.** Let $\mathcal{J}$ be the set of out-of-kernel items. Let $\mathcal{S}$ be the set of constraints of the problem. Given two items v_1 and v_2 in $\mathcal{J}$, we define the *correlation* between these two items as:

$$\mathbf{corr}(v_1, v_2) = \begin{cases} 1 & \text{if } \exists c \in \mathcal{S} \text{ such that } v_1 \in c \wedge v_2 \in c \\ 0 & \text{otherwise} \end{cases}$$

Note that, in order to simplify the implementation of the resulting relations graph, this correlation measure can produce an undirected *unweighted* graph.

Weighted correlation

If two variables appear together in a constraint, their respective nodes are connected and the weight of the edge is the number of constraints they share.

Definition 1.3.3 (Weighted correlation)**.** Let $\mathcal{J}$ be the set of out-of-kernel items. Let $\mathcal{S}$ be the set of constraints of the problem. Let $\mathcal{S}_{x,y} = \{c \in \mathcal{S} \mid x \in c \wedge y \in c\}$ be the set of constraints shared by two variables x and y. Given two items v_1 and v_2 in $\mathcal{J}$, we define the *correlation* between these two items as:

$$\mathbf{corr}(v_1, v_2) = |\mathcal{S}_{v_1, v_2}|$$

Weighted correlation with threshold

If two variables appear together in at least n constraint, their respective nodes are connected and the weight of the edge is the number of constraints they share.

Definition 1.3.4 (Weighted correlation with threshold)**.** Let $\mathcal{J}$ be the set of out-of-kernel items. Let $\mathcal{S}$ be the set of constraints of the problem. Let $\mathcal{S}_{x,y} = \{c \in \mathcal{S} \mid x \in c \wedge y \in c\}$ be the set of constraints shared by two variables x and y. Given two items v_1 and v_2 in $\mathcal{J}$ and the threshold $n \in \mathbb{N}^+$, we define the *correlation* between these two items as:

$$\mathbf{corr}(v_1, v_2) = \begin{cases} |\mathcal{S}_{v_1,v_2}| & \text{if } |\mathcal{S}_{v_1,v_2}| \geq n \\ 0 & \text{otherwise} \end{cases}$$

This third measure brings out only the heavy-weighted edges, "filtering" the graph and discarding the "casual" relations between items. Instances with thousands of variables may lead to graphs with millions of edges. Unfortunately, due to limitations on computing resources, the clustering algorithm can take a long time to complete. For this reason, at the end of Algorithm 6, we decided to prune the graph by removing edges with lower weight, until the number of remaining edges (those with higher value) does fall below a certain threshold, which can be specified in the configuration file through the `MaxGraphEdges` parameter. Pruning can be performed in a dynamic way by gradually increasing the threshold n until the number of edges with a greater weight than n, is less than `MaxGraphEdges`.

1.3.4 Clustering Algorithm

The choice of a good clustering technique is fundamental for the efficiency of our algorithm. The technique has to evaluate the graph created from the model analysis and to give a good communities representation of the problem.

***Clauset-Newmann-Moore* algorithm**

The algorithm of our choice is the one proposed by A. Clauset, M. E. J. Newman and C. Moore[3]. It's an unsupervised (i.e. it doesn't require to know the number of communities nor their sizes before execution) greedy hierarchical agglomeration algorithm based upon the concept of *Modularity* Q, known also as *Greedy Modularity*.

Definition 1.3.5 (Modularity)**.** Let $\mathcal{G} = (V, E)$ be a weighted undirected graph, where:

- $m = |E|$ is the number of edges;

[3]Clauset, Newman, and Moore, "Finding community structure in very large networks".

