Presolve Routines for LP and SDP within Convex.jl

Ramchandran Muthukumar*

Birla Institute of Technology and Science-Pilani, India
ramcha1994@gmail.com
Github handle : ramcha24
Location/Timezone: India, GMT +5:30
Mentor: Madeleine Udell

1. Personal Background

I am a third year undergraduate student pursuing Masters in Mathematics and Bachelors in Computer Science from BITS Pilani, Goa Campus. For the past 2-3 years, I have taken rigorous math courses like Algebra, Real Analysis, Measure and Integration, Topology. In particular I enjoyed the optimization-related courses like - Linear Programming, Non-Linear Optimization and Operations Research.

My first exposure to Convex Optimization and the Stanford Convex Group was actually through a Machine Learning Project back in 2014 where I had to use Prof. Stephen Boyd's implementation of the interior point method for l1-logistic regression. This led me to discover the vast amount of materials hosted on Prof. Boyd's web page for his Convex Optimization courses - EE364a/b - the video lectures, slides, the book, MATLAB files, extra exercises et cetera. I can never be grateful enough to Prof. Boyd for making these available online. For the past one year I have been reading and working my way through Prof. Boyd's book on Convex Optimization and it's been an immense pleasure to attempt the exercise questions (especially from the "Extra Exercises") and along the way I have greatly enjoyed implementing the solutions using CVX, CVXPY, Convex.jl, PICOS.

I think I am very well suited to work on the project proposed, as it requires a good comfort level with the theory of convex optimization and higher level mathematics that I am accustomed to, due to my background in abstract Mathematics courses and the wonderful course on Convex Optimisation. I have worked through two research papers, and I believe that I can convert the ideas in the paper to code and add Pre-solving capabilities to Convex.jl.

2. Programming Skills

I have worked with C, C++, Java, Python, R and Julia before. I usually use C++ for solving algorithmic questions from websites like Hackerrank, Codechef etc.

A MOOC course I completed in summer 2015 - Analytics Edge by MITx involved programming with R and I am quite comfortable with R. I have used Python for machine-learning scripts in the past (working with scikit-learn, numpy libraries).

Recently I helped build *conda* packages for CVXPY to make installation easier. The files for the same are hosted here for public download.

Exercises using CVX that were part of my formal report for a study-oriented-project on campus are available here and some sample code that I have written using Julia - Binary-searching variants in Julia, Quasi-Convex Optimization in Julia.

3. Project Synopsis

Pre-solving is the process of detecting redundancies in an optimization problem and removing them so that the problems that are fed to solvers are properly formulated. The reduced optimization problem is now solved by the solver. This has the two-fold benefit of speeding up the optimization process, and higher accuracy in solutions. Since smaller problems are fed to the solver (eg. SCS), the bottleneck call to the solver has now become significantly faster.

Ideally, we would want to remove the types of redundancy which lead to reduction in total solution time. But this isn't possible. There is also a trade-off between time spent in the pre-solve procedure and the amount of redundancy detected by our procedure.

Motto - Find simple forms of redundancies and find them quick.

Many commercial solver include pre-solve (like CPLEX) but almost no open source solver has an efficient and exhaustive implementation of pre-solve for LP. To the best of our knowledge no first order solver has pre-solve of any kind for SDP problems. Thus adding a pre-solve to Julia-opt greatly increases the competition of open-source solvers.

I intend to write pre-solve routines for Julia-opt rather than write interfaces to Frank Permenter's **frlib** for SDP or

*I would like to thank Madeleine Udell for pointing out the recent paper on Partial Facial Reduction[2]

GLPK for LP. Why? The presolve support for LP in other open source solvers is not as exhaustive as proposed. The presolve support for SDP in **frlib**, uses SeDuMi, but first-order solver like SCS (the default solver in Convex.jl) has been shown to be considerably faster when the problem size scales from [3].

SCS currently has no pre-solve inbuilt and this would help resolve that issue. Also writing optimized Julia code is easier than re-writing in a lower-level language like C and more easily maintained for further improvements in future. This could have the added side benefit of yet another benchmark against other 'fast' languages and give feedback on how the pre-solve routine written completely in Julia would fare against some of the support given in GLPK , CLP.

The next few sections will give a theoretical description of the process 'under-the-hood' of the final pre-solve algorithm. While I have quoted two quality research papers, the mathematics behind both these papers are well understood and acknowledged and their conversion to code can be done to get reliable results with ease.

