Construction of Synthetic CDO Squared

Olof Sivertsson

Department of Information Technology, Uppsala University
Box 337, 751 05 Uppsala, Sweden
olof@olofsivertsson.com

Abstract. We present techniques used in the implementation of an efficient constraint program for the portfolio optimization (PO) problem. This important combinatorial problem in the credit derivatives market arises for example when constructing synthetic collateralized debt obligations (CDOs) squared. A close relationship with the balanced incomplete block design (BIBD) problem exists which we make use of. Due to the large size of typical PO instances, global solving is not possible, instead we embed and solve sub-instances. The high quality of our approximate solutions can be assessed by comparison with a tight lower bound on the cost. Together with detection of BIBDs, symmetry breaking, extended reuse of already solved instances, and existence-checking during search, the performance of the program becomes good enough for constructing optimal portfolios of CDOs squared, with sizes common in the credit derivatives market, within minutes or seconds.

Supervisor:
Pierre Flener

Examiner:
Justin Pearson

Table of Contents

Construction of Synthetic CDO Squared 1
Olof Sivertsson
1 Introduction 3
2 Constraint Programming 4
3 Portfolio Optimization 5
3.1 Short Financial Background 5
3.2 Balanced Incomplete Block Designs 6
3.3 Portfolio Optimization 8
3.4 Lower Bound on λ 9
4 Implementation 18
4.1 Basic Model 18
4.2 Labeling Order and Static Symmetry-Breaking 19
4.3 BIBD Detection and BIBD Model 20
4.4 Checking for Known Non-Existing BIBDs 21
4.5 First Intersection and First Column Optimizations 21
4.6 Checking Feasibility During Search 23
4.7 Solving Using the Complement 24
4.8 Caching Results and Extended Lookup 25
5 Approximately Solving by Embedding Subinstances 27
6 Symmetry-Breaking During Search 28
6.1 Overview 29
6.2 Modeling the Row-Column Relationship 30
6.3 Aggregating Identical Columns 31
6.4 Same-Cardinality Optimization 31
6.5 Applying the Symmetry-Breaking 31
6.6 Comparison to STAB 32
6.7 Alternative Approach 32
6.8 Benchmarks 32
7 Conclusion 33

1 Introduction

The work presented in this report is the result of a M.Sc. in computer science project with the ASTRA Research Group* at Uppsala University. The main objective of this work has been to improve and expand on the work done previously by members of the ASTRA Research Group on the problem of Financial Portfolio Optimization [8]. The focus is on how constraint programming can be used to efficiently solve this problem.

The work includes research into purely theoretical areas, e.g. conditions for the existence of POs and symmetry-breaking to improve performance of the search for POs. But it also includes a lot of practical work. During this work a coherent package for solving PO problems should be developed, something that never was done in [8]. This will be done using the support for constraint programming over finite domains found in SICStus Prolog.**

The end result should show performance improvements over the work presented in [8] and hopefully lead to new insights about the PO problem and how it can be solved efficiently using constraint programming.

By solving the Financial Portfolio Optimization problem efficiently issuers of certain types of financial instruments will benefit directly when their instruments can be made more attractive and because of this hopefully result in better sales.

But this work might prove useful in other areas too, since closely related problems have found use for many different tasks, such as scheduling, design of experiments and constructing error correcting codes. The author is also convinced that the techniques used for symmetry-breaking in matrix models can be used successfully for many other problems that make extensive use of matrix models, with none or only minor modifications.

Finally, this work shows an application of constraint technology that solves a real-world problem in finance and hence shows another example of using constraint technology successfully for a real world problem.

In this introduction we have discussed the motivation for this work and what benefits it could have. The remainder of this paper is organized as follows. Section 2 presents constraint programming briefly, followed by Sect. 3 where we present the problem first in its financial domain and then the abstracted problem and introduce necessary theoretical background and results. Next Sect. 4 discuss the techniques used in the problem solver, Sect. 5 presents how large problems are solved and Sect. 6 discuss the usage of more elaborate symmetry-breaking. Finally Sect. 7 presents conclusions of this work and possible future work.

---

*Analysis, Synthesis, and Transformation/Reformulation of Algorithms in constraint technology. More information at http://www.it.uu.se/research/group/astra/.

**All implementation and benchmarks were done using SICStus Prolog 3.12.2 on Debian GNU/Linux running Linux 2.4.18 on an AMD AthlonXP 2400+ with 256MB RAM.

2 Constraint Programming

Although reasoning through constraints in programming was done already in the 1960's., constraint programming as we know it today with constraint programming languages has been developed mainly in the last 20 years [2].

A CSP, Constraint Satisfaction Problem, is a set of constraints on a collection of variables that limit the domains of these variables. A solution to a CSP is an assignment to each variable such that all constraints are satisfied. There are also COPs, Constraint Optimization Problems. They differ from CSPs by also having a function on the variables that should be minimized or maximized.

The process of writing software to solve CSPs and COPs is known as constraint programming. This is (preferably) done using some constraint solver, such as SICStus Prolog's CLPFD* module [3]. This constraint solver was used to implement the techniques presented in this paper.

When using a constraint solver the first step is to define the variables including their domains, then post constraints on these variables and finally tell the constraint solver to start looking for one or more solutions.

There are some different types of constraints. The most basic constraints are arithmetic constraints, e.g. the sum of a set of variables should be fixed. Then there are constraints that capture more structure, with the most common being `all_different` that makes sure that no two variables given to it are equal. And finally there are reified constraints that links a boolean and an expression together such that the boolean is true when the expression is satisfied, and false when the expression is not satisfied.

To solve a CSP or COP the constraint solver will alternate between what is known as propagation and labeling. Propagation narrows the domains of the variables (if possible) by removing values that are inconsistent with the constraints. When propagation no longer can narrow the domain of any variable further the solver will begin labeling instead. It will do this by choosing a variable and limit the domain of this variable (more about this below). Then the solver will do propagation again for as long as possible, followed by labeling again, and so on.

After doing a couple of iterations with propagation and labeling the solver might reach a state where a variables domain has become empty. This is known as an inconsistent state and forces the solver to backtrack and redo the last labeling (domain-reduction) in another way. If the domain-reduction on this variable has already been done in all possible ways it will redo the domain-reduction prior to that and so on. If the solver exhausts all possibilities it will report back that the problem has no solution.

In which order the constraint solver tries to limit the domains of the variables is decided by the constraint programmer choosing a variable and value order. Commonly used techniques are choosing the variable with the smallest domain first, and value ordering is usually done step by step from the lowest or highest value in the domain, but more elaborate techniques are possible.

*Constraint Logic Programming over Finite Domains

The variable and value ordering used can have a huge impact on both runtime performance and memory requirements. In the optimal case the correct values are chosen for each variable at the first try. In the worst case the variable with the largest domain is tried first, and only the last value tried in that variables domain is correct. In the latter case the solver will try all possibilities for all other variables the maximum number of times in vain.

Different constraints allow for different amounts of propagation. Since more propagation means less wrong labeling can be done, more propagation generally leads to better performance. Usually the more structure is captured by a constraint the more efficient propagation can be achieved. In most cases it is therefore more efficient to post a few constraints that capture a lot of structure instead of many small constraints that fail to capture "the big picture".

For more background information about constraint programming and CSPs and COPs [2] is recommended.

3 Portfolio Optimization

3.1 Short Financial Background

Credit derivatives are used in finance to transfer the risk of a specified financial event happening to a credit asset without transferring the asset itself [16]. The credit derivative that we will deal with in this report is (synthetic) Collateralized Debt Obligations, CDOs, and since this is a credit derivative that consists of Credit Default Swaps, we will begin this short financial background by a brief look at them.

Credit Default Swaps, or CDS for short, is a type of insurance for the holder of a financial asset against some specified financial event. Let us look at an example to make this clear.

Example 1. (Based on the example in [15].) Suppose you are working for a small bank that six months ago issued a three year bond of $1.000.000 to a new IT business. The new IT business pays 10% interest per year. Your bank has now become worried that the IT business may not be able to pay back its debt. Therefore the small bank decides to buy protection against this in the form of a CDS from a derivative issuer. The small bank pays 8% of $1.000.000, i.e. $80.000, divided into quarterly payments of $20.000 to the derivative issuer. In exchange for this the derivative issuer refunds the small bank for its $1.000.000 should the IT business not be able to redeem the bond (perhaps because of bankruptcy) and the quarterly payments stop. Otherwise the small bank is redeemed by the IT business as if the CDS had not existed, but the derivative issuer has also made some money at the expense of the small bank.

So to summarize, a protection buyer does regular payments to a protection seller in exchange for a payment in the event of some pre-defined credit event.

CDS are the most common credit derivative [16] and naturally the issuers of CDS want to minimize their risk by selling on their CDS. And just like ordinary

stock shares can be lumped together into mutual funds, CDS can be lumped together into baskets consisting of different CDS (with varying risk). And this leads us to CDOs.

Collateralized Debt Obligations [14] are credit derivatives that consist of a large number of other credit derivatives. There are cashflow CDOs where the underlying credit derivatives are bonds or loans, and synthetic CDOs where the underlying credit derivatives are CDS.

A CDO is divided into tranches (baskets), each consisting of a fixed sized subset of the underlying credit derivatives. Since different tranches consist of different bonds/loans or CDS, they have different risk (and potential returns) associated with them. There are also CDO-Squared (CDO^2), where the underlying credit derivatives also are CDO tranches. They are hard to risk-analyze because the same underlying credit derivative may be available in many tranches.

An investor can choose to invest in a tranche from a CDO that matches the risk the investor wants. But sometimes an investor may want to put a part of his money into a high-risk tranche and another part into a low- or medium-risk tranche. The issuer of a CDO still want the investor to invest as much as possible of his money in the issuers CDO instead of looking for another issuers CDO.

To be able to convince the investor of this no two tranches should overlap more than absolutely necessary because the investor wants to spread his risks, otherwise he could invest in just one CDO instead. Hence the question arises of how a CDO issuer should construct his CDO to minimize the overlap between any two tranches. And this is exactly the problem we try to solve efficiently in this report. We denote this problem as the Portfolio Optimization (PO) problem [8].

3.2 Balanced Incomplete Block Designs

Before we begin to discuss the abstracted portfolio optimization problem, and how we solve it, we will need to look at a closely related problem, BIBDs.

Balanced Incomplete Block Designs, BIBDs, are mainly used to study algebraic geometry and designing statistical experiments, but can also be used when constructing error correcting codes [10]. We will study BIBDs because the portfolio optimization problem can be viewed as a generalization of BIBDs and will be the foundation we use for solving Portfolio Optimization problems.

