Discrete Optimization – Learning by Examples

Yisu Nie

January 1, 2019

1 Linear Algebra

1.1 Matrix Decomposition

1.1.1 LU Decomposition

Lower-upper decomposition refers to the factorization of matrix $A \in \mathbb{R}^{m \times n}$ to a unit (diagonal elements equal to 1) lower triangular matrix L and an upper triangular matrix U: $A = LU$. LU decomposition can be carried out using Gaussian elimination, but proper ordering of the rows (or columns) in matrix A is needed to avoid L or U becoming singular. In practice, LU decomposition with partial pivoting allows row permutations:

A 3 by 3 example

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 4 & 3 & -1 \\ 3 & 5 & 3 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 3 & -2 & 1 \end{bmatrix}, U = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & -5 \\ 0 & 0 & -10 \end{bmatrix}$$

$$PA = LU, \tag{1.1}$$

where P is $m \times m$ permutation matrix. LU decomposition is an important vehicle for solving linear systems numerically. Starting from a generic linear system:

$$Ax = b, \tag{1.2}$$

with Eq.1.1, we obtain

$$LUx = \widetilde{b}, \quad \widetilde{b} = Pb. \tag{1.3}$$

Introducing an intermediate vector

$$z = Ux, \tag{1.4}$$

then z can be solved by triangular forward-substitution:

$$Lz = \widetilde{b}, \tag{1.5}$$

and similarly x can be solved by triangular back-substitution in Eq.1.4.

Example Here we apply LU decomposition in Julia's linear algebra package with an example that A is not a square matrix.

```
using LinearAlgebra
A = [1 2 -3 1; 2 4 0 7; -1 3 2 0]
println("LU factorize A")
lu(A)
println("LU factorize A transpose")
lu(transpose(A))
```

```
3x4 Array{Int64,2}:
  1  2  -3  1
  2  4   0  7
 -1  3   2  0
LU factorize A
LU{Float64,Array{Float64,2}}
L factor:
3x3 Array{Float64,2}:
  1.0  0.0  0.0
 -0.5  1.0  0.0
  0.5  0.0  1.0
U factor:
3x4 Array{Float64,2}:
 2.0  4.0   0.0   7.0
 0.0  5.0   2.0   3.5
 0.0  0.0  -3.0  -2.5
LU factorize A transpose
LU{Float64,Array{Float64,2}}
L factor:
4x3 Array{Float64,2}:
  1.0       0.0        0.0
 -0.333333  1.0        0.0
 -0.666667  0.571429   1.0
 -0.333333  0.285714  -0.13253
U factor:
3x3 Array{Float64,2}:
 -3.0  0.0  2.0
  0.0  7.0  0.666667
  0.0  0.0  3.95238
```

1.1.2 Cholesky Decomposition

When matrix $A \in \mathbb{R}^{m \times m}$ is symmetric and positive definite, one can decompose A into the product of a lower triangular matrix L and its transpose.

$$A = LL^{\mathrm{T}}. \tag{1.6}$$

symmetric matrix

$$A = A^{\mathrm{T}}$$

positive definite matrix

$$z^{\mathrm{T}} A z > 0, \quad \forall z \neq 0$$

1.1.3 Singular Value Decomposition

Given $m \times n$ matrix A, it can be rewritten as the product of three matrices:

$$A = U\Sigma V^{\mathrm{T}}, \tag{1.7}$$

where U is an $m \times m$ orthogonal matrix, Σ is an $m \times n$ diagonal matrix with non-negative values, and V is an $n \times n$ orthogonal matrix. The diagonal entries of Σ are the singular values, the columns of U are the left singular vectors, and the rows of V^{T} are the right singular vectors. The left singular vectors are eigenvectors of AA^{T}, the right singular vectors are eigenvectors of $A^{\mathrm{T}}A$, and the singular values are the square roots of the eigenvalues of $A^{\mathrm{T}}A$ (or equally, AA^{T}). Geometrically, as A is a linear transformation, singular value decomposition can be interpreted as a rotation(V^{T})-stretch(Σ)-rotation(U) procedure.

orthogonal matrix $A^{\mathrm{T}}\mathrm{A} = \mathrm{I}$

$$A^{\mathrm{T}}A = V\Sigma^{\mathrm{T}}U^{\mathrm{T}}U\Sigma V^{\mathrm{T}} = V\Sigma^2 V^{\mathrm{T}}$$
$$AA^{\mathrm{T}} = U\Sigma V^{\mathrm{T}}V\Sigma^{\mathrm{T}}U^{\mathrm{T}} = U\Sigma^2 U^{\mathrm{T}}$$