In fact, for SDP, where the mathematics is more involved due to the frlib package we have a clearer starting point from their implementation so that we can straightaway discuss design issues for implementing in Julia.

3.1. Pre-solving for LP

Any LP problem can be formulated as:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & l \leq x \leq u \end{aligned}$$

where $x, c, l, u \in \mathbb{R}^n$ and $b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$

Some of the simple bounds l_j, u_j may be $-\infty, \infty$.

Define $L = \{j : l_j > -\infty\}$ and $U = \{j : u_j < \infty\}$

Then the constraints of the LP are:

$$\begin{aligned} Ax^* &= b \\ A^T + y^* &= c \\ (x_j^* - l_j)z_j^* &= 0 \\ (u_j - x_j^*)z_j^* &= 0 \\ z_j^* \geq 0 \quad \forall \quad j \quad &\in \quad \mathrm{L} \\ z_j^* \leq 0 \quad \forall \quad j \quad &\in \quad \mathrm{U} \\ z_j^* = 0 \quad \forall \quad j \quad &\notin \quad L \cup U \end{aligned}$$

We want a pre-solve procedure that would reduce the size of A without creating new non-zero terms.

This also means that linear transformations are not allowed but changes to c, b, l, u are.

We make several passes through A and remove redundancies as and when they are detected. Some types of redundancies are easily understood and in the next subsection we will look at few of them.

3.2. Redundancies in LP

Empty Rows/Columns

When a row i is empty, either the constraint is redundant or infeasible.

When a column j is empty, depending on the bounds on the variable and the objective function, the variable x_j can either be fixed at its bounds or is unbounded.

Row Singletons

$$\exists (i,k) \text{ s.t. } a_{ij} = 0 \quad \forall j \neq k, a_{ik} \neq 0$$

Here, the i^{th} constraint fixes the variable x_k at the value - $\frac{b_i}{a_{ik}}$

Thus we can substitute x_k out of the system and reduce the number of variables by one.

Thus, one reduction could possibly lead to several others and to exploit this, we need to keep a count of number of non-zeros in each constraint and a list containing all singleton constraints.

Every time we fix a variable, the number of non-zeros is updated and the new singleton constraints that arise are added to the list.

Column Singletons

$$\exists (j,k) \text{ s.t. } a_{ij} = 0 \quad \forall i \neq k, a_{kj} \neq 0$$

When this occurs, a 'free' column singleton occurs on variable j along with $l_j = -\infty$ and $u_j = \infty$. Here again we can substitute:

$$x_j = \frac{b_k - \sum_{p \neq j} a_{kp} x_p}{a_{kj}}$$

Removal of column singletons is very advantageous as they result in the removal of one variable and also one constraint.

Whenever our Presolve procedure detects a column singleton x_j , we try to establish that it is *implied free*.

A variable is *implied free* if we can construct new bounds, that are at least as tight as the original bounds. A candidate pair of bounds for the variable is calculated for each a_{ij} as -

$$\begin{aligned} u_{ij} &= \frac{b_i - \sum_{k \in P_{ij}} a_{ik} l_k - \sum_{p \in M_{ij}} a_{ik} u_k}{a_{ij}}, \quad a_{ij} > 0 \\ u_{ij} &= \frac{b_i - \sum_{k \in P_{ij}} a_{ik} u_k - \sum_{p \in M_{ij}} a_{ik} l_k}{a_{ij}}, \quad a_{ij} < 0 \\ l_{ij} &= \frac{b_i - \sum_{k \in P_{ij}} a_{ik} l_k - \sum_{p \in M_{ij}} a_{ik} u_k}{a_{ij}}, \quad a_{ij} < 0 \\ l_{ij} &= \frac{b_i - \sum_{k \in P_{ij}} a_{ik} u_k - \sum_{p \in M_{ij}} a_{ik} l_k}{a_{ij}}, \quad a_{ij} > 0 \end{aligned}$$

For any feasible solution x , it is easy to observe that $l_{ij} \leq x_j \leq u_{ij}$. Thus, we have an implied free column singleton if - $l_j \leq l_{kj} \leq u_{kj} \leq u_j$

Forcing and Dominating Constraints

Let $P_i = \{i : a_{ij} > 0\}$ and $M_i = \{i : a_{ij} < 0\}$. Let's compute the quantities:

$$g_i = \sum_{j \in P_i} a_{ij} l_j - \sum_{j \in M_i} a_{ij} u_j$$
$$h_i = \sum_{j \in M_i} a_{ij} l_j - \sum_{j \in P_i} a_{ij} u_j$$

Clearly, $g_i \leq \sum a_{ij} x_j \leq h_i$ for any solution x. Now if $h_i < b_i$ or $g_i > b_i$, we have an unfeasible constraint.