Definition 2. *A BIBD $\langle v, b, r, k, \lambda \rangle$ consists of b* blocks *$B_1, \ldots, B_b$ each containing a k-element subset of $\{1 \ldots v\}$ and v* coblocks *$V_1, \ldots, V_v$, each containing a r-element subset of $\{1 \ldots v\}$ such that the intersection between any two coblocks equals λ, i.e.*

$$
\begin{aligned}
B_i &\subseteq \{1 \ldots v\} \\
V_i &\subseteq \{1 \ldots b\} \\
\forall i \in \{1 \ldots v\}\ |V_i| &= r \\
\forall j \in \{1 \ldots b\}\ |B_j| &= k \\
\forall i \neq j\ |V_i \cap V_j| &= \lambda
\end{aligned}
$$

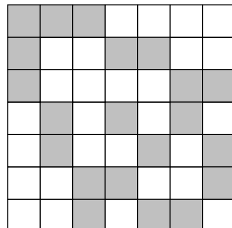

Fig. 1. Incidence matrix for the BIBD $\langle 7, 7, 3, 3, 1\rangle$. Columns represent blocks, rows represent coblocks. Filled squares represent that $[m_{ij}]$ equals one and unfilled squares that it equals zero. Note how rows and columns sum to three and how the intersection (overlap) between any two rows is one.

A BIBD is often represented using an incidence matrix M. An element of this matrix in row i and column j is one if the block B_j contains i (or equivalently, if coblock V_i contains j), and zero otherwise. Hence blocks are represented by columns and coblocks by rows:

$$M = [m_{ij}], i \in \{1 \dots v\}, j \in \{1 \dots b\}$$
$$m_{ij} = 1 \Leftrightarrow i \in B_j,$$
$$m_{ij} = 0 \Leftrightarrow i \notin B_j$$

Example 3. Figure 1 shows an incidence matrix for the BIBD $\langle 7, 7, 3, 3, 1\rangle$. Columns represent blocks, rows represent coblocks.

There are some interesting properties of BIBDs that we will find useful later in this report. First, not all values of the parameters v, b, r, k, and λ result in BIBDs, actually very few do. The following two simple identities are required for the parameters to be able to represent a BIBD.

$$vr = bk \tag{1}$$
$$r(k-1) = \lambda(v-1) \tag{2}$$

When the parameters satisfy these two identities they are said to be *admissible*.

But just because the parameters are admissible does unfortunately not result in a BIBD. An example of this is $v = b = 22, r = k = 7, \lambda = 2$ which are admissible, and still there does not exist a BIBD with these parameters. The problem is that the two identities don't capture the fixed intersection size between any two different coblocks, only the *sum* of the intersections between a particular row and *all* other rows. The pairwise intersection between any two coblocks is captured by the product between the incidence matrix M and its transpose:

$$MM^T = \begin{pmatrix} r & \lambda & \lambda & \cdots & \lambda \\ \lambda & r & \lambda & \cdots & \lambda \\ \lambda & \lambda & r & \cdots & \lambda \\ \cdot & \cdot & \cdot & r & \cdot \\ \lambda & \lambda & \lambda & \cdots & r \end{pmatrix}$$

From this matrix Fisher's inequality for BIBDs can be derived.

Theorem 4 (Fisher [10]). *For any BIBD $\langle v, b, r, k, \lambda \rangle$ we must have $b \geq v$.*

Also, when the incidence matrix M is square, i.e. when $v = b$, the determinant of MM^T and Lagrange's four-squares theorem can be used to get the following theorem.

Theorem 5 (Bruck-Ryser-Chowla [10]). *For any BIBD $\langle v, b, r, k, \lambda \rangle$ which is symmetric, i.e. where $v = b$, we have*

$$\text{for } v \text{ even, } k - \lambda \text{ square; and} \tag{3}$$

$$\text{for } v \text{ odd, } z^2 = (k-1)x^2 + (-1)^{(v-1)/2}\lambda y^2, \text{ where } x, y, z \in \mathbb{Z}, x \neq 0 \tag{4}$$

Using this last theorem we can immediately see that no BIBD $\langle 22, 22, 7, 7, 2 \rangle$ could exist. Still, all these conditions are only necessary and not sufficient, so we can use them only to refute parameters that can't result in a BIBD, but when all these conditions hold we still don't know if the BIBD exists or not. An example of this is $\langle 111, 111, 11, 11, 1 \rangle$ that fulfill all of the conditions above, including the second part of Bruck-Ryser-Chowla ($x = 1,\ y = 1,\ z = 3$), but no BIBD with these parameters exist. Sufficient conditions for the existence of a BIBD are unfortunately not known.*

Finally, let's mention symmetries briefly. Since both rows and columns can be permuted and we still have a (non-)solution to the original problem, we have $v!b!$ symmetries to any BIBD. This will be important to in Sect. 6.

Solving BIBDs using constraint programming has been done multiple times before, especially in the context of symmetry breaking due to the huge number of symmetries that needs to be dealt with during search for non-trivial BIBDs. Some examples of work on this topic are [12], [13], and also [8].

3.3 Portfolio Optimization

The problem of constructing a CDO with v tranches,** each consisting of r credit assets (CDS or bonds/loans) out of a total of b credit assets, such that the intersection between any two tranches is minimized is the problem we tackle in this work. This is known as the Portfolio Optimization (PO) problem [8] and now we present the abstract definition of this problem.

Definition 6. *A PO $\langle v, b, r, \lambda \rangle$ consists of b* blocks *$B_1, \ldots, B_b$ each containing a subset of $\{1 \ldots v\}$ and v* coblocks *$V_1, \ldots, V_v$, each containing a r-element subset of*

*All information from the matrix MM^T need to be captured in identities with v, b, r, k, and λ to get sufficient conditions for the existence of BIBDs, not just "projections" such as the determinant which is used in the theorems above. No one has succeeded in doing this yet.

**The choice of symbols comes from the close relationship to BIBDs.

$\{1 \ldots v\}$ *such that the intersection between any two coblocks is less than* λ, *i.e.*

$$
\begin{aligned}
B_i &\subseteq \{1 \ldots v\} \\
V_i &\subseteq \{1 \ldots b\} \\
\forall i \in \{1 \ldots v\}\ |V_i| &= r \\
\forall j \in \{1 \ldots b\}\ |B_j| &\leq v \\
\forall i \neq j\ |V_i \cap V_j| &\leq \lambda
\end{aligned}
$$

Being a generalization of BIBDs, some POs are actually BIBDs. This occurs when $b \mid vr$ at the same time as every intersection between coblocks has size λ. This knowledge is applied in the implementation presented in Sect. 4.

POs are naturally represented using incidence matrices just as BIBDs, and from this it is easy to realize that any PO has $v!b!$ symmetries. just as BIBDs. The difference compared to BIBDs is that the cardinality of the intersection between any two coblocks is not necessarily equal to λ, but at most λ, and the cardinality of blocks can be anything in the range $1 \ldots v$ as long as the other conditions are satisfied.

This last point about not having a constant cardinality for all blocks means that conditions corresponding to admissibility are not available for POs. Neither does Theorem 4 or Theorem 5 hold for POs.

3.4 Lower Bound on λ

In this section we present a lower bound on the maximum intersection cardinality, λ, between any two coblocks (tranches) from the parameters of a PO. It is also applicable to BIBDs.* This lower bound is useful as a starting point in the search for an optimal solution, i.e. a solution with the smallest possible λ. We will discuss this further when we present the implementation in Sect. 4.1.

Theorem 7 (Corrádi [5,11]). *Let* $\mathcal{F}$ *be the family of* r*-element sets* $V_1, \ldots, V_v$ *and let* B *be their union. If*

$$|V_i \cap V_j| \leq \lambda \quad \forall i \neq j \in \{1, \ldots, v\}$$

then

$$b = |B| \geq \frac{r^2 v}{r + (v-1)\,\lambda}$$

Corollary 8 ([8]). *If we reorder Theorem 7, and add the requirement that* λ *is not negative, we get the following lower bound on* λ

$$\lambda \geq \left\lceil \frac{r(rv-b)}{b(v-1)} \right\rceil \quad \wedge \quad \lambda \geq 0$$

*For BIBDs it can be derived using the admissibility properties found in (1) and (2).

Improving the Lower Bound on λ. In [8] the lower bound on λ of Corollary 8 was used. However, we can prove an improved lower bound on λ that equals Corollary 8 when $b \mid rv$, but results in a tighter lower bound when $b \nmid rv$. The proof of this theorem is presented next, divided into several lemmas that present the necessary underlying results, and finally the main theorem.

Lemma 9. [*]

$$x = \left\lceil \frac{x}{y} \right\rceil (x \bmod y) + \left\lfloor \frac{x}{y} \right\rfloor (y - x \bmod y) \quad \forall x \in \mathbb{Z}, y \in \mathbb{Z}^+$$

Proof. *If* $y \mid x$ *then trivially the right hand side is also* x*. Else consider*

$$x = y \left\lfloor \frac{x}{y} \right\rfloor + (x \bmod y) \tag{5}$$

$$x = y \left\lceil \frac{x}{y} \right\rceil - (y - x \bmod y) \tag{6}$$

Using the shorthand notation $c = \left\lceil \frac{x}{y} \right\rceil$ *and* $f = \left\lfloor \frac{x}{y} \right\rfloor$ *we reorder the second equation to get*

$$y = \frac{x + (y - x \bmod y)}{c}$$

and use it to replace the first occurrence of y *in the first of the two equations above*

$$x = \frac{x + (y - x \bmod y)}{c} f + (x \bmod y)$$

which is equivalent to

$$x \left(1 - \frac{f}{c}\right) = (y - x \bmod y) \frac{f}{c} + (x \bmod y)$$

Dividing both sides by $1 - \frac{f}{c} = \frac{c-f}{c}$*, or equivalently, multiplying both sides by* $\frac{c}{c-f} = c$*, where the latter equality is true because* $c - f = 1$ *when* $y \nmid x$*, we reach*

$$x = c\,(x \bmod y) + f\,(y - x \bmod y)$$

□

Lemma 10. *Let*

$$\sum_{i=1}^{n} a_i = 0 \quad a_i \in \mathbb{Z},\ \forall i \in \{1, \ldots, n-1\} : a_i \geq a_{i+1}$$

then

$$\sum_{i=1}^{k} a_i \geq 0 \quad k \in \{1, \ldots, n\}$$

[*]The result of $a \bmod b$ is the integer leftover when doing the integer division a/b. Precedence is higher than addition/subtraction, but lower than multiplication/division, i.e. we use the definition found in arithmetic as opposed to that found in group theory.

