Implementation of an Algorithm to solve n-folds

Lasse Becker
CAU Kiel

Abstract

This paper introduces an implementation to an algorithm to solve integer linear programs with n-fold structure. The implemented algorithm is first introduced in the paper "Faster Algorithms for Integer Programs with Block Structure" from Eisenbrand et al. (2018). To put the quality of the implementation to a test, benchmarking is used. The experimental results will be used to compare a slightly modified version of the implementation with the initial one.

1 Introduction

Solving integer linear programs is a very hard, but relevant problem in practice. Due to this it is not surprising, that variants and special subsets of the problem are in the region of interest too. One such special subset is the set of n-folds . An n-fold is an integer linear program (ILP) of the form

$$\max\{\langle c, x\rangle | \mathcal{A}x = b, l \leq x \leq u, x \in \mathbb{Z}^{nt}\} \tag{1}$$

where $n, r, s, t \in \mathbb{N}_{>0}$ are general parameters, $b \in \mathbb{Z}^{r+ns}$ is a right-hand-side vector, $l, u \in \mathbb{Z}^{nt}$ are lower and upper bounds, $c \in \mathbb{Z}^{nt}$ is the cost-vector and the constraint-matrix $\mathcal{A} \in \mathbb{Z}^{nt \times r+ns}$ has the following structure

$$\begin{pmatrix} A_1 & A_2 & \cdots & A_n \\ B_1 & 0 & \cdots & 0 \\ 0 & B_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & B_n \end{pmatrix}$$

with $A_1 \dots A_n \in \mathbb{Z}^{s \times t}$ and $B_1 \dots B_n \in \mathbb{Z}^{r \times t}$. In the following sections Δ denotes $||\mathcal{A}||_\infty$. Another important number will be $L_\mathcal{A} := (2s\Delta + 1)^s(2r\Delta(2s\Delta + 1)^s)^r$. There are some papers dealing with the problem of solving n-folds . One of them is from Eisenbrand et al. (2018), providing a description for an algorithm to solve a n-fold in time $n^2t^2\varphi \log^2 nt \cdot (rs\Delta)^{\mathcal{O}(r^2s+rs^2)}$, where φ is the logarithm of the largest entry of the input. Goal of this work was to implement this algorithm, find a (local) run time improvement and evaluating some benchmark tests, to validate the improvement. The algorithm is using a augmenting framework, the idea is to find a feasible solution and improve it. This framework is used by Jansen et al. (2018) too. Due to a more clear formulation, the part finding an initial solution is based on the latter paper. To bring up one of the latest works done on this topic: Cslovjecsek et al. (2020) came up with an algorithm, witch is designed to solve a n-fold in parallel.

2 Implementation

This chapter contains technical and algorithmic details of the implementation used to solve n-folds . To speed up the program the implementation is written in C++. For storing and handling calculations on vectors and matrices the Eigen library[1] is used.

2.1 Maximizing a feasible Solution

This whole section is based on the paper written by Eisenbrand et al. (2018).
Assume n-fold (1) and a feasible vector f. The following algorithm for solving (1) is provided by Eisenbrand et al. (2018).
Depending on some $\lambda \in \mathbb{Z}$ there is a method to gain an improved feasible solution. To find a sufficient improvement different λ's can be tested and the best result will be chosen. The λ's to be tested are 2^k for each $0 \leq k \leq \lceil \log_2(||u - l||_\infty) \rceil$.
Searching for an improvement for a given λ works by solving a, so called, augmenting n-fold

$$\max\{\langle c, x\rangle | \mathcal{A}x = 0, l^* \leq x \leq u^*, x \in \mathbb{Z}^{nt}\} \tag{2}$$

where $l_i^* = \max(\left\lceil \frac{l_i - f_i}{\lambda} \right\rceil, -L_\mathcal{A})$ and $u_i^* = \min(\left\lfloor \frac{u_i - f_i}{\lambda} \right\rfloor, L_\mathcal{A})$ for all $1 \leq i \leq nt$. The notation for a solution of this n-fold is from now on y^s. The improved result will be $f + \lambda y^s$.
The Manhattan-Norm of y^s is limited, therefore an exhaustive search is possible. Instead it is suggested to build a graph each node standing for a partial sum $\sum\limits_{i=1}^{k} \mathcal{A}_i y_i^c$ for each vector y^c from the solution space and for each $0 \leq k \leq nt$. ($\sum\limits_{i=1}^{0} \mathcal{A}_i y_i^c := \vec{0}$) Edges will be between nodes associated with $\sum\limits_{i=1}^{k} \mathcal{A}_i y_i^c$ and $\sum\limits_{i=1}^{k+1} \mathcal{A}_i y^c$. Such an edge will be labeled with y_{k+1}^c and a weight of $c_{k+1} \cdot y_{k+1}$. For k being a multiple of t there are two more restrictions.