A forcing constraint is one where $g_i = b_i$ or $h_i = b_i$. Here, the value of x_j is fixed at its bounds according to the sign of a_{ij} and thus we can fix all variables that occur in the i^{th} constraint.

Removal of Forcing Constraints is highly advantageous as they we remove all variables that are structurally degenerate. We can also use this to detect more column singletons and we can determine l'_{ij}, u'_{ij} for appropriate cases when $a_{ij} \neq 0$ and when g_i or h_i is finite.

$$u_{ij} = \frac{b_i - g_i}{a_{ij}} + l_j, \quad a_{ij} > 0$$
$$u_{ij} = \frac{b_i - h_i}{a_{ij}} + l_j, \quad a_{ij} < 0$$
$$l_{ij} = \frac{b_i - h_i}{a_{ij}} + u_j, \quad a_{ij} > 0$$
$$l_{ij} = \frac{b_i - g_i}{a_{ij}} + u_j, \quad a_{ij} < 0$$

This is called the Dominated Constraint Procedure. Similarly there are concepts of Dominated Columns and Forcing Columns elaborately discussed in the paper by Andersen.

3.3. Algorithm for LP

In [1] , the authors have proposed the following algorithm for Pre-solving a LP -

1. Remove all fixed variables
2. Repeat
   - Check Rows
     - Remove Row Singletons
     - Remove Forcing Constraints
   - Dominated Constraints
     - Remove all dominated constraints
   - Check Columns
     - Remove all free, implied free column singletons
     - Remove column singleton-double combinations
   - Dominated Columns
   - Duplicate Rows
   - Duplicate Columns

   Until no reduction in last pass
3. Remove all empty rows and columns

We also need to maintain a 'stack' of presolve operations and undo a reduction such that, if the primal and dual solutions to the reduced problem are optimal and feasible, so is the solution to the restored problem.

This constitutes a Post-solve procedure to recover solutions to the orginial problem, after we have solved the reduced LP.

Each of the operations in pre-solve has a corresponding post-solve procedure whereby we 'unfix' the fixed variable x_j or constraint i at each iteration.

3.4. Pre-solving for SDP

SDPs with no strictly feasible solutions don't satisfy Slater's conditions for strong duality. This is a frequent problem arising on outputs from the parser, that generate SDP-based relaxations of algebraic problems. This is because the parser does not exploiting the structure of the SDP.

The feasible set of an SDP is the intersection of an affine subspace and the cone of positive-semidefinite matrices. If there are no matrices in the feasible set that are positive definite then strict feasibilty fails.

This results in two issues:
- Lack of strong duality means duality gap is not zero.
- The SDP is unnecessarily large; the feasible set can be reduced.

One such approach for a general Conic Optimization Problem is the **Facial Reduction Algorithm** which aims at finding *faces* of the original PSD cones that contain the feasible set.

Let's look at some terminologies that are needed to discuss the facial reduction algorithm -

Let E be a finite dimensional vector space of $\mathbb{R}$ with inner product $\langle \cdot, \cdot \rangle$

- For a subset $S \subseteq E$, we define LinS to be the Linear Span of all elements of S and $S^{\perp}$ to be the orthogonal complement of Lin$S$
- A **Convex Cone** K is a convex set that satisfies - $x \in K \implies \alpha x \in K \quad \forall \alpha \geq 0$.
- A **Dual Cone** K^* of K is teh set of linear functions non-negative on K - $\{y | \langle y, x \rangle \geq 0, \quad \forall x \in K\}$

- A **face** F of a convex cone K is a convex subset that satisfies the following

$$\frac{a+b}{2} \in F \text{ and a, b, } \in K \implies a, b \in F$$

- A face is **proper** if it is non-empty and not equal to K. Faces of convex cones are also convex cones.
- $\mathbb{S}_+^n$ denotes the convex cone of Positive Semidefinite Matrices
- A set F is a face of $\mathbb{S}_+^n$ iff it equals the set of all $n \times n$ PSD matrices with range contained in a given d-dimensional subsapce.
- A proper face f (and f^*) can be described with a invertible matrix $(U, V) \in \mathbb{R}^{n\times n}$ where the range of $U \in \mathbb{R}^{n\times d}$ equals this subspace and $V \in \mathbb{R}^{n\times n-d} = R(U)^\perp$
- The relative interior of a set S consists of points of the set that are not in the boundary with respect to the affine hull of S. Its used to analyse low-dimensional spaces embedded in higher-dimensional spaces. Mathematically ,

$$relint(S) := \{x \in S : \exists \epsilon > 0, N_e(x) \cap \text{aff}(S) \subseteq S\}$$