Proof. Think of the a_i as a series of non-increasing slopes of line segments each with length one along the x-axis. If we draw the graph with all these line segments adjoined one after another we will always get a "hill". Then no matter at what x-coordinate k we check the "height", representing the sum up to k, it will be larger or equal to zero. $\square$

Lemma 11. *Let*

$$\sum_{i=1}^{n} x_i = a, \quad x_i \in \mathbb{N}, \quad n \in \mathbb{N}^+ \tag{7}$$

then

$$\sum_{i=1}^{n} x_i^2 \geq \left\lceil \frac{a}{n} \right\rceil^2 (a \bmod n) + \left\lfloor \frac{a}{n} \right\rfloor^2 (n - a \bmod n)$$

Proof. By Lemma 7 we can write

$$\sum_{i=1}^{n} x_i = a = \left\lceil \frac{a}{n} \right\rceil (a \bmod n) + \left\lfloor \frac{a}{n} \right\rfloor (n - a \bmod n)$$

Since $(a \bmod n) + (n - a \bmod n) = n$ we can think of this as choices for the x_i

$$\sum_{i=1}^{n} x_i = \underbrace{\left\lceil \frac{a}{n} \right\rceil + \ldots + \left\lceil \frac{a}{n} \right\rceil}_{a \bmod n \text{ times}} + \underbrace{\left\lfloor \frac{a}{n} \right\rfloor + \ldots + \left\lfloor \frac{a}{n} \right\rfloor}_{n - a \bmod n \text{ times}}$$

or equivalently

$$\left|\left\{ i : x_i = \left\lceil \frac{a}{n} \right\rceil \right\}\right| = a \bmod n$$
$$\left|\left\{ i : x_i = \left\lfloor \frac{a}{n} \right\rfloor \right\}\right| = n - a \bmod n$$

and with this choice of the variables x_i we get

$$\sum_{i=1}^{n} x_i^2 = \left\lceil \frac{a}{n} \right\rceil^2 (a \bmod n) + \left\lfloor \frac{a}{n} \right\rfloor^2 (n - a \bmod n)$$

and we have proved that the stated lower bound is attainable.

Next we need to prove that no matter what set of non-negative integers $\{x_1, \ldots, x_n\}$ is chosen we never get below the value of the above expression. Any other choice of the x_i can be written as

$$\left|\left\{ i : x_i = \left\lceil \frac{a}{n} \right\rceil + \delta_i \right\}\right| = a \bmod n$$
$$\left|\left\{ i : x_i = \left\lfloor \frac{a}{n} \right\rfloor + \delta_i \right\}\right| = n - a \bmod n$$

with $\delta_i \in \mathbb{Z}$, and since all the x_i in (7) must still sum to a, the combined effect of all the δ_i on the sum found in (7) is zero

$$\sum_{i=1}^{n} \delta_i = 0 \tag{8}$$

which also implies that for any $k \in \{0 \dots n\}$

$$\sum_{i=1}^{k} \delta_i \quad = \quad - \sum_{i=k+1}^{n} \delta_i \tag{9}$$

What we need to prove is that

$$\left(\sum_{i=1}^{n} x_i^2 \right)_{\textit{arbitrary } x_i} \quad - \quad \left(\sum_{i=1}^{n} x_i^2 \right)_{\textit{proposed } x_i \textit{ for lower bound}} \quad \geq \quad 0$$

By expanding the left hand side of the above expression using the shorthand notation $c = \lceil \frac{a}{n} \rceil$ and $f = \lfloor \frac{a}{n} \rfloor$ we get

$$\begin{aligned} & \left((c+\delta_1)^2 + \dots + (c+\delta_{a \bmod n})^2 + (f+\delta_{a \bmod n \ + \ 1})^2 + \dots + (f+\delta_n)^2 \right) \\ - \quad & \left(c^2 + \dots + c^2 + f^2 + \dots + f^2 \right) \end{aligned}$$

From this we work out the squares and cancel out all c^2 and f^2 terms which results in the following expression

$$\begin{aligned} & 2c\delta_1 + \dots + 2c\delta_{a \bmod n} + 2f\delta_{a \bmod n \ + \ 1} + \dots + 2f\delta_n + \\ & \delta_1^2 + \dots + \delta_{a \bmod n}^2 + \delta_{a \bmod n \ + \ 1}^2 + \dots + \delta_n^2 \end{aligned}$$

Collecting terms into sums and moving out common factors we arrive at

$$2c \left(\sum_{i=1}^{a \bmod n} \delta_i \right) + 2f \left(\sum_{i=a \bmod n \ + \ 1}^{n} \delta_i \right) + \sum_{i=1}^{n} \delta_i^2$$

Apply (9) with $k = a \bmod n$ to get

$$2c \left(\sum_{i=1}^{a \bmod n} \delta_i \right) - 2f \left(\sum_{i=1}^{a \bmod n} \delta_i \right) + \sum_{i=1}^{n} \delta_i^2$$

When $n \mid a$ we have $f = c$ and then the above expression is definitely non-negative since the first two sums cancel each other out, and left is only the rightmost sum of squares. When $n \nmid a$ then $f = c - 1$ and by using this and cancelling terms the following expression with neither c nor f is reached

$$2 \sum_{i=1}^{a \bmod n} \delta_i + \sum_{i=1}^{n} \delta_i^2 \tag{10}$$

The second term is trivially non-negative. If we are able to show that the first term too is non-negative, then the proof is complete.

Forcing the x_i to be a non-increasing series does not contradict with anything we have done so far, so let's do that

$$x_1 \geq x_2 \geq \dots \geq x_{a \bmod n} \geq x_{a \bmod n \ + \ 1} \geq \dots \geq x_n$$

Expand each x_i *and use the same notation as previously to arrive at*

$$c + \delta_1 \geq c + \delta_2 \geq \ldots \geq c + \delta_{a \bmod n} \geq f + \delta_{a \bmod n \,+\, 1} \geq \ldots \geq f + \delta_n$$

and therefore

$$\delta_1 \geq \ldots \geq \delta_{a \bmod n}, \tag{11}$$

$$\delta_{a \bmod n \,+\, 1} \geq \ldots \geq \delta_n \tag{12}$$

Now first consider the case when

$$x_k > x_{k+1}$$

where $k = a \bmod n$*. This is equivalent to**

$$c + \delta_k > c - 1 + \delta_{k+1}$$

or simpler just

$$\delta_k - \delta_{k+1} > -1 \tag{13}$$

Assume that

$$\delta_k < \delta_{k+1} \tag{14}$$

or equivalently that

$$\delta_k - \delta_{k+1} < 0 \tag{15}$$

But (13) together with (15) results in a contradiction since $\delta_i \in \mathbb{Z}$*, so our assumption in (14) must be wrong. Hence*

$$\delta_k \geq \delta_{k+1}$$

and together with (8), (11) and (12) we know that δ_i *represents a non-increasing series that sums to zero. We can then use Lemma 10 to realize that* $\sum_{i=1}^{a \bmod n} \delta_i \geq 0$ *and hence (10) is non-negative when* $x_k > x_{k+1}$*.*

Next consider the case when

$$x_k = x_{k+1} \geq c$$

which gives us

$$x_k \geq c \quad \Leftrightarrow \quad c + \delta_k \geq c \quad \Leftrightarrow \quad \delta_k \geq 0$$

Since x_i *is a non-increasing series,* δ_i *is also a non-increasing series up to and including* $i = k$ *since for this range* $x_i = c + \delta_i$*, and hence all the* δ_i *up to and including* $i = k$ *are non-negative so we have*

$$\delta_k \geq 0 \quad \Rightarrow \quad \forall i \in \{1, \ldots, k\} : \delta_i \geq 0 \quad \Rightarrow \quad \sum_{i=1}^{k} \delta_i \geq 0$$

and we have proved that (10) is non-negative for this case as well.

*Remember that we have already taken care of the case when $n \mid a$ so $f = c - 1$.

Finally consider the case when

$$x_k = x_{k+1} \leq f$$

which gives us

$$x_{k+1} \leq f \quad \Leftrightarrow \quad f + \delta_{k+1} \leq f \quad \Leftrightarrow \quad \delta_{k+1} \leq 0$$

Symmetrically to the previous case, since x_i is a non-increasing series, δ_i is also a non-increasing series from $k+1$ and onwards since for this range $x_i = f + \delta_i$, and hence all the δ_i from $k+1$ and onwards are non-positive so we have

$$\delta_{k+1} \leq 0 \quad \Rightarrow \quad \forall i \in \{k+1, \ldots, n\} : \delta_i \leq 0 \quad \Rightarrow \quad \sum_{i=k+1}^{n} \delta_i \leq 0 \quad \Leftrightarrow \quad \sum_{i=1}^{k} \delta_i \geq 0$$

where (9) is used for the last step. Hence (10) is non-negative for this case too.

*Since we end up with $\sum_{i=1}^{a \bmod n} \delta_i \geq 0$ in all of these exhaustive cases** *we can be sure that the expression in (10) is non-negative and hence the lower bound on $\sum_{i=1}^{n} x_i^2$ has been proved.* □

Definition 12. *The* replication number *or* degree *of x, denoted $d(x)$, is the number of sets in a family $\mathcal{F}$ that contain the element x.*

Lemma 13 ([11]). *Let $\mathcal{F}$ be a family of subsets of some set X. Then*

$$\sum_{A \in \mathcal{F}} \sum_{x \in A} d(x) = \sum_{x \in X} d(x)^2$$

Theorem 14. *Let $\mathcal{F}$ be the family of r-element sets $V_1, \ldots, V_v$ and let B be their union with $b = |B|$. If*

$$|V_i \cap V_j| \leq \lambda \quad \forall i \neq j \in \{1, \ldots, v\}$$

then

$$\lambda \geq \left\lceil \frac{\left\lceil \frac{rv}{b} \right\rceil^2 (rv \bmod b) + \left\lfloor \frac{rv}{b} \right\rfloor^2 (b - rv \bmod b) - rv}{v\,(v-1)} \right\rceil$$

Proof. *$V_1, \ldots, V_v$ can be thought of as a boolean matrix with v rows and b columns with a 1 (0) in row i and column j meaning that $j \in V_i$ ($j \notin V_i$). Using this point of view $d(x)$ is the sum of the xth column.*

By counting the cardinality of all intersections between V_i and all coblocks both column- and row-wise we get

$$\begin{aligned} \forall i \in \{1, \ldots, v\} \ : \ \sum_{x \in V_i} d(x) &= \sum_{j=1}^{v} |V_i \cap V_j| \\ &= |V_i| + \sum_{j \neq i} |V_i \cap V_j| \\ &\leq r + (v-1)\,\lambda \end{aligned}$$

*The cases are exhaustive since there is no number in between the ceiling and the floor.

where the final inequality comes from the fact that no two subsets intersect over more than λ elements. Next sum over all sets V_i in $\mathcal{F}$ to get

$$\sum_{i=1}^{v} (r + (v-1)\,\lambda) \geq \sum_{i=1}^{v} \sum_{x \in V_i} d(x)$$

and then use Lemma 13 to arrive at

$$v\,(r + (v-1)\,\lambda) \geq \sum_{x \in B} d(x)^2$$

which we reorder to get an expression for λ

$$\lambda \geq \frac{\sum_{x \in B} d(x)^2 - rv}{v(v-1)} \tag{16}$$

By counting the number of ones in the matrix both column- and row-wise

$$\sum_{x \in B} d(x) = rv$$

and we can then replace the sum in (16) with the lower bound from Lemma 11, yielding the lower bound

$$\lambda \geq \left\lceil \frac{\left\lceil \frac{rv}{b} \right\rceil^2 (rv \bmod b) + \left\lfloor \frac{rv}{b} \right\rfloor^2 (b - rv \bmod b) - rv}{v\,(v-1)} \right\rceil$$

$\square$

Note that when $b \mid rv$ the bound of Theorem 14 degenerates into the bound of Corollary 8.

