Preprocessing for Linear Programming Problems

Ivet Galabova
Supervisor: Julian Hall

School of Mathematics
University of Edinburgh
I.L.Galabova@sms.ed.ac.uk

15th August 2017

THE UNIVERSITY *of* EDINBURGH

Table of Contents

Linear Programming (LP) Problems

A general bounded linear problem in standard form:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{L} \leq A\boldsymbol{x} \leq \boldsymbol{U} \\ & \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}, \end{aligned}$$

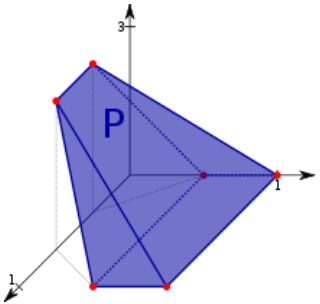
where $A \in \mathbb{R}^{m \times n}$.

- LP arise naturally
- Large numbers of variables and constraints
- Matrices are generally sparse and generated by a modelling program
- Redundancies present

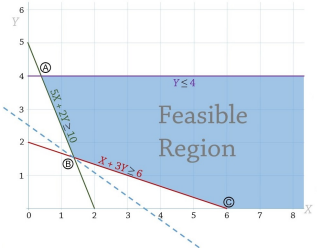
Table of Contents

Reducing dimensions

Reducing the dimension of the problem generally makes it easier to solve.



Higher dimension polytope



Lower dimension polytope

Presolve

A presolve procedure reduces an LP to another of smaller dimension.
Key properties:

- Speed
- Preservation of sparsity
- Ability to recover solution to original problem

Original problem → Presolve → Reduced problem → Solution → Postsolve → Solution to Original problem

Rules

- Apply different elimination rules consecutively
- Easy transformations
  - Empty row
  - Empty column
  - Free row
  - Fixed variable
  - Singleton row
- Keep alert for infeasibility

Rules

- Forcing constraint
  - Only way to satisfy is to push each variable to one of its bounds
  - The constraint and all variables in it are eliminated
  - The objective value and row bounds are updated
- Free column singleton
  - There is always a way to satisfy a constraint by adjusting the value of a free variable
  - Eliminate row and column

Rules

- Implied free column singleton
  - Column singleton with bounds
  - Implied bounds tighter than original bounds
  - Can eliminate like free column singleton
- Duplicate rows & columns
  - One row is a multiple of another
  - Eliminate one of two duplicate rows

Dominated rows rule

- Row bounds can be used to deduce bounds on columns

$$x_1 + x_2 + x_3 = 1, \qquad 0 \leq x_i \leq 2, \forall i \in \{1, 2, 3\}.$$

  - all variables are non-negative and row value is 1
  - implied upper bounds for $x_i, i \in \{1, 2, 3\}$ are all 1
- Implied bounds on columns lead to tighter implied row bounds
- Tighter bounds lead to more eliminations

Doubleton equation rule

- Row i is: $a_{i1}x_1 + a_{i2}x_2 = b \implies x_2 = \dfrac{b - a_{i1}x_1}{a_{i2}}$
- Modify bounds and cost of x_1
- Remove x_2 from problem
- For each row $k \neq i$

$$\sum_{j \neq 2} a_{kj}x_j \quad + \quad a_{k2}x_2$$

$$\sum_{j \neq 2} a_{kj}x_j - \frac{a_{k2}a_{i1}}{a_{i2}}x_1 + \frac{a_{k2}b}{a_{i2}}$$

  - Decrease row bounds by $\dfrac{a_{k2}b}{a_{i2}}$
  - Decrease a_{k1} by $\dfrac{a_{k2}a_{i1}}{a_{i2}}$

Implied bounds detection during presolve

- Dominated columns procedure
  - Dominated rows procedure applied to the dual problem
- During detection of forcing constraints
  - Detect redundant or infeasible constraints
  - Calculate implied bounds on column values
- During detection of dominated columns
  - Calculate new bounds on row duals

Code details

- High performance dual simplex solver `h2gmp`
- C++
- Presolve & Postsolve
- Postsolve much harder to implement

Presolve

- Linked lists for singleton columns and rows
  - Update during each elimination
  - Use column list to calculate implied bounds
- During detection of dominated columns
  - Also calculate new bounds on row duals
  - Calculate implied bounds on column values

Postsolve

- Data structure to hold eliminations
- Stack of such eliminations
- During presolve for each rule applied add element to stack
- During postsolve pop elements from stack
- Other data structures for updates to bounds and costs