- The following lemma stated in [2] is important:

  Lemma 3.1 *K - convex cone and A - affine subspace such that $A \cap K$ is non-empty. Then the following statements are equivalent:*

  1. *$A \cap$ relint K is empty*
  2. *$\exists s \in K^* - K^\perp$ for which the hyperplane $s^\perp$ contains A.*

  Thus if we have strict feasibility then the first condition is satisfied and which means a suitable vector s exists, we call this vector the **reducing certificate** as we can construct a new face at any iteration i as $F_{i+1} = F_i \cap s_i^\perp$.

Thus we would get a sequence of 'faces' , $F_n \subset F_{n-1} \subset \ldots \subset F_0 = K$. Producing this sequence is called **Facial Reduction**.

Facial Reduction Algorithm

1. Begin, Initialize $f_0 \leftarrow K, i = 0$
2. Repeat
   - Find Reducing Certificates i.e, Solve the feasibility $(*)$

$$\begin{aligned} \text{Find} \quad & s_i \in F_i^* - F_i^\perp \\ \text{subject to} \quad & s_i^\perp \text{ contains } A \end{aligned}$$

   - Compute new face, $F_{i+1} = F_i \cap s_i^\perp$
   - Increment i

   Until the feasibility problem $(*)$ is infeasible

In the case of an SDP, we compute a new face by using a matrix $B \in \mathbb{R}^{d\times r}$ with range = Null$(U_i^T S_i U_i)$ and $F_i \cap S_i^\perp = U_i B S_+^r B^T U_i^T$

Facial Reduction Algorithm for SDP

1. Begin, Initialize $U_0 = I_{n\times n}, d_0 = n, i = 0$
2. Repeat
   - Find Reducing Certificates S_i by solving the SDP:

$$\begin{aligned} \text{Find} \quad & S_i \in S^n \\ \text{subject to} \quad & U_i^T s_i^\perp U_i \text{ contains } A \\ & U_i U_i^T . S_i = 1 \\ & C.S_i = 0 \\ & A_j - S_i = 0 \quad \forall 1 \le j \le n \end{aligned}$$

   - Compute New face by finding basis for Null$(U_i^T S_i U_i)$ and setting $U_{i+1} = U_i B$ and $d_i + 1 = \dim(\text{Null}(U_i^T S_i U_i))$
   - Increment i

   Until the feasibility problem $(*)$ is infeasible

But finding the reducing certificates is itself an SDP problem , thus we try to reduce the complexity of finding a reducing certificate at the cost of only partially simplifying the given Conic Optimization Problem.

We approximate f_i with a user-specified convex cone $f_{i,outer}$ that satisfies -

1. $f_i \subseteq f_{i,outer}$
2. Lin $f_i =$ Lin $f_{i,outer}$
3. $f_{i,outer}$ has lower search space complexity.

We can now modify $(*)$ to -

$$\begin{aligned} \text{Find} \quad & S_i \in F_{i,outer}^* - F_{i,outer}^\perp \\ \text{subject to} \quad & S_i^\perp \text{ contains } A \end{aligned}$$

The authors in [2] have proved theorems that check the mathematical validity of such a modification.

The new modified algorithm is called the **Partial Facial Reduction** algorithm and it takes an additional input of a user-specified cone approximation.

Let W be a set containing finite number of $d \times k$ rectangular matrices , the approximations $C(W)$ are such that - $C(W)^* \subseteq S_+^d \subseteq C(W)$. Four such approximations have been analysed in [2] and the following table gives us a clear picture of the final result -

Table 1: *Cone Approximations and resulting simplified complexity*

$C(W)^*$	Search	$\|W\|$
Non-negative Diagonal (D^d)	LP	$O(d)$
Diagonally Dominant (DD^d)	LP	$O(d^2)$
Scaled DD (SDD^d)	SOCP	$O(d^2)$
Factor Width-k (FW_k^d)	SDP	$O\left(\binom{d}{k}\right)$

- **Non-negative Diagonal Matrices**: D^d represents the set:

$$D^d := \{X \in \mathbb{S}^d : X_{ii} \geq 0, X_{ij} = 0 \quad \forall \quad i \neq j\}$$

- **Diagonally Dominant Matrices**: DD^d represents the set:

$$DD^d := \{X \in \mathbb{S}^d : X_{ii} \geq \sum_{j \neq i} |X_{ij}|\}$$

- **Scaled Diagonally Dominant**: SDD^d represents the set :

$$SDD^d := \{DTD | D \in D^d, D_{ii} > 0, T \in DD^d\}$$

- **Factor-width-k Matrices**: These are matrices such that k is the smallest integer for which they can written as sum of PSD matrices that are non-zero only on $k \times k$ principal submatrices.