Example In this example, we use the svd function in Julia to decompose $A \in \mathbb{R}^{3\times4}$. Note that we use option *full* = *true* to force the program return V^{T} as a 4×4 matrix.

```
using LinearAlgebra
A = [1 2 -3 1; 2 4 0 7; -1 3 2 0]
println("Singular Value Decomposition of A")
F = svd(A, full = false);
println("U")
F.U
println("V transpose")
F.Vt
println("Sigma")
Diagonal(F.S)
```

and the result follows

```
3x4 Array{Int64,2}:
  1  2  -3  1
  2  4   0  7
 -1  3   2  0
Singular Value Decomposition of A
U
3x3 Array{Float64,2}:
 -0.265794   0.595678   -0.757972
 -0.952144  -0.0391265   0.303135
 -0.150914  -0.80227    -0.577571
V transpose
4x4 Array{Float64,2}:
 -0.232641  -0.552221   0.0570961  -0.798542
```

```
  0.338159  -0.351551  -0.869058    0.0824561
  0.156139  -0.746516   0.410184    0.500083
 -0.898414  -0.119067  -0.270607    0.324728
Sigma
3x3 Diagonal{Float64,Array{Float64,1}}:
 8.67932
         3.90259
                  2.72749
```

When use reduced size we have: $V^{\mathrm{T}} = \begin{bmatrix} -0.232641 & -0.552221 & 0.0570961 & -0.798542 \\ 0.338159 & -0.351551 & -0.869058 & 0.0824561 \\ 0.156139 & -0.746516 & 0.410184 & 0.500083 \end{bmatrix}$

2 Linear Programming

2.1 Duality

In general, duality equates two different views of the same object [2]. For example, signals can be described either in the time domain or the frequency domain. Similarly, a closed convex set Ω can be represented by a union of internal and boundary points or by taking the intersection of all supporting hyperplanes to Ω.

Convex set

for any $x_1, x_2 \in \Omega, \theta x_1 + (1-\theta)x_2 \in \Omega, \theta \in [0,1]$

Supporting hyperplane

for any $a \neq 0$, if hyperplane $a^{\mathrm{T}}x = a^{\mathrm{T}}x_0$ has the property of $a^{\mathrm{T}}x \leq a^{\mathrm{T}}x_0, \forall x \in \Omega$, then the hyperplane is a supporting hyperplane to Ω at point x_0 (x_0 is a point on the bounday of the convex set)

Example of $f(x) = ax^2$

$f^*(x^*) = \sup_x x^*x - ax^2$

$x_{argmax} = \dfrac{x^*}{2a}$, ($x_{argmax}$ maximize $f(x)$)

$f^*(x^*) = \dfrac{x^{*2}}{4a}$

2.1.1 Conjugate Function

Consider a function $f(x) : \mathbb{X} \subset \mathbb{R}^n \to \mathbb{R}$, its conjugate function is defined by

$$f^*(x^*) = \sup_{x \in \mathbb{X}} \left\{ \langle x^*, x \rangle - f(x) \right\}, \quad x^* \in \mathbb{R}^n, \tag{2.1}$$

and the conjugate function of the conjugate can be obtained:

$$f^{**}(x) = \sup_{x^* \in \mathbb{R}^n} \left\{ \langle x, x^* \rangle - f^*(x^*) \right\}, \quad x \in \mathbb{X}, \tag{2.2}$$

It follows that

$$f(x) \geq f^{**}(x). \tag{2.3}$$

From Eq. 2.1, we have $f(x) \geq \langle x^*, x \rangle - f^*(x^*)$, which implies $f(x) \geq \sup_{x^*}\{\langle x^*, x \rangle - f^*(x^*)\}$, where the right hand side is the definition of $f^{**}(x)$. However, if $f(x)$ is closed and convex, then we have $f(x) = f^{**}(x)$.

2.1.2 Dualization Using Perturbation

Given a general optimization problem $\min_{x \in \mathbb{X}} f(x)$, we introduce a perturbation function $\phi(x, y)$, where the original problem is to minimize $\phi(x, 0)$,

and the perturbed problem is to minimize $\phi(x, y)$ for some $y \neq 0$, also with respect to x. Let

$$v(y) = \inf_x \phi(x, y) \tag{2.4}$$

Using the dual form v^{**} (the conjugate of the conjugate), following Eq. 2.2:

$$v^{**}(y) = \sup_{y^*} \{\langle y, y^* \rangle - v^*(y^*)\} \tag{2.5}$$

and we have equality between the primal and the dual, $v(y) = v^{**}(y)$. When we set $y = 0$, the dual form of the optimization problem reduces to

$$v^{**}(0) = \sup_{y^*} \{-v^*(y^*)\} \tag{2.6}$$

Now we consider the convex conjugate of the pertured function $\phi(x, y)$:

$$\phi^*(x^*, y^*) = \sup_{x,y} \{\langle x^*, x \rangle + \langle y^*, y \rangle - \phi(x, y)\} \tag{2.7}$$