(i) For $k = mt$ $(1 \leq m < n)$ there are only edges to partial sums where components from index $r + 1$ to $r + ms$ for $1 \leq m \leq n$ are zero.

(ii) For $k = nt$ the full sum is meant and so there are only edges to the zero vector.

[1] `https://eigen.tuxfamily.org`

This restrictions are leading to a necessary pruning. Also not the whole partial sum needs to be associated with each node, but the first r components and the components from $r + mt + 1$ to $r + m(t + 1)$, for $mt \leq k \leq m(t + 1)$. This is possible since $\mathcal{A}y^s = \sum_{i=1}^{nt} \mathcal{A}_i y_i^s$ and therefore $\sum_{i=1}^{kt} \mathcal{A}_i y_i^s$ needs to be zero in the components from index $r + 1$ to $r + ms$ for and from $r + (m + 1)s + 1$ to $r + ns$ $1 \leq m \leq n$, due to the n-fold structure. This graph has $nt + 1$ layer (for each k one). A path through this graph starting at the only/ $\overrightarrow{0}$ node in layer 0 ending at $\overrightarrow{0}$ in layer nt, represents a feasible solution of the augmenting n-fold (2). A longest path therefore is associated as an maximal solution of y^s. To find a longest path it is suggested to use the Moore-Bellman-Ford algorithm.

To sum up: given a feasible solution f for (1) the algorithm to find a maximal solution looks like the following.

Algorithm 1 Solve n-fold with Initial Solution f

```
1: $improvement\_found \leftarrow$ True
2: while $improvement\_found$ do
3:     $improvement\_found \leftarrow$ False
4:     for each $\lambda$ in $\{2^0, 2^1, \dots, 2^{\lceil \log_2(||u-l||_\infty) \rceil}\}$ do
5:         $augmenting\_vector \leftarrow$ solution of augmenting $n$-fold for $\lambda$ and $f$.
6:         $candidate \leftarrow f + \lambda \cdot augmenting\_vector$
7:         if $\langle c, candidate \rangle > \langle c, f \rangle$ then
8:             $f \leftarrow candidate$
9:             $improvement\_found \leftarrow$ True
   return $f$
```

2.2 Finding an initial Solution

The algorithm from the previous section needs an initial feasible solution, how to find such is described in this paragraph. This part is completely based on the work of Jansen et al. (2018). The basic concept is simple: The original n-fold will be transformed, such that the transformation has a trivial feasible solution, so the solution can be maximized. A maximum of the transformed n-fold yields a feasible solution for the original ILP. This new n-fold will be:

$$\max\{\langle c^t, x\rangle | \mathcal{A}^t x = b^t, l^t \leq x \leq u^t, x \in \mathbb{Z}^{n(t+r+s)}\}. \quad (3)$$

Let's assume the original ILP (1). To transform the constraint matrix, $\mathcal{A}$ will be widened by $r + s$ columns in the following manner:

Every zero block will be expanded by zero columns, so the n-fold structure remains;

to the first A-block r columns of the unit matrix $\mathcal{I}_r$ will be appended and s zero columns;

to the rest of the A-blocks only zero columns are appended;

every B-block is widened by r zero columns and s columns of the unit matrix $\mathcal{I}_s$.
The result looks like the following:

$$\mathcal{A}^t = \left(\begin{array}{ccc|ccc|c|ccc} A_1 & \mathcal{I}_r & 0 & A_2 & \mathcal{I}_r & 0 & \cdots & A_n & \mathcal{I}_r & 0 \\ B_1 & 0 & \mathcal{I}_s & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & B_2 & 0 & \mathcal{I}_s & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & & B_n & 0 & \mathcal{I}_s \end{array}\right).$$