Postsolve

- Dual solutions can be non-unique
- Methods to calculate bounds on row duals
- Use these bounds to choose a feasible set of dual values
- During postsolve identify basic variables
- Basis matrix must remain non-singular

Matrix changes and checking

- Matrix kept row-wise and column-wise in memory
- Both representations updated during each elimination
- Only doubleton equation rule makes changes to matrix
  - column-wise representation requires more memory
  - row-wise representation changes in place
- Changing matrix expensive and may cause fill-in
- KKT checker for development

Ordering

The order of presolve rules is very important

Remove all fixed variables
repeat
    Remove all row singletons
    Remove all forcing constraints
    ...
    Dominated columns
    ...
until (condition is met)
end

Ordering

- The order of presolve rules is very important
- Singleton column with zero cost rule
  - Singleton columns used to derive bounds on row duals
  - Rule called too early: fewer reductions
  - Rule called later: more reductions
- Different orderings may be good for different classes of problems

Ordering: Singleton column with zero cost rule

Remove all fixed variables
repeat
    Remove all row singletons
    Remove all forcing constraints
    ...
    Remove all zero cost singleton columns
    Dominated columns
    ...
until (condition is met)
end

Remove all fixed variables
repeat
    Remove all row singletons
    Remove all forcing constraints
    ...
    Dominated columns
    Remove all zero cost singleton columns
    ...
until (condition is met)
end

Presolve Summary

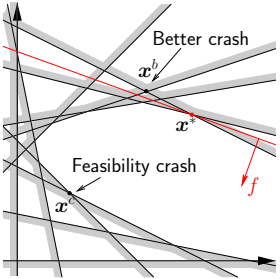
- Presolve transforms an LP into a smaller, easier to solve LP
- Beneficial due to redundancies
- Essential part of any LP solver
- What now?
  - Further reductions
  - Better implementation
  - Other preprocessing techniques

Table of Contents

Overview

A linear programming problem is

$$\begin{aligned} \min \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.} \quad & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned}$$



$$\boldsymbol{x}^0 \xrightarrow{\text{Crash}} \boldsymbol{x}^k \xrightarrow{\text{Crossover}} \boldsymbol{x}^b \xrightarrow{\text{Simplex}} \boldsymbol{x}^*$$

Motivation

- A good starting basis is essential for the solution of large problems
- The Idiot crash is a heuristic aiming to improve feasibility
- Implemented by John Forrest in CLP
- Seeking primal feasibility prior to primal simplex

$$\boldsymbol{x}^0 \xrightarrow{\text{Crash}} \boldsymbol{x}^k \xrightarrow{\text{Crossover}} \boldsymbol{x}^b \xrightarrow{\text{Simplex}} \boldsymbol{x}^*$$

General idea

The idea is to solve many small problems instead of one big one

In turn

- Fix all components of $\boldsymbol{x}$ except x_1
- Fix all components of $\boldsymbol{x}$ except x_2
- Fix all components of $\boldsymbol{x}$ except x_3
- ...

During the execution parameters are adjusted according to progress

The Idiot Crash

The Idiot Crash structure

Initialize $\boldsymbol{x}^0$
For $I \in 1..\text{nMajor}$
  For $i \in 1..\text{nMinor}$
    $\forall j \in \{1..n\} \quad \min_{x_j} g(\boldsymbol{x})$
  end

end

$$\min_{x_j} g(\boldsymbol{x})$$

$$\Updownarrow$$

Minimize a quadratic function in one variable

The function minimized

Two goals:

$$\min \boldsymbol{c}^T\boldsymbol{x} \quad \text{and} \quad \sum_i e_i(\boldsymbol{x})^2$$

where $e_i(\boldsymbol{x})$ is the infeasibility of constraint i.
The function minimized for each column:

$$\min_{x_j} \quad \mu\tilde{\boldsymbol{c}}^T\boldsymbol{x} + \sum_{i\in 1..m} e_i(\boldsymbol{x})^2 = g(\boldsymbol{x})$$

$$\Updownarrow$$

$$\min_{x_j} \quad \mu\left(c_j + \sum_i a_{ij}\lambda_i\right)x_j + \sum_{i\in 1..m} e_i(\boldsymbol{x})^2 = g(\boldsymbol{x})$$