- A_{v_1,v_2} is the entry of the adjacency matrix that represents the graph, or the weight of the edge connecting nodes v_1 and $v_2 \in V$;
- k_v is the degree of the node v;
- c_v is the community to which the node v belongs;
- $\delta(\cdot,\cdot)\colon V \times V \mapsto \mathbb{N}$ is the function defined as:

$$\delta(v_1, v_2) = \begin{cases} 1 & \text{if } c_{v_1} = c_{v_2} \\ 0 & \text{if } c_{v_1} \neq c_{v_2} \end{cases} \tag{1.1}$$

The *Modularity* Q is defined as:

$$Q = \frac{1}{2m} \sum_{v,w \in V} \left[A_{v,w} - \frac{k_v k_w}{2m} \right] \delta(v, w) \tag{1.2}$$

The gist of the method is to maximize the modularity measure to obtain a well-structured communities graph. As described in Algorithm 7, at first each community corresponds to a single node of the graph (lines 7-10). At every iteration, the algorithm finds the two communities that, if merged together, lead to the maximum positive variation of Modularity (lines 12-13, 21-22). As the two communities are merged, the structure of the graph reflects those changes (line 20); that's why the modularity of the graph itself can increase. The algorithm stops when the modularity reaches its maximum peak, that is when every possible merge gives a negative modularity variation ($\max(\Delta Q) < 0$, lines 16-17).

The algorithm has a computational complexity of $O(md \log n)$, where m is the number of edges, n is the number of vertices and d is the depth of the *dendrogram*[4] describing the communities structure.

Alternative algorithms

An interesting alternative based upon the maximization of Modularity is the *Louvain* algorithm[5]. It's a greedy unsupervised algorithm and has a computational complexity of $O(n \log^2 n)$, where n is the number of vertices of the graph it has to analyze.

Generally speaking, the *Louvain* algorithm allows better time performances; however, we explain in section 3.3 why this was not the optimal choice for our work.

[4] A hierarchical clustering *dendrogram* is a diagram showing the grouping of clustered items as a function of modularity.

[5] Vincent D. Blondel et al. "Finding community structure in very large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008). DOI: `https://iopscience.iop.org/article/10.1088/1742-5468/2008/10/P10008`.

Algorithm 7 Clauset-Newmann-Moore *Algorithm*

procedure BUILD COMMUNITIES(G)
input
    $\mathcal{G} = (V, E)$: the graph of the model: V is the set of vertices and E is the set of edges
output
    $\mathcal{C}$: the set of found communities
begin
    $\mathcal{C} \leftarrow \emptyset$
    for all $v \in V$ **do**
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{\{v\}\}$
    end for
    Compute $Q \leftarrow$ MODULARITY(G)
    $\Delta Q \leftarrow$ VARIATION OF MODULARITY(G)
    $H \leftarrow$ MAXIMUM ENTRY FOR EACH ROW(ΔQ)
    while H is not empty **do**
        Pop maximum entry $(\delta q, v, w)$ from H
        if $\delta q \leq 0$ **then**
            break
        end if
        Update modularity $Q \leftarrow Q + \delta q$
        Merge c_v and c_w in graph $\mathcal{G}$ and in list $\mathcal{C}$
        $\Delta Q \leftarrow$UPDATE($\Delta Q, v, w$)
        $H \leftarrow$UPDATE(H, v, w)
    end while
    return $\mathcal{C}$
end procedure

Chapter 2

Implementation

After designing the algorithm, we studied the performances in terms of time, memory, correctness and goodness of the attained solutions. In order to do that, we implemented the whole algorithm extending the original *Kernel Search* implementation in `Java` language, which has been provided to students of *Optimization Algorithms* course.

For the nature of *Clustered Kernel Search*, we worked especially on the *buckets construction* phase. The remaining code has the same flow of the original implementation, but has been refactored and re-organized to simplify teamwork.

In the next paragraphs we present some of the choice we made for data structures and procedures in the algorithm coding.

2.1 Graph Data Structure

One crucial decision we had to take regarded the data structure used to represent the items graph. Next we briefly comment the different solutions we explored.