The transformed right-hand-side vector will be $b^t = b - \mathcal{A}l$. Now it is already possible to give a feasible solution f^t for (3). If i is an index of a newly inserted column of $\mathcal{A}^t$, $f^t_i = b^t_i$ and otherwise $f^t_i = 0$.
The bounds for the transformed n-fold will be chosen as follows $l^t = \vec{0}$ and u^t will 0 in new variables and $u - l$ in the old ones.
To make it possible to derive a feasible solution for (1) from a maximal solution of (3) the cost vector c^t is chosen as follows: If i is an index of an original variable, then $c^t_i = 0$. So increasing a solution in this variable will not result in a worse solution. To penalise using newly inserted variables $c^t_i = 1$ if $b^t_i \geq 0$ and $c^t_i = -1$ else.
Due to the transformed cost vector a maximal solution $f^{t\max}$ of (3), will minimize the usage of new variables, so if they are all 0 in $f^{t\max}$, then $\langle c^t, f^{t\max} \rangle = 0$ and the remaining variables will solve $\mathcal{A}x = b^t$. Shifting this rest back by adding l we gain a feasible solution f for (1). If $f^{t\max}$ is still using artificially added variables, then the original n-fold (1) has no solution.

2.3 Adjustments and Improvements

In the following section concrete adjustments and improvements to the pure algorithm, introduced by Eisenbrand et al. (2018), are introduced. In other words in this section discusses the differences between the algorithmic ideas and the concrete implementation.

Classes and reducing a augmenting Graph

It is obviously sensible to bring up a class for n-folds , this is called NFILP and it keeps not the whole constraint matrix, but the A- and B -blocks. Also this class provides an method to get a transformed version of itself and a corresponding feasible solution. Further a NFILP object holds the capability to maximize a given feasible solution as described previously. The first major adjustment made is not to capture the whole graph to solve the augmenting n-fold (2) in memory. Instead the graph will be hold only nodes reachable from the start node in layer 0. This is sensible since the goal is to find a path through the graph. To achieve this the following classes are implemented: Node represents a node holding only the current longest path to get there together with the current maximal costs. Of course also the value of the partial sum is hold. Two nodes with the same value, inside the same layer will be interpreted as the same. To to prevent node duplication inside a layer, a class NodeSet is implemented inheriting from a hash map (unordered_map), where the partial sum is the key to the corresponding node. An NodeSet object provides a method to update a node. Updating a node in a layer means inserting the node if is not contained jet and updating the longest path and costs of the node otherwise. The class for solving augmenting n-folds like (2), is Layer . A layer object belongs to a specific augmenting ILP, therefore it holds an attribute for λ. Since such an object corresponds to a layer of the augmenting graph, it has a number from 0 to $n \cdot t$, this is also the length of the paths stored in the nodes of the layer. This nodes will be stored in a NodeSet object. Since a layer belongs to one variable of the augmenting ILP and stores therefore only the appropriate entry of l^* and u^* (in attributes l_star and u_star). Very important is the functionality to increment a Layer .When layer x is incremented afterwards it corresponds to layer $x + 1$. During incrementaiton a layer for each node only the children will be added to the next NodeSet . This way only nodes reachable from the previous layer are stored, getting back to where we started. Now the only thing left to do in order to solve an augmenting ILP, is initializing a layer and incrementing it until the last layer is reached. Now the longest path and therefore a maximum solution to the augmenting ILP is stored in the Node of the last layer. The following graphic sums up what is described above and gives an overview of the classes implemented and their functionality.

Overflow on $L_{\mathcal{A}}$

While running debuging tests on some earlier implementation the algorithm returned some wrong answers, due to a overflow on $L_{\mathcal{A}}$. This leads to the wrong lower and upper bounds (l^*, u^*) when it comes to solve an augmenting n-fold . In detail this is, because of the definitions $l_i^* = \max(\left\lceil \frac{l_i - f_i}{\lambda} \right\rceil, -L_{\mathcal{A}})$ and $u_i^* = \min(\left\lfloor \frac{u_i - f_i}{\lambda} \right\rfloor, L_{\mathcal{A}})$. If an overflow in the variable holding $L_{\mathcal{A}}$ occurs, minimum and maximum might be chosen wrong, since the variable might now be negative. Such an overflow happens pretty soon, due to the very fast growing $L_{\mathcal{A}}$. In order to handle this l_i^* and u_i^* are directly chosen as $\max(\left\lceil \frac{l_i - f_i}{\lambda}\right.$ and $\left\lfloor \frac{u_i - f_i}{\lambda}\right.$. When it happens that this might not be the right choice a warning will be output to the console.