When we choose $x^* = 0$, we obtain

$$\begin{aligned}
\phi^*(0, y^*) &= \sup_{x,y} \{\langle y^*, y \rangle - \phi(x, y)\} && \text{(2.8a)} \\
&= \sup_y \left\{ \langle y^*, y \rangle - \inf_x \phi(x, y) \right\} && \text{(2.8b)} \\
&= \sup_y \{\langle y^*, y \rangle - v(y)\} && \text{(2.8c)} \\
&= v^*(y^*) && \text{(2.8d)}
\end{aligned}$$

Now, we can have the dual problem stated as

$$\phi(\mathbf{x}, \mathbf{y} = \mathbf{0}) = v(0) = v^{**}(0) = \sup_{y^*} \{-v^*(y^*)\} = \sup_{\mathbf{y}^*} \{-\phi^*(\mathbf{x}^* = \mathbf{0}, \mathbf{y}^*)\} \tag{2.5}$$

Assuming strong duality here $v(0) = v^{**}(0)$. Otherwise, for weak duality $v(0) > v^{**}(0)$. Also, keeping the minus sign in the end for later convenience in deriving Lagrangian.

Lagrangian Dual Consider the general form of a convex optimization problem as follows:

$$\begin{aligned}
&\min\ f(x) && \text{(2.6a)} \\
&s.t.\quad h(x) = 0 && \text{(2.6b)} \\
&\qquad g(x) \leq 0 && \text{(2.6c)}
\end{aligned}$$

Every dualization in optimization is associated with a perturbation function

and a perturbed problem

$$\min\ f(x) \tag{2.7a}$$
$$s.t.\quad h(x) + y_1 = 0 \tag{2.7b}$$
$$g(x) + y_2 \leq 0 \tag{2.7c}$$

With the pertubation specified in Eq. 2.7, we define function ϕ as

$$\phi(x, y_1, y_2) = \begin{cases} f(x) & h(x) + y_1 = 0, g(x) + y_2 \leq 0 \\ \infty & \text{otherwise} \end{cases} \tag{2.8}$$

By definition in Eq. 2.7, one writes

$$-\phi^*(x^* = 0, y_1^*, y_2^*) = -\sup_{\substack{x \in \mathbb{X} \\ h(x) + y_1 = 0 \\ g(x) + y_2 \leq 0}} \{\langle 0, x\rangle + \langle y_1^*, y_1\rangle + \langle y_2^*, y_2\rangle - f(x)\} \tag{2.9a}$$
$$= -\sup_{x, z \geq 0} \{\langle y_1^*, -h(x)\rangle + \langle y_2^*, -g(x) - z\rangle - f(x)\} \tag{2.9b}$$
$$= \begin{cases} \inf_x \{f(x) + \langle y_1^*, h(x)\rangle + \langle y_2^*, g(x)\rangle\} & \mathrm{y}_2^* \geq 0 \\ \infty & \text{otherwise} \end{cases} \tag{2.9c}$$

Sign of y_2^* has to be non-negative. Otherwise, $\langle y_2^*, z\rangle$ becomes unbounded below.

To this end, we recogozie the **Lagrangian function** (with slight variation in notation from above):

$$\mathcal{L}(x, \mu, \lambda) = f(x) + \mu^{\mathrm{T}} h(x) + \lambda^{\mathrm{T}} g(x), \tag{2.4}$$

and the **Lagrangian dual function** is

$$g(\mu, \lambda) = \inf_x \left\{ f(x) + \mu^{\mathrm{T}} h(x) + \lambda^{\mathrm{T}} g(x) \right\}, \tag{2.5}$$

and the **Lagrangian dual problem** is

$$\sup_{\mu, \lambda} g(\mu, \lambda). \tag{2.6}$$

2.1.3 Linear Programming Duality

Now we apply Lagrangian duality to derive the dual of linear program:

$$\max\ c^{\mathrm{T}} x \tag{2.7a}$$
$$s.t.\quad Ax \leq b \tag{2.7b}$$
$$x \geq 0 \tag{2.7c}$$

Define the Lagrangian:

$$\mathcal{L}(x, \lambda, \mu) = -c^{\mathrm{T}}x + \lambda^{\mathrm{T}}(Ax - b) - \mu^{\mathrm{T}}x$$
$$\lambda \geq 0, \quad \mu \geq 0$$

Stationary condition w.r.t. x, requires

$$-c^{\mathrm{T}} + \lambda^{\mathrm{T}}A - \mu^{\mathrm{T}} = 0$$

and the Lagrangian dual function becomes

$$g(\lambda, \mu) = \left(-c^{\mathrm{T}} + \lambda^{\mathrm{T}}A - \mu^{\mathrm{T}}\right)x - \lambda^{\mathrm{T}}b = -\lambda^{\mathrm{T}}b$$

and the dual problem using the Lagrangian is

$$\max\ -\lambda^{\mathrm{T}}b \tag{2.8a}$$
$$s.t. \quad -c^{\mathrm{T}} + \lambda^{\mathrm{T}}A - \mu^{\mathrm{T}} = 0 \tag{2.8b}$$
$$\lambda \geq 0, \quad \mu \geq 0 \tag{2.8c}$$