Furthermore, when the expression inside the ceiling above is not an integer, you can possibly get the same lower bound on λ when changing the parameters slightly. For example, this could be used to get a lower bound on the number of blocks b needed to reach a particular λ.

The expression inside the ceiling of the lower bound in Corollary 8 becomes negative when $b > rv$. For example $\langle 5, 13, 2 \rangle$ gives $\lambda \geq -\frac{3}{26}$. The corresponding expression inside the ceiling of the improved lower bound presented above gives $\lambda \geq 0$ for the same example. Further, the improved lower bound is never negative.

Corollary 15. *The lower bound of Theorem 14 is always non-negative.*

Proof. *By Lemma 7*

$$\left\lceil \frac{rv}{b} \right\rceil (rv \bmod b) + \left\lfloor \frac{rv}{b} \right\rfloor (b - rv \bmod b) = rv$$

and considering the above left expression as a summation of b terms, $(rv \bmod b)$ ceilings and $(b - rv \bmod b)$ floors, and since these terms are all non-negative, squaring them results in a larger or equal value, and we get

$$\left\lceil \frac{rv}{b} \right\rceil^2 (rv \bmod b) + \left\lfloor \frac{rv}{b} \right\rfloor^2 (b - rv \bmod b) \geq rv$$

which is equivalent to

$$\left\lceil \frac{rv}{b} \right\rceil^2 (rv \bmod b) + \left\lfloor \frac{rv}{b} \right\rfloor^2 (b - rv \bmod b) - rv \geq 0$$

Hence the numerator of Theorem 14 is non-negative, and since the denominator $v(v-1)$ is positive, the fraction, i.e. Theorem 14, is always non-negative. □

Even with this improved lower bound on λ there are (many) cases where the lower bound is not possible to reach. Consider $\langle 9, 8, 3 \rangle$, with Corollary 8 we get a lower bound $\lambda \geq \lceil 0.8906\ldots \rceil = 1$ and with Theorem 14 we get $\lambda \geq \lceil 0.9166\ldots \rceil = 1$. But $\langle 9, 8, 3 \rangle$ can only be solved with $\lambda \geq 2$. The unrounded bound of Theorem 14 is closer to reality than the one of Corollary 8, but it is still too small.

In some cases this improved bound means we can be sure no solution exists to cases which were an open question previously. For example $\langle 10, 350, 100 \rangle$ now has $\lambda \geq \lceil 21.\bar{1} \rceil = 22$ (and a solution does exist with $\lambda = 22$, see [8]) whereas with the previously used bound we only got $\lambda \geq \lceil 20.63\ldots \rceil = 21$. Hence we can be sure that no solution with $\lambda = 21$ exists, something that required a separate proof previously [9].

Evaluating the Improved Bound. Let us subtract the bound of Corollary 8 from the improved bound of Theorem 14 to compare them (both expressions without the ceiling). We use the following shorthand notations

$$c = \left\lceil \frac{rv}{b} \right\rceil \quad f = \left\lfloor \frac{rv}{b} \right\rfloor \quad x = rv \bmod b$$

We start with

$$\frac{c^2 x + f^2 (b - x) - rv}{v (v - 1)} - \frac{r (rv - b)}{b (v - 1)}$$

Transform this expression to use a common denominator and then multiply both the numerator and denominator by b to get

$$\frac{b^2 c^2 x + b^2 f^2 (b - x) - br^2 v^2}{b^2 v (v - 1)}$$

Eliminate $b^2 c^2$ and $b^2 f^2$ using (5) and (6) to get

$$\frac{(rv + (b - x))^2 x + (rv - x)^2 (b - x) - br^2 v^2}{b^2 v (v - 1)}$$

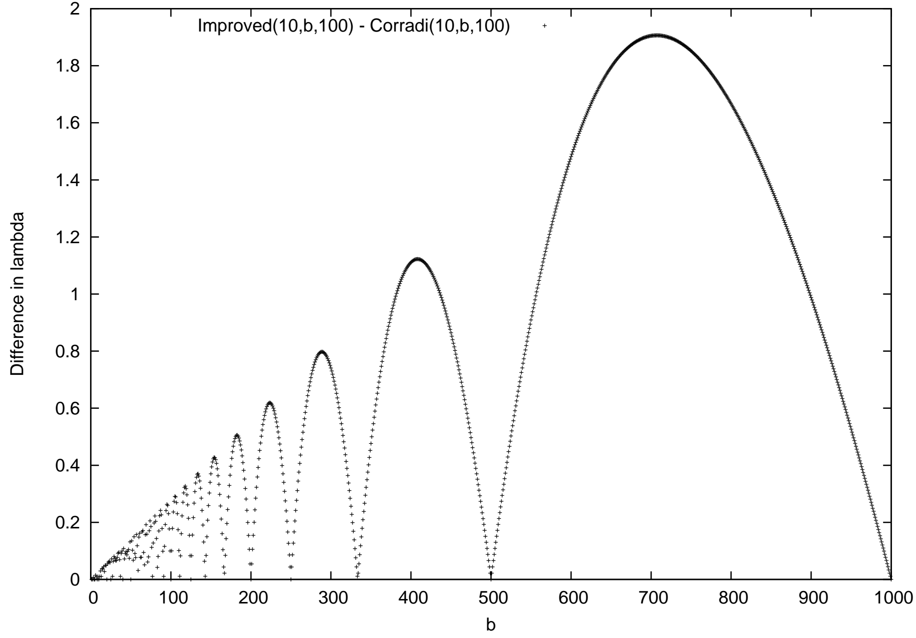

Fig. 2. Difference between improved and previously used lower bound on λ.

If we expand and then simplify this expression, a lot of terms cancel out and we are left with only

$$\frac{x\,(b-x)}{bv\,(v-1)} = \frac{(rv \bmod b)\,(b - rv \bmod b)}{bv\,(v-1)}$$

and because $0 \leq (rv \bmod b) < b$ the expression above is never negative, and zero only when $b \mid rv$. We summarize this into a proposition.

Proposition 16. *The difference between the improved lower bound on λ of Theorem 14 and the old lower bound of Corollary 8 is*

$$\frac{(rv \bmod b)\,(b - rv \bmod b)}{bv\,(v-1)} \geq 0$$

It is interesting to look at the difference between the improved and the previously used lower bound on λ. Figure 2 shows how the difference between the two, both without the ceiling, as the number of credits b in $\langle 10, b, 100, \lambda \rangle$ is varied. It can be seen that they are equal when $b \mid rv$ and large between any two such points, and the difference increases when b is increased until $b = 1000$, from where λ becomes zero (and both lower bounds are $\lambda \geq 0$).

As we can see the difference between the improved and the previously used lower bound on λ, when both are unrounded, is for some values of v, b, r greater

than one. It is therefore possible that the lower bound on λ of Corollary 8 in some cases were *two* steps from the lower bound of Theorem 14 when rounding is taking into consideration.

4 Implementation

Section 3 presented the purely theoretical result of this work, now let's turn to the problem of how to use constraint programming to construct a constraint program that finds POs.

4.1 Basic Model

Just as in [8] we transform the problem from a constraint optimization problem to a regular CSP. We first try to solve the problem using the lower bound on λ, and if this is not possible we increase λ one step at a time until we find a solution. We of course use the improved lower bound on λ from Theorem 14 instead of Corollary 8 which was used in [8].

To construct the CSP for the PO problem we model the problem using a boolean matrix with b columns and v rows. On this boolean matrix we post the constraint that the sum of any row must equal r. The pairwise overlap between any two rows must not be bigger than λ but this constraint is not added until during search. This is done by posting `scalar_product` constraints after a row has been completely labeled. On the subject of exactly how to do this, two different techniques were tried. The first posts the `scalar_product` constraints between the newly labeled row and all not yet labeled rows. The second technique delays posting the constraints until the last possible moment. It does this by posting the `scalar_product` constraints involving a particular row and all previously labeled rows just before labeling of that row begins.

Using `scalar_product` constraints is different from the model used in [8] which modeled the cardinality constraint on row intersections by using reified conjunction constraints between pairs of elements from the same positions on each row, and then limiting the sum of the reified variables.

Table 1 shows the difference in runtime performance between the three different techniques for a few portfolio optimization problems. We can see that using `scalar_product` constraints roughly halves the runtime and that delaying constraint-posting makes no difference.

Regarding why posting `scalar_product` constraints is faster than using reified conjunction constraints, it is probably because this method leads to a lot fewer constraints and hence less overhead in CLPFD of SICStus Prolog. Posting the constraints at the latest possible moment makes little difference, probably because propagation for a constraint is not run until the domain of a variable involved changes.

Problem				Results			
v	b	r	λ	#btks	time_{old} [s]	$\text{time}_{\text{scalar_product}}$ [s]	$\text{time}_{\text{scalar_product_latest}}$ [s]
9	37	12	3	746750	39.58	24.56	24.42
10	15	6	2	96822	14.12	4.17	4.33
10	25	8	2	2492	0.22	0.10	0.09
10	37	14	6	6932	1.28	0.62	0.63
10	38	10	2	85238	10.00	5.07	4.95

Table 1. Performance comparison of stating the pairwise intersection in different ways for some POs.

4.2 Labeling Order and Static Symmetry-Breaking

During search for a solution the solver might reach a state where it will need to backtrack and reconsider the choices it has done previously for where in the matrix there should be a one and where a zero. When it then makes some other choices we don't want them to be symmetric to the case the solver just backtracked out of, because that will only lead to backtracking again. To prevent this from happening we could break all symmetry, but since this problem has $v!b!$ symmetries this is not feasible for other than the most trivial problem.

By posting lexicographical ordering constraints on the rows and the columns,* often known as Lex^2 and described in [7], we can in a computationally cheap way break some, but not all, symmetry. This technique is often applied, and was used in [8]. We will look at a more elaborate technique for breaking symmetry in Sect. 6.

Since the problem has constraints on rows and on the cardinality of row intersections we do labeling row by row to allow the constraints to efficiently prune the search space. To allow the lexicographical ordering constraint between rows to work efficiently we should do the labeling of rows from left to right. Trying 1 before 0 is more efficient than the other way around, because it allows partially labeled rows to be rejected earlier based on too much overlap with the other rows.

With this labeling strategy it is more efficient to change from lexicographic ordering of the columns to anti-lexicographic ordering of the columns to not work against the labeling strategy, i.e. flipping the inequality from $\leq_{lex}$ to $\geq_{lex}$ between rows and columns. This implies that we must order the columns too using anti-lexicographic ordering to not lose solutions [7].

Even though not described in the same detail previously, everything described so far in this section was done in [8] as well.

