Erasmus University Rotterdam
Erasmus School of Economics

Bachelor Thesis BSc^2 Econometrics & Economics

An Enquiry into an Improved Branch-and-Bound Algorithm for the Knapsack Problem with Conflict Graph

Abstract

The knapsack problem with conflict graph consists of packing items into a knapsack such that total profit is maximised, while not violating the knapsack capacity constraint and avoiding the packing of incompatible items. This research replicates a proposed improved branch-and-bound algorithm, and compares the obtained results with other custom branch-and-bound algorithms and a general purpose solver. The improved algorithm depends on an upper bound based on finding a clique cover, and a branching scheme based on solving the 0-1 knapsack problem with dynamic programming. It has been found that the proposed algorithm outperforms the other solvers when larger amounts of items are considered and conflict graph densities are relatively high. Additionally, a GPU-based implementation for the branching scheme is proposed, which showed significant improvements for higher knapsack capacities.

Name student: V.T. Schouten

Student ID number: 449249

Supervisor: R. Hoogervorst

Second assessor: Dr. W. van den Heuvel

Date final version: July 4, 2020

Contents

1 Introduction

The Knapsack Problem with Conflict Graph (KPCG), otherwise known as the disjunctively constrained knapsack problem, extends upon the classical 0-1 Knapsack Problem (KP). The latter is a well-known problem in the field of combinatorial optimisation. It consists of filling a knapsack with items, with each considered item having a weight and profit associated with it, and the knapsack being constrained by a weight capacity. The goal of the 0-1 KP is to maximise total profit under the given constraint. The KPCG adds incompatible items to the problem, i.e. items which cannot be placed in a knapsack together. The KP occurs frequently in the field of operations research. For example, the KP presents itself as a subproblem in the Bin Packing Problem (BPP) in operations research, which also holds for the KPCG in the case of Bin Packing Problem with Conflicts (BPPC) (Sadykov and Vanderbeck, 2013). In an era where logistics become increasingly difficult to manage, research on topics like this is becoming progressively more important. Simultaneously, academics are still improving on existing frameworks, take Bettinelli, Cacchiani, and Malaguti (2017) with their work on the KPCG. Hence, both in practical and scientific context there is incentive to examine this topic.

The main goal of this research is to replicate and validate the results of Bettinelli et al. (2017). To achieve that goal, various methods are used to solve the KPCG. The problem will first be formulated as two Mixed Integer Programs (MIPs), which will be solved using a general purpose solver. Moreoever, multiple custom Branch-and-Bound (B&B) algorithms will be implemented: a generic version as proposed by Sadykov and Vanderbeck (2013); and two improved versions of the generic version, one with an improved branching scheme, the other utilising both the improved branching scheme and an improved upper bound (Bettinelli et al., 2017). Extending on the proposed methods, a parallel computing implementation of the improved branching scheme is proposed and applied. All methods will be compared on amount of instances solved to optimality, solving time, and amount of nodes visited. Additionally, the ratio of the upper bounds with respect to the optimal solution will be used to determine the performance of each of the upper bounds.

It has been found that for moderate knapsack capacities all custom B&B algorithms perform well, and are faster than a general purpose solver. With increased knapsack capacities, their performance degrades rather quickly, yet they are still preferred over a general purpose solver when the amount of items considered is more than moderate and graph densities are relatively high. Moreover, it is still apparent that the proposed branching scheme and upper bound increase the performance of the B&B algorithm. The main contribution of this research is the parallel implementation of the branching scheme proposed by Bettinelli et al. (2017). It has been found that for large knapsack capacities, computing the branching condition on the GPU results in moderate to large speedups in solving times, depending on the amount of items considered.

The paper is outlined as follows. In Section 2, a general description of the problem to be researched is given, followed by a review on existing literature on the subject in Section 3. Subsequently, in Section 4 the different methods of solving the KPCG will be elaborated upon. Section 5 will then treat the data needed to perform the research. Subsequently, Section 6 will discuss the obtained results, after which

Section 7 will conclude the paper with conclusions and a discussion on the limitations of the research.

2 Problem Description

The main task at hand is replicating and validating the findings of Bettinelli et al. (2017), and ultimately adding to their framework in a contributing manner. Consequently, the KPCG will be explained in detail. The KPCG extends upon the classical 0-1 KP by adding the possibility of having incompatible items, implying that those items cannot be packed into the knapsack. To further specify, the aim of the KPCG is to find the subset of items that fits in the knapsack while maximising the associated profit, taking into account that incompatible items may not be placed together in the knapsack. The incompatible items are represented by an undirected conflict graph $G = (V, E)$, with $|V| = n$ and where any edge $(i, j) \in E$ denotes that the items i and j are incompatible. The density of such a graph is then defined as the ratio between $|E|$ and the amount of edges in a complete graph with the same amount of vertices. To formulate the KPCG, some parameters are to be defined: the knapsack capacity c and the set of items $1, \ldots, n$, with each item having a weight w_i and profit p_i $(i = 1, \ldots, n)$. Following Yamada, Kataoka, and Watanabe (2002), we assume the following without loss of generality:

(i) $c > 0$ & w_i, $p_i > 0$ $(i = 1, ..., n)$

(ii) $\sum_{i=1}^{n} w_i > c$ & $w_i < c$ $(i = 1, ..., n)$

(iii) Items are sorted in non-increasing profit-weight ratio, i.e. $p_i/w_i < p_{i+1}/w_{i+1}$ $(i = 1, ..., n-1)$

Assumption (i) simply states that the capacity, weights and profits are to be positive. Assumption (ii) denotes that the sum of the weights of all items should be greater than the capacity and that each item should have a weight less than the capacity. If this were not the case, the problem would become trivial. Due to the nature of the knapsack problem, assumption (iii) is to facilitate the solving process, also known as implementing a greedy approach. This owing to the fact that items having a higher profit relative to its weight intuitively have a higher precedence over other items. Hence, in the solving process, the item at the top of the item list should be the most profitable item available to be added to the knapsack.

3 Literature Review

There are many approaches to solve combinatorial optimisation problems, yet *direct* approaches can be inefficient or do not always find a feasible solution (Lawler and Wood, 1966). B&B algorithms are well-studied and widely implemented, as it facilitates the solving process by dividing a difficult problem into easier subproblems. The process branches over solution sets, not considering branches incompatible with the bounding condition. Computation time thus depends on the amount of distinct subproblems created, which can prove to be very valuable for larger problem instances.

3.1 Branch-and-Bound in Practice

Nowadays B&B is a widely-implemented solving approach. Take the field of engineering, where Demeulemeester and Herroelen (1992) proposed a B&B algorithm for the multiple resource-constrained project scheduling problem. The problem concerns the scheduling of a construction project, with the goal to minimise its duration subject to technological precedence and resource constraints. This entails that an activity can only be started when preceding activities have finished, and resources have limited availability per period. Efficiently solving this problem may greatly benefit construction managers, and issuers of projects.

In the context of transport economics, Haffner, Monticelli, Garcia, and Romero (2001) proposed a B&B algorithm for transmission network expansion planning. The goal of the aforementioned problem is to guide future investment in transmission equipment such that the long-term transmission network is optimal. Transportation is a part of everyone's life, be it a single individual or a firm, and B&B can be an important tool in shaping an efficient network.

The Vehicle Routing Problem (VRP) is an important and diverse problem in the field of operations research, as many firms have to deal with it. Zhang, Qin, Zhu, and Lim (2012) propose a B&B approach in solving the VRP with toll-by-weight scheme. The VRP normally assumes that the cost to traverse each edge is equal to its length. In China however, expressways require the driver to pay a toll based on distance travelled and vehicle weight. In that context, it is crucial for firms to efficiently route all vehicles to avoid excessive and unneeded cost.

Miralles, García-Sabater, Andrés, and Cardós (2005) propose B&B procedures for the newly introduced Assembly Line Worker Assignment and Balancing Problem (ALWABP). The problem describes allocation of tasks to stations as well as available workers to stations. The amount of workers is limited, and each task takes a different time to complete depending on who executes the task, even having some tasks which cannot be completed by some workers. The procedure is applied to the real world environment which motivated their research to begin with: a sheltered work centre for disabled. Solving the ALWABP in this context helps those centres to provide jobs to the disabled, while considering their individual limitations. This example shows how the field of operations research may be extended to include certain social aims, besides economic and productive aims.

In a more unexpected application of B&B algorithms, Pupko, Pe'er, Hasehawa, Graur, and Friedman (2002) implemented it to reconstruct ancestral gene sequences with among-site-rate-variation. The proposed method appears to efficiently find the global most likely ancestral-sequence reconstruction with maximum likelihood. Reconstruction of ancestral sequences is an important part of evolutionary biology research, as it may ultimately help in synthesising certain hormones, subsequently opening up new ways for treating illnesses.

3.2 Origins of Branch-and-Bound

One of the first to describe a framework for a B&B algorithm were Land and Doig (1960). This was in the context of discrete programming problems, in which some or all variables where restricted to be

nonnegative integers. They described a process in which first the LP relaxation of the problem was solved, i.e. ignoring the constraints for nonnegative integer values of the variables. This would yield an upper bound on the objective value. Then a single fractional variable is selected and its neighbouring integer values determined, adding each as a constraint to the resulting two subproblems. The objective values of all encountered subproblems are recorded in a list. The maximum objective value in that list will then be set as the new upper bound, and its entry disregarded from that list. The subproblem resulting in the new best upper bound will serve as the next branching point. This process is repeated, not branching on infeasible solutions. When an integer solution is found with objective value higher than those of the upper bounds remaining in the list, the process is terminated and the optimal solution is found. This could be described as pruning, be it at the end of the procedure.

While the idea of Land and Doig (1960) was promising, it proved to be somewhat difficult to implement given the storage limitations of computers available at that time. Dakin (1965) proposed a similar algorithm which required far less memory than it's predecessor. Instead of solving each arising subproblem and saving the corresponding simplex-tableau, each branch is followed until an infeasible or integer solution is found. This significantly reduces the storage requirement as only a list of node-specific information has to be saved, yet it may result in visiting parts of the tree which would otherwise not have been visited. Additionally, to ensure integral values, Dakin (1965) applies bounding constraints instead of exact integer constraints like Land and Doig (1960). The algorithm proposed by Dakin (1965) showed to be a great improvement over the mixed-integer algorithm proposed by Gomory (1960), solving the same instance in less than eight minutes while the mixed-integer algorithm of Gomory (1960) was not able to find a solution in over 2000 iterations.

Shortly after, researchers began improving upon the framework of Land and Doig (1960) or extended it to more specific cases, such as the traveling salesman problem (Little, Murty, Sweeney, and Karel, 1963), the plant location problem (Efroymson and Ray, 1966), and more relevant for this research the KP (Balas, 1965; Kolesar, 1967). The aforementioned papers are just a small overview of early B&B papers, yet since then many have adapted the framework and it has become a staple method in the world of combinatorial optimisation. Next to the abundance of literature on B&B algorithms, its importance is additionally emphasised when looking at state-of-the-art general purpose solvers. Some well-known and widely-used solvers are `CPLEX`, `Gurobi`, `LINDO`, and `Xpress-MP`. What these solvers all have in common is that their main solving approach is B&B-based (Atamtürk and Savelsbergh, 2005).

3.3 0-1 Knapsack Problem

The classical 0-1 KP has been well-studied over the last century. Two main approaches come to mind for solving this problem, namely Dynamic Programming (DP) and B&B. DP algorithms are less affected by correlation between profits and weights, yet quickly become impractical for greater capacity sizes (Toth, 1980). Procedures to reduce the amount of variables have been proposed by multiple authors such that DP remains viable, but the effectiveness of variable reduction is limited (Toth, 1980). To that extent, many B&B algorithms have been proposed for the KP, one after the other improving the upper bound. Until then, the solution to the continuous knapsack problem as put forward by Dantzig (1957) was used

as an upper bound, which corresponds to the Linear Programming (LP) relaxation of the 0-1 KP. The branching scheme has remained the same throughout, selecting an item and generating child nodes where the item is either set to one or zero. Martello and Toth (1977) proposed a new upper bound for the knapsack problem, improving on that of Dantzig (1957). Given a list of items ordered on profit-weight ratio, Dantzig (1957) filled the knapsack up to item l, with item $l+1$ not fitting in its entirety for the residual capacity. After that, the residual capacity is filled with item $l+1$. Martello and Toth (1977) proposed a maximum of two upper bounds, distinguished by the decision of whether item $l+1$ is added or not, after the knapsack has been filled up to item l. If item $l+1$ is not added, the residual capacity is filled with item $l+2$. Otherwise, item $l+1$ is added in its entirety, and item l is fractionally removed until maximum capacity is reached. The proposed upper bound proved to be a slight improvement over that of Dantzig (1957).

Martello and Toth (1988) then proposed an upper bound involving the solving of a subset of items on the residual capacity through an elementary binary decision tree, and procedures to reduce the size of a KP instance, as such yielding favourable results. Later, Martello and Toth (1997) discussed an upper bound based on the continuous knapsack solution, but with adding valid inequalities on the cardinality of an optimal solution and solving the Lagrangian relaxation of it. They showed that for large amounts of items and correlated problems the results were promising, solving all instances rather quickly, while other algorithms could not. Since then, literature on the classical 0-1 KP is somewhat sparse. Mostly derivations and extensions thereof appear to be researched, such as the two-constraint 0-1 KP (Martello and Toth, 2003) or the multidimensional KP (Boyer, El Baz, and Elkihel, 2010).

3.4 Knapsack Problem with Conflict Graph

The KPCG is one of those extensions of the classical 0-1 KP which only surfaced relatively recently, with Yamada, Kataoka, and Watanabe (2002) being the first to introduce the KPCG. They utilised a greedy algorithm for the KPCG, incorporating conflict checking to obtain a lower bound, which is then improved upon through a 2-opt neighbourhood search exchange. For the upper bound, the LP-relaxation and subsequently the Lagrangian relaxation of the KPCG is considered. These heuristic approaches are then implemented in the exact method of implicit enumeration. Due to the lower bound possibly not being that strong, an interval reduction algorithm is suggested. This method takes an estimate of the lower bound with the aim of reducing the search interval of the B&B algorithm. If the estimated lower bound does not result in a feasible solution, it may be used as a stronger upper bound. This process may then be repeated until an optimal solution is found.

Moreover, Hifi and Michrafy (2006) proposed a reactive local search algorithm with tabu list for solving the KPCG. It's reactive in the sense that if the solutions tend to cycle often, the algorithm reacts and *degrades* the solution set, diversifying the search and escaping local optima. Configurations which lead to these cycles are then considered tabu, preventing the search to visit known unwanted solution sets. Similar to Yamada et al. (2002), a greedy approach is used to obtain an initial feasible solution, after which an improved feasible solution is constructed through the swapping of already selected items or performing a neighbourhood replacement procedure. Hifi and Michrafy (2007) continued research on

the subject and implemented a B&B algorithm for the KPCG. The upper bound is straight-forward and derived from the LP-relaxation of the KPCG, while the lower bound is a slight alteration of the algorithm proposed in Hifi and Michrafy (2006). In order to improve the algorithms performance, various reduction strategies are applied to the upper bound in an attempt to reduce the size of the subproblems. Additionally, a dichotomous search is suggested to update the lower bound, similar in concept to the interval reduction approach proposed by Yamada et al. (2002), in order to speed up the solving process.

3.5 Bin Packing Problem with Conflicts

Conflict graphs have been introduced to other related problems as well. The BPPC has a conflict graph where the edges denote items which cannot be packed into the same bin. Consequently, each subproblem in the BPPC takes the form of a KPCG. The BPPC is a combination of the BPP and the Vertex Coloring Problem (VCP). Naturally, without any conflicts the BPPC reduces to the BPP, while the BPPC with all item weights being zero reduces to the VCP (Muritiba, Iori, Malaguti, and Toth, 2010). Jansen and Öhring (1997) were one of the first to explicitly touch upon the concept of the BPPC, albeit in the context of job scheduling. Generalising the BPP, jobs are to be assigned to machines, with total execution time per machine being constrained and some jobs not being able to be performed on the same machine. They suggested various polynomial time heuristic algorithms, and yielded upper bounds with a worst-case approximation ratio of 2.7 for perfect conflict graphs. This application can be seen as process assignment, which is one of the real-world applications of the BPPC. Next to the inherent implications of the BPPC for specific distributional packing problems, it may also be applied in the context of examination scheduling (Laporte and Desroches, 1984) and even parallel computing (Jansen, 1999).

Epstein and Levin (2008) tried to improve upon the work of Jansen and Öhring (1997), thus proposing additional approximate methods and as well as extending the framework to the on-line BPPC. With the introduction of weighting systems and removal of small problematic subgraphs, the heuristic procedures ultimately yielded upper bounds with a theoretical worst-case approximation ratio of 2.5 for perfect conflict graphs. Gendreau et al. (2004) also put forward multiple heuristics and provided benchmark instances, which are still used to this day. The most promising heuristic dealt with clique calculation in combination with a first fit decreasing procedure to fill the bins. Moreover, they proposed an improved lower bound through solving a Transportation Problem in order to obtain a solution for packing the bins while taking the the conflict constraints into account.