Rearranging terms and substitute out μ, we end up with the classic form of the dual to the liner program 2.7:

$$\min\ b^{\mathrm{T}}\lambda \tag{2.9a}$$
$$s.t. \quad A\lambda \geq c \tag{2.9b}$$
$$\lambda \geq 0 \tag{2.9c}$$

3 Lagrangian Relaxation COMPLETED

3.1 Theory

3.1.1 Basic Theorem

Lagrangian relaxation is a constraint decomposition method, where model constraints are separated to an easy and a hard group. The reduced problem with the subset of the easy constraints can be solved rapidly in practice. Consider an integer program as follows:

Aslo spell as Lagrangean relaxation

$$z_{IP} = \min\ cx \tag{3.1a}$$
$$s.t. \quad Ax \geq b \tag{3.1b}$$
$$Dx \geq e \tag{3.1c}$$
$$x \in \mathbb{Z}_+^n \tag{3.1d}$$

The complicating (bad) constraints in Eq. (3.1b) can be cast to the objective function with a pricing variable u to try to ensure feasibility.

$$z_{LR} = \min\ cx + u(b - Ax) \tag{3.2a}$$
$$s.t. \quad Dx \geq e \tag{3.2b}$$
$$x \in \mathbb{Z}_+^n \tag{3.2c}$$

z_{LR} is called the Lagrangian relaxation of the original integer program z_{IP}, and z_{LR} is an underestimate of z_{IP}. For a quick proof, assuming x^* is the

optimal solution of the integer program:

$$z_{IP} = cx^* \geq cx^* + u(b - Ax^*) \tag{3.3a}$$ x^* is feasible in IP $\rightarrow b - Ax^* \leq 0$ and $u \geq 0$

$$cx^* + u(b - Ax^*) \geq cx + u(b - Ax) = z_{LR} \tag{3.3b}$$ x^* is feasible in LR

$$z_{IP} \geq z_{LR} \tag{3.3c}$$

To obtain the best lower bound, we search the space of the dual vector u, and introduce the Lagrangian dual:

$$z_{LD} = \max_{u \geq 0} z_{LR} = \max_{u \geq 0} \min_{x} cx + u(b - Ax) \tag{3.3da}$$

$$s.t. \quad Dx \geq e \tag{3.3db}$$

$$x \in \mathbb{Z}^n_+ \tag{3.3dc}$$

3.1.2 Advanced Theorem

The remaining constraint set after the transformation can be denoted as $Z = \{\mathbb{Z}^n_+ : Dx \geq e\}$. If this set contains a finite number of (extreme or integer) points $t = 1, \ldots, T$. The Lagrangian dual can be viewed alternatively as:

$$z_{LD} = \max_{u \geq 0} z_{LR} = \max_{u \geq 0} \min_{t=1,\ldots,T} cx^t + u(b - Ax^t). \tag{3.3e}$$

This can be rewritten to a linear program:

$$z_{LD} = \max \mu \tag{3.3fa}$$

$$s.t. \quad \mu - u(b - Ax^t) \geq cx^t, \quad t = 1, \ldots, T, \tag{3.3fb}$$ μ is the smallest value among t

$$u \geq 0, \mu \in \mathbb{R}^1 \tag{3.3fc}$$

Introducing λ^t as the dual variables for the constraints, the dual of the linear program can be deduced:

$$z_{LD} = \min \sum_{t=1}^{T} (cx^t)\lambda^t \tag{3.3ga}$$

$$s.t. \quad \sum_{t=1}^{T} \lambda^t = 1 \tag{3.3gb}$$

$$\sum_{t=1}^{T} (Ax^t)\lambda^t \geq b, \tag{3.3gc}$$ $\sum_{t=1}^{T} \lambda^t b = b$

$$\lambda \in \mathbb{R}^T_+ \tag{3.3gd}$$

With this form, the Lagrangian dual can be further expressed as

$$z_{LD} = \min\{cx : \ Ax \geq b, x \in conv(Z)\} \tag{3.3h}$$

The convex hull of set Z is defined by the extreme points x^t, noted as

$$conv(Z) = \{x : x = \sum_{t=1}^{T} x^t \lambda^t, \sum_{t=1}^{T} \lambda^t = 1, \lambda^t \geq 0\} \tag{3.3i}$$

Comparing the above formulation in Eq. 3.3h with the IP problem in Eq. 3.1, one can view the Lagrangian dual problem as optimizing the objective function within the intersection of the complicating constraints and the convex hull of the easy constraints.