Zero Column checking

To find a initial solution a transformation of the original n-fold is build. The constraint matrix of this new n-fold contains at least $(n - 1) \cdot r$ zero columns. So there is a lot of not

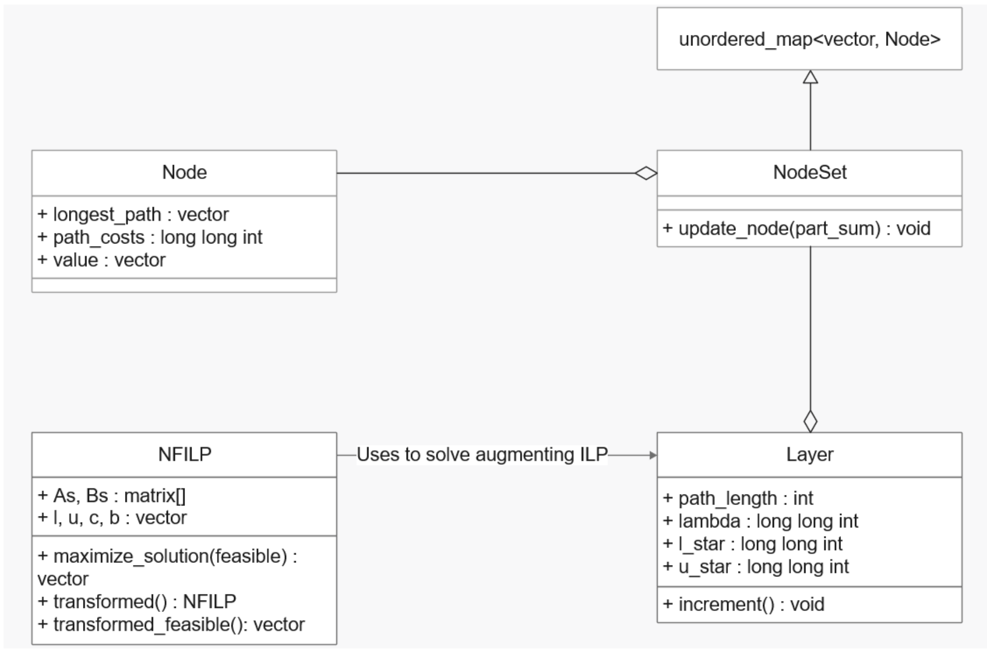

Figure 1: UML sketch of implemented classes

necessary work done when it comes to calculating the children of a layer in an augmenting graph. The children of a node with value (partial sum) v in layer x will have values $\{(v + l^*_{x+1}\mathcal{A}_{x+1}), (v + (l^*_{x+1} + 1)\mathcal{A}_{x+1}), \dots, (v + u^*_{x+1}\mathcal{A}_{x+1})\}$. If $\mathcal{A}_{x+1}$ is zero, this set only contains only v. Repeating the procedure for each node in a layer brings overhead, since we actually know that each child will have the same value as their parent. So before iterating over each node in a layer the "zero-column-checking algorithm" checks weather the next column of the constraint matrix is zero. If this is the case the either l^*_i or u^*_i will be saved as next edge to take (depending on witch results in more costs) and the nodes will stay the same. How well this improvement works will be discussed in Section 4.

3 Creation of Test Cases

This paragraph introduces the method used to generate test instances, to give the reader full power to interpret the results from this paper. The main idea is build on an algorithm to generate general ILP's

$$\max\{\langle c, x\rangle | \mathcal{A}x = b, l \leq x \leq u, x \in \mathbb{Z}^n\}3$$

The same idea is applied to get instances for n-folds .

The first naive idea is to choose l, u, c, b and $\mathcal{A}$ completely random, an instance generated in such manner might have no solution. In practice the chance to gain a feasible ILP with the naive method is actually very poor.