Muritiba et al. (2010) were the first to discuss an exact method to solve the BPPC, next to proposing yet again improvements upon both lower and upper bounds. They proposed a Branch-and-Price (B&P) algorithm, with the pricing problem being the KPCG. The lower bounds of the algorithm correspond to the LP-relaxation of the Set Covering (SC) formulation of the BPPC, while the upper bounds are obtained through a so-called population heuristic. The population heuristic consists of diversifying a set of upper bounds obtained through known greedy algorithms and performing a local tabu search on them. It may be noted that the SC formulation of the BPPC is a minimisation problem. Their computational results showed to be a fair improvement on those obtained in previously published literature on this

subject.

Subsequently, Elhedhli, Li, Gzara, and Naoum-Sawaya (2011) proposed a special-purpose B&P algorithm with a different lower bounding scheme and branching rule, solving the majority of instances left unsolved by Muritiba et al. (2010). Then Sadykov and Vanderbeck (2013) proposed an efficient generic B&P algorithm, which proved to outperform those algorithms proposed by Muritiba et al. (2010) and Elhedhli et al. (2011). This algorithm differs in several aspects from the aforementioned B&P algorithms. Looking at the branching rule, Muritiba et al. (2010) branch on the largest fractional variable, while Elhedhli et al. (2011) follow the scheme proposed by Ryan and Foster (1981). Sadykov and Vanderbeck (2013) follow the branching scheme as proposed by Vanderbeck (2011), which was developed with the aim of preserving the structure of the subproblems when branching.

Another interesting difference is the approach of solving the subproblems. Muritiba et al. (2010) use a greedy approach and opt for `CPLEX` to solve the subproblem in an exact manner if no improved solution is found heuristically. Elhedhli et al. (2011) utilise `CPLEX` with conflict constraints based on maximal cliques. Sadykov and Vanderbeck (2013) however propose a B&B algorithm for solving the KPCG subproblem in case of arbitrary conflict graphs, and a DP approach for the case of interval conflict graphs. The proposed B&B algorithm posed to be quicker than `CPLEX` for instances with conflict graph densities greater than 10%. When instances with a greater amount of items per bin possible are considered and the conflict graph has no particular structure, the difficulty of solving the problem was found to be higher.

3.6 GPU Computing

The usual course of action when programming any algorithm is to execute it sequentially sing the Central Processing Unit (CPU) of a computer, as the CPU is designed to handle logical operations of a machine. However, when problems become larger, even modern multi-core CPUs can struggle when solving some problems. Graphics Processing Units (GPUs) were tradionally designed to handle image processing, for which they have to perform thousands of simple operations at once. GPUs have evolved tremendously throughout the last decades, and even consumer-grade GPUs now show large potential in the context of parallel computing. Parallel computing may be defined as the act of solving a problem by splitting its domain into multiple parts, and solving each part simultaneously by using multiple physical processors (Navarro, Hitschfeld-Kahler, and Mateu, 2014). As Navarro et al. (2014) note, a problem may be parallelisable in multiple ways. A problem can be data-parallel or task-parallel, both of which intrinsically belong to the GPU and CPU, respectively. Data-parallel problems consist of different data values which have to be applied to the same function. Task-parallel problems consists of different tasks being applied to a common data stream. Due to GPUs having a Single-Instruction Multiple-Threads architecture, a data-parallel problem may see benefit of execution on the GPU (Boyer, El Baz, and Elkihel, 2012).

GPU computing is a relatively recent development in the context of parallel computing, yet very promising. Boyer et al. (2017) present a survey on the advancements of GPU computing in the context of operations research. Many metaheuristics able to exploit parallelism are discussed, but the main exact methods suitable for GPU implementation are the simplex method, DP, and B&B. With such parallelised methods, various problems have been attempted to solve throughout the years, such as the KP (Boyer

et al., 2012), traveling salesman problem (Carneiro, Muritiba, Negreiros, and Lima de Campos, 2011), and the flow-shop scheduling problem (Chakroun, Melab, Mezmaz, and Tuyttens, 2013).

Boyer et al. (2012) proposed a parallel implementation of solving the KP with DP. The program was implemented using NVIDIA's Compute Unified Device Architecture (CUDA) platform, which could be seen as the standard for such implementations. The inner loop over the possible knapsack capacities of the DP algorithm is executed in parallel on the GPU. Their results showed a speedup of factor of around 26 compared to the traditional sequential implementation.

B&B can be a challenge to implement in a parallel manner, as the tree structure often is irregular. Nevertheless, Lalami and El Baz (2012) proposed a parallel B&B algorithm for the KP, where both branching and bounding computations are parallelised. The algorithm transfers the list of nodes to the GPU, in which first the branching operations are performed in parallel, then the bounds for all nodes are computed simultaneously, and lastly the best lower bound is found among the nodes. Then the updated list of nodes is transferred back to the CPU, after which the process is repeated. This implementation showed to decrease the time needed to solve instances by around 20 times.

Gmys, Mezmaz, Melab, and Tuyttens (2016) were the first to propose a B&B algorithm employed entirely on the GPU. Most GPU-based implementations still have to communicate relevant information back to the CPU in order for the algorithm to continue, like the algorithm of Lalami and El Baz (2012). Gmys et al. (2016) perform all of the branching, bounding, selecting, and pruning operations on the GPU. This is possible due to the use of an Integer-Vector-Matrix data structure, as opposed to the conventional Linked-List based data structures. Data transfer times are an important factor in whether GPU implementation is advantageous, hence there being no need for communication between the CPU and GPU until the problem instance is solved is an interesting development. In the context of flow-shop scheduling problem, their algorithm is compared to a *regular* parallel B&B algorithm, and it was able to solve all instances faster with a factor of 3.3 on average.

3.7 Accessibility of GPU Programming

The main drawback of GPU computing is that CUDA, or similar platforms like OpenCL, may not appeal to the average researcher, as it comes with learning a new programming language and requiring an integral understanding of computer architecture. Clarkson, Fumero, Zakkak, Xekalaki, Kotselidis, and Luján (2018) aimed to bypass that hurdle by introducing a new Java-based Virtual Machine (VM) called `TornadoVM`. It allows the user to run parts of their program on parallel devices with little adjustment to their Java code and without needing excessive knowledge about the intricacies of the hardware. `TornadoVM` helps to execute code on OpenCL-compatible devices and is able to do so on any Java Virtual Machine (JVM) compatible architecture (Clarkson et al., 2018).

`TornadoVM` consists of three layers: an Application Programming Interface (API), runtime, and Just-In-Time (JIT) compiler. The API ensures that with code annotations, the compiler can recognise parts of the program which are to be executed on a parallel device. The runtime analyses data dependencies, optimises data transfers, and manages communication between the host and its devices. Lastly, the JIT compiler generates and optimises code for the different devices dynamically. Clarkson et al. (2018) showed

for a Kinect Fusion benchmark, provided through SLAMBench (Nardi, Bodin, Zia, Mawer, Nisbet, Kelly, Davison, Luján, O'Boyle, Riley, Topham, and Furber, 2015), an increase in performance between 18 and 150 times with `TornadoVM`, compared to the standard Java reference.

4 Methodology

4.1 MIP Models

Two MIP models will be examined for the KPCG, both of which will be solved with a general purpose solver. The binary decision variable x_i $(i = 1, ..., n)$ denotes whether an item is selected or not. Model (1a)-(1d) has objective function (1a), which maximizes the total profit associated with the selected items. Constraint (1b) ensures that the weight of the selected items does not exceed the knapsack capacity. Constraint (1c) makes sure that incompatible items are not simultaneously selected. Lastly, constraint (1d) tells us that the decision variables are binary.

$$\begin{aligned}
\max \quad & \sum_{i=1}^{n} p_i x_i && && (1a)\\
\text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq c && && (1b)\\
& x_i + x_j \leq 1 && (i,j) \in E && (1c)\\
& x_i \in \{0,1\} && i = 1, ..., n && (1d)
\end{aligned}$$

Model (2a)-(2d) is equivalent to model (1a)-(1d), but yields a stronger LP-relaxation bound due to the formulation of conflict constraint (2c). The constraint states that at most one item in clique C may be added to the knapsack, for all clique sets C in the family of cliques ξ. Given an undirected graph $G = (V, E)$, a clique set is defined as a set vertices $C \subseteq V$ such that any distinct pair of vertices in C is associated with an edge in E, i.e. $\forall i, j \in C \wedge i \neq j \implies (i, j) \in E$. This translates to C inducing a complete subgraph of G. As defined by Bettinelli et al. (2017), a family of cliques ξ on graph G is a set of cliques such that $\forall (i, j) \in E)$, items $i, j \in V$ belong to some clique $C \in \xi$. The family of cliques will be determined with a heuristic approach (Bettinelli et al., 2017). It takes a random edge $(i, j) \in E$ such that either items i, j are not yet part of any clique. A maximal clique will be built for those items i, j. This entails going through the (sorted) list of items and adding an item to the clique if that item is adjacent to all items in the current clique. This process is repeated until all edges are in at least one clique. Following Bettinelli et al. (2017), model (2a)-(2d) will be applied to the dense instances and model (1a)-(1d) to the sparse instances.

$$\begin{aligned}
\max \quad & \sum_{i=1}^{n} p_i x_i && && (2a)\\
\text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq c && && (2b)\\
& \sum_{i \in C} x_i \leq 1 && C \in \xi && (2c)\\
& x_i \in \{0,1\} && i = 1, ..., n && (2d)
\end{aligned}$$

4.2 Branch-and-Bound Algorithm

A general description of a B&B procedure boils down to an intelligent search over the space of all feasible solutions. In the context of KPs, the process can be represented by a tree. The root of the tree corresponds to no items having been added to the knapsack. Given a greedy approach of item ordering, subtrees will be made based on (not) locking the first item in the list of available items into the knapsack. An upper bound will be calculated for the entire subtree, which may result in termination of the branching procedure at that node if the found upper bound is lower than the current best solution, i.e. the (lower) bound. When branching, an internal upper bound specific to the item in consideration will ultimately determine whether an item will be locked into the knapsack at that point in the tree.

In describing the B&B framework, we will follow Bettinelli et al. (2017) closely, as such we will first clarify some notation. Each node in the B&B tree is associated with a subproblem defined by the two sets of items S and F. $S \subseteq V$ is a stable set of items currently packed in the knapsack, while $F \subseteq V$ is the set of unassigned items ordered on profit-weight ratio. A stable set could be seen as the opposite of a clique set, defined as a set of vertices $S \subseteq V$ such that for any distinct pair of vertices in S there is no edge in E, i.e. $\forall u, v \in S \wedge u \neq v \implies (u, v) \notin E$. To simplify notation, we denote the profits and weight corresponding to some solution $Z \subseteq V$ by $p(Z)$ and $w(Z)$, respectively. The current global lower bound will be abbreviated to LB. Then, $N(i)$ denotes the set of nodes adjacent to node i, i.e. $N(i) = \{j \in V : (i, j) \in E\}$. Lastly, $KP(Z, \bar{c})$ denotes the value of the optimal solution for the 0-1 KP, given some set of items Z and capacity $\bar{c}$.

Algorithm 1: Generic B&B scheme for the KPCG: $BB(S, F, LB)$

```
Data: Set of items S currently in the knapsack; set of available items F; current lower bound
      LB.
if LB < p(S) then
    LB = p(S)
if Upperbound(F,c-w(S)) + p(S)≤ LB then
    return
for i ∈ F do
    if UB_i > LB then
        F = F \ {i}
        BB(S ∪ {i}, F \ N(i), LB)
    else
        break
```

Having defined those, Algorithm 1 shows a generic B&B scheme in which all proposed variations of the algorithm will be implemented (Bettinelli et al., 2017). In describing the algorithm, we will take the implementation of Sadykov and Vanderbeck (2013) as reference point, which we'll denote by $SV13$ henceforth. The scheme first checks if the profits of the current solution S exceed the lower bound, if that's the case a new global lower bound has been found. Then an upper bound for the available items F on residual capacity $c - w(S)$ is calculated. Sadykov and Vanderbeck (2013) use the LP-relaxtion of model (1a)-(1d) for the upper bound.

This entails solving model (3a)-(3c) and possibly results in a non-integer solution, thus we will refer to this bound as ${}_{frac}KP$. If that upper bound plus $p(S)$ is lower than the current bound, the subproblem does not improve upon the current solution and the scheme will not continue at that node. Continuing the

scheme, an internal upper bound UB_i will be calculated for each item $i \in F$. The upper bound following the $SV13$ algorithm equals $UB_i = p(S)+(c-w(S))(p_i/w_i)$, which translates to calculating the maximum profit possible to obtain by filling the residual capacity with item i. If that upper bound appears to be an improvement on the current best found objective value, branching will continue by adding item i to S and removing neighbours $N(i)$ from F. Initially, the function as defined in Algorithm 1 will be called as $BB(\varnothing, V, 0)$.

$$\max \quad \sum_{i=1}^{n} p_i x_i \tag{3a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{3b}$$

$$0 \leq x_i \leq 1 \quad i = 1, ..., n \tag{3c}$$

4.2.1 Branching Scheme

Bettinelli et al. (2017) propose an improved internal upper bounding procedure to that of Sadykov and Vanderbeck (2013). Instead of filling the residual weight of the knapsack with item $i \in F$, they propose to solve the 0-1 KP for the residual weight with items {i,...,n}, ignoring possible conflicts. Mathematically, this entails $\widetilde{UB_i} = p(S)+KP(\{i, ..., n\}, c-w(S))$, which we'll denote by ${}_{pre}KP$ from now on. As suggested by Bettinelli et al. (2017), the optimal solution $KP(Z, \bar{c})$ will be determined using a DP approach (Toth, 1980). It can clearly be seen that $\widetilde{UB_i} < UB_i$, which should make itself apparent in the relative amount of nodes visited. Bettinelli et al. (2017) compute $K(Z, \bar{c})$ in a preprocessing phase for all $i = 1, \ldots, n$, $Z = \{i, \ldots, n\}$, $\bar{c} = 0, \ldots, c$. This is not done in this research due to time constraints, which may result in increased solving times compared to Bettinelli et al. (2017).

4.2.2 Weighted Clique Cover Bound

A weighted clique cover is a set of cliques K_r $(r = 1, \ldots, R)$ of graph G, of which each has weight Π_r $(r = 1, \ldots, R)$, such that $\sum_{r=1,\ldots,R:i\in K_r} \Pi_r \geq p_i, \forall i \in V$ (Bettinelli et al., 2017). The weight of a clique cover is subsequently defined as $\sum_{r=1}^{R} \Pi_r$. Held et al. (2012) proposed an upper bound using a weighted clique cover for the maximum weight stable set problem (MWSSP), which Bettinelli et al. (2017) adopted and afterwards modified for the KPCG. This makes sense due to the fact that the KPCG is an extension of the MWSSP. The aim of the MWSSP is to find the maximum weight stable set of a graph G without some capacity constraint, with each item having weight w_i $(i = 1, \ldots, n)$. The KPCG extends upon the MWSSP by letting those weights function as profits p_i $(i = 1, \ldots, n)$, adding weights w_i $(i = 1, ..., n)$ for each item, and adding a capacity constraint. The optimal objective value of the MWSSP is less than or equal to the weight of any clique cover of G, consequently the weight of any such clique cover is an upper bound for the MWSSP (Held et al., 2012). Given the similarities between the MWSSP and the KPCG, any upper bound for the MWSSP also is an upper bound for the KPCG (Bettinelli et al., 2017).

In discussing Algorithm 2, we'll first introduce an additional parameter q_i, denoting the *residual weight* of item i. Initially, q_i is set equal to the item's profit for each item $i \in V$. While there still exists an item with positive residual weight, the index $\bar{i}$ for which q_i is smallest but positive is found. A maximal clique

Algorithm 2: Weighted Clique Cover

$q_i = p_i \quad i \in V$
$r = 0$
while $\exists i \in V : q_i > 0$ **do**
  $\bar{i} = \text{argmin}\{q_i : q_i > 0, i \in V\}$
  $r = r + 1$
  Find clique $K_r \subseteq \{j \in N(\bar{i}) : q_j > 0\}$
  $K_r = K_r \cup \{\bar{i}\}$
  $\Pi_r = q_{\bar{i}}$
  $q_j = q_j - q_{\bar{i}} \quad \forall j \in K_r$

K_r is found heuristically for that item $\bar{i}$. Lastly, the weight Π_r of clique K_r is set to the residual weight of item $\bar{i}$, and q_i is subtracted from the residual weights of all items in the clique. This procedure yields an upper bound for the KPCG equal to $\sum_{r=1}^{R} \Pi_r$, which we'll denote by CC. It should be noted that the stronger the capacity constraint of the KPCG is, the weaker CC may be as an upper bound.