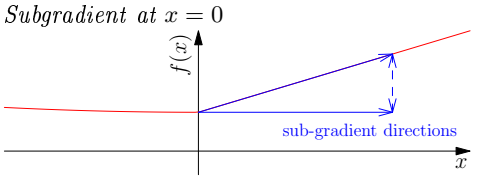
3.2 Algorithm

3.2.1 Subgradient Algorithm

The Lagrangian dual (LD) function in Eq. (3.3e) is a maximization problem over a piece-wise affine and concave function. However, the function is non-differentiable [6]. The subgradient algorithm can be used to solve this problem. The algorithm is stated in Algorithm 1:

1 Initial guesses of dual solution $u^0 = 0$ and primal solution x^0 ;
2 Set iteration index $t = 1$;
3 **do**
4 Solve Lagrangian relaxation problem in Eq. (3.1) to obtain the lower/dual bound and optimal solution x^t ;
5 Compute subgradient $s^t = b - Ax^t$;
6 Update dual solution $u^{t+1} = \max\{0, u^t + \gamma^t s^t\}$. This update means moving along the subgradient direction with a step size controlled by coefficient γ^t;
7 **while** *Change in the multiplier* $\left|u^t - u^{t-1}\right| > \tau$ *(τ is a small positive number) and* $t < t^{max}$ *(t^{max} represents maximum iteration number)*;
8 Best solution at algorithm termination u^{t^*}, x^{t^*}.

Algorithm 1: Subgradient algorithm for solving the Lagrangian dual



For any function $f(x)$ (no need to be convex), if

$$f(x) - f(x_0) \geq s^T(x - x^0), \quad \forall x$$

Then s^T is a subgradient at $x = x_0$.

The subgradient algorithm was first developed by Shor [7]. The convergence property is governed by the step size law. A formula that has been proved effective was proposed in [5]:

$$\gamma^t = \alpha^t \frac{z^*_{IP} - z_{LR}(u^t)}{\|b - Ax^t\|};$$

$$\alpha^t = \begin{cases} 2, & \text{t=0;} \\ 0.5\alpha^{t-1}, & z_{LR}(u^t) \ N^{\text{th}} \text{ decrease failure} \\ \alpha^{t-1}, & \text{else.} \end{cases}$$

Here, $\|b - Ax^t\| := \sum_i (b_i - \sum_j a_{ij} x^t)^2$. And z^*_{IP} can be replaced by any known values (obtained from feasible solutions). This update formula does not guarantee convergence to optimality [3].

3.3 Example

The illustrative example we use is modified from the example given by Fisher [3]. The implementation of the example IP in Julia/JuMP is shown as below

```
using JuMP
@time begin
```

```
c = [-16, -10, 0, -4]
A = [-8, -2, -1, -4]
B1 = [-1, -1, 0, 0]
B2 = [0, 0, -1, -1]

ip = Model()
@variable(ip, x[1:4], Bin)
@objective(ip, Min, dot(c,x))
@constraint(ip, dot(A,x) >= -10)
@constraint(ip, dot(B1,x) >= -1)
@constraint(ip, dot(B2,x) >= -1)
opt = solve(ip)
end
print(ip)
println("Model status: ", opt, " | Objective value: ", getobjectivevalue(ip))
println("x = ", getvalue(x))
```

The IP can be solved to optimality using open-source solver CBC [4] from COIN-OR, with the best objective function value at -16.

Computational Infrastructure for Operations Research (https://www.coin-or.org/) website hosts a collection of open-source software for scientific computing, in particular mathematical optimization.

```
  1.910062 seconds (611.85 k allocations: 26.214 MB, 0.38% gc time)
Min -16 x[1] - 10 x[2] - 4 x[4]
Subject to
 -8 x[1] - 2 x[2] - x[3] - 4 x[4] >= -10
 -x[1] - x[2] >= -1
 -x[3] - x[4] >= -1
 x[i] in {0,1} for all i in {1,2,3,4}
Model status: Optimal | Objective value: -16.0
x = [1.0,0.0,0.0,0.0]
```

The Lagrangian subproblem is created by relaxing the first constraint—it is the complicating constraint since all the decision variables appear—and shift the relaxed constraint to the objective function with the penalty factor u. We set the initial guess of $u = 0$, which in fact is equivalent to dropping the constraint.

```
using JuMP
c = [-16, -10, 0, -4]
A = [-8, -2, -1, -4]
B1 = [-1, -1, 0, 0]
B2 = [0, 0, -1, -1]
u = 0

lr = Model()
@variable(lr, x[1:4], Bin)
@objective(lr, Min, dot(c,x) + u*(-10 - dot(A,x)))
@constraint(lr, dot(B1,x) >= -1)
@constraint(lr, dot(B2,x) >= -1)
```

```
opt = solve(lr)
print(lr)
println("u = ", u)
println("Model status: ", opt, " | Objective value: ", getobjectivevalue(lr))
println("x = ", getvalue(x))
```

