Optimization Algorithms
Final Exam Project Report

Simone Bondi

January 30, 2026

Abstract

This report describes six solvers for the Disjunctively Constrained Knapsack Problem (DCKP) that were implemented by the candidate, of which three are exact solvers.

1 Disjunctively Constrained Knapsack Problem

The Knapsack Problem (KP) is a very well-known problem which consists in determining which items to choose out of a finite set of items, each having an assigned profit and weight, in order to maximize the total profit while maintaining the total weight below a fixed capacity. The Disjunctively Constrained KP (DCKP) adds disjunctive conflicts between pairs of items. Mathematically, let c be the knapsack capacity, n the number of items, w_i the weight and p_i the profit of the i_{th} item. Additionally let the set $E \subseteq \{(i,j), 1 \leq i < j \leq n\}$ be the set of conflicts, i.e. the set of pairs of items which cannot be selected at the same time, and $m = |E|$. Formulated as an optimization problem (1):

$$\begin{aligned} \max_{x} \quad & \sum_{j=1}^{n} x_j p_j \\ \text{subject to} \quad & \sum_{j=1}^{n} x_j w_j \leq c, \\ & x_i + x_j = 1 \quad \forall (i,j) \in E \\ & x_j \in \{0,1\}, j = 1 \dots n \end{aligned}$$

Both the KP and the DCKP are NP-complete problems. Although there exist dynamic programming problems for the KP with pseudo-polynomial complexity in case c is bounded, such is not the case for DCKP. As such, it is often solved directly as an Integer Linear Programming (ILP) problem.

2 Problem relaxations

Two relaxations of DCKP are used to obtain upper bounds or find approximate solutions in the solvers described in this report.

2.1 Fractional Knapsack Problem

The Fractional KP is a relaxation of the base KP where the items are allowed to be partially selected and conflicts are not considered at all. Formulated as an optimization problem (2):

$$\begin{aligned}
\max_{x} \quad & z_f = \sum_{j=1}^{n} x_j p_j \\
\text{subject to} \quad & \sum_{j=1}^{n} x_j w_j \leq c, \\
& 0 \leq x_j \leq 1, j = 1 \dots n
\end{aligned}$$

FKP can be quickly solved optimally with greedy strategy, consisting of sorting the items by descending profit/weight ratio $\frac{p_i}{w_i}$ and then greedily choosing the items with highest ratio. The execution time is dominated by the sort, which can be done in $O(n \log n)$. The resulting z_f^* is used as an upper bound to DCKP.

2.2 Lagrangian Relaxation of Fractional DCKP

The Lagrangian relaxation of DCKP (LDCKP) consists in allowing fractional item selection and relaxing the conflict constraints as soft bounds with the Lagrangian formulation. Let $\lambda_k \geq 0$ be the Lagrange multiplier for the k_{th} disjunctive constraint, or equivalently $\lambda_{e=(i,j)}$ if $e = (i,j)$ is the k_{th} element of E with respect to some ordering (3):

$$\begin{aligned}
\max_{x,\lambda} \quad & L = \sum_{j=1}^{n} x_j p_j + \\
& \quad \sum_{(i,j) \in E} \lambda_{(i,j)} (1 - x_i - x_j) \\
\text{subject to} \quad & \sum_{j=1}^{n} x_j w_j \leq c, \\
& 0 \leq x_j \leq 1, j = 1 \dots n \\
& \lambda_k \geq 0, k = 1 \dots m
\end{aligned}$$

By fixing λ and by means of simple mathematical steps L can be rewritten to obtain an equivalent FKP problem. Let the set of conflicts concerning item j be $\partial_j = \{(x,y) \in E \mid j = x \vee j = y\}$, then (2.2):

$$L^*(\lambda) = \sum_{j=1}^{n} x_j(p_j - \sum_{e \in \partial_j} \lambda_e) + \sum_{e \in E} \lambda_e$$

Given a fixed λ, the solver described in Section 2.1 can be used to compute a feasible $x*$ for Equation 3. $L^*(\lambda)$ is then trivial to compute. Of course, $L(0) = z_f^*$ (Equation 2). Next, we want to choose λ to minimize this quantity while $\lambda_k \geq 0, k = 1 \dots m$, since that will give us the optimal solution to Equation 3. It can be proven that $L^*(\lambda)$ is a piecewise convex function of λ, and where it is differentiable its gradient with respect to λ is such that $\frac{\partial L^*}{\partial \lambda_{(i,j)}} = 1 - x_i^* - x_j^*$.