4.2.3 Capacitated Weighted Clique Cover Bound

As Bettinelli et al. (2017) noted, the weighted clique cover bound does not take the knapsack capacity into account. Hence the weighted clique cover bound is extended to incorporate the capacity constraint. The proposed method either finds a complete or partial weighted clique cover, it being partial if the knapsack is saturated before a complete clique cover has been found. The set of items not covered by the partial clique cover Ξ is defined as $\underline{V} = \{i \in V : \sum_{K_r \in \Xi : i \in K_r} \Pi_r < p_i\}$, such that the set of fully covered items equals $\bar{V} = V \setminus \underline{V}$. Furthermore, we define the load W_r of a clique K_r as in equation (4), from which follows that the load-over-weight ratio of a clique K_r may not exceed the lowest weight-profit ratio of the items in K_r (Bettinelli et al., 2017).

$$W_r \leq \Pi_r \min_{j \in K_r} \left\{ \frac{w_j}{p_j} \right\} \tag{4}$$

$$\sum\nolimits_{K_r \in K} W_r = c \tag{5}$$

$$\min_{K_r \in \Xi} \left\{ \frac{\Pi_r}{W_r} \right\} \geq \max_{j \in \underline{V}} \left\{ \frac{p_j}{w_j} \right\} \tag{6}$$

Theorem 1 in Bettinelli et al. (2017, p. 461) then states two conditions which needs to be satisfied for the weight $\sum_{K_r \in \Xi} \Pi_r$ of a partial clique cover Ξ such that it is a valid upper bound for the KPCG, as shown in equation (5) and (6). Equation (5) denotes that the sum of the loads of all cliques should equal the knapsack capacity, while equation (6) describes that the most valuable items should be part of the clique cover.

Algorithm 3 incorporates these two constraints, continuously satisfying equation (6) and terminating when equation (5) is satisfied or a complete clique cover is found. First, the index corresponding to the item with the smallest weight-profit ratio is found, given that its residual weight q_i is positive. Similar to before, a maximal clique containing item $\bar{i}$ is then found heuristically. Subsequently, the load and weight of the clique are updated. To that extent, first the item t having the lowest residual weight in the clique is determined. Bearing equation (4) in mind, the load W_r is then set to the the minimum of q_t multiplied

Algorithm 3: Capacitated Weighted Clique Cover

$q_i = p_i \quad i \in V$
$r = 0$
while $\exists i \in V : q_i > 0 \wedge \sum_{h=1}^{r} W_h < c$ **do**
  $\bar{i} = \text{argmin}\{w_i/p_i : q_i > 0, i \in V\}$
  $r = r + 1$
  Find clique $K_r \subseteq \{j \in N(\bar{i}) : q_j > 0\}$
  $K_r = K_r \cup \{\bar{i}\}$
  $t = \text{argmin}\{q_t : q_i > 0, t \in K_r\}$
  $W_r = \min\{q_t w_{\bar{i}}/p_{\bar{i}}, c - \sum_{h=1}^{r-1} W_h\}$
  $\Pi_r = W_r/(w_{\bar{i}}/p_{\bar{i}})$
  $q_j = q_j - q_t \quad \forall j \in K_r$

with the minimum weight-profit ratio in the clique, and the residual load with respect to the capacity. The weight of the clique is then set conform with equation (6) and the minimum residual weight q_t in the clique is subtracted from the residual weights of all its nodes. Similarly to CC, the upper bound equals $\sum_{r=1}^{R} \Pi_r$, which we'll denote by ${}_{cap}CC$. Propositions 7 and 8 in Bettinelli et al. (2017, p. 463) state that ${}_{cap}CC$ should theoretically dominate CC and ${}_{frac}KP$, respectively. In the case of CC, the proposition is conditional on the fact that both ${}_{cap}CC$ and CC consider the same cliques in the same order. If that were the case, ${}_{cap}CC$ would surely dominate CC as an upper bound, given that it takes the capacity into account. If the conflict constraints were to be disregarded, the algorithm would only encounter cliques with one item, and the algorithm would terminate when the knapsack capacity is reached. This would results in the ${}_{frac}KP$ upper bound. Hence including the conflict constraints would yield a tigher upper bound for the KPCG than ${}_{frac}KP$.

4.3 GPU-based Branching Scheme

The internal upper bound ${}_{pre}KP$ can be transformed to be implemented in a parallel manner on a GPU. Let us denote this implementation by ${}_{par}KP$. In order to achieve that, the DP algorithm of Toth (1980) is adjusted to the parallel DP framework of Boyer et al. (2012). The DP approach for the KP has two dimensions, namely the item and weight count. In each item iteration, it computes whether it is worth adding it to the knapsack for all possible capacities, given information about the objective values for different capacities derived from items which have preceded it. From this, we can deduce that along the item dimension there are dependencies. However, the weight dimension has no dependencies and could thus be computed simultaneously.

To that extent, Algorithm 4 shows the sequential part of the parallel DP algorithm for the KP. It computes for each item $k = 0, \ldots, n$ the maximum profit for each capacity value $l = 0, \ldots, \bar{c}$ up to item k. inputDP and outputDP alternately switch their function in the method for better memory management (Boyer et al., 2012). This can be done as only outputDP is changed in the internal loop, and that array needs to serve as input in the next iteration. More technically, creating new outputDP arrays is inefficient in the case of our implementation. The input and output array have to be streamed to the GPU for each item iteration. If the memory address of those arrays remain unchanged, memory occupancy is decreased, and the chance of errors due to changed memory addresses is mitigated. Ultimately, either outputDP or inputDP will contain the optimal objective value, depending on n.

Algorithm 4: Dynamic Programming Knapsack Problem: KP(Z,$\bar{c}$)

Data: Set of items Z ($|Z| = n$) and capacity $\bar{c}$
Create arrays inputDP and outputDP, containing zero values
for $k \leftarrow 0$ **to** n **do**
  if k even **then**
    parallelLoop(inputDP,outputDP,$\bar{c}$,w_k,p_k)
  else
    parallelLoop(outputDP,inputDP,$\bar{c}$,w_k,p_k
)
if k even **then**
  return outputDP[$\bar{c}$]
else
  return inputDP[$\bar{c}$]

Algorithm 5 shows what will be executed simultaneously on the GPU for all $l = 0, \ldots, \bar{c}$. If an item fits the capacity, i.e. $w_k \leq l$, it is checked whether adding the item is more profitable than excluding it for that particular capacity. If that's not the case, or the item doesn't fit the capacity, the previous profit value for that capacity is kept.

Algorithm 5: Parallel Internal Loop: parallelLoop(inputDP,outputDP,$\bar{c}$,w_k,p_k)

Data: Arrays inputDP and outputDP of length $\bar{c}$, capacity $\bar{c}$, weight w_k, and profit p_k
parallel for $l \leftarrow 0$ **to** $\bar{c}$ **do**
  if $w_k \leq l$ **then**
    if *inputDP(l)* $<$ *inputDP[*$l - w_k$*]* $+p_k$ **then**
      outputDP[l] = inputDP[$l - w_k$] $+p_k$
    else
      outputDP(l) = inputDP[l]
  else
    outputDP[l] = inputDP[l])

It is well known and easily observed that the sequential DP algorithm for the KP runs in pseudopolynomial time, namely O(cn). Strictly speaking that doesn't change for the parallel implementation, as the number of operations does not change. However, the potential speedup factor does depends on the problem size. As the algorithm is parallelised along the capacity dimension, this is where the most noticeable speedup will be observed first. Transferring the relevant data to the GPU might not even be worth it for smaller capacities. It seems likely that noticeable speedups will become apparent when the knapsack capacity sufficiently large enough such that the parallel implementation is worth it. In that case, greater knapsack sizes will potentially make even better use of it, as effective parallelisation will take place a larger amount of times. Strictly speaking, if the capacity is sufficiently large and enough processing units are available, all c iterations can efficiently be performed concurrently on the GPU. That is the desired scenario at which perceptible differences will likely be obtained.

5 Data

The data used in this research has been obtained at `or.dei.unibo.it`, as generated and used by Bettinelli et al. (2017). The data is split into two groups based on conflict graph densities. As put forward by Sadykov and Vanderbeck (2013), the first group has randomly generated conflict graphs with densities ranging from 0.1 to 0.9 and consists of eight classes. The first four classes have $n \in \{120, 250, 500, 1000\}$

items, respectively. The item weights are drawn from a uniform distribution such that $w_i \in U[20, 100]$ $(i = 1, \ldots, n)$ and the knapsack capacity c equals 150. The last four classes have $n \in \{60, 120, 249, 501\}$ items, respectively, with the items having been generated in triplets such that an exact packing in the knapsack is formed. Sadykov and Vanderbeck (2013) clarify these triplets further, stating that the weight of every third item is determined in such a way that an optimal solution requires $n/3$ bins filled to capacity with three items. To be more specific, this entails that the weight of every third item i_{3s} $(s = 1, \ldots, n/3)$ should equal $w_{i_{3s}} = c - w_{i_{3s-1}} - w_{i_{3s-2}}$. Akin to the first four classes, the item weights are defined as $w_i \in U[250, 500]$ $(i = 1, \ldots, n)$ and the knapsack capacity c equals 1000. The approach of generating all abovementioned values ultimately originates from Gendreau, Laporte, and Semet (2004), who built on the test instances proposed by Falkenauer (1996) for the BPP. However, the KPCG requires that profits are associated with each of the items, unlike the BPP from which the aforementioned parameter values were derived. Bettinelli et al. (2017) have generated the profit values using two approaches: a random approach, where $p_i \in U[1, 100]$ $(i = 1, \ldots, n)$; and a correlated approach with $p_i = w_i + 10$ $(i = 1, \ldots, n)$. These characteristics are summarised in Table 1.

Table 1: Summary of Class Characteristics for Dense Datasets

Class	n	c	w_i	p_i^{rand}	p_i^{corr}
1	120	150	$U[20, 100]$	$U[1, 100]$	$w_i + 10$
2	250	150	$U[20, 100]$	$U[1, 100]$	$w_i + 10$
3	500	150	$U[20, 100]$	$U[1, 100]$	$w_i + 10$
4	1000	150	$U[20, 100]$	$U[1, 100]$	$w_i + 10$
5	60	1000	$U[250, 500]$	$U[1, 100]$	$w_i + 10$
6	120	1000	$U[250, 500]$	$U[1, 100]$	$w_i + 10$
7	249	1000	$U[250, 500]$	$U[1, 100]$	$w_i + 10$
8	501	1000	$U[250, 500]$	$U[1, 100]$	$w_i + 10$

Additionally, a capacity multiplier $(m \in \{1, 3, 10\})$ is applied to obtain instances where more items fit in the knapsack. In total 4320 instances are available of the first group, and each set of instances is identified by a capital letter, followed by an integer number denoting the capacity multiplier. The capital letters C and R denote the profits in the dataset to be correlated or random, respectively. Due to the characteristics of class four and the datasets with capacity multiplier 10 and the time constraints under which this research has been performed, two decisions have been made on how and if they're to be solved. A subset containing half the instances of class 4 is considered, alternately picking nine instances from class four. This ensures that the ratio of density values over the dataset remains constant, and that the chosen instances are reasonably spread over the class. Datasets with capacity multiplier 10 are not considered at all, as it has been found that solving the other datasets already took more time than expected.

The second group of datasets concerns instances with conflict graphs exhibiting sparse density values. Bettinelli et al. (2017) consider graph densities of 0.001, 0.002, 0.005, 0.01, 0.02, and 0.05. For each of these densities, $n \in \{500, 1000\}$ items are considered for knapsack capacities $c \in \{1000, 2000\}$. Item weights are once more uniformly distributed such that $w_i \in U[1, 100]$ $(i = 1, \ldots, n)$, which altogether results in 24 classes. The decision on these parameter values take heavy inspiration from Hifi and Michrafy

(2007) and Yamada et al. (2002), with both papers using a similar framework with slightly different values compared to what has been applied here. The associated profit values are defined the same as for the first group, of which for each class 10 instances are generated for both the random and correlated variants. These characteristics are summarised in Table 2.

Table 2: Summary of Class Characteristics for Sparse Datasets

Class	n	c	w_i	p_i^{rand}	p_i^{corr}
1	500	1000	$U[1, 100]$	$U[1, 100]$	$w_i + 10$
2	500	2000	$U[1, 100]$	$U[1, 100]$	$w_i + 10$
3	1000	1000	$U[1, 100]$	$U[1, 100]$	$w_i + 10$
4	1000	2000	$U[1, 100]$	$U[1, 100]$	$w_i + 10$

This results in two datasets for this group, distinguished by its profits being either random or correlated. Due to the sparse instances not being the main focus of the research and time constraints, a subset of the datasets are considered. Half of the instances are considered, such that the ratio of density values over each dataset is constant and instances are chosen alternately for every two instances.

6 Results

All solvers are implemented in Java, with custom code for the B&B algorithms and the MIP models using the `CPLEX 12.6.3.0` library. The workstation used for testing is equipped with an Intel i5-3570K 3.4 GHz CPU and 16 GB RAM, running the Windows 10 operating system. The solvers are subject to a time limit of 1800 seconds per instance. Any deviation from the aforementioned will be explicitly stated. All Java code can be found in Appendix B.

6.1 Upper Bound Performance

To analyse the performance of each of the implemented upper bounds, their optimality gaps in percentages are determined. These values are obtained by computing $100(UB - z^*)/z^*$, with z^* denoting the optimal (or best-known) objective value corresponding to the instance of interest.

Tables 3 and 4 show descriptive statistics on the optimality gaps of the proposed upper bounds, aggregated by dataset and density, respecitvely. Upper bound ${}_{frac}KP$ performs moderately well, yet its performance becomes worse for increased knapsack capacities and densities, as shown in Table 3 and 4, respectively. As discussed in Section 4.2, ${}_{frac}KP$ is the LP-relaxation of model (1a)-(1d), without conflict constraints and resulting in a non-integer solution. Both these factors explain the conditions for ${}_{frac}KP$ to decrease in performance.

Upper bound CC shows better performance in datasets with higher capacity, and performs worse for lower conflict graph densities, as shown in Table 3 and 4, respectively. As discussed in Section 4.2.2, tighter capacity constraints would indeed theoretically result in a higher optimality gap, due to CC ignoring the capacity constraint of the KPCG. Lower conflict graph densities entails that the majority of the cliques added to CC will only contain one item. That in combination with the absence of a capacity

constraint will result in higher optimality gaps for lower densities. Compared to ${}_{frac}KP$ and ${}_{cap}CC$, CC's performance is the worst overall.

The performance of ${}_{cap}CC$ worsens under the same conditions as ${}_{frac}KP$, yet in all cases outperforms it, and in most cases with a large factor. As discussed in Section 4.2.3, ${}_{cap}CC$ should indeed theoretically dominate ${}_{frac}KP$ and CC under some conditions. Due to the approach of obtaining the cliques being heuristic based, the proposition of Bettinelli et al. (2017) as discussed in Section 4.2.3 does not hold theoretically, but is nevertheless confirmed numerically.

Table 3: Upper Bounds Percentage Optimality Gaps (Minimum, Average, Maximum)

	${}_{frac}KP$			CC			${}_{cap}CC$		
Dataset	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
R1	0.48	29.81	116.79	4.91	579.09	3,023.49	0.02	18.88	57.41
R3	2.12	96.94	393.20	4.91	209.28	946.35	0.26	40.21	141.21
C1	0.00	4.15	24.07	101.19	2,978.41	20,868.46	0.00	2.92	15.49
C3	0.21	20.74	162.13	26.65	544.88	3,535.39	0.12	14.80	107.82

Note: Results aggregated by dataset.

Table 4: Upper Bounds Percentage Optimality Gaps (Minimum, Average, Maximum)

	${}_{frac}KP$			CC			${}_{cap}CC$		
Density	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
0.1	0.00	6.54	29.96	72.84	2,269.63	20,868.46	0.00	4.67	29.86
0.2	0.00	9.61	45.63	55.67	1,755.85	15,955.00	0.00	6.19	30.24
0.3	0.00	16.61	66.32	24.90	1,405.09	12,689.00	0.00	8.18	32.86
0.4	0.00	18.65	78.02	29.03	1,175.53	10,472.73	0.00	11.12	38.47
0.5	0.00	26.88	126.07	40.31	962.63	8,292.31	0.00	15.80	61.99
0.6	0.00	36.18	159.26	24.09	781.78	6,566.33	0.00	20.78	74.87
0.7	0.00	50.73	218.36	24.10	606.64	5,013.00	0.00	28.09	90.23
0.8	0.00	71.12	278.98	10.67	450.12	3,636.95	0.00	35.78	115.75
0.9	0.95	107.88	393.20	4.91	293.96	2,243.96	0.65	42.22	141.21

Note: Results aggregated by density.

The obtained results for the upper bounds coincide quite well with those of Bettinelli et al. (2017). ${}_{frac}KP$ should not come as a surprise, due to the method being exact. In this case, small deviations are explained by the fact that a subset of class 4 is examined in this research. Moreover, ${}_{cap}CC$ also compares very well, even though it is a heuristic-based method. As heuristics are not exact, discrepancies may occur and may explain deviations. Interestingly enough, our implementation of CC outperforms that of Bettinelli et al. (2017) in the majority of cases, except for graph densities lower than 0.4.