In [8] strict lexicographical ordering, $>_{lex}$, was used for rows since repeating rows implies that $\lambda = r$. A solution to such a problem would have the leftmost r columns all ones, and the remaining columns all zero. Problems with $\lambda = r$ might not seem very interesting, but they can occur when solving POs using

*$\text{row}_1 \leq_{lex} \text{row}_2 \leq_{lex} \ldots \leq_{lex} \text{row}_v$ and similar for columns.

BIBD					Result			
v	b	r	k	λ	#btks$_{\text{bibd-solver}}$	time$_{\text{bibd-solver}}$ [s]	#btks$_{\text{po-solver}}$	time$_{\text{po-solver}}$ [s]
7	7	3	3	1	0	0.00	2	0.00
8	14	7	4	3	57	0.01	209	0.02
9	18	8	4	3	95	0.01	168384	6.04
9	24	8	3	2	9	0.01	51487	1.99
10	30	9	3	2	23	0.02	?	>3600

Table 2. Difference in runtime performance between solving a BIBD using the BIBD- and PO-solvers showing that detecting BIBDs is advantageous.

the technique described in Sect. 5. Therefore we allow them by using $\geq_{lex}$ for both rows and columns, not only columns.

4.3 BIBD Detection and BIBD Model

We said in Sect. 3 that some POs are actually BIBDs. BIBDs are easier to solve than POs because there are constraints on the columns, and also the cardinality constraint between different rows is an equality instead of an upper bound.

Because of this the implementation will test if problem instances could be BIBDs prior to starting the search for a solution. The same technique was applied by the authors of [8] although they did the check manually. The implementation presented by this work does this automatically by checking for admissibility, that Fisher's inequality holds, and that the first part, i.e. v even, of the Bruck-Ryser-Chowla theorem holds.* If all of these are satisfied the implementation will try to solve the problem as a BIBD instead of a PO, and only if that fails will it revert back to try to solve the problem as a PO.**

The constraint programming model we use for BIBDs is exactly the same matrix model that is used for POs, except we add the constraints on the column sum and that the overlap between any two rows is exactly λ. We also use the same labeling strategy, even though in some cases labeling a row, followed by a column, and so on, could be more beneficial as shown in [7].

The checks that needs to be done to detect possible BIBDs are more or less free in terms of runtime performance, but the end result when the instance is a BIBD is a substantial performance increase, as can be seen in Table 2. For $\langle 10, 30, 9, 3, 2\rangle$ the PO-solver failed to find a solution within one hour and was cancelled, hence the question-mark in the table.

---

*The second part, i.e. v odd, of the Bruck-Ryser-Chowla theorem was considered too difficult to implement.

**If, instead of failing, we reach a timeout when trying to solve the problem as a BIBD, we assume it will timeout should we try to solve it is a PO too.

4.4 Checking for Known Non-Existing BIBDs

As mentioned above, whenever the implementation finds parameters that satisfy the properties for admissibility, and some more properties of BIBDs, it will try to solve the problem as a BIBD.

As we saw in Sect. 3.2 admissibility or the other properties of BIBDs presented in that section does not guarantee that a BIBD exists. Therefore, before we start searching for a BIBD, the implementation will check the parameters against a list of parameters known to not result in BIBDs even though the parameters satisfy admissibility and other properties. We use a subset of the list presented in [4] and [6], with parameters in the range we are interested in.

When the implementation finds parameters that can't possibly result in a BIBD it will direcly try to solve the problem as a PO instead. It is important to remember that it may still be possible to solve the problem as a PO.*

The cases when we come across admissible parameters that are known to not result in a BIBD are rare. But it is at the same time a very computationally cheap check to do that will avoid an almost certain timeout, or at least a big search when we are able to exhaust all possibilities, when we actually do come across these cases.

4.5 First Intersection and First Column Optimizations

Proposition 17. *Any solution to $\langle v, b, r, \lambda \rangle$ where the maximum intersection-cardinality is less than λ can be turned into a solution with a maximum coblock intersection-cardinality of λ.*

Proof. *Consider a solution to $\langle v, b, r, \lambda \rangle$ where the maximum cardinality of all intersections is less than λ. Reorder the coblocks to have the two coblocks in the intersection with the maximum cardinality as coblock V_v and V_{v-1}.*

By replacing an element in the last coblock V_v, but not the second last coblock V_{v-1}, with an element in V_{v-1}, the cardinality of the intersection between these two coblocks is increased by one.

Since no other intersection will have its cardinality increased by more than one when we do this, and since we can repeat this procedure any number of times until we reach $\lambda = r$ we have proved Proposition 17. □

To find a solution to $\langle v, b, r, \lambda \rangle$ at least one intersection can, due to Propostion 17, have cardinality λ. Because of the symmetry involved we can reorder any solution so that this is the first intersection.

Consider how the solver chooses the first two rows initially. For the first row, the solver will first try with r ones to the left and the remaining elements all zero. Continuing with the second row the solver will try with λ ones to the left resulting in an intersection of cardinality λ between the first two rows, then $r - \lambda$ zeros, followed by the remaining $r - \lambda$ ones and then the final elements of the row are all zero. This is the most common case, the other case is when

*In reality, whether this is possible or not is an open question.

$\lambda > r$,* and then the solver will for the second row instead try with r ones, followed by all zeros, giving an intersection of cardinality r between the first two rows.

If no solution exists with this choice of the first two rows, or equivalently the first intersection, then the solver should backtrack and first choose another second row and if all choices for the second row fail too, backtrack and choose another first row. But any other choice of the first two rows that has an intersection of cardinality λ can always be transformed into the choice described above by permuting the columns.** Hence using another choice for the elements of the first intersection will only lead to testing rows that are symmetric to rows that have already been tested unsuccessfully. Thus, the choice of the first two rows in the incidence matrix need never be reconsidered by the solver.

Only when there is no solution, or when we enumerate all solutions, will this method give any performance gain, since only then will the solver backtrack all the way to the choice of the first and second row. With this method fewer rows will need to be tried before the solver can be sure no solution exists. The number of available symmetries to a fully labeled matrix is decreased from $v!b!$ to $(v-2)!b!$, that is by a factor $v(v-1)$.

The technique described above applies to both POs and BIBDs. But for BIBDs we can do more. The constant column sum of BIBDs and the anti-lexicographic ordering of rows implies that the first column must be k ones at the top and the remaining elements all zero. If this wasn't the case then the anti-lexicographic constraint would not be satisfied. We can't fix the second column though as we don't have any constraint between pairwise columns as we had with rows.

By fixing the first column for BIBDs we get a small performance improvement in some cases and not only when there is no solution, or when we enumerate all solutions. Table 3*** shows the runtime performance impact for some BIBDs.

We can see that using this optimization leads to some small performance improvements. For example for $\langle 15, 21, 7, 5, 2\rangle$ the number of backtracks is decreased slightly. A decrease is to be expected as this instance is not a BIBD even though it satisfies basic necessary conditions for BIBDs, and fixing elements of the matrix limits the backtracking possible.

Fixing the first column and first row (but not the second row) has been done previously in [7], but there it was only evaluated as a technique for breaking symmetry in BIBDs and was not tested *together* with other symmetry breaking techniques. This led the authors of [7] to dismiss this technique.

---

*This case is included for completeness to be able to correctly solve instances no matter how far λ is increased.

**Remember that at least one intersection will have cardinality λ and we make this the first intersection as described above so this is no limitation.

***We don't use the checking for known non-existing BIBDs here as presented in Sect. 4.4 to allow the search to exhaust all possibilities of $\langle 15, 21, 7, 5, 2\rangle$ and see what impact it has on performance.

Problem					Plain		Fix row 1,2, col 1	
v	b	r	k	λ	#btks	time [s]	#btks	time [s]
8	14	7	4	3	57	0.01	57	0.00
9	18	8	4	3	96	0.01	95	0.01
9	24	8	3	2	9	0.01	9	0.01
10	30	9	3	2	23	0.01	23	0.02
15	21	7	5	2	111421	12.63	111400	12.68
19	19	9	9	4	12294	1.58	11922	1.38
25	25	9	9	3	46105	7.16	46015	6.98

Table 3. Performance of finding the first solution to different BIBDs using both plain solving and solving when first intersection and first column fixed.

4.6 Checking Feasibility During Search

The anti-lexicographic ordering constraints on rows and columns are not only an efficient technique to break relatively much symmetry, it also allows us to say more about how a solution must be formed and use this knowledge to construct more efficient models. Section 4.5 showed one application, now we will look at another, beginning with an example.

Example 18. Consider the partially labeled matrix for the problem $\langle 10, 15, 6, 2\rangle$ in Fig. 3. After labeling the four first rows, the solver has reached a state where the first two columns can no longer be used while labeling the remaining rows since rows are ordered anti-lexicographically. Hence, any row must consist of elements only in the singled out lower right part of the incidence matrix. But for this to work this part of the matrix must be *a PO on its own* and must therefore satisfy Theorem 14, with the same λ and r, but b and v decreased accordingly. In this case it does not satisfy it as we get $\lambda \geq \lceil \frac{16}{7} \rceil = 3$ for $v = 7, b = 13, r = 6$, but we are trying to solve the problem with $\lambda \leq 2$.

The above technique naturally works just as well for any other PO or BIBD. After labeling a row the solver looks in what column the last labeled row has its first element, any column to the left of this one is then unusable. For the resulting subproblem with the newly labeled row as the first row, and possibly fewer columns, the value of Theorem 14 is compared against the value we want the global problem to have. Too big and we can infer that this last labeled row will never lead to a solution and let the solver directly choose another one.

There are two special cases to consider, when the last labeled row has no first element, i.e. the last labeled row is completely empty, then we can't do anything. This will only happen when r equals zero, and hence the entire matrix will be empty.* Another special case is when there is only one row left, then we don't check anything either, since the concept of intersection with all other rows becomes meaningless.

*POs with r = 0 sometimes occur when using the technique found in Sect. 5.

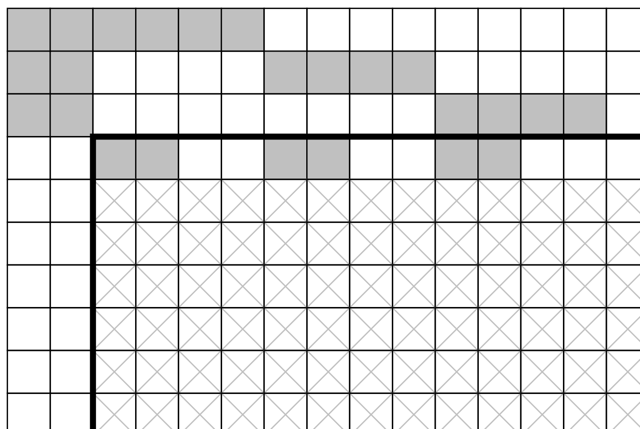

Fig. 3. Incidence matrix for the partially labeled problem instance $\langle 10, 15, 6, 2\rangle$. Grey elements represent a one, white elements a zero, and a crossed out element means the element has not yet been labeled. The bold rectangle represents a sub-problem that should satisfy Theorem 14, but since it does not, backtracking will occur.