Because we can quickly compute L^*, it makes sense to use a numerical optimization approach, although numerical experiments show that in the candidate's implementation computing the profits immensely dominates over both solving the FKP problem and applying the subgradient method, limiting the appropriate maximum number of iterations to small values ($\sim$ 20 iterations) in case this solver needs to be run many times. The Candidate did not have enough time to find a solution to this problem, and in fact it will only be used in the Relaxation solver described in Section 3.2 as it only needs to be run once.

Plotting the value of $L*$ vs the iteration k show that in such a little amount of iterations the best solution is to normalize the gradient and scaling it by a constant factor. Let $g(k) = \frac{\nabla L(\lambda)}{\nabla \lambda_{(k)}}$ and $\alpha > 0$ be the constant step size, then the subgradient step used by the candidate is $\lambda_{(k)} = \lambda_{(k-1)} - \alpha \frac{g(k)}{\|g(k)\|_2}$.

3 inexact solvers (avg)

In this section the three inexact solvers developed by the candidate are described.

3.1 Greedy solver

The greedy solver starts from an empty knapsack. It then iterates over the items in order of descending $\frac{p}{w}$, taking each item if there is enough capacity to accomodate it and that it does not conflict with any better item that was already taken (worse items cannot have been taken).

3.2 Relaxation solver

This solver first runs the LDCKP relaxation solver described in Section 2 with an especially extended number of iterations to obtain an initial solution. It will then drop all partially chosen items, meaning all items j such that $0 < x_j < 1$. Then, iterating over the items in order of **ascending** $\frac{p}{w}$, it will greedily drop all items j that conflict with better items that have been chosen, namely all items j such that $\exists\ (i,j) \in E \mid x_i = 1$. Finally, it will improve the solution with the same strategy as the greedy solver described in Section 3.1, although this also has to make sure not to conflict with worse items, as they may have been chosen by the relaxation.

It is worth noting that if the FKP relaxation was used the result would always be exactly equal to that of the Greedy solver.

3.3 Hillclimb solver

The Hillclimb solver is a local search algorithm implementation using the Hillclimb strategy. The moveset is composed of an add move, which adds item i to the knapsack, and a swap move, which swaps two items $i \mid x_i = 1$ and $j \mid x_j = 0$. At each step, the solver iterates over all feasible moves at the current solution, evaluating the immediate profit and keeping track of which move results in the best improvement. Once the iteration terminates, it applies the best move, effectively moving to a neighbouring solution that has an higher profit.

4 Exact solvers

In this section the three exact solvers developed by the candidate are described.

4.1 CP-SAT solver

This solver uses the CP-SAT solver from Google OR-Tools to solve the problem directly, exactly as described in the professor's paper and due to this it will not be described further. For this work it is intended to be a reference, as it quickly finds the optimum for most problems.

4.2 Implicit Enumeration solver

This solver was described in Yamada et. al [2]. This algorithm performs what is essentially a BFS on a binary tree where each distinct node at level j represents the items (path) taken up until that level, $x_{1:j}$ (using MATLAB notation). Because there could be 2^n nodes, it prunes unpromising or unfeasible nodes.

At each level j (i.e. item j, in descending order of $\frac{p}{w}$), it prepares a list of nodes over which to iterate at the next level, Ω_{j+1}. To start the algorithm, Ω_0 is set to a node representing an empty knapsack with no choices made. For each node in Ω_j, meaning items 1 to $j-1$ have been decided, its two children nodes are evaluated, one with $x_j = 1$ and one with $x_j = 0$. If the nodes are unfeasible, or if there is a better node at the current level, or if the upper bound of the node is less than some lower bound to the problem, the nodes are not added to Ω_{j+1}.

There being a "better" node at the current level means that such node has better or equal profit, better or equal free space, and has just a subset of conflicting items with index $> j$ with respect to the current node. In the candidate implementation, this is a linear search over all nodes in the current level and when enabled is an important part of the execution time, to the point where the candidate experimented with removing it completely, and although this is not evaluated in the numerical results due to a lack of time, this resulted in extremely fast solve times for some problems (5 to 10 times faster than CP-SAT) but Ω becoming so large that the solver runs out of memory in other problems. It is important to note that, for computational efficiency, each node contains a "conflict set", or a sorted set of indices of items not yet in the knapsack that conflict with items already in the knapsack, which is the biggest contributor to memory usage.