The solution we obtain with the relaxed problem is lower than the true optimum as expected, and the optimal value of the decision variable is different.

```
Min -16 x[1] - 10 x[2] - 4 x[4]
Subject to
 -x[1] - x[2] >= -1
 -x[3] - x[4] >= -1
 x[i] in {0,1} for all i in {1,2,3,4}
u = 0
Model status: Optimal | Objective value: -20.0
x = [1.0,0.0,0.0,1.0]
```

To search for the true optimal solution with the subgradient algorithm, we introduce the relaxed subproblem:

```
  using JuMP
  using Gadfly
  using DataFrames

  # model parameters
  c = [-16, -10, 0, -4]
  A = [-8, -2, -1, -4]
  B1 = [-1, -1, 0, 0]
  B2 = [0, 0, -1, -1]

  function solve_lagrangian_sub(u)
      lr = Model()
      x = @variable(lr, [1:4], Bin)
      @objective(lr, Min, dot(c,x) + u*(-10 - dot(A,x)))
      @constraint(lr, dot(B1,x) >= -1)
      @constraint(lr, dot(B2,x) >= -1)
      opt = solve(lr)
      return opt, getobjectivevalue(lr), getvalue(x)
  end

  function calc_step(t, u_old, s, PB, lr_obj, lr_obj_old, alpha_old, no_impr)
      # check objective value improvement
      if lr_obj < lr_obj_old
          no_impr = no_impr + 1
      else
          no_impr = 0
      end
      # calculate step size alpha
      if no_impr > 0
```

```
            alpha = 0.5*alpha_old
        else
            alpha = alpha_old
        end
        # calculate gradient
        g = (PB - lr_obj)/s^2
        u = max(0, u_old + s*alpha*g)
        t = t + 1
        return u, t, alpha, no_impr
    end

    t = 0
    LB = []
    UB = []
    while t <= 200
        # first iteraion
        if t == 0
            u = 0
            _, obj, x = solve_lagrangian_sub(u)
            alpha = 2
            obj_old = obj
            no_impr = 0
            PB = 0
            best = Inf
        end
        append!(LB, obj)
        # calculate subgradient and check convergence
        s = -10 - dot(A, x)
        if abs(s) < 1e-3
            break
        end
        # update best feasible solution
        if s <= 0
            best = min(best, obj-u*s)
        end
        append!(UB, best)
        # update u
        u, t, alpha, no_impr = calc_step(t, u, s, PB, obj, obj_old, alpha, no_impr)
        # record previous objective function value
        obj_old = obj
        # solve Lagrangian subproblem
        opt, obj, x = solve_lagrangian_sub(u)
    end

df1 = DataFrame(x=2:length(UB), y=convert(Array{Float64,1}, UB[2:end]), bound="Upper")
df2 = DataFrame(x=2:length(LB), y=convert(Array{Float64,1}, LB[2:end]), bound="Lower")
df = vcat(df1, df2);

  my_plot = plot(df, x=:x, y=:y, color=:bound, Geom.step, Guide.xlabel("Iteration"), Guide.ylabel("Objective"));
  draw(PDF("./images/lagrangian-relaxation-sub-gradient-converge.pdf", 3inch, 3inch), my_plot)
```

After 200 iterations, the best lower bound obtained is -18, and the best

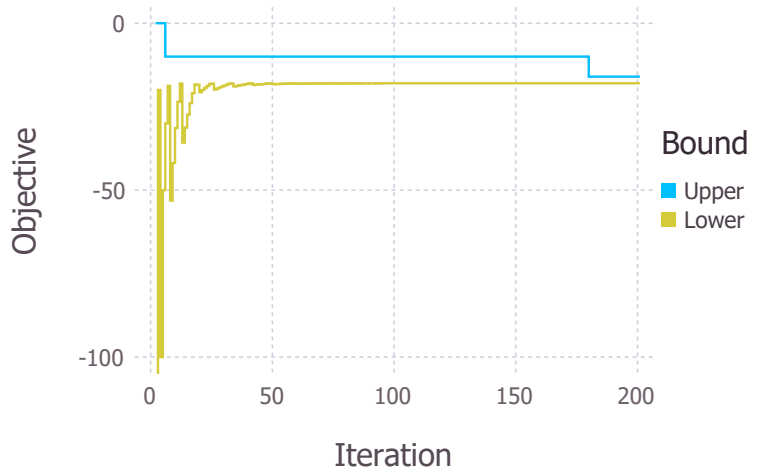


Figure 1: Convergence plot of the subgradient algorithm

upper bound is -16 (which is the optimal solution). It is worth noting that the Lagrangian relaxation is as good as the linear programming relaxation of the original IP. LP relaxation is obtained by solving the same IP problem with relaxed integrality constraints, where in Julia/JuMP we can use `{solve(model, relaxation=true)`.