Problem				Without check		Check during search	
v	b	r	λ	#btks	time [s]	#btks	time [s]
10	15	6	2	96822	4.330	138	0.010
11	11	5	2	764	0.050	45	0.010
15	15	4	1	3167791	279.97	2335130	228.99
16	8	3	1	729	0.080	11	0.000

Table 4. Results comparing running the program that solves POs with and without the check for feasibility of the current partial solution during search.

Last but not least, Table 4 shows the performance improvements we can get with this technique for some different POs. As we can see we can get substantial runtime performance increases. Note that the problems in Table 4 have been picked because for them this technique is useful; many other POs are not constrained enough for the check done during search to fail, and hence for them this is only a (small) runtime overhead.

4.7 Solving Using the Complement

When the implementation has found the incidence matrix representing a PO or BIBD we have actually found the solution to another problem as well, the complement. To find the complement to a PO or BIBD represented as an incidence matrix just switch every 1 to a 0 and vice versa.

For the complement the dimensions of the incidence matrix is of course not changed, and the row sum is naturally replaced by $b - r$, and in the same way

Fig. 4. Figure used to show that the resulting parameters of the complement to an arbitrary PO are $\langle v, b, b-r, b-2r+\lambda\rangle$.

Problem				Solving problem		Complement				Solving complement	
v	b	r	λ	#btks	time [s]	v	b	r	λ	#btks	time [s]
10	38	28	20	1360590	91.24	10	38	10	2	85238	5.01
10	31	22	15	992711	52.09	10	31	9	15	4044363	152.22
10	15	6	2	3251	0.16	10	15	9	5	138	0.02

Table 5. Benchmarks showing the impact of solving the complement instead of the problem directly.

the column sum of BIBDs are replaced by $v-k$. To realize what the cardinality of the intersection between pairwise rows, λ, is in the complement look at Fig. 4. λ for the complement is the width of the last rectangle, containing only non-filled elements. The width of this rectangle is the total width b minus the width of each of the other rectangles, hence it is

$$b - \lambda - (r - \lambda) - (r - \lambda) = b - 2r + \lambda$$

To summarize, a solution to the PO $\langle v, b, r, \lambda\rangle$ (or the BIBD $\langle v, b, r, k, \lambda\rangle$) also yields a solution to the complement $\langle v, b, b-r, b-2r+\lambda\rangle$ ($\langle v, b, b-r, v-k, b-2r+\lambda\rangle$).

Table 5 shows that the performance impact can be substantial for some problems. Problems are listed in such a way that the complement has a smaller row sum, r, than the original problem.

Unfortunately no technique for knowing when to solve the complement instead of the problem directly has been found. Therefore the complement is not tried by the implementation, but will need to be tried manually.

We will see that the complement can be used to increase performance in other occasions as well in the next section.

4.8 Caching Results and Extended Lookup

For each problem the implementation tries to solve, it will result in either a successfully found solution, a timeout prior to finding a solution, or an exhaustion of the search space, i.e. this problem has no solution.

Each of these results is saved, including the incidence matrix, so that later on, should the same problem resurface, the implementation will not need to tackle it again, but instead can just look up the result.

But this technique can be extended to increase the performance of the implementation by looking for harder problems that have already been solved with success and whose solutions can be adapted to be solutions to this problem.

First, if we are to solve the PO $\langle v, b, r, \lambda \rangle$ and it is not available in the cache, but $\langle v, b', r, \lambda \rangle$ is available as a successfully found solution with $b' < b$ then the solution to $\langle v, b', r, \lambda \rangle$ is also a solution to $\langle v, b, r, \lambda \rangle$. We take the first b' columns from $\langle v, b', r, \lambda \rangle$ and the remaining $b - b'$ columns are set to all zeros.

In the same way, if we are to solve $\langle v, b, r, \lambda \rangle$ and it is not available in the cache, but $\langle v', b, r, \lambda \rangle$ is available with a successfully found solution with $v' > v$ then the solution to $\langle v', b, r, \lambda \rangle$ is also a solution to $\langle v, b, r, \lambda \rangle$. We take the first v rows from $\langle v', b, r, \lambda \rangle$ and the remaining $v' - v$ rows are ignored.

We can also do the converse, i.e. look for easier problems that the program has already tried to solve and where the result was not a success. Before we present how we do this we need to state two assumptions. Firstly, we assume that more columns always make a problem easier to solve, so that if a problem with more columns timed out, so will a problem with fewer columns. Secondly, we assume that fewer rows always make a problem easier to solve, so that if a problem with fewer rows timed out, so will the problem with more rows.

If we are to solve $\langle v, b, r, \lambda \rangle$ and it is not available in the cache, but $\langle v, b', r, \lambda \rangle$ is available with a cached non-success,* with $b' > b$ then the implementation will not be able to find a solution to $\langle v, b, r, \lambda \rangle$ either. In the same way, if we are to solve $\langle v, b, r, \lambda \rangle$ and it is not available in the cache, but $\langle v', b, r, \lambda \rangle$ is available with a cached non-success with $v' < v$ then the program will not be able to find a solution to $\langle v, b, r, \lambda \rangle$ either.

In the implementation the above technique is called *extended cache lookup*. When solving larger problems by embedding smaller instances (see Sect. 5), it is common for the same problems to appear multiple times, and also for problems of roughly the same size to appear many times, so this technique has a large impact on the performance of the implementation

The caching described in this section is implemented separately from the checking for known non-existing BIBDs described in Sect. 4.4 even though nothing would prevent us from merging the two. As it is implemented parameters will first be checked against the cache, and then, if not found in the cache, against the set of parameters that are known to not result in a BIBD, and this will then result in an entry in the cache.

Finally, we can make use of the complement presented in Sect. 4.7 again. When the extended cache lookup can't find a solution to our problem an extended cache lookup is done for the complement instead, and only if this fails too, does the program need to search for a solution.

*Timeout was reached when trying to solve $\langle v, b', r, \lambda \rangle$ or no solution exists to this problem.

m	$\langle v, b_1, r_1, \lambda_1\rangle$	$\langle v, b_2, r_2, \lambda_2\rangle$	$m \cdot \lambda_1 + \lambda_2$	exists?
10	$\langle 10, 32, 09, 2\rangle$	$\langle 10, 30, 10, 3\rangle$	23	$\surd$
11	$\langle 10, 31, 09, 2\rangle$	$\langle 10, 09, 01, 1\rangle$	23	$\surd$
11	$\langle 10, 30, 09, 2\rangle$	$\langle 10, 20, 01, 0\rangle$	22	$\surd$

Table 6. Embedding candidates found for $\langle 10, 350, 100, 23\rangle$.

5 Approximately Solving by Embedding Subinstances

With the implementation working as presented in the previous section we can solve POs. Unfortunately though, solving real-life instances will take many, many hours, perhaps even days or more! The sizes of the real-life instances are so big that the number of combinations and symmetries explode (remember it's $v!b!$).

All is not lost though as we can try to find smaller instances that, when their solutions are put together, solve the original big instance, though not necessarily optimally. This is exactly what was done in [8] and since this technique hasn't been improved in any way apart from now being an integral part of the program, the presentation in this section is kept brief.

The technique is known as embedding of subinstances, and to find the subinstances we use constraint programming as well. To keep things easy we choose to have as few different embedded instances as possible per problem. But since not all problems can be neatly divided into m equally sized parts, we use two embedded instances for every problem, one quotient-instance and one remainder-instance.

Before we present the CSP that we use to model and solve this problem, let's look at an example showing what candidates for embedding subinstances we would expect to get in this case.

Example 19. Consider $\langle 10, 350, 100, 23\rangle$. To solve this problem we will look for embedding candidates consisting of two possible POs $\langle 10, b, r, \lambda\rangle$ and $\langle 10, b', r', \lambda'\rangle$, and a factor m, such that the incidence matrix of the solution to $\langle 10, b, r, \lambda\rangle$ repeated horizontally m times followed by the incidence matrix of the solution to $\langle 10, b', r', \lambda'\rangle$ equals a feasible, but not necessarily optimal, solution to the original problem. If we choose to try and construct such embeddings where b and b' are both less than 36[*] we end up with the embedding candidates of Table 6. This is the same example as occurs in [8] but since we use a tighter lower bound on λ we don't get as many embedding candidates, but the probability of the subinstances being solvable increases, and in this case they are all solvable. In this case we are also able to find a solution with $\lambda \leq 22$.

[*] Choose a too big value here and the subinstances will still be too difficult to solve directly.

The CSP used to find embeddings $m \cdot \langle v, b_1, r_1, \lambda_1 \rangle + \langle v, b_2, r_2, \lambda_2 \rangle$ of a problem $\langle v, b, r, \lambda \rangle$ is shown in the equations below.

$$b = mb_1 + b_2 \tag{17}$$
$$r = mr_1 + r_2 \tag{18}$$
$$\lambda \leq m\lambda_1 + \lambda 2 \tag{19}$$
$$1 \leq b_i \leq t \quad \text{for } i = 1, 2 \tag{20}$$
$$0 \leq r_i \leq b_i \quad \text{for } i = 1, 2 \tag{21}$$
$$\lambda_i = \text{Thm.13}\,(v, b_i, r_i) \quad \text{for } i = 1, 2 \tag{22}$$

First, t is a threshold on how large subinstances are allowed to be, and is specified when calling the predicate that computes them. It should be chosen such that subinstances are solvable in a reasonable amount of time. Typical values are between 20 and 40 when v is between 20 and 10 respectively.

Second, (19) warrants an explanation. Since POs have intersections where the cardinality is *less* or equal to a λ, we could in some rare cases reorder the rows in an incidence matrix representing a solution such that the intersection cardinalities of the total solution is decreased. Any attempt to detect and do this is not done by the implementation though.

Let's also note one implementation detail. Since we use constraint programming over finite domains (finite integers), we have to use some "trick" to compute the floor and ceiling of fractions that is found in the expression for Theorem 14. Actually, the floor is easy, as division just discards the remainder. For the ceiling we use a reified constraint to represent if the division works out with or without a remainder as shown below.

$$Mod = Numerator \text{ mod } Denominator$$
$$Mod > 0 \Leftrightarrow Carry$$
$$Ceiling = Numerator / Denominator + Carry$$

As a final point, note that we might not reach an optimal solution by solving embeddings. In Example 19 the optimal value for λ is 22 by Theorem 14 and we also managed to reach this with one of the embeddings. But in other cases this is not possible, especially when the threshold on the number of columns is lower. It is a balance between being able to solve the problem quickly and being able to reach an optimal solution.

Further details on embeddings can be found in [8].

6 Symmetry-Breaking During Search

Up until now we have broken symmetry using (anti-)lexicographical ordering constraints on rows and columns, and also by fixing the first intersection (and column, in the case of BIBDs). These techniques post their constraints before labeling begins.

Fig. 5. Partially labeled incidence matrix of total size 4×5. Grey elements represent a one, white elements a zero, and a crossed out element means the element has not yet been labeled.