Adjacency matrix

We first started with a raw matrix array to represent the graph. However, this approach was not found to be the best, since large portions of the matrix remain empty and we needed to keep track which row and column belonged to which node (and variable).

Sparse adjacency matrix

The second idea was to use a raw *sparse matrix* to represent the edges in the graph. Still a map that linked a variable to an index of the matrix had to be stored, making this solution not the most space efficient.

Implicit graph

We implemented an undirected graph using proper `Node`, `Edge` and `Graph` classes, but we encountered some problems in terms of high computational time when translating the model constraints into a graph. The main reason is that, as algorithm 6 shows, we need 3 nested cycles in order to detect "*relations*" between the variables.

Furthermore, every time we add a new node or a new edge, checks must be done to verify that the node is not already in the graph, and that the edge links two existing nodes. This approach failed early due to the long time required to build large graphs for medium and large models (*e.g.* few thousands of variables and millions of edges).

Third-party library

We tried to use `MutableValueGraphs` from the Google `Guava`[1] library, in the hope to speed up the construction of the graph. Unfortunately it seemed not to solve our problem.

Adjacency map

We used a `HashMap` having a `Node` (an item) as the key for each entry, and a `HashMap` of `Node` objects as the corresponding value. The outermost `HashMap` gives information about to which node a particular one is directly connected whereas the innermost `HashMap` stores the weights of the edges. The absence of a node v in the innermost map of node w means that the two considered nodes v and w are not directly connected. This approach is the fastest and the most space efficient among the ones listed; it provides a fast (constant) access to node neighbors and edges. That's why we stuck with this implementation.

2.2 Clustering Data Structures

For the *Clauset-Newman-Moore* algorithm[2] to work properly, we needed three essential objects: ΔQ, ΔQ *heap* and H.

Modularity Variation Matrix

ΔQ is the matrix containing the variation of modularity associated to the merge of two connected communities. It has to change every time two communities are merged together and has to grant a quick access to the stored values. We decided to implement it as a sparse matrix, in the form of a two level of depths `HashMap`: the outermost `HashMap` act as the rows of the matrix whereas the innermost `HashMap` acts as the columns of the matrix. Every missing entry is considered by default to be 0 and the maps' indices are the nodes of the

[1] Google Guava - Github. URL: `https://github.com/google/guava`

[2] Clauset, Newman, and Moore, "Finding community structure in very large networks".

graph, granting a direct and quick access to the elements of the matrix. At each iteration of the algorithm the row related to the community that was involved in a merge is removed.

Modularity Variation Heaps

In order to grant a constant-time access to the maximum element of each row of the ΔQ matrix, two specific data structures have been used in the implementation of *Clustered Kernel Search* algorithm.

- The first one is denoted as ΔQ *heap*. It refers to the ΔQ matrix, which stores each row as a **max heap**. This solution allows fast adaptation in case of changes on ΔQ. It has been implemented in `Java` as a `HashMap` of type `PriorityQueue`, where each entry of the `HashMap` refers to the corresponding row in ΔQ. At each iteration, the row related to the merged community is emptied.
- The second one is denoted as H. It contains the maximum entry of the whole max heap. It grants constant-time access to the maximum variation of modularity in order to find the two communities to merge. It is implemented as a `PriorityQueue` where each element refers to an element of ΔQ *heap* and adapts every time the other data structure changes. At each iteration of the algorithm the maximum entry is removed from H and the communities that it refers to are merged together.

Chapter 3

Experimental results

In order to verify whether our approach could help the performances of the *Kernel Search* framework, we tested the *Clustered Kernel Search* on a collection of instances and we compared the results to the ones obtained with the original version of the *KS* algorithm.

More specifically, we aimed to understand if the improvement of the solution obtained with the *CKS* framework was worth the additional time spent to compute the buckets.

We express the following results in terms of *computational time* needed to find a solution, and in terms of *quality* of the solutions.