The paper authors also go on to describe an interval reduction method for which the lower bound passed to the solver is made to exponentially decrease from an upper bound (computed using one of the relaxations described in section 2) to a lower bound (computed using one of the inexact solvers described in Section 3). This was attempted by the Candidate, but found unsuitable for the very short time deadlines used in the numerical experimentations, and thus the Candidate's implementation just uses the lower bound computed by the greedy solver described in 3.1.

4.3 Branch and Bound solver

This solver was developed by the candidate after reasoning on the weaknesses of the Implicit Enumeration solver, mainly how slowly it improves the lower bound. Instead of evaluating each level sequentially, this algorithm does not have an explicit consideration of levels and is instead allowed to freely explore the search space, reasoning being that when there are a lot of conflicts, it is to be expected that good solutions may include items from lower levels, even if the most promising items are found at the very first levels. A single priority queue Ω orders the nodes by descending upper bound, which is initialized with a single empty node. At each iteration, the node with maximum upper bound is dequeued from Ω, and the two children are checked for feasibility and whether their upper bound is greater than or equal to the current best solution - children that satisfy both are added to the queue, and the solution found by the relaxation is greedily improved with the same strategy described in the Relaxation Solver (Section 3.2) and used as lower bound if better than the current one, allowing more nodes to be pruned especially in early iterations.

5 Numerical Experiments

This section describes the dataset used, then presents and discusses the results for all six algorithm described in Sections 3 and 4.

5.1 Hardware

All experiments were run on a plugged-in laptop with a i7-12700H CPU, which has a maximum frequency of 4.7GHz and 24MB of L3 cache, and two 8GiB

SODIMM DDR4 RAM sticks clocked at 3.2GHz. The operating system is set to maximum performance mode.

Only the CP-SAT solver uses all 14 cores (20 threads) of the CPU, while the rest of the solvers are running on a single thread due to lack of time to develop a multi threaded solution.

5.2 Dataset

For these experiments, the candidate used a subset of the dataset used by Bettinelli et. al in [1], considering 271 instances out of the original 4800, and limited the execution time of each algorithm to 30 seconds. This results in a worst-case execution time between two and two and a half hours for each algorithm, allowing for fast experimentation at the expense of not having much data to develop statistics with - the candidate deemed this tradeoff more than acceptable, this being a didactic project.

The original dataset consists of randomly generated instances classified in the following manner, where each resulting class generates 10 instances:

- by conflict graph density (where 0 means no conflicts at all and 1 means that all items conflict with each other), ranging from 0.001 to 0.9;
- by the profit of each item being independent from its weight (R class) or correlated to it (C class);
- densities 0.1 to 0.9 are divided into three classes representing a linear scale of the knapsack capacity. Classes C10 and R10 have ten times the capacity of C1 and R1.
- densities 0.1 to 0.9 are divided in two groups by whether the knapsack has small capacity and the items have small weight, or whether the knapsack and items are big;
- again for densities 0.1 to 0.9, each group is divided in four classes by increasing number of items;
- for densities 0.001 to 0.05, by number of items and capacity.

Any combination of these results in a different class, for example "density 0.1 R10 class 4".

The subset was formed by restricting the conflict graph densities to 0.01, 0.1 and 0.9, which allows testing the various algorithm at different density extremes. To further limit the number of instances, from density 0.1 to 0.9 only two instances for each class are taken.

Importantly, no data is available on the optimum value for each instance. Because CP-SAT is extensively tested and reports it found the optimum in most of the instances, its results are used for this purpose. Where available, we will not only consider the usual optimality gap using the bounds reported by each algorithm, but also consider a "true" optimality gap with the lower bound reported by each algorithm and the optimal solution reported by CP-SAT.

5.3 Results

5.3.1 Relaxations

Figure 1, admittedly quite noisy, will show real-time data collected from the *bnb* solver by replacing the usage of the FKP solver with the LDCKP solver and running it multiple times on instance 10 of C10 class 7, density 0.1 with two different values for step size α: 2.0 and 10.0.

It may be useful to remember that the very first iteration of LDCKP is equivalent to the output pass of the FKP solver, as $\lambda = 0$ means that the disjunctive constraints are not weighed in the cost function (see Section 2).