Now we will present a technique for doing symmetry-breaking during labeling/search by posting additional symmetry-breaking constraints at suitable moments during the search. The aim is of course to break more symmetry than before and doing this in a way that results in a net decrease in the required runtime to find a solution.

While the techniques used so far do not break enough symmetry, and the program is sometimes exploring symmetries to already failed search trees, breaking all symmetries is not feasible due to the immense number of symmetries of the incidence matrix for typical problem-sizes.

What we present in this section is closely related to STAB, presented in [12] and [13], and originated from it. But it also differs from STAB on some key issues that we will highlight.

6.1 Overview

The aim of both STAB, as presented in [12] and [13], and what we present in this section, is to avoid exploring choices that are symmetric to other already explored choices. To accomplish this both techniques break (almost) *all* symmetry that leave the current partial assignment of the incidence matrix unchanged. The set of all such symmetries is known as the *stabilizer* subgroup to the symmetries of the partial assignment, and since we need to compute this set to post the constraints that break all symmetry, the technique is referred to as STAB. Such symmetry-breaking constraints are constructed and posted after the labeling of every row.

Example 20. Consider the partially labeled incidence matrix in Fig. 5. We want to find all row and column orderings that leave the matrix unchanged. These are symmetries we can then break (except of course the identity symmetry). This example is continued later in the text.

To find all the row and column symmetries that keep the current assignment unchanged we construct and solve an additional CSP. All possible solutions to this CSP are used to post new symmetry-breaking constraints. Then the search for a solution to the original CSP is continued until we reach a partial assignment where we choose to repeat this procedure.

To model this problem into a CSP we use one list of variables that represents the permuted column numbers, `columns` $= [c_1, \ldots, c_b]$, and one list that represents the permuted row numbers, `rows` $= [r_1, \ldots, r_v]$. On these variables we post the following constraints

- `all_different(rows)`
- `all_different(columns)`
- If and only if the element at the intersection between row i and column j in the partial assignment incidence matrix is equal to one, is the element at the intersection between row r_i and column c_j equal to one.

This last item is keeping the relationship between rows and columns intact. The next section presents how this constraint is implemented.

6.2 Modeling the Row-Column Relationship

To store the partial assignment incidence matrix, henceforth denoted as A, we use two lists. One, `ColToRow`, that for each column lists the rows where the intersection between a chosen column and every row is one, and the other one, `RowToCol`, lists columns where the intersection between a chosen row and every column is one.

Example 21. Continuing our example, the lists describing A are

- `ColToRow` $= [[1], [1], [2], [2], []]$
- `RowToCol` $= [[1, 2], [3, 4]]$

We want to find all orderings of columns and rows that don't change the matrix A. If there exists such a reordering that moves column i to c_i and row j to r_j then $A(r_j, c_i) = A(j, i)$, otherwise A would be changed by this reordering.

This means all rows that had a certain element in column i also must have the same element in column c_i, but where the elements are reordered according to the reordering of the rows. The above argument is valid when we interchange rows and columns too.

Since we have only two elements in the matrix A, zero and one, we can restrict ourselves to look only on those elements that are one. If we make sure that all of the elements that are one are put into the correct position, then all the remaining elements must be zero.

More formally we can describe this as

$$\forall \text{ rows } i : \forall \text{ columns } j : j \in \texttt{RowToCol}(i) \iff c_j \in \texttt{RowToCol}(r_i) \tag{23}$$

and the above equation is of course also true when row and column are interchanged and at the same time `RowToCol` is exchanged for `ColToRow`, and these are the formulations used for implementing this with CLPFD in SICStus Prolog.

Because a change to the domain of a row only can cause changes to the domains of columns and vice versa as noted above, the constraint is divided into two parts.

More technically, this constraint is implemented using the basic constraint predicate `relation/3` of CLPFD in SICStus Prolog. We do this by letting this constraint take a domain variable representing a permuted column number, a mapping from column numbers to row numbers, and a permuted row number, all representing one position in the incidence matrix where there is a one. This constraint is then true if there is a mapping from the permuted column number to the permuted row. Of course, we do exactly the same thing but with rows and columns exchanged.

6.3 Aggregating Identical Columns

Identical columns in the partially labeled incidence matrix will, because of the anti-lexicographical ordering of columns, always be adjacent to each other. Since they are all equal they can be freely permuted without changing the partial assignment matrix.

Therefore we leave them all out, except for one column used to represent them all, when constructing the permutations.

When we then post the symmetry-breaking constraints we could from each generated permutation expand it with all permutations possible of the identical columns, and to break *all* symmetry that leaves the partial assignment unchanged we must do this.

But in reality the number of possibilities becomes too large if we do this and therefore we post the constraints as is from the computation instead.

6.4 Same-Cardinality Optimization

A simple observation allows us to add an implied constraint that improves performance when solving the CSP that generates all symmetries that keep the partial assignment unchanged. The observation is that a column (row) can only be permuted into another column (row) with the same number of ones or it will not be a symmetry to the current partial assignment incidence matrix.

This is implemented by limiting the domain of the possible column-positions a column could be moved to, so that it only contains positions with the same cardinality.

6.5 Applying the Symmetry-Breaking

Once we have computed the permutations we should use them to break symmetries of the next row that leave the already labeled rows unchanged. Hence we should force an ordering of the next row such that we don't explore such a symmetry. We do this by posting constraints stating that the next row is lexicographically larger than all of its generated column-permutations.

Note that we actually don't need to know the row-permutations here, since permutations of rows will have no effect on the choice of the next row to be labeled.

6.6 Comparison to STAB

As mentioned previously the technique presented above was inspired by the work on the symmetry-breaking technique STAB. We will now present a list with differences between the two.

- We differ between rows and columns in a way not done with STAB since a column cannot be permuted into a row and vice versa.
- We divide the problem into two parts, where one deals with making sure mappings to rows are correct, and the other makes sure mappings to columns are correct.
- Our technique does not reason in terms of graph theory, but in terms of the incidence matrix.
- We only post constraints making sure the ones in the matrix are put in the correct position, the zeros are then implicitly put in the remaining positions and that is correct. Hence there is no need to state constraints for both the ones and the zeros as STAB does.

6.7 Alternative Approach

Before this technique was implemented using `relation/3` it was implemented as a new global constraint in CLPFD of SICStus Prolog in a way almost identically to what is described in [12] and [13]. But the performance of this implementation was very poor. This can probably be contributed to the author's limited experience with writing global constraints.

6.8 Benchmarks

Table 7 shows the result of solving some POs and BIBDs using this technique compared to doing static symmetry-breaking.

We can see that it is faster in some cases and slower in others. This is not too surprising actually since STAB as presented in [12] never offered any big improvement in the case when only one solution is computed. When all solutions are computed it is claimed in [12] that STAB is more than one order of magnitude faster than other techniques, such as doing just Lex^2.

What is not mentioned in [12], nor in [13], is that a big part of the benchmarks upon which this statement rests are just multiples of much smaller BIBDs, especially of the BIBDs $\langle 7, 7, 3, 3, 1\rangle$ and $\langle 6, 10, 5, 3, 2\rangle$.* Since the STAB-solver has support for aggregating identical columns, but the Lex^2-solver does not even though you could easily factor out a common factor m in $\langle v, mb, mr, k, m\lambda\rangle$ this will unfairly skew the results in favor of the STAB-solver. If we filter out all the multiples of smaller instances found in [12] then the geometric mean of the speedup will surely decrease significantly.

** Table 2. Results when computing all solutions* in [12]

Problem					Static		Dynamic	
v	b	r	k	λ	#btks	time [s]	#btks	time [s]
10	15	6	-	2	138	0.01	102	0.15
10	25	8	-	2	4170	0.14	2248	0.33
10	33	13	-	5	23254	1.28	7383	0.60
10	38	10	-	2	85238	5.59	13968	1.50
12	23	13	-	7	47726	2.28	43439	2.37
7	7	3	3	1	0	0.01	0	0.02
7	63	27	3	9	0	0.01	0	0.05
9	18	8	4	3	95	0.01	65	0.12
10	15	6	4	2	44	0.01	65	0.13
11	11	5	5	2	15	0.00	8	0.21

Table 7. Comparison of static and dynamic symmetry-breaking for some POs and BIBDs.

Also, it is questionable if the BIBDs in these benchmarks do as much backtracking of whole rows as many other BIBDs would need. This could be significant since in [12] the stabilizer subgroup is not computed until after a row has been backtracked for the first time.

Let's get back to the implementation presented here. The memory usage of this implementation is quite large and this limits its usefulness, especially since the memory-handling of SICStus Prolog (version 3.12.2) is implemented in such a way that no more than 256MB of memory can be used on 32-bit architectures. We hit this limit for fairly small problems, instances bigger than those presented in Table 7 will exit with memory exhaustion because of this.

Therefore, and since we haven't seen a performance improvement across all problems but instead faster for some instances and slower for others, the dynamic symmetry breaking is not used in the implementation.

7 Conclusion

In the credit derivatives market the problem arises of how to construct synthetic collateralized debt obligations (CDOs) squared in an optimal way such that ultimately sales can be maximized. This problem is known as the portfolio optimization (PO) problem. We have presented the techniques used in the construction of a constraint-based solver for this problem that explores the close relationship with the balanced incomplete block design (BIBD) problem. Due to the large size of typical PO instances, global solving was not possible, instead we embedded and solved sub-instances. We proved a new tight lower bound on the cost that was used to assess the high quality of our approximate solutions. By using detection of BIBDs, symmetry breaking, extended reuse of already solved instances, and existence-checking during search, the performance of the solver becomes good enough for constructing possibly optimal portfolios of CDOs squared, with sizes common in the credit derivatives market,

PO				Solving PO		BIBD alt.					Solving BIBD	
v	b	r	λ	#btks	time [s]	v	b	r	k	λ	#btks	time [s]
10	33	15	6	?	>3600	11	33	15	5	6	29552	2.79
19	20	9	4	?	>3600	19	19	9	9	4	11922	1.29

Table 8. A lot of performance can be won by instead of trying to solve the original PO you solve an alternative BIBD whose solution is a solution to the original PO.

within minutes or seconds. Finally we presented a method of doing dynamic symmetry-breaking that could be explored further.

Limitations With no constraint on the column sum for POs solving problems with the number of rows v bigger than around 15 is often hard. Since we in practice use only Lex^2, symmetry-breaking is not as extensive as it could be.

Related Work. The financial portfolio optimization problem has very recently been solved with great success using local search as presented in [1]. Instances that can only be solved by embeddings when using the global search implementation presented in this report, can be solved directly when using the local search techniques in [1].

Future Work. Proving or disproving that the column sums can always be constrained to be in the range $\lfloor \frac{rv}{b} \rfloor \leq k \leq \lceil \frac{rv}{b} \rceil$ would be interesting. Also, generalizing the notion of embeddings to any linear combination of several sub-instances could perhaps prove beneficial in some cases.