So there is the temptation to create some reasonable lower and upper bounds with some vector in between. Multiplying this vector with some random matrix yields a right-hand-side vector, such that together with the constraint matrix and the lower and upper bounds a feasible ILP is build.
The actual used technique is granting $p \in \mathbb{N}$ feasible solutions for the generated instance $Ax = b$. This will be described in the following: First step is to generate p vectors $f_1 \dots f_p \in \mathbb{Z}^{nt}$. They will be the feasible solutions to the generated test case. To fulfill this for each $1 \leq i, j \leq p$ two conditions must hold:

(i) $l \leq f \leq u$

(ii) $\mathcal{A} f_i = b = \mathcal{A} f_j$

where $\max\{\langle c, x\rangle | \mathcal{A}x = b, l \leq x \leq u, x \in \mathbb{Z}^n\}$ is the finished test instance.
Since c is not restricted by the f_i's it may be chosen randomly inside some given bounds.
To fulfill (i) l and u might be chosen as the pairwise minimum and maximum of the f_i. Actually the l and u are also incremented pairwise by 1, to potentially gain more feasible solutions.
Restriction (ii) gives the following linear system: $(f_{ij})_1 \cdot \mathcal{A} + (f_{ij})_2 \cdot \mathcal{A} + \dots + (f_{ij})_3 \cdot \mathcal{A} = 0$ where $1 \leq i, j \leq p$, $f_{ij} := f_i - f_j$ and $\mathcal{A}_i$ is the i-th column of $\mathcal{A}$. Since the equations are highly linear dependent

$$\Big(\forall i < j \colon f_{ij} = f_i - f_j = (f_i - f_{i+1}) + (f_{i+1} - f_{i+2}) + \dots + (f_{j-1} - f_j)\Big)$$

the equation system reduces drastically.

Writing the reduced system as a matrix: $\begin{pmatrix} f_{12} \\ f_{23} \\ f_{34} \\ \vdots \\ f_{(p-1)p} \end{pmatrix}$.

With a Gauß-like algorithm this can be resolved to a matrix of the form:

$$\begin{pmatrix} * & & 0 & * & \dots & * \\ & \ddots & & \vdots & \ddots & \vdots \\ 0 & & * & * & \dots & * \end{pmatrix}$$

where $* \in \mathbb{Z}$. Finding a feasible vector roughly reduces to choosing arbitrary entries (of course some bounds needs to be given, we will call them lower and upper matrix bound) for the second-part of the solution. Depending on this the first-part can be calculated. Note: During this calculations the second-part might change.
Every feasible vector can now be treated as a row of $\mathcal{A}$ and a ILP with at least p solutions generated.
This algorithm can be easily adopted to generate random n-folds with p solutions.

4 Evaluation

The presentation of the experimental results to compare the implementation with and without "zero-column-check" is done in this section. The idea is to reduce the run time during the part of finding an initial solution, like described in Section 2.3. Because the number of added zero columns to a transformed constrained matrix, depends on the parameter r it is only sensible to have a look weather the improvement pays off for growing r.

4.1 About the sample Set

The generated samples will all have at least two feasible solutions randomly chosen between $-\vec{1}0$ and $\vec{1}0$. The matrix bounds are -4 and 4 and for each instance $\Delta \leq 960$, $||u-l||_\infty \leq 39$, and $-\vec{3}0 \leq c \leq \vec{3}0$. The sample set provides test instances for each $r \leq 10$. In the set for each $n \leq 3$, $s \leq 2$ and $t \leq 2$ instances with corresponding parameter equally arranged. The total number of tested cases is 84 for $s \leq 3$ and 36 for grater s.

4.2 Technical Disclaimer

In practice there are restrictions how long a test can run and how much memory can be used. So computations using more than 200 GB of memory or more than 4 hours to complete were interrupted. This is also logged and the handling of this is exceptions is described for each evaluation.

4.3 Results on "zero-column-check" Improvement

As the problem to solve ILP's is NP-Hard, it is not surprising that even for low n, r, s and t a lot of bench marking tests failed, due to exceeded run time or memory. But first take a look on the results where the algorithms passed. The bench marking results from running each test case for each algorithm can be viewed in Figure 2. For small instances with $s \leq 2$, the "zero column checking" seems to struggle with the branching overhead and has a worse average run time than the pure algorithm. For grater s the improvement seems to pay off and this version stays constantly faster, until $s = 10$ is reached. For $s = 10$ the native implementation has the lower average run time again. The difference is not big, but this little outlier tells that even for grater s the improvement is not set to be finally better than the pure algorithm. But note that no test instances with $n = 3$ are passed for $s = 9$ and $s = 10$, what relativize the outlier a bit. Interestingly the run time seems not to increase for growing n-fold instances, when having a look at Figure 2. That this impression is false becomes clear when having a look at the failed tests. The proportion of tests exceeding time or memory drastically grows with increasing size of the test instances like illustrated in Figure 3.