It is easy to observe that:

$$D^d = FW_1^d \subseteq DD^d \subseteq SDD^d = FW_2^d \subseteq \cdots \subseteq FW_d^d = S_+^d$$

Another aspect of the procedure is to find those reducing certificates that minimize the dimension of the next face. In relation to the same, the following lemma has been proved in [2]:

Lemma 3.2 *Let M be a subspace of S^d , A matrix X maximising $\sum_{i=1}^{|W|}$ rank X_i over $M \cap C(W)^*$ is given by any optimal solution (X, X_i, T) to the following SDP:*

$$\begin{aligned} \textit{maximize} \quad & \sum_{i=1}^{|W|} Tr(T_i) \\ \textit{subject to} \quad & X \in M \\ & X = \sum W_i X_i W_i^T \\ & X_i \succeq T_i \\ & I \succeq T_i \succeq 0 \end{aligned}$$

The first two approximations were polyhedral. The above problem reduces to an LP in those cases.

We can now substitute this with appropriate modifications to find reducing certificates in the Partial Facial Reduction Algorithm for SDP given above. The paper[2] also discusses the important issue of recovery of dual solutions.

3.5. Algorithm for SDP

Finally we present a high-level description of the Pre-solve steps for an SDP

Input - Primal-Dual pair and Approximation Type

Algorithm -

1. Identify sequences of faces containing primal(dual) feasible set using approximations
2. Construct the reduced primal problem
3. Solve reduced primal-dual pair
4. Recover solution to primal(dual) and attempt recovery to dual(primal)

Output - Recovered solutions to Input

3.6. Computational Benefits

Computational benefits for both the pre-solve procedures have been discussed in [1] and [2] after analysing them with testing benchmarks.

4. Current Ecosystem of Presolve Routines

Currently the Pre-solve routines mentioned for LP have been implemented either in whole, in parts or with some variations in JOptimizer , lpsolve-5.5 , CPLEX , GLPK.

JOptimizer seems to implement the most among other open source alternatives but it isn't used widely or cited. CPLEX is the industry standard but not open source. GLPK has some basic presolve routines inbuilt for LP but they don't seem to be as extensive as proposed in [1].

For SDP , the authors of [2] have written matlab package frlib which interfaces with Sedumi solver. But they only seem to have implemented for the approximations of 'D' and 'DD'.

Central to Julia's success so far has been readability and its speed. Addition of the presolve feature to optimization problems would enhance Julia's performance and preference for users.

5. Implementation and Timeline

Sample Code

Presolve will by default be set to false (convention). It will be specified in the solve!() function as a parameter, and based on type of problem we will execute the presolve.

We will need to maintain a vector of the sequence of operations performed . They could possibly be identified by a tuple of unique method_id,var_change array and constraint_change array which will help us later post-solve and recover solutions to orginal problem.

The codebase of JOptimizer could serve as a starting point to look at how these functions are implemented there and then we could implement them in Convex.jl.

For lp_presolve(), we will write many layers of helper functions and there purpose is mostly self-explanatory from the function names.

1. Utility Functions:

```
remove_fixed_variables(),
sparsity_pattern(),
remove_zero_randc(),
make_lp()
```

2. Row functions:

```
check_empty_rows(),
remove_row_singletons(),
remove_duplicated_rows()
```

3. Column Functions:

```
check_empty_columns(),
remove_forcing_constraints(),
remove_dominated_constraints(),,
remove_coloumn_singleton(),
remove_dominated_columns(),
remove_duplicated_columns()
```