6.2 Dense instances

Following the method of Dolan and Moré (2002), performance profiles of the different solvers for datasets R1 and C1 are shown in Figure 1 and 2, respectively. The performance profiles plot the cumulative distribution function for the *performance ratio* against τ. The performance ratio is defined as the solving time of a solver for a problem instance, normalised against the fastest solving time for that instance.

Then for each τ, the graph shows for each solver the fraction of instances solved at most τ times slower than the fastest solver.

Figure 1 and 2 show that the custom B&B algorithms outperform `CPLEX` for these datasets. We can see that $SV13$ is the fastest algorithm for 87% of all instances for the random dataset, and solves 95% of the instances within four times the solving time of the fastest algorithm ($\tau = 4$). Compare this to $BCM17$, only being the fastest algorithm for 2% of the instances and only solving 27% at $\tau = 4$. $SV13$ with ${}_{pre}KP$ performs similar to $BCM17$, only showing itself to be a slight improvement over it for this dataset. For the correlated dataset the results for the B&B algorithms are a bit more ambiguous. Still $SV13$ is the fastest algorithm with 50% of the instances solved the fasted, followed closely by $SV13$ with ${}_{pre}KP$, then by $BCM17$. However, around $\tau = 4$ their performances seem to converge somewhat, with $BCM17$ lagging behind a bit.

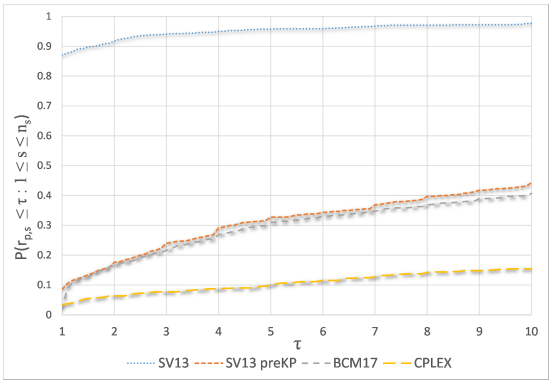

Figure 1: Performance Profiles for SV13, SV13 with ${}_{pre}KP$, BCM17, and CPLEX model (2a)-(2d) for dataset R1

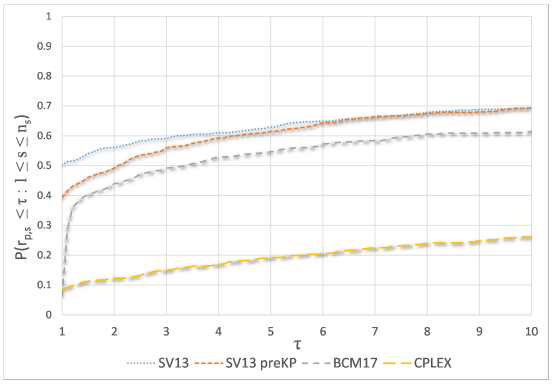

Figure 2: Performance Profiles for SV13, SV13 with ${}_{pre}KP$, BCM17, and CPLEX model (2a)-(2d) for dataset C1

Figures 5 and 6 show more elaborate results for the datasets with capacity multiplier one, aggregated by class and density, respectively. For the random dataset, all B&B algorithms seem to have no real difficulty with the instances. `CPLEX` is not doing bad in absolute term, but not really well in either. It has been noted as well that the time to build a clique family in the majority of cases far exceeds the solving times of the B&B algorithms. This, in combination with time constraints, has led to the decision to not solve class four with `CPLEX`. The column for (2a)-(2d) in Table 6 thus shows somehwat biased results, yet including class four would only have increased its values, thus relatively the B&B algorithms would still come out superior in this case. $SV13$ appears to have the greatest difficulty with the instances of class eight and those with a lower density. Noticing the amount of nodes $SV13$ visits compared to the other algorithms, it performs well. This is likely due to the simple structures implemented in the algorithm, saving computational time in that respect. $SV13$ with ${}_{pre}KP$ and $BCM17$ perform very similarly, from solving times to nodes visited, although $SV13$ with ${}_{pre}KP$ seems to have a slight edge over $BCM17$. Instances of class four and eight appear the most difficult for these methods. There is some variation in solving times for the different density levels, yet firm deductions cannot be made from Table 6 on the performance impact for these datasets.

Table 5: Computational Results for Datasets with Capacity Multiplier 1

	$SV13$			$SV13$ with $_{pre}KP$			$BCM17$			(2a)-(2d)	
Class	Solved	Time	Nodes	Solved	Time	Nodes	Solved	Time	Nodes	Solved	Time
						R1					
1	90	0.001	204.0	90	0.011	34.9	90	0.129	32.1	90	1.627
2	90	0.006	475.0	90	0.069	61.0	90	0.087	56.6	90	4.045
3	90	0.067	1,767.4	90	0.482	150.7	90	0.631	142.0	90	12.659
4	45	0.255	3,396.8	45	3.146	363.3	45	5.003	347.4	x	—
5	90	0.001	236.0	90	0.010	16.1	90	0.011	14.9	90	1.048
6	90	0.016	1,116.8	90	0.058	25.1	90	0.062	24.4	90	2.099
7	90	0.428	7,185.1	90	0.521	44.7	90	0.540	44.3	90	6.649
8	90	10.960	46,645.5	90	4.203	65.7	90	4.246	65.7	90	184.801
						C1					
1	90	0.347	18,202.4	90	0.014	79.0	90	0.019	77.0	90	3.591
2	89	51.066	565,414.2	90	0.096	153.0	90	0.110	149.9	90	11.862
3	85	45.262	472,444.9	90	0.534	273.8	90	0.633	269.4	90	25.096
4	31	17.300	2,251,476.0	45	3.683	718.7	45	4.909	705.6	x	—
5	90	0.139	15,186.7	90	0.028	135.6	90	0.029	132.6	90	1.035
6	90	2.958	93,479.0	90	0.112	163.9	90	0.111	161.9	90	1.675
7	90	11.493	242,237.3	90	0.339	152.4	90	0.347	152.4	90	3.095
8	89	56.034	817,845.0	90	1.643	200.8	90	1.676	200.8	90	14.041

Note: Results aggregated by class. * denotes values subject to change before final version. x denotes classes which have not been attempted to solve.

Table 6: Computational Results for Datasets with Capacity Multiplier 1

	$SV13$			$SV13$ with $_{pre}KP$			$BCM17$			(2a)-(2d)	
Density	Solved	Time	Nodes	Solved	Time	Nodes	Solved	Time	Nodes	Solved	Time
						R1					
0.1	75	8.325	22,994.5	75	1.399	35.5	75	1.392	35.5	69	2.733
0.2	75	3.167	15,895.0	75	0.725	33.7	75	0.738	33.7	69	4.065
0.3	75	1.249	10,489.0	75	0.540	40.9	75	0.573	40.9	69	3.350
0.4	75	0.704	9,121.5	75	0.762	53.9	75	0.834	53.9	70	5.362
0.5	75	0.341	6,601.2	75	0.731	64.1	75	0.835	64.1	70	7.861
0.6	75	0.126	3,299.1	75	0.771	92.2	75	1.023	92.2	70	10.976
0.7	75	0.050	1,722.9	75	0.786	134.0	75	1.160	132.6	71	19.775
0.8	75	0.021	739.5	75	1.125	113.3	75	1.403	107.1	71	41.655
0.9	75	0.013	334.1	75	1.472	128.3	75	1.751	104.4	71	174.617
						C1					
0.1	64	99.466	614,350.4	75	0.213	78.6	75	0.216	78.6	70	3.294
0.2	70	23.067	391,564.5	75	0.368	105.5	75	0.373	105.5	70	4.203
0.3	70	19.102	396,999.9	75	0.491	134.0	75	0.500	134.0	70	4.061
0.4	75	28.577	429,176.3	75	0.343	129.8	75	0.353	129.8	70	4.061
0.5	75	30.363	462,309.3	75	0.390	169.6	75	0.442	169.6	70	6.701
0.6	75	16.160	429,030.7	75	0.724	268.1	75	0.946	268.1	70	8.271
0.7	75	3.370	151,099.8	75	0.613	265.6	75	0.737	265.1	70	12.165
0.8	75	0.644	52,238.2	75	0.900	303.9	75	1.196	301.8	70	13.272
0.9	75	0.115	9,530.8	75	1.487	366.5	75	1.692	343.6	70	21.621

Note: Results aggregated by density. (2a)-(2d) column subject to change before final version.

The correlated dataset shows that indeed the proposed improvements of Bettinelli et al. (2017) are important, with $SV13$ beginning to exhibit difficulty solving the instances. Relative to the other algorithms, solving times are very high, amount of nodes visited becomes very large, and it is not able to solve all instances to optimality anymore. Table 6 shows these cases mainly consist of instances with lower densities. Similar to the random dataset, both $SV13$ with $_{pre}KP$ and $BCM17$ perform well, with the amount of nodes visited being low and dominating $SV13$ and `CPLEX`. The difference in upper bounds

used by the two algorithms do not seem to have much of an impact on their respective solving capabilities for these datasets.

Now we'll discuss the datasets with capacity multiplier three. First it must be said that $SV13$ is not considered and `CPLEX` for MIP model (2a)-(2d) only partially. $SV13$ had inconsistent performance for the previous datasets, and the other B&B algorithms were designed to improve upon it. The solving times of `CPLEX` for model (2a)-(2d) were relatively slow and misleading for the previous datasets. The solving times themselves are reliable, yet in order to run `CPLEX`, a clique family must be built first. It has been found by examining a small amount of instances that the time to build a clique family is at most moderate, around 60 seconds for the highest density, when item amounts don't exceed 250. If there are more than 250 items, the building time can be moderate for lower densities to excessive for higher densities, mostly far exceeding the time limit of 1800 seconds. To this extent, classes three, four and eight will be excluded for now. These factors in combination with the available time for this research has resulted in the decision to exclude or to partially examine these methods for the following datasets.

Figure 3 and 4 show the performance profiles for $SV13$ with ${}_{pre}KP$ and $BCM17$ for the datasets R3 and C3, respectively. Contrary to the previous performance profiles, a clear indication of the superior performance of $BCM17$ is apparent for both datasets. In both cases $BCM17$ is the fastest solver for approximately 73% and 66% of the instances for the random and correlated datasets, respectively. $SV13$ with ${}_{pre}KP$ on the other hand is only the fastest for roughly 14% and 10%, respectively. Within two times ($\tau = 2$) the solving time of the fastest algorithm, $BCM17$ solves about at 84% of R3 and 74% of C3, yet $SV13$ with ${}_{pre}KP$ solves approximately 50% and 43% respectively. Assuming these algorithms to be faster than $SV13$ and `CPLEX`, these findings show great potential for $BCM17$, especially in this comparison the potential of ${}_{cap}CC$. It may be noted that the algorithms are not able to solve all instances anymore, with $BCM17$ converging to approximately 85% and 75% solved for the random and correlated datasets, respectively.

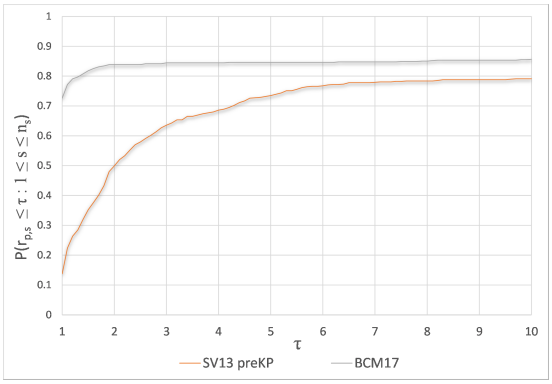

Figure 3: Performance Profiles for SV13 with ${}_{pre}KP$ and BCM17 for dataset R3

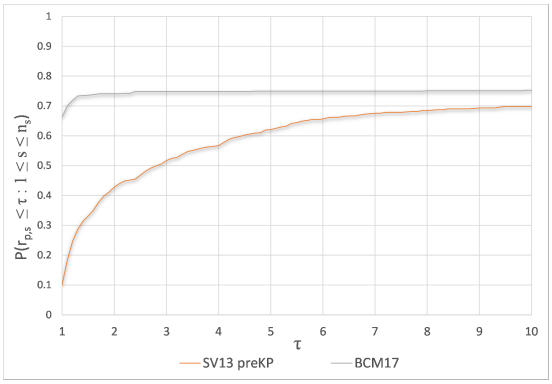

Figure 4: Performance Profiles for SV13 with ${}_{pre}KP$ and BCM17 for dataset C3

Table 7 and 8 show more detailed computational results for these datasets, aggregated by class and density, respectively. As discussed before, it was decided to not solve all classes with `CPLEX`, which is also the reason why (2a)-(2d) is absent from Table 8, as it would be an unfair comparison. This time around, both algorithms seem to show more difficulty in solving the instances. As Table 7 shows, classes three,

four, seven and eight cannot be solved entirely anymore for the random dataset, while for the correlated dataset the majority of the classes cannot be solved in their entirety anymore. In other words, classes

Table 7: Computational Results for Datasets with Capacity Multiplier 3

	$SV13$ with $_{pre}KP$			$BCM17$			(2a)-(2d)		
Class	Solved	Time	Nodes	Solved	Time	Nodes	Solved	Time	Clique
				R3					
1	90	10.035	11,432.5	90	13.130	8,658.1	90	1.656	0.484
2	90	183.529	119,724.0	90	156.070	72,221.4	90	6.914	13.228
3	51	401.918	185,811.8	55	315.803	98,320.9	x	—	—
4	16	470.918	570,795.6	15	550.49	80,744.7	x	—	—
5	90	9.766	1,915.2	90	3.253	759.4	90	0.780	0.027
6	90	81.385	11,568.2	90	31.028	5,584.0	90	1.853	0.534
7	84	418.752	41,376.8	89	187.186	20,254.3	90	8.088	13.250
8	45	588.135	40,973.1	61	664.291	24,274.7	x	—	—
				C3					
1	87	156.857	78,455.6	90	59.110	64,754.1	90	2.394	0.484
2	58	251.979	228,007.8	67	307.494	236,131.3	90	28.870	13.228
3	36	474.870	495,290.1	36	321.007	107,732.0	x	—	—
4	12	673.312	1,823,530.0	12	706.072	392,596.4	x	—	—
5	90	86.721	33,838.2	90	40.058	15,572.8	90	1.024	0.027
6	85	261.271	81,047.6	90	134.872	40,060.8	90	2.386	0.534
7	57	125.013	31,637.3	68	178.709	27,109.5	90	31.721	13.250
8	51	137.056	19,640.5	61	319.428	28,509.4	x	—	—

Note: Results aggregated by class. x denotes classes which have not been attempted to solve.

Table 8: Computational Results for Datasets with Capacity Multiplier 3

	$SV13$ with $_{pre}KP$			$BCM17$		
Class	Solved	Time	Nodes	Solved	Time	Nodes
				R3		
0.1	66	146.309	11,297.4	66	138.031	11,297.4
0.2	55	168.376	33,359.9	56	184.115	33,208.4
0.3	56	331.224	80,175.5	56	254.302	72,872.8
0.4	51	282.786	108,944.7	55	250.254	60,636.3
0.5	52	294.211	93,927.7	60	239.140	64,264.0
0.6	60	265.844	93,698.9	63	146.488	22,384.2
0.7	66	203.753	145,197.1	74	248.247	25,459.2
0.8	75	184.495	47,506.5	75	108.035	5,928.8
0.9	75	64.917	8,900.9	75	48.610	1,818.8
				C3		
0.1	65	81.425	6,257.3	66	72.487	7,734.3
0.2	52	91.552	23,995.8	54	115.856	52,142.7
0.3	49	135.210	54,176.6	51	97.712	71,977.2
0.4	48	408.726	167,408.8	51	308.866	165,593.4
0.5	32	458.638	240,090.5	41	325.131	234,412.1
0.6	40	296.679	203,112.7	41	116.592	53,323.2
0.7	50	280.199	308,182.5	60	224.924	73,459.1
0.8	65	189.454	93,189.9	75	317.326	85,441.1
0.9	75	111.991	21,140.4	75	82.010	10,302.4

Note: Results aggregated by density.

with higher amounts of items available strain the performance of both B&B algorithms. The amount of nodes visited for both methods per datasets are more than substantial, although $BCM17$ has more favourable values in virtually all cases. Looking at how many nodes $SV13$ visited relatively in the case of normal capacity, it's performance on these datasets would likely be much worse than $SV13$ with ${}_{pre}KP$ and $BCM17$. Looking at density values, both low and high density values appear to be mostly solved and relatively quickly at that, as shown in Table 8. Both B&B algorithms seem to struggle most with densities between 0.3 and 0.7, having a great amount of nodes visited and high solving times. Eventhough both methods suffer the same performance-degrading characteristics, $BCM17$ still shows to be the superior algorithm in amount of instances solved, solving times, and amount of nodes visited. Still, for item amounts up to 250, `CPLEX` appears to be faster than the B&B algorithms even when including the clique family building time. When more items are considered, the B&B algorithms are plausibly superior, assuming the clique family building time exceeds the time limit in the majority of cases, especially for higher densities.