The mixed results received from doing dynamic symmetry-breaking have raised many more questions than it answered, and it is definitely an area where more work should be spent.

Another interesting thing would be to try and change the parameters of a problem such that it can possibly be solved as a BIBD and still be a solution to the original PO. This would in a way be a continuation of the work done in Sect. 4.8. The argument for why the BIBD alternatives are solutions to the original POs is the same as presented there. As can be seen from Table 8, the runtime improvement is huge, from not being able to solve the problem in an hour to solving the problem in just a few seconds.

Acknowledgments. The author wishes to thank Pierre Flener and Justin Pearson of the ASTRA Research Group at Uppsala University for all their inspiring feedback and suggestions.

References

1. M. Ågren, P. Flener, and J. Pearson. Incremental algorithms for local search from existential second-order logic. In P. van Beek, editor, *Proceedings of CP'05*, volume 3709 of *LNCS*, pages 47–61. Springer-Verlag, 2005.

2. K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
3. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of PLILP'97*, volume 1292 of *LNCS*, pages 191–206. Springer-Verlag, 1997.
4. C. J. Colbourn and J. H. Dinitz, editors. *The CRC Handbook of Combinatorial Designs*. CRC Press, 1996.
5. K. Corrádi. Problem at Schweitzer competition. *Mat. Lapok*, 20:159–162, 1969.
6. J. Dinitz. Handbook of combinatorial designs, new results, 2005. Available at http://www.emba.uvm.edu/~dinitz/newresults.html. Accessed 28th July, 2005.
7. P. Flener, A. M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In P. Van Hentenryck, editor, *Proceedings of CP'02*, volume 2470 of *LNCS*, pages 462–476. Springer-Verlag, 2002.
8. P. Flener, J. Pearson, and L. G. Reyna. Financial portfolio optimisation. In M. Wallace, editor, *Proceedings of CP'04*, volume 3258 of *LNCS*, pages 227–241. Springer-Verlag, 2004.
9. I. Gent and N. Wilson. Minimizing pairwise intersections problem. Personal communication to Justin Pearson, October 2004.
10. M. Hall, Jr. Block designs. In E. F. Beckenbach, editor, *Applied Combinatorial Mathematics*, chapter 13, pages 369–405. John Wiley & Sons, 1964.
11. S. Jukna. *Extremal Combinatorics*. Springer-Verlag, 2001.
12. J.-F. Puget. Symmetry breaking using stabilizers. In F. Rossi, editor, *Proceedings of CP'03*, volume 2833 of *LNCS*, pages 585–599. Springer-Verlag, 2003.
13. J.-F. Puget. Using constraint programming to compute symmetries, 2003. Available at http://4c.ucc.ie/~bms/SymCon03/Papers/Puget.pdf.
14. Wikipedia. Collateralized debt obligations - wikipedia, the free encyclopedia, 2005. Available at http://en.wikipedia.org/wiki/Collateralized_debt_obligations. Accessed 18th May, 2005.
15. Wikipedia. Credit default swap - wikipedia, the free encyclopedia, 2005. Available at http://en.wikipedia.org/wiki/Credit_default_swap. Accessed 18th May, 2005.
16. Wikipedia. Credit derivative - wikipedia, the free encyclopedia, 2005. Available at http://en.wikipedia.org/wiki/Credit_derivative. Accessed 18th May, 2005.

Villkorsteknik och finansiella instrument

Olof Sivertsson

Department of Information Technology, Uppsala University
Box 337, 751 05 Uppsala, Sweden
olof@olofsivertsson.com

Bakgrund

Aktier och aktiefonder är exempel på finansiella instrument som många av oss är bekanta med. En del är även bekanta med något mer ovanliga finansiella instrument som optioner och hedgefonder. Collaterized Debt Obligations (CDO) och CDO squared (CDO^2) är det däremot få utanför finanssektorn som känner till, trots att de är finansiella instrument med en marknad värd flera miljarder euro.

Dessa instrument skapas av någon av aktörerna på marknaden, bl.a. Merrill Lynch och J.P. Morgon Chase, för att sedan säljas vidare till investerare världen över. En CDO skapas utifrån andra mindre komplexa finansiella värdepapper, medan en CDO^2 skapas utifrån andra CDO. I båda fallen säljs de vidare i mindre delar, vardera innehållande samma antal värdepapper. Varje sådan mindre del kallas för en *tranche*.

Varje utgivare vill att investerare ska köpa så mycket som möjligt av deras CDO. Samtidigt är det viktigt för investerare att sprida sina investeringar för att undvika hög risk att förlora pengar. En utgivare av en CDO vill därför se till att investerare ska kunna köpa mer än en tranche från samma CDO och ändå uppnå en önskad risknivå.

För att åstadkomma detta försöker utgivaren skapa en *optimal* CDO där varje par av tranches har så få värdepapper som möjligt som tillhör dem båda två. Figur 1 innehåller ett exempel på en sådan optimal CDO. Den består av fem tranches, vardera bestående av sju värdepapper, utifrån sammanlagt 17 värdepapper, där varje par av tranches har max två värdepapper som tillhör dem båda två.

Villkorsteknik

Problemet med att skapa en optimal CDO utifrån ett totalt antal värdepapper och ett antal tranches, vardera med lika många värdepapper, kan angripas via *villkorsteknik*.

Figur 1. Matris som representerar en CDO. Varje rad representerar en tranche. En ifylld ruta visar att en viss tranche innehåller ett värdepapper som motsvaras av den kolumn den ifyllda rutan befinner sig i.

Med villkorsteknik, till skillnad från programmeringsspråk som t.ex. Java eller C#, så beskriver vi inte *hur* problemet ska lösas utan enbart vilka villkor som en lösning på problemet måste ha. Sedan låter vi en dator söka efter lösningar som uppfyller dessa villkor. Om sökningen når ett tillstånd där en lösning är omöjlig, fortsätter sökningen från den position där senaste gissningen gjordes, tills alla möjligheter är uttömda eller vi nått en lösning.

Modell av problemet

För att lösa problemet via villkorsteknik används en matrismodell (jmf Figur 1) med en rad för varje tranche och en kolumn för varje värdepapper. En etta (nolla) på rad i och kolumn j motsvarar att tranche i innehåller (inte) värdepapper j. Eftersom varje tranche ska ha lika många värdepapper lägger vi till villkoret att summan av varje rad är konstant.

Det behövs även ett villkor som minimerar antalet värdepapper som finns i båda två av ett godtyckligt par av tranches. Genom att använda en formel som säger hur stort detta antal minst måste vara, kan sökningen börja därifrån. Om det visar sig vara omöjligt att uppnå detta värde, ökas det ett steg i taget tills problemet är löst. Villkoret som används är att skalärprodukten mellan varje par av rader i matrisen är mindre än det beräknade värdet.

Sökningen efter en lösning fortgår sedan rad för rad i matrisen.

Symmetri och Lex^2

Om sökningen hittar en lösning så finns det också andra lösningar med enda skillnaden att rader och/eller kolumner i matrisen bytt ordning inbördes. Detta kallas för symmetrier till dessa lösningar. På samma sätt finns symmetrier till matriser som inte kan vara lösningar (icke-lösningar).

Om sökningen efter ett tag måste ompröva en gissning så vill vi inte spendera tid med att söka efter lösningar som är symmetrier till den funna icke-lösningen.

Tyvärr är det generellt en praktisk omöjlighet att helt förhindra att symmetrier testas, ty antalet möjliga symmetrier är väldigt stort. Antalet villkor som skulle behöva användas för att helt förhindra detta blir så stort att sökningen skulle spendera allt för mycket tid med att kontrollera att dessa villkor var uppfyllda.

En anti-lexikografisk* ordning av både rader och kolumner i matrisen, Lex^2, kan dock förhindra testing av många, men inte alla, symmetrier, samtidigt som det är en enkel och snabb metod.

Dela upp problemet

Tyvärr är det generellt omöjligt att skapa en optimal CDO av verklig storlek, d.v.s. med ett par hundra värdepapper och ett tiotal tranches, med ovanstående metod eftersom det är för många möjligheter som måste avfärdas under sökningen.

Istället delas problemet upp i mindre delar som löses var för sig, för att sedan sättas ihop till en lösning på det ursprungliga problemet. En matris som är en lösning på det ursprungliga problemet skapas genom att lägga matriser som representerar lösningar till de mindre problemen efter varandra på rad. Uppdelningen görs också den via villkorsteknik.

Tillsammans med några optimeringar fungerar denna metod bra för att lösa problem av verklig storlek. Risken finns emellertid att den optimala lösningen inte nås. Det vägs dock upp av möjligheten att snabbt hitta en lösning.

Optimeringar som används är bland annat att återanvända lösningar till ett problem som lösningar till ett annat när så är möjligt och undersöka om problemet har sådan struktur att ytterligare villkor kan adderas för att öka prestanda. Dessutom fixeras de två första raderna i matrisen eftersom, om de ändrades, skulle det leda till testning av matriser som är symmetriska med redan testade matriser.

Mer symmetri

För att försöka få bättre prestanda undersöktes också möjligheten att minska antalet symmetrier som testades i onödan.

Arbetet utgick från publicerade artiklar om en teknik vid namn STAB. Idéen är att för varje rad som sökningen når läggs nya villkor till. Dessa villkor hindrar sökningen att komma till lägen som är symmetriska med det nuvarande.

Resultatet blev dock inte så bra som man hoppats på. För vissa problem halveras tiden innan en lösning hittats, ibland ännu snabbare. Tyvärr så ökar tiden innan en lösning hittas på många andra problem. Dessutom är minnesanvändningen såsom det är implementerat väldigt stor, vilket påverkar prestanda och hur stora problem som kan lösas.

Slutsats

Det program som konstruerats med de ovan beskrivna metoderna, och några till, för att skapa en optimal CDO utifrån givna parametrar löser de flesta verkliga problem på ett antal minuter, eller i gynnsamma fall enbart sekunder.

Om projektet fortsätter skulle prestanda kunna förbättras ytterligare. Nästa naturliga steg är en fortsatt undersökning av hur man förhindrar att symmetrier testas i onödan, bl.a. genom att mer djupgående undersöka metoder liknande STAB.

Även om den metod som presenteras ovan fungerar bra, har nyutvecklade metoder inom *Local Search* gjort att problemet kan lösas ännu effektivare. Local Search startar med något som liknar en lösning för att sedan ändra den steg för steg för att komma allt närmare en lösning. Med denna teknik kan verkliga problem lösas direkt utan att delas upp i mindre bitar. □

En utförligare och mer teknisk presentation av detta magisterexamensprojekt i datavetenskap finns i den fullständiga rapporten Construction of Synthetic CDO Squared *som kan hämtas på adressen* `http://olofsivertsson.com/xjob`.

* $x_1 \geq x_2 \geq \ldots \geq x_n$ är en anti-lexikografisk ordning, där $x \geq y$ betyder att x aldrig har en etta senare än y i de sekvenser av ettor och nollor de består av.