All the instances, for both *Kernel Search* and *Clustered Kernel Search*, have been executed on a AMD Ryzen 5 2500U processor (2000 MHz, 4 cores, 8 logical processors) and 8 GB of RAM memory.

3.1 Parameters configuration

For the comparison between the two algorithms, we kept the same configuration parameters values where possible. Our configuration is listed in the following table.

Parameter	***KS* and *CKS* values**
`Presolve`	2
`Time Limit`	900
`MipGap`	$1 \cdot 10^{-12}$
`Threads`	8
`Sorter`	Sorting by Value and Absolute Reduced Cost
`KernelBuilder`	`KernelBuilderPositive`
`KernelSize`	0.1
`TimeLimitKernel`	20
`NumIterations`	2
`TimeLimitBucket`	30
`MaxGraphEdges`	1000

The main differences is in the Bucket construction parameters. We decided to compare the *Kernel Search*'s *Default bucket builder* strategy with our most interesting strategies: *Simple clusters with privileged elements* (PRI) and *Aggregated clusters with balanced sizes* (AGG).

While the *Kernel Search* worked with a buckets relative dimension of 0.55, the *Clustered Kernel Search* worked with one of 0.34.

For PRI strategy we worked with 20% of the most promising items. Due to limited resources, we also introduced a cap, set to 1000, on the maximum number of edges in the graph.

3.2 Tested instances

We tested the two frameworks on the instances reported in the following table. The instances come from the `MipLib` Collection[1], and they all represent problems of *set partitioning*.

Instance name	Diff.	Variables	Binary v.	Constr.	Edges
`30n20b8`	easy	18380	18318	576	6668485
`assign1-10-4`	open	572	520	52	399
`ds`	hard	67732	67732	656	-
`eil33-2`	easy	4516	4516	32	9514770
`rococoC10-001000`	easy	3117	2993	1293	113446

3.3 Results and comments

In this section we present the results obtained by the framework testing, along with some remarks we could observe.

[1] *MIPLIB 2017 - The Mixed Integer Programming Library*. 2019. URL: `https://miplib.zib.de`.

3.3.1 General remarks

First, some considerations about the execution times are needed. The *Clustered Kernel Search* extends with new functionalities the original *Kernel Search* framework, therefore we expected it to require additional time. Our aim was to inquire whether the additional time spent for the clustering process could considerably improve the solution.

Intuitively, more dense models required more time for the graph to be constructed; the same happened for the clustering process. However, we observed how these times did not change considerably between the two strategies we tested (AGG, PRI). Unfortunately, for dense-structured problems, the trade-off between the graph building and clustering time and the improvement of the solution resulted with *Clustered Kernel Search* was not worth it.

Second, a consideration about the memory used by the program. During the whole testing phase, we had to deal with the saturation of memory heap dedicated to our process. We introduced some constraints on the maximum number of edges, because their large number had a big impact on the clustering algorithm. Some of the instances were found to have a number of edges in the order of millions.

We also tried to enlarge the memory space dedicated to the heap, but with no use: the time spent by the `Java` Garbage Collector to clean up unused portions of the memory overtook the actual computational time.

For our limited computational resources, we could not execute the algorithm on graphs having a large number of edges; we empirically set that limit around 1000. However, we can suppose that - with more time and memory available - the general quality of the buckets could have improved, since a larger number of relations between the items could have been considered in the clustering process.

Third, a consideration about the clustering algorithm. The communities-clustering method proposed by Clauset, Newman and Moore[2] is not the most efficient nor the latest clustering algorithm published in literature. We tested separately a different procedure, the *Louvain* algorithm[3], which can deal with a large number of edges more efficiently, despite of not being deterministic.

However, our purpose was not to complete the clustering process in the fastest time, but to understand if this approach could lead us to some interesting results; therefore, we decided not to insert a random element in the algorithm.