The plots will show the iteration k on the x axis and the upper bound $L^*(k)$ on the y axis. Because the software used to plot the data is receiving the data in real time and interpreting it as a continuous plot, straight and oblique lines return from the end of the last run of the LDCKP solver to the start of the next (i.e. from one node to the other of *bnb*). These should be ignored.

This plots allows visualizing in a very intuitive way, especially in real time as will be presented in the oral exam, the effect of the same parameter alpha on different "instances", as *bnb* "pins" the first j items and j can vary extensively given that the solver is allowed to freely explore the search space. Overlaying multiple plots with different colors, like done in Figure 1, shows how different values of α affect different solutions.

It can be seen by comparing the blue and green top plateauing lines that, if the initial upper bound is good enough, a smaller α allows the solver to reach lower values. However if the initial upper bound is very weak this value of alpha is not enough, as can be seen in the jagged blue lines with shallow slope at the center.

A bigger alpha instead fails to get close to the optimal value in the very top "flat" jagged green lines, as can be seen by comparing them with the slightly lower flat blue lines, but it allows improving bad upper bounds by a lot, as can be seen by the $1/x$-looking green lines below and comparing them with the jagged shallow blue lines at the center, which represent similar problems.

Such behaviour is hard to predict as it varies a lot between instances, and makes the choice of α a guessing game, which coupled with the high execution time required to improve the upper bound, justifies the use of the FKP relaxation in all solvers (apart from *relax*, which runs one-time).

5.3.2 Inexact solvers

Table 1 to 8 show average results for the inexact solvers, separated by correlation, problem scale and density. The Opt Gap reported is the actual optimality gap with respect to the optimal solution obtained by CP-SAT - instances for which CP-SAT did not prove optimality are not considered in the averages.

The time taken by the relax solver is always unsuprisingly high, as the parameters for the subgradient method were set to unusually high values, as described in 3.2.

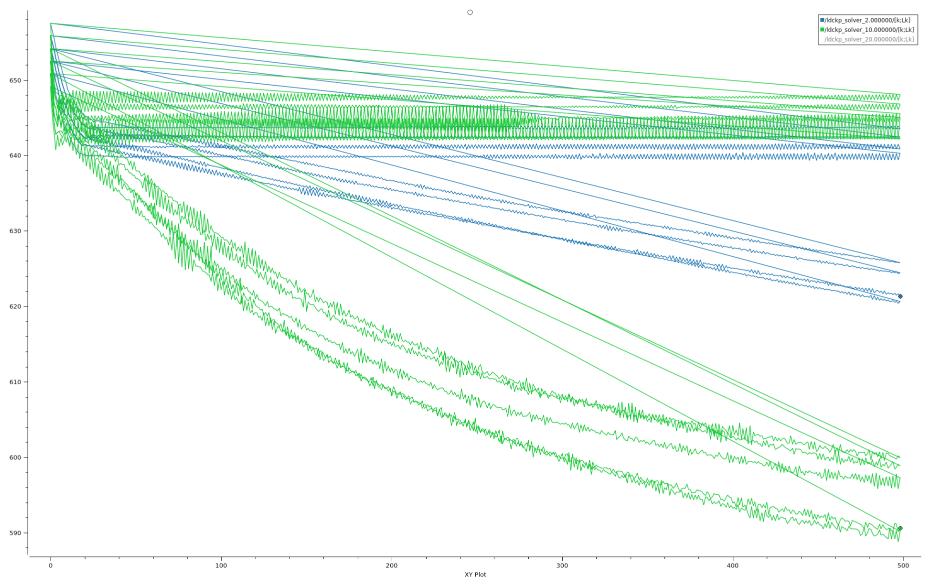

Figure 1: LDCKP solver run on subsequent problems generated by BNB, with $\alpha = 2.0$ (blue) and $\alpha = 10.0$ (green)

In datasets where there is a large number of items, or there are few conflicts, or the knapsack capacity is very large, meaning where there are lots of moves to choose from, the *hillclimb* solver is unsurprisingly unacceptably slower relative to the other inexact algorithms. However, it is surprisingly more effective than the other algorithms when the knapsack capacity is large, especially when the weights and profits are correlated, as can be seen by the lower optimality gap in the Correlated instances with scale 10.

The *relax* solver has always an optimality gap at least better than that of greedy. This is to be expected, as it is greedily improving the solution obtained from a tighter relaxation than FKP, which as discussed would be the same as *greedy*. It is interesting however how little the improvement is on average.

Overall, the results, especially the execution time, justify the use of the Greedy strategy in the *bnb* solver.