Bettinelli et al. (2017) show slightly different results than those mentioned in here. They managed to solve all instances for the datasets considered in this research, with solving times almost never exceeding one second. Most noticeably in Table 7 and 8, this is not the case with our implementation. This is most likely due to Bettinelli et al. (2017) precomputing the values of ${}_{pre}KP$, and possibly some tricks to implement their code more efficiently. While for the datasets with normal capacity the amount of nodes visited coincide relatively well, with an increased capacity the results diverge considerably from theirs. What does correspond is that their proposed branching scheme and upper bound does indeed improve upon the $SV13$ algorithm, decreasing solving time and amount of nodes visited.

6.3 Sparse instances

Table 9 shows the computational results for $BCM17$ and `CPLEX` for model (1a)-(1d) for the random dataset with sparse graph densities. `CPLEX` performs well, with fast to moderate solving times, solving virtually all instances. Instances with density 0.05 are the most difficult to solve for this dataset, with increased solving times when either the amount of available items or the capacity increases. In the case where 1000 items are considered for a capacity of 2000, it cannot solve the instances of density 0.05 anymore. Given that `CPLEX` is rather quick for this dataset, the time limit for $BCM17$ is set to 900 seconds, instead of 1800. With this time limit, $BCM17$ is not able to solve any of the instances for this dataset.

Bettinelli et al. (2017) already noted that for datasets with sparse graph densities, $BCM17$ showed undesirable results, which is confirmed by our results. As we have noted throughout this research, correlated datasets are more difficult to solve for all proposed methods. To this extent, the choice has been made to not further examine the correlated dataset with sparse instances, as the verdict about the performance of $BCM17$ will likely not differ much from that of the random dataset.

Table 9: Results Random Sparse Instances

RANDOM					
		$BCM17$		(1a)-(1d)	
Items/Cap	Density	Solved	Time	Solved	Time
500/1000	0.001	0	—	5	0.98
	0.002	0	—	5	1.04
	0.005	0	—	5	2.10
	0.01	0	—	5	3.07
	0.02	0	—	5	4.10
	0.05	0	—	5	19.68
500/2000	0.001	0	—	5	0.79
	0.002	0	—	5	1.13
	0.005	0	—	5	2.13
	0.01	0	—	5	3.63
	0.02	0	—	5	4.81
	0.05	0	—	5	203.52
1000/1000	0.001	0	—	5	0.04
	0.002	0	—	5	0.03
	0.005	0	—	5	0.06
	0.01	0	—	5	0.10
	0.02	0	—	5	0.21
	0.05	0	—	5	169.38
1000/2000	0.001	0	—	5	0.03
	0.002	0	—	5	0.03
	0.005	0	—	5	0.07
	0.01	0	—	5	0.12
	0.02	0	—	5	1.81
	0.05	0	—	0	—

Note: $BCM17$ performed under reduced time limit of 900 seconds.

6.4 GPU-based Branching Scheme

Some aspects of the workstation had to be changed or needed to be introduced in the context the implementation of ${}_{par}KP$. The workstation used for testing was running on the Ubuntu 20.04 LTS operating system, as required by `TornadoVM`. The used GPU was an NVIDIA GeForce RTX 2060. It is based on NVIDIA's new Turing architecture, equipped with 6GB GDDR6 of memory and 1920 processing cores. The Java code modified for use with `TornadoVM` can be found in Appendix B.5 Listing 10. Tables 10 and 11 show the computational results for $BCM17$ with ${}_{par}KP$ aggregated by class for datasets R1 and R3, respectively. Given the limited computational time available, for the datasets with capacity multiplier three, a subset of instances has been chosen under a reduced time limit of 900 seconds. The subset was derived the same way as has been done for class four earlier. That is, alternately picking nine instances, such that density ratios remain constant and the instances are spread equally over the original dataset. The computational results of the sequential implementation of $BCM17$ under the same conditions has been included in the tables as well for reason of comparison. ${}_{par}KP$ should not show dissimilar behaviour towards different graph densities than ${}_{pre}KP$, because at the base they perform the same operations. This fact makes it that meaningful inferences could not be made for those results. Nevertheless, for those interested, those results can be found in Appendix A, Tables 12 and 13.

Table 10: Computational Results for Dense Datasets with Capacity Multiplier 1

	$BCM17$		$BCM17$ with $_{par}KP$	
Class	Solved	Time	Solved	Time
	R1			
1	90	0.013	90	0.214
2	90	0.087	90	0.589
3	90	0.631	90	2.272
4	45	5.003	45	11.182
5	90	0.011	90	0.120
6	90	0.062	90	0.340
7	90	0.540	90	1.217
8	90	4.246	90	3.465
	C1			
1	90	0.019	90	0.354
2	90	0.110	90	0.979
3	90	0.633	90	3.223
4	45	4.909	45	15.019
5	90	0.029	90	0.662
6	90	0.111	90	1.453
7	90	0.347	90	1.395
8	90	1.676	90	3.586

Note: Results aggregated by class. All instances solved under reduced time limit of 900 seconds.

It can be seen in Table 10 that $BCM17$ with $_{par}KP$ performs worse than its sequential counterpart for datasets R1 and C1. This must be because the capacities for these datasets are relatively low, and the program is not able to efficiently parallelise the code over the GPU. The only exception being class eight for R1, where $BCM17$ with $_{par}KP$ is 0.781 seconds faster on average. This is likely due to its characteristics, having a capacity of 1000, in combination with a relatively high amount of items, namely 501.

Subsequently we'll have a look at the results for the (subsets of the) datasets with multiplier three, as shown in Table 11. The first four classes show $BCM17$ with $_{par}KP$ to be worse than $BCM17$ for the random dataset. Solving times do not significantly improve for all classes, except class four. Although an improvement in solving time is seen for that class, for all four classes $BCM17$ with $_{par}KP$ solves less instances to optimality within the time limit. However, the last four classes are more interesting, showing $BCM17$ with $_{par}KP$ to be an improvement over $BCM17$ in both solving times and amount of instances solved to optimality. Class seven and eight show a decrease in solving times of 30% and 26%, respectively. Class seven is now solved in its entirety (under the reduced time limit), and 68% more instances are solved for class eight.

The correlated dataset shows the same pattern as the random dataset. $BCM17$ with $_{par}KP$ shows no improvement over $BCM17$ for the first four classes, yet for the last four classes the differences in favour of $BCM17$ with $_{par}KP$ are very noticeable. Solving times show a decrease of 20% for class eight, while for classes six and seven this is approximately 60%. However, the difference in amount of instances solved to optimality for this dataset is minimal, although still in favour of $BCM17$ with $_{par}KP$.

Overall it may be noted that for the last four classes the performance of $BCM17$ with $_{par}KP$ compared

to $BCM17$ is noticeably well, which cannot be said for the first four classes. Note that first four classes have a capacity of 450, while the last four have a capacity of 3000. As discussed in Section 4.3, performance was expected to increase when the capacity was sufficiently large, which appears to be the case for a capacity of 3000.

Table 11: Computational Results for Dense Datasets with Capacity Multiplier 3

	$BCM17$		$BCM17$ with ${}_{par}KP$	
Class	Solved	Time	Solved	Time
	R3 Subset			
1	45	8.279	45	28.684
2	44	130.544	42	124.156
3	23	155.825	22	155.611
4	12	349.675	11	247.779
5	45	3.129	45	2.342
6	45	29.727	45	16.967
7	42	164.741	45	114.712
8	19	295.745	32	235.326
	C3 Subset			
1	45	27.498	44	89.174
2	30	181.527	25	109.768
3	16	226.821	15	200.872
4	7	383.755	7	354.872
5	45	35.755	45	28.079
6	44	135.883	45	58.205
7	32	138.789	33	54.638
8	26	106.013	30	84.292

Note: Results aggregated by class. All instances solved under reduced time limit of 900 seconds.

7 Conclusion & Discussion

The purpose of this research was to replicate and validate the results of Bettinelli et al. (2017), and extend upon their model. In replicating their results we have found that for the datasets with dense graph densities, all examined B&B algorithms perform well for low capacities, except for $SV13$ on the random set of instances. The B&B algorithms outperform `CPLEX`, especially if the time-consuming part of building the clique family is taken into account. $BCM17$ outperformed $SV13$ with ${}_{pre}KP$ relatively easy for the examined datasets with higher capacity. $BCM17$ however is not suitable for datasets with sparse graph densities, i.e. lower than 0.1. `CPLEX` is the preferred approach for solving those datasets. Extending upon their framework, it is found that for sufficiently large knapsack capacities, parallel implementation of the branching scheme on a GPU can result in moderate to large speedups in solving times.

Overall, an increased capacity or increased amount of items considered greatly strains performance of the B&B algorithms. However, the performance degradation of an increased capacity can partly be overcome with parallel implementation of ${}_{pre}KP$. Moreover, datasets are substantially harder to solve when the profits are correlated to the item's weight, compared to datasets where they are random. The density of the conflict graph is also an important factor on performance, with midrange densities, those

between 0.2 and 0.7, showing an increased amount of instances not solved to optimality and increased solving times. Graph densities lower than 0.1 cannot be solved at all with the proposed B&B algorithm.

To some extent we have validated the results of Bettinelli et al. (2017). Namely, the solving times found in this research greatly exceed those presented by Bettinelli et al. (2017), and not all instances which they were able to solve could be solved by our implementation. However, this is likely due to our implementation not precomputing the values of ${}_{pre}KP$. Still, it has been shown that the proposed branching scheme ${}_{pre}KP$ and upper bound ${}_{cap}CC$ do indeed increase the performance of algorithm compared to the generic version proposed by Sadykov and Vanderbeck (2013). The $BCM17$ algorithm shows a large reduction in solving times, amount of nodes visited, and is able to solve more instances than comparative algorithms. However, `CPLEX` still seems the best option when the amount of items available is moderate, i.e. does not exceed 250, and graph densities are moderate to low.

The main limitation of this research was time. With how the B&B algorithm was implemented in this paper, a large amount of time would have been needed to apply all algorithms to all available datasets. That time was not available, and thus some choices had to be made on which solvers were to be used and which datasets were to be solved. Secondly, it should have been noted earlier that the time needed to build a clique family was considerable. If those times were recorded, relevant statistics of those times could have been shown for all datasets. Lastly, the absence of knowledge on parallel programming posed to be a hurdle. It was a journey to delve into the subject, but it might have made things more difficult than they had to be. Initially, the intention was to apply parallel computing to the algorithm in a broader context, but this proved to be too ambitious. However, this leaves all the more room for other researchers to improve upon what is proposed.

The parallel implementation of the internal upper bound is an interesting addition, but it is only a small part of the algorithm as a whole. Although a challenge to efficiently implement, B&B algorithms show large potential for parallel computing. I would suggest future research on the topic of KPCGs to look into ways to further parallelise the proposed algorithm.

Acknowledgements

I would like to thank the team behind `TornadoVM` for their quick responses to problems I encountered in setting up and utilising `TornadoVM`. To be more precise, a special thanks to Juan Fumero and Thanos Stratikopoulos, who were my main points of contact with the team, and both have been very helpful.

References

Atamtürk, A., & Savelsbergh, M. (2005). Integer-Programming Software Systems. *Annuals of Operations Research*, *140*, 67–124. https://doi.org/10.1007/s10479-005-3968-2

Balas, E. (1965). An Addative Algorithm for Solving Linear Programs with Zero-One Variables. *Operations Research*, *13*(4), 517–546. https://doi.org/10.1287/opre.13.4.517

Bettinelli, A., Cacchiani, V., & Malaguti, E. (2017). A Branch-and-Bound Algorithm for the Knapsack Problem with Conflict Graph. *INFORMS Journal of Computing*, *29*(3), 457–473. https://doi.org/10.1007/BF00226291

Boyer, V., El Baz, D., & Elkihel, M. (2010). Solution of Multidimensional Knapsack Problems via Cooperation of Dynamic Programming and Branch and Bound. *European Journal of Industrial Engineering*, *4*(4), 434–449. https://doi.org/10.1504/EJIE.2010.035653

Boyer, V., El Baz, D., & Elkihel, M. (2012). Solving knapsack problems on GPU. *Computers & Operations Research*, *39*(1), 42–47. https://doi.org/10.1016/j.cor.2011.03.014

Boyer, V., El Baz, D., & Salazar-Aguilar, M. (2017). GPU Computing Applied to Linear and Mixed-Integer Programming. In *Advances in GPU, Research and Practice* (pp. 247–271). Elsevier. https://doi.org/10.1016/B978-0-12-803738-6.00010-0

Carneiro, T., Muritiba, A. E., Negreiros, M., & Lima de Campos, G. A. (2011). A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU, In *2011 23rd International Symposium on Computer Architecture and High Performance Computing*. https://doi.org/10.1109/SBAC-PAD.2011.20

Chakroun, I., Melab, N., Mezmaz, M., & Tuyttens, D. (2013). Combining Multi-Core and GPU Computing for Solving Combinatorial Optimization Problems. *Journal of Parallel and Distributed Computing*, *73*(12), 1563–1577. https://doi.org/10.1016/j.jpdc.2013.07.023

Clarkson, J., Fumero, J. P., Zakkak, F., Xekalaki, M., Kotselidis, C., & Luján, M. (2018). Exploiting High-Performance Heterogeneous Hardware for Java Programs using Graal, In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. https://doi.org/10.1145/3237009.3237016

Dakin, R. (1965). A Tree-Search Algorithm for Mixed Integer Programming Problems. *The Computer Journal*, *8*(3), 250–255. https://doi.org/10.1093/comjnl/8.3.250

Dantzig, G. (1957). Discrete-Variable Extremum Problems. *Operations Research*, *5*(2), 266–277. https://doi.org/10.1287/opre.5.2.266

Demeulemeester, E., & Herroelen, W. (1992). A Branch-and-Bound Procedure for the Multiple Resource-Constrained Project Scheduling Problem. *Management Science*, *38*(12), 67–124. https://doi.org/10.1287/mnsc.38.12.1803

Dolan, E., & Moré, J. (2002). Benchmarking Optimisation Software with Performance Profiles. *Mathematical Programming*, *91*, 201–213. https://doi.org/10.1007/s101070100263

Efroymson, M., & Ray, T. (1966). A Branch-Bound Algorithm for Plant Location. *Operations Research*, *14*(3), 361–368. https://doi.org/10.1287/opre.14.3.361

Elhedhli, S., Li, L., Gzara, M., & Naoum-Sawaya, J. (2011). A Branch-and-Price Algorithm for the Bin Packing Problem with Conflicts. *INFORMS Journal on Computing*, *23*(3), 404–415. https://doi.org/10.1287/ijoc.1100.0406

Epstein, L., & Levin, A. (2008). On Packing With Conflicts. *SIAM Journal of Optimization*, *19*(3), 1270–1298. https://doi.org/10.1137/060666329

Falkenauer, E. (1996). A Hybrid Grouping Genetic Algorithm for Bin Packing. *Journal of Heuristics*, *2*, 5–30. https://doi.org/10.1007/BF00226291

Gendreau, M., Laporte, G., & Semet, F. (2004). Heuristic and Lower Bounds for the Bin Packing Problem with Conflicts. *Computers & Operations Research*, *31*(3), 347–358. https://doi.org/10.1016/S0305-0548(02)00195-8

Gmys, J., Mezmaz, M., Melab, N., & Tuyttens, D. (2016). A GPU-Based Branch-and-Bound Algorithm using Integer-Vector-Matrix Data Structure. *Parallel Computing*, *59*, 119–139. https://doi.org/10.1016/j.parco.2016.01.008

Gomory, R. (1960). *An Algorithm for the Mixed Integer Problem*. Santa Monica, CA, RAND Corporation. https://www.rand.org/pubs/research_memoranda/RM2597.html

Haffner, S., Monticelli, A., Garcia, A., & Romero, R. (2001). Specialised Branch-and-Bound Algorithm for Transmission Network Expansion Planning. *IEEE Proceedings - Generation, Transmission and Distribution*, *148*(5), 482–488. https://doi.org/10.1049/ip-gtd:20010502

Held, S., Cook, W., & Sewell, E. C. (2012). Maximum-Weight Stable Sets and Safe Lower Bounds for Graph Coloring. *Mathematical Programming Computation*, *4*, 363–381. https://doi.org/10.1007/s12532-012-0042-3

Hifi, M., & Michrafy, M. (2006). A Reactive Local Search-Based Algorithm for the Disjunctively Constrained Knapsack Problem. *Journal of Operations Research*, *57*(6), 718–726. https://doi.org/10.1057/palgrave.jors.2602046

Hifi, M., & Michrafy, M. (2007). Reduction Strategies and Exact Algorithms for the Disjunctively Constrained Knapsack Problem. *Computers & Operations Research*, *34*(9), 2657–2673. https://doi.org/10.1016/j.cor.2005.10.004