Fourth, a consideration about the time spent to compose the graphs and to cluster them. Overall the two phases required a similar amount of time for PRI and AGG strategies, as expected. In the case of more pronounced differences we think that this was due to the different state the memory was into when we ran the tests: the differences in time are imputable to the `Java` Garbage Collector

[2] Clauset, Newman, and Moore, "Finding community structure in very large networks".
[3] Blondel et al., "Finding community structure in very large networks".

and his actions to free unused, occupied, areas of memory.

3.3.2 Experimental data

Let us now consider the raw data coming from the tests:

Results with *Kernel Search*

Instance	Exact sol.	KS best sol.	Total time (s)	Buckets number
`30n20b8`	302	553	3,04	2
`assign1-10-4`	422	423	140,72	2
`ds`	57,23457	infeasible	-	2
`eil33-2`	811,279	934,008	10,29	2
`rococoC10-001000`	11460	14781	12,36	2

Results with PRI, *Clustered Kernel Search*

Instance	Best sol.	Total time(s)	Graph time (s)	Clustering time (s)	Buckets number
`30n20b8`	infeasible	41,20	16,10	6,90	24
`assign...`	423	$\sim 2,02 \cdot 10^2$	0,15	0,69	3
`ds`	-	out of time	-	-	-
`eil33-2`	934,008	$\sim 1,36 \cdot 10^2$	113,20	1,71	5
`rococo...`	16739	71,4	0,33	4,26	856

Results with AGG, *Clustered Kernel Search*

Instance	CKS best sol.	Total time(s)	Graph time (s)	Clustering time (s)	Buckets number
`30n20b8`	402	26,3	15,70	6,70	3
`assign...`	425	$\sim 2,01 \cdot 10^2$	0,15	0,70	3
`ds`	-	out of time	-	-	-
`eil33-2`	1046,8	$\sim 1,13 \cdot 10^2$	101,39	2,25	3
`rococo...`	14375	7,87	0,37	3,87	3

Notice how, for the AGG strategy, the number of buckets is known a priori: the procedure builds the buckets with a pre-defined relative size (given by the parameter `BucketSize`). On the contrary, for the PRI strategy, the buckets number can provide some relevant information for the execution.

Accordingly to the data we obtained, similarly to the *Kernel Search* framework, our algorithm was able to produce a solution for all the instances for both our strategies, except for the `ds`.

Using the *Clustered Kernel Search* with PRI strategy for `30n20b8`, every obtained sub-problem was unfeasible. Considering the number of buckets generated in the process, we observed that the sub-problems were too small: each bucket did not include a sufficient number of items to make the problem solvable.

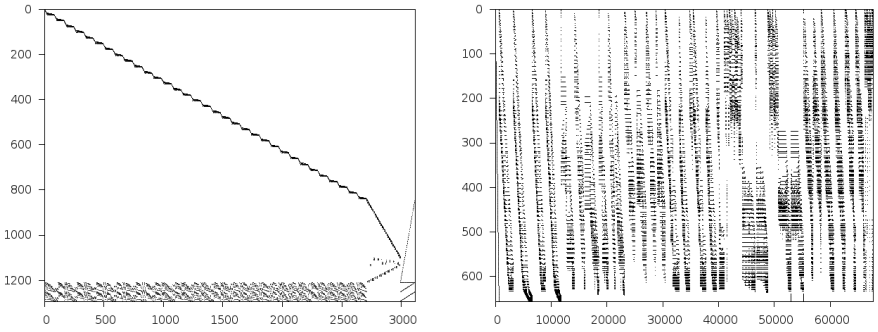
Comparing the *Kernel Search* framework with the *Clustered Kernel Search* with PRI strategy, the data suggest that the *KS* is the best one in terms of time

and quality of the solutions. Even when the *CKS* managed to return solutions comparable to the ones found by the *KS*, the total time was higher. The actual time spent by *CKS* for solving the sub-problems was still higher than the time used by the *KS* in every instance: we concluded that this *CKS* variant built low quality buckets.