5.3.3 Exact solvers

Just like before, tables 9 to 16 show average results for the exact solvers separated by correlation, problem scale and density. In addition to the previous columns, this also presents the estimated optimality gap, which is the optimality gap that can be computed from the lower and upper bound of each solver indipendently from the true optimum value. A zero valued actual optimality gap but high valued estimated optimality gap shows weakness in lowering the upper bound. In correlation to that, the column LB T[s] (Time for Lower Bound) shows the time required to find the best solution that the solver found before terminating. Low values but high solver times can show that the solver finds a solution very quickly but needs a lot of time to prove that it is optimal.

Solved shows the count of solved instances per group. The optimum is 16 for the dense instances and 40 for the extremely sparse ones.

First, in sparse instances both *bnb* and *ienum* are too slow at finding a solution and do not manage to solve any instance, albeit they reach close to optimality. *bnb* has, however, both a better solution (lower actual optimality gap) and better upper bound estimates (lower estimated optimality gap) than *ienum*. This is most likely due to the fact that *ienum* is too slow and is stuck at the upper levels, while *bnb* is allowed to go to deeper "levels", which allows it to discard solutions with very high upper bounds (i.e. that have too much freedom) earlier. All solvers are constantly finding better solutions, as can be seen by the lower bound times being similar to the solver times, meaning that they usually find good solutions even at the very last iterations.

The only time *ienum* is better than *bnb* is in some R3 instances, where the former manages to solve more instances than the latter. However, the average actual optimality gap having value 0 and the lower bound time being very low shows that *bnb* found the optimum value early but then failed to prove optimality.

It is interesting how in all averages the CP-SAT solver times equals the lower bound times, meaning that it usually proves optimality right when it finds the best solution. Note that the lower bound timestamp is updated only when the solver reports a solution that has higher profit than the previously reported one. This exploits the *SolutionCallback* mechanism of the solver.

bnb comes very close to the optimum in in all very dense instances but the ones belonging to C3 and C10, which prove to be the hardest dense instances as was also shown by Bettinelli et. al [1]. This can be seen by the actual optimality gap being zero or close to zero.

6 Conclusions

It would be interesting to get more details about which algorithm performs better in which instances, but due to a lack of time (and the dataset subset being particularly small) no conclusions were made on that front. For example, the candidate fails to find reasons why the datasets are organized the way they are and instead would argue that these statistics are not particularly useful. Instead of grouping the data solely by density, correlation and scale it would be interesting to try grouping it by capacity, number of items, some measure of correlation and other numerical measures. This would allow more descriptive statistics - even in the original paper it seems to the candidate that the way to interpret the data is just to have an application in mind and make decisions based on the class that is closest to the one described. The candidate believes this is not very useful for drawing conclusions for developing more general solvers.

References

[1] Andrea Bettinelli, Valentina Cacchiani, and Enrico Malaguti. A branch-and-bound algorithm for the knapsack problem with conflict graph. *INFORMS Journal on Computing*, 29(3):457–473, 2017.

[2] Yamada Takeo. Heuristic and exact algorithms for the disjunctively constrained knapsack problem. *Jōhō-shori Gakkai Ronbunshi*, 43(9):2864–2870, 09 2002.

Table 1: C0 - inexact solvers

	Time [us]			Opt Gap [%]		
Solver	greedy	hillclimb	relax	greedy	hillclimb	relax
density						
0.01	330	8907	9948	1.43	37.07	0.47

Table 2: C1 - inexact solvers

	Time [us]			Opt Gap [%]		
Solver	greedy	hillclimb	relax	greedy	hillclimb	relax
density						
0.1	42	211	16679	5.83	12.68	5.83
0.9	337	604	171279	33.54	7.63	34.42

Table 3: C3 - inexact solvers

	Time [us]			Opt Gap [%]		
Solver	greedy	hillclimb	relax	greedy	hillclimb	relax
density						
0.1	121	877	16832	5.59	12.10	5.30
0.9	436	1099	171112	63.60	22.02	59.68

Table 4: C10 - inexact solvers

	Time [us]			Opt Gap [%]		
Solver	greedy	hillclimb	relax	greedy	hillclimb	relax
density						
0.1	547	4788	17862	27.10	8.14	22.64
0.9	446	1097	176672	63.67	22.27	61.51