Jansen, K. (1999). An Approximation Scheme for Bin Packing with Conflicts. *Journal of Combinatorial Optimization*, *3*, 363–377. https://doi.org/10.1023/A:1009871302966

Jansen, K., & Öhring, S. (1997). Approximation Algorithms for Time Constrained Scheduling. *Information and Computing*, *2*(1), 85–108. https://doi.org/10.1006/inco.1996.2616

Kolesar, P. (1967). A Branch and Bound Algorithm for Knapsack Problem. *Management Science*, *13*(9), 723–735. https://doi.org/10.1287/mnsc.13.9.723

Lalami, M., & El Baz, D. (2012). GPU Implementation of the Branch and Bound Method for Knapsack Problems, In *IEEE 26th International Parallel and Distributed Processing Symposiom Workshops and PhD Forum*, Shanghai. https://doi.org/10.1109/IPDPSW.2012.219

Land, A., & Doig, A. (1960). An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, *28*(3), 497–520. https://doi.org/10.2307/1910129

Laporte, G., & Desroches, S. (1984). Examination Timetabling by Computer. *Computers & Operations Research*, *11*(4), 351–360. https://doi.org/10.1016/0305-0548(84)90036-4

Lawler, E., & Wood, D. (1966). Branch-and-Bound Methods: A Survey. *Operations Research*, *14*(4), 699–719. https://doi.org/10.1287/opre.14.4.699

Little, J., Murty, K., Sweeney, D., & Karel, C. (1963). An Algorithm for the Traveling Salesman Problem. *Operations Research*, *11*(6), 972–989. https://doi.org/10.1287/opre.11.6.972

Martello, S., & Toth, P. (1977). An Upper Bound for the Zero-One Knapsack Problem and a Branch and Bound Algorithm. *European Journal of Operational Research*, *1*(3), 169–175. https://doi.org/10.1016/0377-2217(77)90024-8

Martello, S., & Toth, P. (1988). A New Algorithm for the 0-1 Knapsack Problem. *Management Science*, *34*(5), 633–644. https://doi.org/10.1287/mnsc.34.5.633

Martello, S., & Toth, P. (1997). Upper Bounds and Algorithms for Hard 0-1 Knapsack Problems. *Operations Research*, *45*(5), 768–778. https://doi.org/10.1287/opre.45.5.768

Martello, S., & Toth, P. (2003). An Exact Algorithm for the Two-Constraint 0-1 Knapsack Problem. *Operations Research*, *51*(5), 826–835. https://doi.org/10.1287/opre.51.5.826.16757

Miralles, C., García-Sabater, J., Andrés, C., & Cardós, M. (2005). Branch and Bound Procedures for Solving the Assembly Line Worker Assignment and Balancing Problem: Application to Sheltered Work Centres for Disabled. *Discrete Applied Mathematics*, *156*(3), 352–367. https://doi.org/10.1016/j.dam.2005.12.0125

Muritiba, A., Iori, M., Malaguti, E., & Toth, P. (2010). Algorithms for the Bin Packing Problem with Conflicts. *INFORMS Journal on Computing*, *22*(3), 401–415. https://doi.org/10.1287/ijoc.1090.0355

Nardi, L., Bodin, B., Zia, M., Mawer, J., Nisbet, A., Kelly, P., Davison, A., Luján, M., O'Boyle, M., Riley, G., Topham, N., & Furber, S. (2015). Introducing SLAMBench, a Performance and Accuracy Benchmarking Methodology for SLAM, In *IEEE International Conference on Robotics and Automation*. https://doi.org/10.1109/ICRA.2015.7140009

Navarro, C., Hitschfeld-Kahler, N., & Mateu, L. (2014). A Survey on Parallel Computing and its Applications in Data-Parallel Problems. *Communications in Computational Physics*, *15*(2), 285–329. https://doi.org/10.4208/cicp.110113.010813a

Pupko, T., Pe'er, I., Hasehawa, M., Graur, D., & Friedman, N. (2002). A Branch-and-Bound Algorithm for the Inference of Ancestral Amino-Acid Sequences when the Replacement Rate Varies among Sites: Application to Evolution of Five Gene Families. *Bioinformatics*, *18*(8), 1116–1123. https://doi.org/10.1093/bioinformatics/18.8.1116

Ryan, D., & Foster, B. (1981). An Integer Programming Approach to Scheduling. In A. Wren (Ed.), *Computer Scheduling of Public Transport Urban Passenger and Vehicle and Crew Scheduling* (pp. 269–280). North-Holland, Amsterdam.

Sadykov, R., & Vanderbeck, F. (2013). Bin Packing with Conflicts: A Generic Branch-and-Price Algorithm. *INFORMS Journal of Computing*, *25*(2), 244–255. https://doi.org/10.1287/ijoc.1120.0499

Toth, P. (1980). Dynamic Programming Algorithms for the Zero-One Knapsack Problem. *Computing*, *25*, 29–45. https://doi.org/10.1007/BF02243880

Vanderbeck, F. (2011). Branching in Branch-and-Price: a Generic Scheme. *Mathematical Programming*, *130*, 249–294. https://doi.org/10.1007/s10107-009-0334-1

Yamada, T., Kataoka, S., & Watanabe, K. (2002). Heuristic and Exact Algorithms for the Disjunctively Constrained Knapsack Problem. *Information Processing Society of Japan*, *43*(9), 2864–2870. http://id.nii.ac.jp/1001/00011486/

Zhang, Z., Qin, H., Zhu, W., & Lim, A. (2012). The Single Vehicle Routing Problem with Toll-by-Weight Scheme: A Branch-and-Bound Approach. *European Journal of Operational Research*, *220*(2), 295–304. https://doi.org/10.1016/j.ejor.2012.01.035

Appendix A Computational Results

Table 12: Computational Results for Dense Datasets with Capacity Multiplier 1

	$BCM17$		$BCM17$ with $_{par}KP$	
Density	Solved	Time	Solved	Time
	R1			
0.1	75	1.392	75	2.166
0.2	75	0.738	75	1.494
0.3	75	0.573	75	1.742
0.4	75	0.834	75	1.813
0.5	75	0.835	75	1.601
0.6	75	1.023	75	1.853
0.7	75	1.160	75	1.680
0.8	75	1.403	75	2.284
0.9	75	1.751	75	1.937
	C1			
0.1	75	0.216	75	1.733
0.2	75	0.373	75	2.371
0.3	75	0.500	75	2.010
0.4	75	0.353	75	1.612
0.5	75	0.442	75	1.642
0.6	75	0.946	75	1.950
0.7	75	0.737	75	2.859
0.8	75	1.196	75	4.653
0.9	75	1.692	75	4.164

Note: Results aggregated by density. All instances solved under reduced time limit of 900 seconds.

Table 13: Computational Results for Dense Datasets with Capacity Multiplier 3

	$BCM17$		$BCM17$ with $_{par}KP$	
Density	Solved	Time	Solved	Time
	R3 Subset			
0.1	31	93.289	33	55.141
0.2	25	96.470	28	145.061
0.3	25	156.145	26	154.064
0.4	24	89.864	27	158.425
0.5	28	153.218	26	77.128
0.6	30	81.960	32	133.229
0.7	32	79.368	35	64.364
0.8	40	123.635	40	66.460
0.9	40	57.880	40	21.786
	C3 Subset			
0.1	32	18.989	33	37.861
0.2	27	63.923	25	46.967
0.3	25	61.208	25	75.569
0.4	22	209.401	23	141.515
0.5	19	287.535	16	134.078
0.6	20	94.124	20	84.488
0.7	29	191.732	27	106.736
0.8	31	80.185	35	98.781
0.9	40	97.806	40	65.825

Note: Results aggregated by density. All instances solved under reduced time limit of 900 seconds.

Appendix B Java Code

B.1 Miscellaneous

Listing 1: Main class

```
import java.io.IOException;
import java.util.*;

import ilog.concert.IloException;

public class Main {
   private static InstanceLoader loader;
   private static Scanner in;
   private static long startTime;
   private static ArrayList<Instance> instanceList;
   private static String solver, foldername, ubString;
   private static boolean preKP, CC, capCC, clique, proper;

   /**
    * Main method, calls choice menus and subsequently the solvers.
    * Ultimately writes all relevant results to excel.
    */
   public static void main(String[] args) throws IOException, IloException {
      loader = new InstanceLoader();
      in = new Scanner(System.in);
      instanceList = new ArrayList<Instance>();

      menu();

```

```
25          System.out.println("Finished solving at "+java.time.LocalTime.now());
26
27          ExcelWriter writer = new ExcelWriter(foldername,solver, ubString," all");
28          writer.writeFile(instanceList);
29
30          in.close();
31      }
32
33      /**
34       * Solves the set of instances using CPLEX
35       * @throws IOException
36       */
37      private static void cplexSolve() throws IloException, IOException {
38          cplexMenu();
39          int k = 1;
40          instanceList = loader.loadInstanceList(foldername);
41
42          for (Instance i : instanceList) {
43              startTime = System.currentTimeMillis();
44              CplexModel cplex = new CplexModel(i,clique);
45
46              cplex.solve();
47              i.setSolvingTime((double) (System.currentTimeMillis() - startTime)/1000);
48              i.setObjectiveValue(cplex.getObjectiveValue());
49              i.setNodesVisited(cplex.getNodesVisited());
50              if (cplex.isOptimal()) {i.setSolved();}
51
52              System.out.printf("\nAmount of instances solved: %d\n\n", k++);
53          }
54      }
55
56      /**
57       * Solves the set of instances using a Branch-and-Bound algorithm
58       * @throws IOException
59       */
60      private static void bbSolve() throws IOException {
61          bbMenu();
62          int k = 1;
63          instanceList = loader.loadInstanceList(foldername);
64
65          for (Instance i : instanceList) {
66              BranchBoundModel bbModel = new BranchBoundModel(i,preKP, CC, capCC);
67              startTime = System.currentTimeMillis();
68
69              bbModel.solve();
70
71              i.setSolvingTime((double) (System.currentTimeMillis() - startTime)/1000);
72              i.setObjectiveValue(bbModel.getObjectiveValue());
73              i.setNodesVisited(bbModel.getNodesVisited());
74              if (bbModel.isSolved()) {i.setSolved();}
75              i.setUB(bbModel.getUB(i.getItemList(), 0));
76
77              System.out.printf("\nAmount of instances solved: %d\n\n", k++);
78          }
79
80      }
81
82      /**
83       * Menu to determine choice of dataset and which solver will be used
84       * @throws IOException
85       */
86      private static void menu() throws IloException, IOException {
87          proper = false;
```

```
88
89          System.out.print("Enter name of dataset to be loaded (C1, C3, C10, R1, R3, R10, SR, SC):
                ");
90          foldername = in.nextLine();
91
92          while (proper == false) {
93             System.out.print("Which solver would you like to use?\n"
94                   + "   - 0 for Branch-and-Bound\n"
95                   + "   - 1 for CPLEX\nEnter choice: ");
96             switch(in.nextInt()) {
97                case 0: System.out.printf("\nBranch-and-Bound selected\n\n"); proper = true;
                       bbSolve(); break;
98                case 1: System.out.printf("\nCPLEX selected\n\n"); proper = true; cplexSolve();
                       break;
99                default: System.out.printf("\nInput mismatch\n\n");
100            }
101         }
102      }
103
104      /**
105       * Choice menu to determine which CPLEX model to use
106       */
107      private static void cplexMenu() {
108         proper = false;
109         while (!proper) {
110            System.out.printf("Enter 1 for CPLEX model with clique conflict constraints, 0
                   otherwise: ");
111            switch(in.nextInt()) { case 0: proper = true; clique = false; solver = "CPLEX MIP 1";
                   break;
112                                case 1: proper = true; clique = true; solver = "CPLEX MIP 2"; break;
113                                default: System.out.printf("\nInput mismatch, try again.\n\n");}
114         }
115         ubString = "N.A.";
116      }
117
118      /**
119       * Choice menu for Branch-and-Bound.
120       * Returns integer value according to the choice made used to determine the B&B variant
121       */
122      private static void bbMenu() {
123         proper = false;
124         preKP = false;
125         CC = false;
126         capCC = false;
127
128         while (!proper) {
129            System.out.print("Which variant of the Branch-and-Bound algorithm would you like to
                   use?\n"
130                  + "   - 0 for SV13\n"
131                  + "   - 1 for SV13 with preKP\n"
132                  + "   - 2 for SV13 with capCC\n"
133                  + "   - 3 for SV13 with CC\n"
134                  + "   - 4 for BCM17\n"
135                  + "Enter Choice: ");
136            switch(in.nextInt()) {
137            case 0: System.out.printf("\nSV13 selected\n");
138                  proper = true; solver = "B&B SV13"; ubString = "fracKP"; break;
139            case 1: System.out.printf("\nSV13 with preKP selected\n"); preKP = true;
140                  proper = true; solver = "B&B SV13 preKP"; ubString = "fracKP"; break;
141            case 2: System.out.printf("\nSV13 with capCC selected\n"); capCC = true;
142                  proper = true; solver = "B&B SV13 capCC"; ubString = "capCC"; break;
143            case 3: System.out.printf("\nSV13 with CC selected\n"); CC = true;
144                  proper = true; solver = "B&B SV13 CC"; ubString = "CC"; break;
```

```
145         case 4: System.out.printf("\nBCM17 selected\n"); preKP = true; capCC = true;
146               proper = true; solver = "B&B BCM17"; ubString = "capCC"; break;
147         default: System.out.printf("\nInput mismatch, try again.\n\n");
148         }
149      }
150   }
151
152 }
```

Listing 2: Item class

```
1  import java.util.*;
2
3  /**
4   * Class describing an item in the KPCG
5   * @author Victor Schouten, 449249
6   *
7   */
8  public class Item {
9     private final int index;
10    private final int profit;
11    private final int weight;
12    private final double pwratio;
13    private boolean clique;
14    private int residWeight;
15    private ArrayList<Item> conflicts;
16    private ArrayList<Integer> conflictIndices;
17
18    /**
19     * Constructor
20     */
21    public Item(int index, int profit, int weight) {
22       this.index = index;
23       this.profit = profit;
24       this.weight = weight;
25       this.pwratio = (double) profit / weight;
26       this.conflicts = new ArrayList<Item>();
27       this.conflictIndices = new ArrayList<Integer>();
28       this.clique = false;
29       this.residWeight = profit;
30    }
31
32    /**
33     * Adds the input item to the list of conflicts. Also saves the corresponding index to a
          list.
34     * @param i - conflict item
35     */
36    public void addConflict(Item i) {
37       conflicts.add(i);
38       conflictIndices.add(i.getIndex());
39    }
40
41    /**
42     * Returns the list of conflict items
43     */
44    public List<Item> getConflicts(){
45       return conflicts;
46    }
47
48    public void pwSort() {
49       Collections.sort(conflicts, new Comparator<Item>() {
50            @Override
51            public int compare(Item item1, Item item2) {
```

```
52              if(item1.getRatio() < item2.getRatio())
53                 return 1;
54               else if (item1.getRatio() == item2.getRatio())
55               return 0;
56               else
57               return -1;
58            }});
59     }
60
61     /**
62      * returns the index of this item instance
63      */
64     public int getIndex() {
65        return index;
66     }
67
68     /**
69      * returns the profit associated with this item instance
70      */
71     public int getProfit() {
72        return profit;
73     }
74
75     /**
76      * returns the weight associated with this item instance
77      */
78     public int getWeight() {
79        return weight;
80     }
81
82     /**
83      * returns the profit/weight ratio associated with this item instance
84      */
85     public double getRatio() {
86        return pwratio;
87     }
88
89     /**
90      * called when adding the item to a clique.
91      * The corresponding boolean variable clique is set to true,
92      * such that it can be checked if the instance is in a clique
93      */
94     public void addedToClique() {
95        clique = true;
96     }
97
98     /**
99      * returns true if the item instance is in a clique
100     */
101    public boolean inClique() {
102       return clique;
103    }
104
105    /**
106     * Sets residual weight for this item instance (needed for capCC bound)
107     */
108    public void setResidWeight(int q) {
109       residWeight = q;
110    }
111
112    /**
113     * returns the residual weight associated with this item instance
114     */
```

```
115     public int getResidWeight() {
116        return residWeight;
117     }
118
119     public void reset() {
120        residWeight = profit;
121        clique = false;
122     }
123
124     @Override
125     public boolean equals(Object o) {
126        if (o == this)
127           return true;
128        if (!(o instanceof Item))
129           return false;
130        Item i = (Item) o;
131        return this.index == i.index;
132     }
133
134     @Override
135     public String toString() {
136        return "Item number " +index+ ", profit = " +profit+ ", weight = " + weight + ",
               profit/weigh ratio = " +pwratio+
137             ", conflicts = " + conflictIndices;
138     }
139
140
141
142 }
```