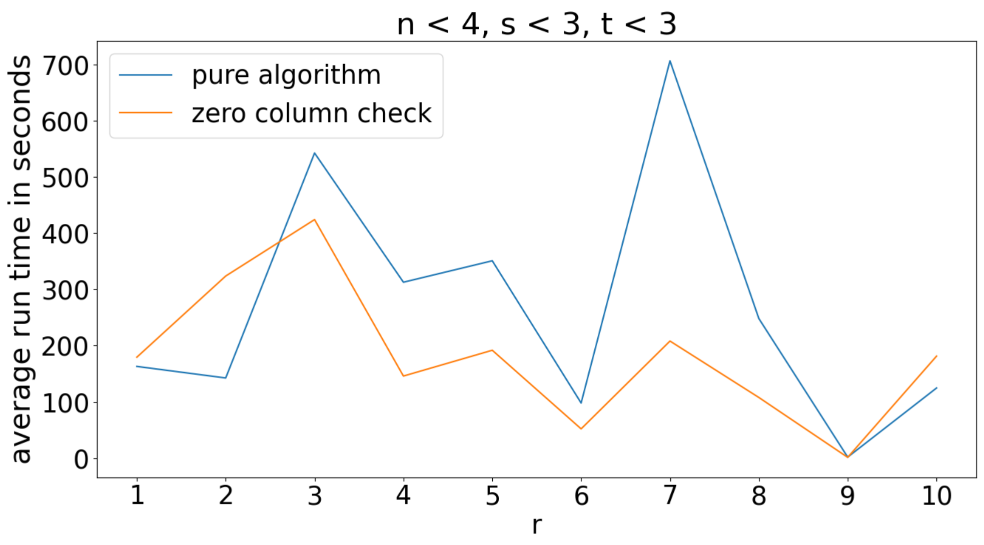

Figure 2: Average run time

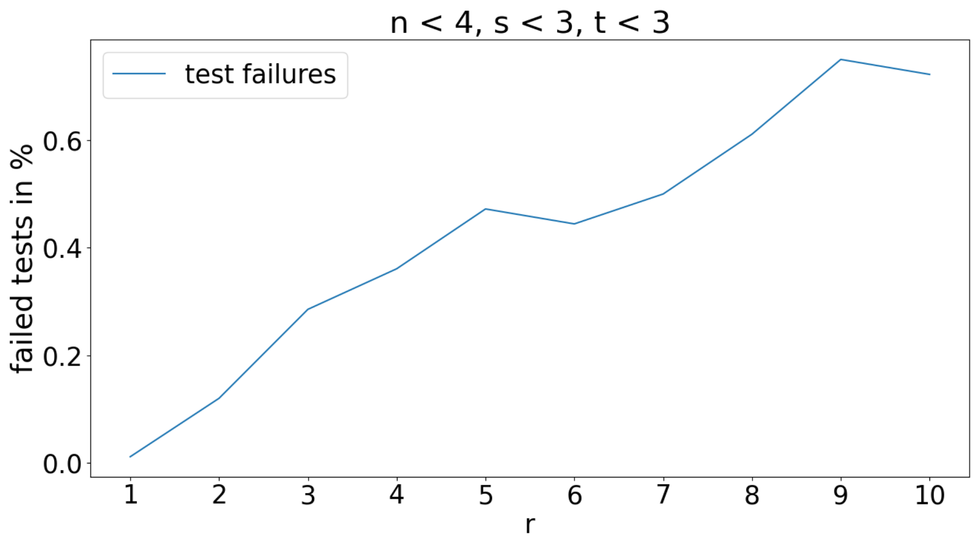

Figure 3: Percentage of Tests exceeding Time or Memory Resources

5 Conclusion

Checking for zero columns has an improving effect on the run time, sadly it is not constantly better than the native implementation. Further work on this could bring a more reliable improvement. The goal should be to bring the ”zero-column-checking” algorithm constantly ahead in comparison to the pure implementation, when this is achieved the improvement presented in this paper can be absolutely recommended. Over all it is an improvement with low implementing costs.

References

Cslovjecsek, J., F. Eisenbrand, C. Hunkenschröder, L. Rohwedder, and R. Weismantel (2020). Block-structured integer and linear programming in strongly polynomial and near linear time.

Eisenbrand, F., C. Hunkenschröder, and K.-M. Klein (2018). Faster algorithms for integer programs with block structure.

Jansen, K., A. Lassota, and L. Rohwedder (2018). Near-linear time algorithm for n-fold ilps via color coding.