Table 5: R0 - inexact solvers

	Time [us]			Opt Gap [%]		
Solver	greedy	hillclimb	relax	greedy	hillclimb	relax
density						
0.01	284	35964	9845	2.79	52.09	1.42

Table 6: R1 - inexact solvers

	Time [us]			Opt Gap [%]		
Solver	greedy	hillclimb	relax	greedy	hillclimb	relax
density						
0.1	40	504	17185	3.96	17.20	3.96
0.9	306	980	190625	12.66	21.73	19.16

Table 7: R3 - inexact solvers

	Time [us]			Opt Gap [%]		
Solver	greedy	hillclimb	relax	greedy	hillclimb	relax
density						
0.1	114	2206	16403	2.75	22.81	1.35
0.9	504	1374	181314	23.71	22.47	27.65

Table 8: R10 - inexact solvers

	Time [us]			Opt Gap [%]		
Solver	greedy	hillclimb	relax	greedy	hillclimb	relax
density						
0.1	432	13902	18203	12.70	12.47	12.43
0.9	500	1329	180933	23.71	22.47	26.22

Table 9: Correlated Sparse - exact solvers

	Solver T[s]			Act. Opt G[%]			Est. Opt G[%]			LB T[s]			Solved[/40]		
Density	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum
0.01	16.30	30.04	30.03	0.00	0.28	1.43	0.08	6.34	9.24	16.30	30.04	30.03	20	0	0

Table 10: C1 - exact solvers

	Solver T[s]			Act. Opt G[%]			Est. Opt G[%]			LB T[s]			Solved[/16]		
Density	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum
0.1	0.20	4.02	9.81	0.00	0.20	0.20	0.00	0.36	0.50	0.20	4.02	9.81	16	14	12
0.9	1.41	8.91	7.20	0.00	0.00	0.00	0.00	0.80	0.53	1.41	8.91	7.20	16	12	13

Table 11: C3 - exact solvers

	Solver T[s]			Act. Opt G[%]			Est. Opt G[%]			LB T[s]			Solved[/16]		
Density	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum
0.1	0.24	18.97	29.23	0.00	1.91	4.71	0.00	2.08	5.31	0.24	18.97	29.23	16	6	1
0.9	2.60	12.28	11.91	0.00	7.27	11.77	0.00	14.18	17.56	2.60	12.28	11.91	16	10	10

Table 12: C10 - exact solvers

	Solver T[s]			Act. Opt G[%]			Est. Opt G[%]			LB T[s]			Solved[/16]		
Density	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum
0.1	22.29	30.08	30.03	0.00	0.26	27.10	0.83	5.45	20.60	22.29	30.08	30.03	6	0	0
0.9	2.65	12.24	11.91	0.00	6.81	11.94	0.00	30.24	31.37	2.65	12.24	11.91	16	10	10

Table 13: Random Sparse - exact solvers

	Solver T[s]			Act. Opt G[%]			Est. Opt G[%]			LB T[s]			Solved[/40]		
Density	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum
0.01	0.13	30.04	30.03	0.00	0.78	2.79	0.00	9.73	15.39	0.13	30.04	30.03	40	0	0

Table 14: R1 - exact solvers

	Solver T[s]			Act. Opt G[%]			Est. Opt G[%]			LB T[s]			Solved[/16]		
Density	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum
0.1	0.25	3.51	4.22	0.00	0.00	0.47	0.00	0.37	2.19	0.25	3.51	4.22	16	15	14
0.9	0.94	2.18	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.94	2.18	0.02	16	16	16

Table 15: R3 - exact solvers

	Solver T[s]			Act. Opt G[%]			Est. Opt G[%]			LB T[s]			Solved[/16]		
Density	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum
0.1	0.18	3.04	6.06	0.00	0.00	0.30	0.00	0.02	0.77	0.18	3.04	6.06	16	15	13
0.9	2.12	11.86	7.43	0.00	0.00	1.10	0.00	10.95	5.73	2.12	11.86	7.43	16	10	14

Table 16: R10 - exact solvers

	Solver T[s]			Act. Opt G[%]			Est. Opt G[%]			LB T[s]			Solved[/16]		
Density	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum	cpsat	bnb	ienum
0.1	7.75	26.67	30.03	0.00	0.69	12.49	1.25	10.90	30.34	7.75	26.67	30.03	14	3	0
0.9	2.22	12.18	11.78	0.00	0.00	1.10	0.00	28.44	25.52	2.22	12.18	11.78	16	10	10