Listing 3: Instance class

```
1  import java.util.*;
2
3  /**
4   * Class describing a problem instance for the KPCG
5   * @author Victor Schouten, 449249
6   *
7   */
8  public class Instance {
9     private final int capacity;
10    private final int n;
11    private final String filename;
12    private final String dataset;
13    private int dataclass;
14    private double density;
15    private int nodesVisited;
16    private double objectiveValue;
17    private double solvingTime;
18    private int solved;
19    private LinkedList<Item> itemList;
20    private LinkedList<Edge> edgeList;
21    private double finalUB;
22
23    /**
24     * Constructor
25     */
26    public Instance(int c, int n, String fileformat, String dataset) {
27       this.itemList = new LinkedList<Item>();
28       this.n = n;
29       this.capacity = c;
30       this.filename = fileformat;
31       this.dataset = dataset;
```

```
32        this.solved = 0;
33        this.nodesVisited = -1;
34        this.objectiveValue = -1;
35        this.solvingTime = -1;
36        this.finalUB = -1;
37        this.edgeList = new LinkedList<Edge>();
38        extractFromFileName();
39     }
40
41     /**
42      * Adds a new item to the list of items corresponding to this data instance
43      */
44     public void addItem(Item item) {
45        itemList.add(item);
46     }
47
48     /**
49      * Saves conflicting items, index are based on unsorted list of items
50      * @param index1 - first index of the set of conflicting items
51      * @param index2 - second index of the set of conflicting items
52      */
53     public void addConflicts(int index1, int index2) {
54        Item temp1 = itemList.get(index1);
55        Item temp2 = itemList.get(index2);
56        temp1.addConflict(temp2);
57        temp2.addConflict(temp1);
58        edgeList.add(new Edge(temp1, temp2));
59     }
60
61     public LinkedList<Edge> getEdgeList(){
62        return edgeList;
63     }
64
65     public Edge getEdge(Item item1, Item item2) {
66        for (Edge e : edgeList) {
67           if (item1.equals(e.getFirstItem()) || item2.equals(e.getFirstItem())) {
68              if (item2.equals(e.getSecondItem()) || item1.equals(e.getSecondItem()))
69                 return e;
70           }
71        }
72        return null;
73     }
74
75     /**
76      * Sorts the list of items in nonincreasing profit/weight ratio
77      */
78     public void pwSort() {
79        Collections.sort(itemList, new Comparator<Item>() {
80              @Override
81              public int compare(Item item1, Item item2) {
82                if(item1.getRatio() < item2.getRatio())
83                   return 1;
84                 else if (item1.getRatio() == item2.getRatio())
85                 return 0;
86                 else
87                 return -1;
88              }});
89     }
90
91     /**
92      * Extracts the dataclass and density of the data instance from the filename
93      */
94     private void extractFromFileName() {
```

```
95          String[] values = filename.split("[_\\-]");
96          if (isSparse()) {
97             String dString = values[3].substring(1, values[3].length());
98             density = Double.parseDouble(dString);
99             dataclass = 0;
100         } else {
101            dataclass = Integer.parseInt(values[1]);
102            density = Double.parseDouble(values[4]);
103         }
104      }
105
106      /**
107       * Returns true if the data instance belongs to the set with sparse densities
108       */
109      private boolean isSparse() {
110         return dataset.equals("SR") || dataset.equals("SC");
111      }
112
113      /**
114       * Returns item list associated with this data instance
115       */
116      public LinkedList<Item> getItemList(){
117         return itemList;
118      }
119
120      /**
121       * Returns item from the list of items based on its index
122       */
123      public Item getItemWithIndex(int index) {
124         for (Item i : itemList) {
125            if (index == i.getIndex())
126               return i;
127         }
128         return null;
129      }
130
131      /**
132       * returns total amount of items
133       */
134      public int getItemCount() {
135         return n;
136      }
137
138      /**
139       * Returns knapsack capacity of this data instance
140       */
141      public int getCapacity() {
142         return capacity;
143      }
144
145      /**
146       * Returns the dataclass of this data instance (0 for sparse sets)
147       */
148      public int getDataclass() {
149         return dataclass;
150      }
151
152      /**
153       * Returns conflict density of this data instance
154       */
155      public double getDensity() {
156         return density;
157      }
```

```
158
159     /**
160      * Sets the amount of nodes visited during the solving process
161      */
162     public void setNodesVisited(int visited) {
163        nodesVisited = visited;
164     }
165
166     /**
167      * Returns the amount of nodes visited during the solving process
168      */
169     public int getNodesVisited() {
170        return nodesVisited;
171     }
172
173     /**
174      * Sets the objective value obtained with the solving process
175      */
176     public void setObjectiveValue(double d) {
177        objectiveValue = d;
178     }
179
180     /**
181      * Returns the objective value obtained with the solving process
182      */
183     public double getObjectiveValue() {
184        return objectiveValue;
185     }
186
187     /**
188      * Sets the time needed by the solving process
189      */
190     public void setSolvingTime(double time) {
191        solvingTime = time;
192     }
193
194     /**
195      * Returns the time needed by the solving process
196      */
197     public double getSolvingTime() {
198        return solvingTime;
199     }
200
201     /**
202      * Called when the instance is solved to optimality
203      */
204     public void setSolved() {
205        solved = 1;
206     }
207
208     /**
209      * Returns 1 if the instance has been solved to optimality
210      */
211     public int getSolved() {
212        return solved;
213     }
214
215     public void setUB(double UB) {
216        finalUB = UB;
217     }
218
219     public double getUB() {
220        return finalUB;
```

```
221     }
222
223     public void reset() {
224        for (Item i : itemList)
225           i.reset();
226     }
227
228     @Override
229     public String toString() {
230        if (isSparse()) {
231           return "Instance information: capacity = " +capacity+ ", n = " +n+ ", dataset = "
                  +dataset+
232                ", density = " +density;
233        } else {
234           return "Instance information: capacity = " +capacity+ ", n = " +n+ ", dataset = "
                  +dataset+
235                ", class = " +dataclass+ ", density = " +density;
236        }
237     }
238
239
240 }
```

Listing 4: Edge class

```
1
2  public class Edge {
3     private Item item1;
4     private Item item2;
5     private boolean covered;
6
7     public Edge(Item item1, Item item2) {
8        this.item1 = item1;
9        this.item2 = item2;
10       this.covered = false;
11    }
12
13    public Item getFirstItem() {
14       return item1;
15    }
16
17    public Item getSecondItem() {
18       return item2;
19    }
20
21    public void setCovered() {
22       covered = true;
23    }
24
25    public boolean isCovered() {
26       return covered;
27    }
28 }
```

Listing 5: InstanceLoader class

```
1  import java.io.*;
2  import java.util.*;
3
4  /**
5   * Class describing a loader of instances for the KPCG
6   * @author Victor Schouten, 449249
```

```
 *
 */
public class InstanceLoader {
   private ArrayList<Instance> instanceList;

   /**
    * Constructor
    */
   public InstanceLoader() {
      instanceList = new ArrayList<Instance>();
   }

   /**
    * Loads instances from a folder given the foldername and returns Instance list.
    * Folders are assumed to be in the workspace in location data/foldername
    */
   public ArrayList<Instance> loadInstanceList(String foldername) {
      String dataset = foldername;
      String folderlocation = "data/" +foldername;
      load(folderlocation, dataset);
      return instanceList;
   }

   /**
    * Loads all instances from a folder and them to instanceList
    */
   private void load(String folderlocation, String dataset) {
      System.out.printf("\nLoading dataset...\n");
      long startTime = System.currentTimeMillis();
      File folder = new File(folderlocation);
      File[] listOfFiles = folder.listFiles();
      //int k = 1;

      for (File file : listOfFiles) {
         try{
               Scanner in = new Scanner(file);
               int capacity = 0;
               int n = 0;
               Instance temp = null;
               boolean conflicts = false;

               while(in.hasNext()){
                  // Read line
                   String line= in.nextLine();
                   String[] values = line.split("\\s+");

                   if (values[0].equals("param")) {
                     if (values[1].equals("n")) {
                        String nString = values[3].substring(0, values[3].length()-1);
                        n = Integer.parseInt(nString);
                     } else if (values[1].equals("c")) {
                        if (values[3].charAt(values[3].length()-1) == ';') {
                           String cString = values[3].substring(0, values[3].length()-1);
                           capacity = Integer.parseInt(cString);
                        } else {
                           capacity = Integer.parseInt(values[3]);
                        }
                     } else {
                        temp = new Instance(capacity, n, file.getName(), dataset);
                     }
                     continue;
                   }
```

```
                if (values[0].equals(";") || line.isEmpty()) {
                  continue;
                } else if (values[0].equals("set")) {
                  conflicts = true;
                  continue;
                }

                if (!conflicts) {
                  int index = Integer.parseInt(values[1]);
                    int profit = Integer.parseInt(values[2]);
                    int weight = Integer.parseInt(values[3]);
                    temp.addItem(new Item(index,profit,weight));
                } else {
                  int index1 = Integer.parseInt(values[1]);
                    int index2 = Integer.parseInt(values[2]);
                    temp.addConflicts(index1, index2);
                }

            }

            in.close();

            temp.pwSort();
          instanceList.add(temp);

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
       //if (k++ == 3) {break;}
       //break;
    }
    System.out.printf("Dataset loaded in %.2f seconds\n\n", (double)
        (System.currentTimeMillis() - startTime)/1000);
  }
}
```

B.2 Clique Heuristic

Listing 6: Clique class

```
import java.util.*;

/**
 * Class describing a clique set for Items
 * @author Victor Schouten, 449249
 *
 */
public class Clique {
  private ArrayList<Item> itemList;
  private double weight;
  /**
   * Constructor
   */
  public Clique() {
    this.itemList = new ArrayList<Item>();
    this.weight = 0.0;
  }

  /**
   * Adds an item to this clique instance
   */
  public void addItem(Item i) {
```

```
23        itemList.add(i);
24        i.addedToClique();
25     }
26
27     /**
28      * Returns list of items in the clique
29      */
30     public ArrayList<Item> getCliqueItems(){
31        return itemList;
32     }
33
34     public void setWeight(double w) {
35        weight = w;
36     }
37
38     public double getWeight() {
39        return weight;
40     }
41
42
43  }
```

Listing 7: CliqueHeuristic class

```
1  import java.util.*;
2
3  /**
4   * Class describing a heuristic approach in finding a maximal clique and a family of cliques
5   * @author Victor Schouten, 449249
6   *
7   */
8  public class CliqueHeuristic {
9
10    /**
11     * Constructor
12     */
13    public CliqueHeuristic() {}
14
15    /**
16     * Finds a maximal clique for item i given a list of items itemList
17     *  and returns the found clique
18     */
19    public Clique findMaxClique(Edge e, Instance i) {
20       Clique clique = new Clique();
21       Item item1 = e.getFirstItem();
22       Item item2 = e.getSecondItem();
23       clique.addItem(item1);
24       clique.addItem(item2);
25
26       for (Item j : item1.getConflicts()) {
27          if (isAdjacent(clique.getCliqueItems(),j) && !clique.getCliqueItems().contains(j)) {
28             for (Item k : clique.getCliqueItems())
29                i.getEdge(k,j).setCovered();
30             clique.addItem(j);
31          }
32       }
33       return clique;
34    }
35
36    public Clique findCliqueForCover(Item i, List<Item> itemList) {
37       Clique clique = new Clique();
38       clique.addItem(i);
39
```

```
40        for (Item j : itemList) {
41           if (isAdjacent(clique.getCliqueItems(),j) && j.getResidWeight() > 0)
42              clique.addItem(j);
43        }
44        return clique;
45     }
46
47     /**
48      * Finds a family of cliques for instance i and returns the list of cliques
49      */
50     public ArrayList<Clique> findCliqueFam(Instance i){
51        ArrayList<Clique> cliqueFam = new ArrayList<Clique>();
52        long startTime = System.currentTimeMillis();
53        for (Edge e : i.getEdgeList()) {
54           if (System.currentTimeMillis()-startTime > 1800000)
55              return null;
56           if (!e.isCovered()) {
57              Clique clique = findMaxClique(e, i);
58              cliqueFam.add(clique);
59           }
60        }
61        return cliqueFam;
62     }
63
64     /**
65      * Returns true when item i is adjacent to all items in the list cliqueItems
66      */
67     private boolean isAdjacent(List<Item> cliqueItems, Item i) {
68        for (Item j : cliqueItems)
69           if (!j.getConflicts().contains(i)) {return false;}
70        return true;
71     }
72  }
```

B.3 CPLEX

Listing 8: CplexModel class

```
1  import java.util.*;
2
3  import ilog.concert.*;
4  import ilog.cplex.*;
5
6  /**
7   * Class describing a cplex solver for the KPCG in two formulations
8   * @author Victor Schouten, 449249
9   *
10  */
11 public class CplexModel {
12    private IloCplex cplex;
13    private Map<Item, IloNumVar> varMapDecision;
14    private Instance instance;
15    private CliqueHeuristic heuristic;
16
17    /**
18     * Constructor
19     */
20    public CplexModel(Instance instance, boolean clique) throws IloException {
21       this.cplex = new IloCplex();
22       this.varMapDecision = new HashMap<>();
23       this.instance = instance;
24       this.heuristic = new CliqueHeuristic();
```

```
25
26          addDecisionVariables();
27          addCapacityConstraints();
28
29          if (clique) {
30             addCliqueConflictConstraints();
31          } else {
32             addConflictConstraints();
33          }
34
35          addObjective();
36       }
37
38       /**
39        * Adds decision variable x_i
40        */
41       private void addDecisionVariables() throws IloException {
42          for (Item i : instance.getItemList()) {
43             IloNumVar var = cplex.boolVar();
44             varMapDecision.put(i,var);
45          }
46       }
47
48       /**
49        * Adds capacity constraints
50        * Sum_{i=1}^n weight_i*x_i <= capcity, for all C in Xi
51        */
52       private void addCapacityConstraints() throws IloException {
53          IloNumExpr lhs = cplex.constant(0);
54          IloNumExpr rhs = cplex.constant(instance.getCapacity());
55
56          for (Item i : instance.getItemList()) {
57             IloNumVar var = varMapDecision.get(i);
58             IloNumExpr prod = cplex.prod(i.getWeight(),var);
59             lhs = cplex.sum(lhs,prod);
60          }
61
62          cplex.addLe(lhs, rhs).setName("Capacity Constraint");
63       }
64
65       /**
66        * Adds conflict constraints using edges
67        * x_i + x_j<= 1, for all (i,j) in E
68        */
69       private void addConflictConstraints() throws IloException {
70          IloNumExpr rhs = cplex.constant(1);
71
72          for (Item i : instance.getItemList()) {
73             IloNumVar var1 = varMapDecision.get(i);
74             for (Item k : i.getConflicts()) {
75                IloNumVar var2 = varMapDecision.get(k);
76                IloNumExpr lhs = cplex.sum(var1,var2);
77                cplex.addLe(lhs, rhs);
78             }
79          }
80       }
81
82       /**
83        * Adds conflict constraints based on clique notation.
84        * Sum_{i in C} x_i <= 1, for all C in Xi
85        */
86       private void addCliqueConflictConstraints() throws IloException {
87          IloNumExpr rhs = cplex.constant(1);
```

```
88          ArrayList<Clique> cliqueFam = heuristic.findCliqueFam(instance);
89
90          for (Clique c : cliqueFam) {
91             IloNumExpr lhs = cplex.constant(0);
92             for (Item i : c.getCliqueItems()) {
93                IloNumVar var = varMapDecision.get(i);
94                lhs = cplex.sum(lhs,var);
95             }
96             cplex.addLe(lhs, rhs);
97          }
98       }
99
100      /**
101       * Adds objective function
102       *    Sum_{i=1}^n profit_i*x_i
103       */
104      private void addObjective() throws IloException {
105         IloNumExpr obj = cplex.constant(0);
106
107         for (Item i : instance.getItemList()) {
108            IloNumVar var = varMapDecision.get(i);
109            IloNumExpr prod = cplex.prod(i.getProfit(),var);
110            obj = cplex.sum(obj,prod);
111         }
112
113         cplex.addMaximize(obj);
114      }
115
116      /**
117       * Solves the MIP, imposing a time limit of 1800 seconds.
118       * Node list file strategy 3 is applied in order to avoid running out of memory.
119       */
120      public boolean solve() throws IloException {
121         cplex.setParam(IloCplex.DoubleParam.TiLim, 1800);
122         cplex.setParam(IloCplex.Param.MIP.Strategy.File, 3);
123         return cplex.solve();
124      }
125
126      /**
127       * returns the objective value obtained through the solving process
128       */
129      public double getObjectiveValue() throws IloException {
130         return cplex.getObjValue();
131      }
132
133      /**
134       * returns the amount of nodes visited in the solving process
135       */
136      public int getNodesVisited() throws IloException {
137         return cplex.getNnodes();
138      }
139
140      /**
141       * Returns true when the solver has solved the problem to optimality
142       */
143      public boolean isOptimal() throws IloException {
144         return cplex.getStatus().toString().equals("Optimal");
145      }
146   }
```