4 Benders' Decomposition WAIT

4.1 Theory

4.1.1 Basic Theorem

Benders' Decomposition is a variable-based decomposition scheme, firstly proposed to address mixed-integer programs (MIPs) [1]. Consider a typical MIP in the follwing form:

$$z_{MIP} = \min\ cx + dy \tag{3.3aa}$$
$$s.t.\quad Ax + By \geq e \tag{3.3ab}$$
$$x \in \mathbb{Z}_+^m, y \in \mathbb{R}_+^n \tag{3.3ac}$$

Benders' Decomposition algorithm for MIPs is based on projection methods, where the feasible region of the MIP $Q_0 = \{(x, y) \in \mathbb{Z}_+^m \times \mathbb{R}_+^n\}$ is reformulated to a lower dimensional domain with only the discrete variables and a single continuous variable $Q_r = \{(x, \sigma) \in \mathbb{Z}^m \times \mathbb{R}\}$. To derive such a pro-

Project of polyhedron P

For $P = \{(x, y) \in \mathbb{R}^m \times \mathbb{R}^n\}$,

Projection onto the x space

$Proj_x(P) = \{x \in \mathbb{R}^m : \exists y \in \mathbb{R}^n \text{that} (x, y) \in P\}$

jection, consider solving the the subproblem as a linear problem with integer variables fixed at $x = \bar{x}$:

$$z_{SLP} = \min dy + \cancel{c\bar{x}} \tag{3.3ba}$$
$$s.t. \quad By \geq e - A\bar{x} \tag{3.3bb}$$
$$y \in \mathbb{R}^n_+ \tag{3.3bc}$$

Dropped constant term with fixed integers

Now consider the dual problem of z_{SLP}:

$$z_{DSLP} = \max u(e - A\bar{x}) \tag{3.3ca}$$
$$s.t. \quad uB \leq d \tag{3.3cb}$$
$$u \in \mathbb{R}^p_+ \tag{3.3cc}$$

Feasible region does NOT depend on x value

From LP duality, if z_{SLP} is feasible, it follows that

$$\sigma = z_{SLP} = z_{DSLP} = \max_{t=1,\ldots,T} u^t(e - A\bar{x}), \tag{3.3d}$$

where u^t are the extreme points of $U = \{u \in \mathbb{R}^p_+ : uB \leq d\}$. On the other hand, infeasible z_{SLP} suggests that $\exists v \in V$ such that $v(e - A\bar{x}) > 0$. The infeasible instances can be suppressed by augmenting constriants $v(e - A\bar{x}) \leq 0, \forall v \in V$, which are equivalent to

This is a result from Farkas Lemma and $V = \{v \in \mathbb{R}^p_+ : uB \leq 0\}$

Extreme rays of V and U are the same if they are not empty.

$$v^r(e - A\bar{x}) \leq 0, \quad r = 1, \ldots, R, \tag{3.3e}$$

when v^r are the extreme rays of V. To this end, the reformulated MIP problem can be stated as

$$z_{RMIP} = \min cx + \sigma \tag{3.3fa}$$
$$s.t. \quad u^t(e - Ax) \leq \sigma, \quad t = 1, \ldots, T \tag{3.3fb}$$
$$v^r(e - Ax) \leq 0, \quad r = 1, \ldots, R \tag{3.3fc}$$
$$x \in \mathbb{Z}^m_+ \tag{3.3fd}$$

σ is the maximum as shown in Eq. (3.3d)

Constraints apply to all $x \in \mathbb{Z}^m_+$

If the orginal problem z_{MIP} is feasible and bounded, z_{MIP} and z_{RMIP} are equivalent. However, the number of constraints in the reformulated problem Eq. 3.3f are very large in genernal. A typical strategy is to relax the feasible region by considering a subset of extreme points and rays, and gradually add constraints to improve the relaxation in an iterative procedure. The relaxed version of z_{RMIP} is employed the master problem in the Benders'

Exhaustive enumeration over extreme points and rays of $U = \{u \in \mathbb{R}^p_+ : uB \leq d\}$ is very expensive, if not prohitive.

Decomposition scheme, denoted as

$$z_{MP} = \min\ cx + \sigma \tag{3.3ga}$$
$$s.t.\quad u^t\,(e - Ax) \le \sigma, \quad t = 1, \ldots, T_r \tag{3.3gb} \qquad T_r \le T$$
$$v^r\,(e - Ax) \le 0, \quad r = 1, \ldots, R_r \tag{3.3gc} \qquad R_r \le R$$
$$x \in \mathbb{Z}_+^m \tag{3.3gd}$$

4.1.2 Extended Forms

Logic Based Benders' Decomposition

Generalized Benders' Decomposition

4.2 Algorithm

Benders' Decomposition is a flexible solution paradigm that can be implemented in a various forms. In the following discussion, we assume the use of the dual form in subproblems.