The *Clustered Kernel Search* with AGG had the same applicability as the *Kernel Search*, except in the case of `ds`. In terms of total time the *KS* was overall the best framework, with an interesting exception in `rococoC10-001000`: our strategy managed to compute a better solution in less total time. In terms of quality of the solutions, the two were pretty much comparable: they found a better solution for a couple of instances both and the discrepancies were, overall, comparable. In terms of time spent to solve the buckets the *KS* was overall the best, except for `rococoC10-001000` and `30n20b8`: in the latter we had a comparable time.

To conclude, it's important to notice that our approach was based upon the assumption that a mathematical model is suitable to be decomposed in variables clusters. Clearly, not every model has this property.

For example, in Figure 3.1, we clearly notice how `ds` is a more homogeneously interconnected model than `rococoC10-001000`. In fact, even if the second one has a higher number of constraints, the clustering was significantly faster and profitable in terms of solution quality.



(a) (b)

Figure 3.1: The graphical representation of instances `rococoC10-001000` (3.1a) and `ds` (3.1b). Each points indicates the presence of a certain variable (column) into a certain constraint (row).

Some instances can benefit from the clustering approach more than others; for those that couldn't not only the resources spent are not worth it, but also the resulting decomposition into sub-problems opposes the resolution. For this reason, whether to choose a clustering approach remains an empirical question, strongly instance dependent.

Chapter 4

Conclusions

We observed that our approach doesn't improve *KS* in terms of efficiency, mainly because of the time required to construct the graph and to identify communities within it. In order to improve the speed of *KS*, we think that the focus must be on factors such as parallelization of the sub-problems' resolution processes. Anyway, in some cases, studying the problem structure through a process like the one we adopted, can lead to better solution, such as it happened with the `30n20b8` and `rococoC10-001000` instances.

We hope that this approach to bucket construction can inspire similar approaches to MIP optimization problems, involving a study of the problem structure through an unsupervised learning algorithm. Probably there exists a better way to define some sort of correlation between variables than the criterion we have chosen. Furthermore, adopting a more advanced clustering algorithm, like the *Louvain* algorithm can speed up the entire process for sure.

Further developments may include also the combined use of some of the strategies illustrated in section 1.3.2. For example, *Aggregated clusters with balanced sizes* and *Simple clusters with privileged elements* may be combined in order to obtain fixed-size buckets but with overlapping items that looks promising.

Although the clustering approach is a time-demanding process, it may become useful in contexts like the following: consider a problem in which the model structure is known a priori, for instance suppose that the constraints represents the total costs of some combination of items, which must not exceed a certain budget. The variables represent the quantity to buy, while the coefficients represent the unit price. Suppose that an optimization problem of this type have to be solved frequently. Then we can solve the clustering phase just the first time (since the model structure does not change), and then use its results to construct the buckets every time we have to solve a specific instance of that problem (eventually ignoring the basis variables which appear inside the clusters).

Bibliography

Angelelli, Enrico, Renata Mansini, and M. Grazia Speranza. "Kernel search: A general heuristic for the multi-dimensional knapsack problem". In: *Computers and Operations Research* 37.11 (2010). Metaheuristics for Logistics and Vehicle Routing, pp. 2017–2026. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2010.02.002.

Blondel, Vincent D. et al. "Finding community structure in very large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008). DOI: https://iopscience.iop.org/article/10.1088/1742-5468/2008/10/P10008.

Clauset, Aaron, M. Newman, and Cristopher Moore. "Finding community structure in very large networks". In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 70 (Jan. 2005), p. 066111. DOI: 10.1103/PhysRevE.70.066111.

MIPLIB 2017 - The Mixed Integer Programming Library. 2019. URL: https://miplib.zib.de.