4. For Testing Purposes:

```
check_progrees(),
is_same_sparsity(),
compare_bounds(),
detect_lp(),
detect_sdp()
```

And, here is a rough outline of how the top level function would be implemented and called -

```
function detect_lp()
        # will set is_lp = true if just a LP
        if is_lp
        # extract inequality constraints L,U,c,A,b
end


function solve!(p::Problem,..,use_presolve=true)

        if (use_presolve && is_lp)
        new_p = lp_presolve(p::Problem)
        elseif (use_presolve && is_sdp)
        newp = sdp_presolve(p::Problem)
        end
        ...
end


function lp_presolve(p::Problem)
iteration = 0
reduction_done = false

while(!reduction_done)

        check_empty_rows
        remove_fixed_variables()
        check_progress()

        remove_row_singletons()
        check_progress()

        remove_forcing_constraints()
        check_progress()

        if(iteration < 5)
                compares_bounds()
                checkprogress()

        remove_dominated_constraints()
        check_progress()

        #Check and remove three kinds
        #Free Columns
        #Implied free Column Singletons
        #Singleton-Doubleton combos

        check_column_singletons()
        check_progress()

        #Dominated Columns
        remove_dominated_columns()
        checkProgress()

        #Duplicate Rows
        remove_duplicate_row()

        #Duplicate columns
        remove_duplicate_column()

        remove_zero_randc()
        end
        problem p = make_lp()
end
```

For writing the presolve for sdp ,we will mostly port the code from frlib package and make appropriate calls to SCS instead of SeDuMi solver.

We would have multiple Julia files (each serving a small

purpose and exporting one or two functions to be used by the main facialreduction function).

facebase.jl would contain the structure necessary to work with faces, and some utility functions to check if an element is in the dual cone, linear span etc. To compute the interesection of two faces.

facialred.jl builds the LP obtained in the feasibility problem and solves it. Similarly solve the dual problem. It uses 'generator' (or extreme-rays) for the user-given approximation.

reduceprg.jl recovers the primal and dual solutions from the reduced program.

solutil.jl has most of the utility functions which test if a matrix is PSD, implementation of the line-search for dual recovery etc.

5.1. Testing Benchmarks

The focus of the tests would be to ensure code is sufficiently optimized and accuracy is maintained and performance in general is better than without the presolve option inside Convex.jl

We shall also document the the possible differences between solution time with and without presolve in comparison to other open source solvers presolve support.

We shall test our code at significant checkpoints with the following standard problem instances.

1. Testing with LP benchmark for speed improvements - **LP-Instances**
2. Testing with SDP problem instances for speed improvements - **SDP-Instances**

5.2. Proposed Schedule

Community Bonding: Closer look at Joptimizer's implementation and frlib's implementations and discussing design decisions with Mentor.

Week 1-2: Utility and Row procedures for LP

Week 3-4: Dominated and Forcing Constraints

Week 5: Columns - Free and Implied Free

Week 6: Dominated and Forcing Columns

Week 7: Duplicated Rows and Columns

Week 8: Post-solve LP and LP Benchmark

Week 9: Utility Procedure for SDP

Week 10-11: Conic Approximation Support

Week 12-14: Partial Facial Reduction

6. Issues to tackle

Most of the *Sparse Matrix Storage* formats that exist currently are good for matrix multiplication and storage-efficient but very slow for indexing. Since our presolve procedures depend heavily on indexing, we need to come up with a workaround.

One possible solution is to store the matrix as dense if it isn't too large and execute the presolve routine. Then we can convert it to a sparse matrix after presolve so that we can now feed it to the solvers.

This of course has the downside that we aren't making use of the sparsity of the matrix. Some clever storage format using ideas like masking etc needs to be explored.

Also, when the matrix is sparse, we need to be careful about the arithmetic operations performed on the elements of the matrix to make sure they are numerically stable and consistent (for eg., dividing by small values close to zero could lead to inaccuracy).

The summations required to calculate for Column Singletons and Forcing constraints could potentially be inefficient when the matrix is sparse. We could be adding zero multiple times if we iterated through each and every index without accounting for sparsity.

This would be the trade-off to consider - exploiting the sparse structure vs efficient access and indexing.

Another issue that could arise is, if the values being subtracted were very close then this could result in loss of significance. This would mean despite the straightforward formulas discussed in [1], using modified equivalent forms could make the algorithm numerically more stable.

References

[1] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Mathematical Programming*, 71:221–245, 1995.

[2] Permenter, Frank and Parrilo, Pablo. Partial facial reduction: simplified, equivalent SDPs via approximations of the PSD cone. *arXiv:1408.4685*, 2014

[3] O'Donoghue, Brendan and Chu, Eric and Parikh, Neal and Boyd, Stephen. Operator splitting for conic optimization via homogeneous self-dual embedding. *arXiv:1312.3039*, 2013