The Idiot Crash

The Idiot Crash structure

Initialize $\boldsymbol{x}^0$, μ^0, $\boldsymbol{\lambda}^0 = \mathbf{0}$
For $I \in 1..\text{nMajor}$
  For $i \in 1..\text{nMinor}$

$$\forall j \in \{1..n\} \quad \min_{x_j} g(\boldsymbol{x}) = \mu \left(c_j + \sum_i a_{ij}\lambda_i \right) x_j + \sum_{i \in 1..m} e_i(\boldsymbol{x})^2$$

  end
  Choose new μ or $\boldsymbol{\lambda}$
end

Every few major iterations μ is reduced

Parameters

At the end of each major iteration the values of $\boldsymbol{\lambda}$ or μ are re-adjusted:

- Start with μ^0, $\boldsymbol{\lambda}^0 = \mathbf{0}$, nMinor $= 2$
- At the end of each major iteration:
  - Until 10% reduction in primal infeasibility set $\mu^{k+1} = \mu^k/10$
- Set nMinor $= 105$
  - The first major iteration with a new μ, $\boldsymbol{\lambda}$ is unchanged
  - The next few major iterations set $\lambda_i^{k+1} = \dfrac{1}{\mu} e_i(\boldsymbol{x}^k)$
  - Reduce μ and repeat

Further details

- During initial major iterations μ is bigger so $\boldsymbol{\lambda}$ ensures reducing primal infeasibility has enough weight

$$\min_{x_j} \quad \mu \left(c_j + \sum_i a_{ij} \lambda_i \right) x_j + \sum_{i \in 1..m} e_i(\boldsymbol{x})^2$$

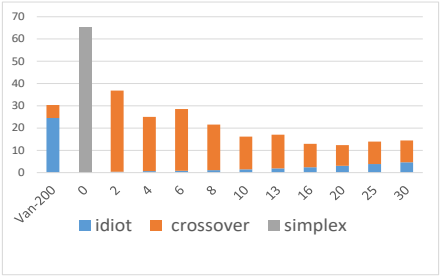
- Cycles through all columns
  - Random starting point
  - Alternating direction
- Early termination of a major iteration is possible
  - Condition on moving average of expected progress

Performance

- Generally useless
  - Single variable change limitation
- Solves Quadratic Assignment Problems (QAPs)
  - Combinatorial optimization problem, special case of facility location
  - Well known for being hard to solve
  - The Idiot crash reduces solution time significantly

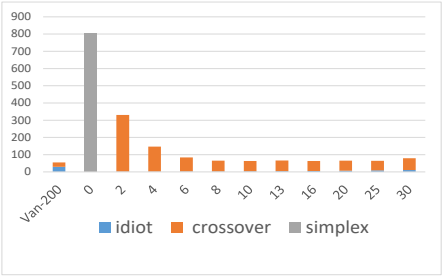
Quadratic Assignment Problems

QAP12
Solution time (s)



nMajor

QAP15
Solution time (s)



nMajor

Idiot alone

$$\begin{array}{ll} \min & f_P = \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.} & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{array} \qquad \text{(P)}$$

$$\begin{array}{ll} \max & f_D = \boldsymbol{b}^T \boldsymbol{y} \\ \text{s.t.} & A^T \boldsymbol{y} \neq \boldsymbol{c} \end{array} \qquad \text{(D)}$$

Idiot applied to (P)

- Primal feasible
- Near-optimal objective $f_P \geq f^*$

Idiot applied to (D)

- Dual feasible
- Objective $f_D \leq f^*$

Observations

				Time (s)		
Problem	LB	UB	f^*	Bounds	Idiot+Crossover	Simplex
qap08	141.39	204.57	203.50	0.55	0.61	1.80
qap12	481.50	523.60	522.89	3.85	12.64	65.21
qap15	950.05	1042.05	1040.99	11.53	64.48	803.30

- QAPs require integer solution to linearization
- Fast bounds on f^* accelerate branch & bound algorithms

Branch and bound acceleration

Branch and bound structure

Initialize stack of problems with LP relaxation of MIP
While stack is not empty
    Choose problem P from stack
    Solve P
        If solution of P is worse than best integer solution prune tree
        Else find violated constraint, add new problems to stack
end

- Fast bounds on objective value beneficial

Branch and bound acceleration

- Generally a basic solution needed at a node
- Idiot does not give basic solution
- With basic solution
  - Use crossover to get basic solution
- Without basic solution
  - Skip heuristics

Crash Summary

- Finding a good starting basis beneficial but difficult
- The Idiot crash has limitations
  - Sensitive to parameters
  - Can have disastrous effects
  - Single variable change
- Effective on some problems
- Scope for accelerating B&B

Table of Contents

Conclusion

- Both presolve and crash essential for large sparse problems
- Implementation is difficult, postsolve and crossover in particular
- No universal solution:
  - Some problems are solved more slowly with presolve or crash
  - Rules beneficial for classes of problems