B.4 Branch-and-Bound

Listing 9: BranchBoundModel class

```
import java.util.*;

public class BranchBoundModel {
   private Instance instance;
   private final int capacity;
   private int nodesVisited;
   private boolean solved, preKP, CC, capCC;
   private double globalLB;
   private long startTime;
   private LinkedList<Item> solution;


   public BranchBoundModel(Instance instance, boolean preKP, boolean CC, boolean capCC) {
      this.instance = instance;
      this.preKP = preKP;
      this.CC = CC;
      this.capCC = capCC;
      this.capacity = instance.getCapacity();
      this.solved = false;
      this.nodesVisited = 0;
      this.globalLB = 0.0;
      this.solution = new LinkedList<Item>();
   }

   public void solve() {
      LinkedList<Item> solSet = new LinkedList<Item>();

      System.out.println("Commencing solving procedure at " +java.time.LocalTime.now());
      startTime = System.currentTimeMillis();
      branchBound(solSet,instance.getItemList(),0,globalLB);

      if (System.currentTimeMillis()-startTime < 1800000)
         solved = true;

      System.out.printf("Solving procedure finished.\n");
   }

   @SuppressWarnings("unchecked")
   private void branchBound(LinkedList<Item> solSet, LinkedList<Item> freeSet, int currWeight,
       double currProfit) {
      if (System.currentTimeMillis()-startTime > 900000)
         return;

      LinkedList<Item> newSolSet = new LinkedList<Item>();
      LinkedList<Item> newFreeSet = new LinkedList<Item>();
      int newCurrWeight;
      double newCurrProfit;

      nodesVisited++;

      // update globalLB if current solution exceeds it
      if (globalLB < currProfit) {
         globalLB = currProfit;
         solution = solSet;
      }

      // Prune if UB is lower than best found solution globalLB
      if (currProfit + getUB(freeSet, currWeight) <= globalLB)
         return;

      int count = 0;
      for (Item i : freeSet) {
```

```
62          double internalUB = currProfit;
63
64          if (preKP) {
65             internalUB += internalBCM17(freeSet, currWeight, count);
66             count++;
67          } else
68             internalUB += internalSV13(i, currWeight);
69
70          if (internalUB > globalLB) {
71             newSolSet = (LinkedList<Item>) solSet.clone();
72             newFreeSet = (LinkedList<Item>) freeSet.clone();
73             newFreeSet.remove(i);
74             if (capacity - currWeight >= i.getWeight()) {
75                newCurrProfit = currProfit + i.getProfit();
76                newCurrWeight = currWeight + i.getWeight();
77                newSolSet.add(i);
78                for (Item j : i.getConflicts())
79                   newFreeSet.remove(j);
80                branchBound(newSolSet,newFreeSet,newCurrWeight,newCurrProfit);
81             }
82          } else
83             break;
84       }
85    }
86
87    public double fracKP(LinkedList<Item> freeSet, int currWeight) {
88       double UB = 0.0;
89       int k = 0;
90
91       while (currWeight < capacity && k < freeSet.size()){
92             Item i = freeSet.get(k);
93             if (currWeight + i.getWeight() <= capacity) {
94                UB += i.getProfit();
95                currWeight += i.getWeight();
96             } else {
97                UB += (capacity - currWeight)*i.getRatio();
98                currWeight = capacity;
99             }
100            k++;
101      }
102      return UB;
103   }
104
105   public double capCC(LinkedList<Item> freeSet) {
106      CliqueHeuristic heuristic = new CliqueHeuristic();
107      LinkedList<Clique> cliqueFam = new LinkedList<Clique>();
108      double totalLoad = 0.0;
109      double UB = 0.0;
110
111      while (totalLoad < capacity) {
112         double min1 = Double.MAX_VALUE;
113         Item minItem = null;
114         int negCount = 0;
115
116         for (Item i: freeSet) {
117            if (1/i.getRatio() < min1 && i.getResidWeight() > 0) {
118               min1 = 1/i.getRatio();
119               minItem = i;
120            } else if (i.getResidWeight() <= 0)
121               negCount++;
122         }
123
124         if (negCount==freeSet.size() || minItem == null)
```

```
125             break;
126
127          Clique clique = heuristic.findCliqueForCover(minItem, freeSet);
128          int min2 = Integer.MAX_VALUE;
129
130          for (Item i : clique.getCliqueItems()) {
131             if (i.getResidWeight() < min2 && i.getResidWeight() > 0)
132                min2 = i.getResidWeight();
133          }
134
135          double load = Math.min(min1*min2,capacity - totalLoad);
136          clique.setWeight(load/min1);
137          cliqueFam.add(clique);
138          totalLoad += load;
139
140          for (Item i : clique.getCliqueItems())
141             i.setResidWeight(i.getResidWeight()-min2);
142
143          UB += clique.getWeight();
144       }
145       instance.reset();
146       return UB;
147    }
148
149    public double CC(LinkedList<Item> freeSet) {
150       CliqueHeuristic heuristic = new CliqueHeuristic();
151       LinkedList<Clique> cliqueFam = new LinkedList<Clique>();
152       double UB = 0.0;
153
154       while (true) {
155          int min = Integer.MAX_VALUE;
156          Item minItem = null;
157          int negCount = 0;
158
159          for (Item i: freeSet) {
160             if (i.getResidWeight() < min && i.getResidWeight() > 0) {
161                min = i.getResidWeight();
162                minItem = i;
163             } else if (i.getResidWeight() <= 0)
164                negCount++;
165          }
166
167          if (negCount==freeSet.size() || minItem == null)
168             break;
169
170          Clique clique = heuristic.findCliqueForCover(minItem, freeSet);
171          clique.setWeight(min);
172          cliqueFam.add(clique);
173
174          for (Item i : clique.getCliqueItems())
175             i.setResidWeight(i.getResidWeight()-min);
176
177          UB += clique.getWeight();
178       }
179       instance.reset();
180       return UB;
181    }
182
183    private int internalBCM17(LinkedList<Item> freeSet, int currWeight, int count) {
184       int n = freeSet.size()-count;
185       int c = capacity-currWeight;
186       int matrixDP[][] = new int[n+1][c+1];
187
```

```
188       for (int i = 0; i <= n; i++) {
189          for (int j = 0; j <= c; j++) {
190             if (i == 0 || j == 0)
191                matrixDP[i][j] = 0;
192             else if (freeSet.get(i+count-1).getWeight() <= j)
193                matrixDP[i][j] = Math.max(freeSet.get(i+count-1).getProfit() +
194                      matrixDP[i-1][j-freeSet.get(i+count-1).getWeight()], matrixDP[i-1][j]);
195             else
196                matrixDP[i][j] = matrixDP[i-1][j];
197          }
198       }
199       return matrixDP[n][c];
200    }
201
202    private double internalSV13(Item i, int currWeight) {
203       return (capacity-currWeight)*i.getRatio();
204    }
205
206    public double getObjectiveValue() {
207       return globalLB;
208    }
209
210    public boolean isSolved() {
211       return solved;
212    }
213
214    public int getNodesVisited() {
215       return nodesVisited;
216    }
217
218    public LinkedList<Item> getSolutionSet(){
219       return solution;
220    }
221
222    public double getUB(LinkedList<Item> freeSet, int currWeight) {
223       double UB = 0.0;
224
225       if (capCC)
226          UB += capCC(freeSet);
227       else if (CC)
228          UB += CC(freeSet);
229       else
230          UB += fracKP(freeSet, currWeight);
231
232       return UB;
233    }
234
235 }
```

B.5 GPU-Based Branching Scheme

Listing 10: ParallelBBModel class

```
1  package parallel;
2
3  import uk.ac.manchester.tornado.api.TaskSchedule;
4  import uk.ac.manchester.tornado.api.annotations.Parallel;
5  import uk.ac.manchester.tornado.api.Policy;
6  import uk.ac.manchester.tornado.api.collections.types.Matrix2DDouble;
7  import uk.ac.manchester.tornado.api.collections.math.TornadoMath;
8  import java.util.*;
9
```

```
10 public class ParallelBBModel {
11    private Instance instance;
12    private final int capacity;
13    private int nodesVisited;
14    private boolean solved, preKP, CC, capCC;
15    private double globalLB;
16    private long startTime;
17    private LinkedList<Item> solution;
18    private int v_par[];
19    private TaskSchedule ts1;
20    private TaskSchedule ts2;
21    private int weights[];
22    private double profits[];
23    private int globalSize;
24    private double inputDP[];
25    private double outputDP[];
26
27    public ParallelBBModel(Instance instance, boolean preKP, boolean CC, boolean capCC) {
28       this.instance = instance;
29       this.preKP = preKP;
30       this.CC = CC;
31       this.capCC = capCC;
32       this.capacity = instance.getCapacity();
33       this.solved = false;
34       this.nodesVisited = 0;
35       this.globalLB = 0.0;
36       this.globalSize = instance.getItemList().size();
37       this.solution = new LinkedList<Item>();
38       this.v_par = new int[2];
39       this.inputDP = new double[capacity+1];
40       this.outputDP = new double[capacity+1];
41       this.weights = new int[globalSize];
42       this.profits = new double[globalSize];
43
44       if (preKP){
45          // Create TaskSchedules
46          ts1 = new TaskSchedule("Inner-Loop-Internal-UB-BCM17-even")
47             .streamIn(inputDP,outputDP,weights,profits,v_par) //
48             .task("taskBCM17",ParallelBBModel::parallelInternalBCM17,inputDP,outputDP,weights,profits,v_par)
                   //
49             .streamOut(outputDP); //
50          ts2 = new TaskSchedule("Inner-Loop-Internal-UB-BCM17-odd")
51             .streamIn(inputDP,outputDP,weights,profits,v_par) //
52             .task("taskBCM17",ParallelBBModel::parallelInternalBCM17,outputDP,inputDP,weights,profits,v_par)
                   //
53             .streamOut(inputDP); //
54       }
55    }
56
57    public void solve() {
58       LinkedList<Item> solSet = new LinkedList<Item>();
59
60       System.out.println("Commencing solving procedure at " +java.time.LocalTime.now());
61       startTime = System.currentTimeMillis();
62       branchBound(solSet,instance.getItemList(),0,globalLB);
63
64       if (System.currentTimeMillis()-startTime < 900000)
65          solved = true;
66
67       System.out.printf("Solving procedure finished.\n");
68    }
69
70    @SuppressWarnings("unchecked")
```

```
71    private void branchBound(LinkedList<Item> solSet, LinkedList<Item> freeSet, int currWeight,
          double currProfit) {
72       if (System.currentTimeMillis()-startTime > 900000)
73          return;
74
75       LinkedList<Item> newSolSet = new LinkedList<Item>();
76       LinkedList<Item> newFreeSet = new LinkedList<Item>();
77       int newCurrWeight;
78       double newCurrProfit;
79
80       nodesVisited++;
81
82       // update globalLB if current solution exceeds it
83       if (globalLB < currProfit) {
84          globalLB = currProfit;
85          solution = solSet;
86       }
87
88       // Prune if UB is lower than best found solution globalLB
89       if (currProfit + getUB(freeSet, currWeight) <= globalLB)
90          return;
91
92       int currSize = freeSet.size();
93       int count = 0;
94       for (int k = 0; k < currSize;k++) {
95          Item i = freeSet.get(k);
96          double internalUB = currProfit;
97
98          if (preKP) {
99             for (int j = 0; j <= capacity-currWeight; j++){
100               if (k%2==0)
101                  inputDP[j]=0;
102               else
103                  outputDP[j]=0;
104            }
105            internalUB += internalBCM17(freeSet,capacity-currWeight,currSize,count++);
106         } else
107            internalUB += internalSV13(i, currWeight);
108
109         if (internalUB > globalLB) {
110            newSolSet = (LinkedList<Item>) solSet.clone();
111            newFreeSet = (LinkedList<Item>) freeSet.clone();
112            newFreeSet.remove(i);
113            if (capacity - currWeight >= i.getWeight()) {
114               newCurrProfit = currProfit + i.getProfit();
115               newCurrWeight = currWeight + i.getWeight();
116               newSolSet.add(i);
117               for (Item j : i.getConflicts())
118                  newFreeSet.remove(j);
119               branchBound(newSolSet,newFreeSet,newCurrWeight,newCurrProfit);
120            }
121         } else
122            break;
123
124      }
125   }
126
127   public double internalBCM17(LinkedList<Item> freeSet,int cap, int size, int count) {
128      v_par[1] = cap+1;
129
130      for (int k = count; k < size; k++) {
131         v_par[0] = k;
132
```

```
133         Item i = freeSet.get(k);
134         profits[k] = i.getProfit();
135         weights[k] = i.getWeight();
136 
137         if (k%2==0)
138            ts1.execute();
139         else
140            ts2.execute();
141      }
142 
143      if (size-1%2==0)
144         return outputDP[cap];
145      else
146         return inputDP[cap];
147   }
148 
149   public static void parallelInternalBCM17(double inputDP[], double outputDP[], int weights[],
       double profits[], int v_par[]){
150      final int k = v_par[0];
151      final int cap = v_par[1];
152 
153      for (@Parallel int c = 0; c < inputDP.length; c++){
154         if (c < cap){
155            if (weights[k] <= c) {
156               if (inputDP[c] < inputDP[c-weights[k]] + profits[k])
157                  outputDP[c] = inputDP[c-weights[k]] + profits[k];
158               else
159                  outputDP[c] = inputDP[c];
160            } else
161               outputDP[c] = inputDP[c];
162         }
163      }
164   }
165 
166   public double fracKP(LinkedList<Item> freeSet, int currWeight) {
167      double UB = 0.0;
168      int k = 0;
169 
170      while (currWeight < capacity && k < freeSet.size()){
171           Item i = freeSet.get(k);
172           if (currWeight + i.getWeight() <= capacity) {
173              UB += i.getProfit();
174              currWeight += i.getWeight();
175           } else {
176              UB += (capacity - currWeight)*i.getRatio();
177              currWeight = capacity;
178           }
179           k++;
180      }
181      return UB;
182   }
183 
184   public double capCC(LinkedList<Item> freeSet) {
185      CliqueHeuristic heuristic = new CliqueHeuristic();
186      LinkedList<Clique> cliqueFam = new LinkedList<Clique>();
187      double totalLoad = 0.0;
188      double UB = 0.0;
189 
190      while (totalLoad < capacity) {
191         double min1 = Double.MAX_VALUE;
192         Item minItem = null;
193         int negCount = 0;
194 
```

```
195         for (Item i: freeSet) {
196            if (1/i.getRatio() < min1 && i.getResidWeight() > 0) {
197               min1 = 1/i.getRatio();
198               minItem = i;
199            } else if (i.getResidWeight() <= 0)
200               negCount++;
201         }
202
203         if (negCount==freeSet.size() || minItem == null)
204            break;
205
206         Clique clique = heuristic.findCliqueForCover(minItem, freeSet);
207         int min2 = Integer.MAX_VALUE;
208
209         for (Item i : clique.getCliqueItems()) {
210            if (i.getResidWeight() < min2 && i.getResidWeight() > 0)
211               min2 = i.getResidWeight();
212         }
213
214         double load = Math.min(min1*min2,capacity - totalLoad);
215         clique.setWeight(load/min1);
216         cliqueFam.add(clique);
217         totalLoad += load;
218
219         for (Item i : clique.getCliqueItems())
220            i.setResidWeight(i.getResidWeight()-min2);
221
222         UB += clique.getWeight();
223      }
224      instance.reset();
225      return UB;
226   }
227
228   public double CC(LinkedList<Item> freeSet) {
229      CliqueHeuristic heuristic = new CliqueHeuristic();
230      LinkedList<Clique> cliqueFam = new LinkedList<Clique>();
231      double UB = 0.0;
232
233      while (true) {
234         int min = Integer.MAX_VALUE;
235         Item minItem = null;
236         int negCount = 0;
237
238         for (Item i: freeSet) {
239            if (i.getResidWeight() < min && i.getResidWeight() > 0) {
240               min = i.getResidWeight();
241               minItem = i;
242            } else if (i.getResidWeight() <= 0)
243               negCount++;
244         }
245
246         if (negCount==freeSet.size() || minItem == null)
247            break;
248
249         Clique clique = heuristic.findCliqueForCover(minItem, freeSet);
250         clique.setWeight(min);
251         cliqueFam.add(clique);
252
253         for (Item i : clique.getCliqueItems())
254            i.setResidWeight(i.getResidWeight()-min);
255
256         UB += clique.getWeight();
257      }
```

```
258         instance.reset();
259         return UB;
260      }
261
262      private double internalSV13(Item i, int currWeight) {
263         return (capacity-currWeight)*i.getRatio();
264      }
265
266      public double getObjectiveValue() {
267         return globalLB;
268      }
269
270      public boolean isSolved() {
271         return solved;
272      }
273
274      public int getNodesVisited() {
275         return nodesVisited;
276      }
277
278      public LinkedList<Item> getSolutionSet(){
279         return solution;
280      }
281
282      public double getUB(LinkedList<Item> freeSet, int currWeight) {
283         double UB = 0.0;
284
285         if (capCC)
286            UB += capCC(freeSet);
287         else if (CC)
288            UB += CC(freeSet);
289         else
290            UB += fracKP(freeSet, currWeight);
291
292         return UB;
293      }
294
295   }
```