4.2.1 Classic Implementation

The classic implementation presented in the following, iterates over the master problem z_{MP} and the subproblem z_{DSLP}.

4.2.2 Modern Algorithm

A major disadvantage of the classic Benders' method is that the master problem is solved multiple times, where previously eliminated nodes can be revisited as the branch-and-cut trees are constructed from scratch each iteration. In contrast, the modern approach avoids rework by solving the master problem only once. The modern Benders' approach relies on modern MILP solvers with programming functionalities such as callbacks and lazy constraints.

The modern implementation of Benders' approach can be stated as follows:

Note that in Algorithm 3, one can manage the frequency of callback for generating lasy constraints and also choose to apply a limited memory to the active set of the lasy constraints.

Dummy example for callback in Julia

```
function say_name(firstname, lastname,
                  callback::Function)
    println(firstname)
    callback()
end

function say_lastname(lastname)
    println(lastname)
end

say_name("John", "Doe", say_lastname)
```

Lazy constraints

A portion of constraints can be removed from the full model specification and put into a pool of the so-called *lazy constraints*. The lazy constraints are temporarily ignored by the optimization solver, until a solution is generated. At such a solution, the lazy constraints are examined and any violated ones

Set iterator $i = 1$ and initial guess of integer assignment x^1;
Set the upper bound $UB = \infty$ and the lower bound $LB = -\infty$;
while *Gap between bounds* $|UB - LB| > \tau$ *(τ is a small positive number)* **do**
    Solve the subproblem in Eq. (3.3c) at x^i to obtain the lower/dual bound and optimal solution x^t ;
    if *optimal solution at* u^i **then**
        Add an optimality cut $u^i (e - Ax) \leq \sigma$ to z_{MP};
        Update $UB = \min\{UB, u^i(e - Ax^i)\}$
    else if *unbounded solution* **then**
        Add a feasibility cut $v^i (e - Ax) \leq 0$ to z_{MP} ;
    Solve the master problem in Eq. (3.3g) with generated cuts, with the optimal solution x^* and σ^* ;
    Update $LB = cx^* + \sigma^*$;
    Set $i = i + 1$, $x^i = x^*$;
end
Best integer solution x^* obtained;
Solve the subproblem Eq. (3.3b) at x^* to obtain optimal y^*

Algorithm 2: Classic Benders' Decomposition algorithm

Set iterator $i = 1$ and initial set of active constraint $\mathcal{A}^1 = \varnothing$;
do
    Run branch-and-cut for the master problem z_{MP} with the current active constraint set $\mathcal{A}^i$;
    At integer incumbent x^i, trigger callback to solve the subproblem in Eq. (3.3c) ;
    if *optimal solution at* u^i **then**
        Add a lazy constraint $u^i (e - Ax) \leq \sigma$ to the active set;
    else if *unbounded solution* **then**
        Add a lazy constraint $v^i (e - Ax) \leq 0$ to the active set;
    Set $i = i + 1$;
until *Optimality condition with no lazy constraints violation*;
Best integer solution x^* obtained;
Solve the subproblem Eq. (3.3b) at x^* to obtain optimal y^*

Algorithm 3: Modern Benders' Decomposition algorithm

4.3 Example

The example used here is borrowed from [6].

$$\begin{aligned} \min \quad & -4x_1 - 7x_2 - 2y_1 - 0.25y_2 + 0.5y_3 && \text{(3.3ha)} \\ s.t. \quad & -2x_1 - 3x_2 - 4y_1 + y2 - 4y_3 \geq -9, && \text{(3.3hb)} \\ & -7x_1 - 5x_2 - 12y_1 - 2y_2 + 4y_3 \geq -11, && \text{(3.3hc)} \\ & x \leq 3, x \in \mathbb{Z}^2_+, y \in \mathbb{R}^3_+. && \text{(3.3hd)} \end{aligned}$$

5 Bibliography

References

[1] Jacques F Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische mathematik*, 4(1):238–252, 1962.

[2] Dimitri P Bertsekas, Angelia Nedi, Asuman E Ozdaglar, et al. *Convex analysis and optimization*. Athena Scientific, 2003.

[3] Marshall L Fisher. An applications oriented guide to lagrangian relaxation. *Interfaces*, 15(2):10–21, 1985.

[4] John Forrest and Robin Lougee-Heimer. Cbc user guide. *INFORMS Tutorials in Operations Research*, pages 257–277, 2005.

[5] Michael Held, Philip Wolfe, and Harlan P Crowder. Validation of subgradient optimization. *Mathematical programming*, 6(1):62–88, 1974.

[6] Michael Jünger, Thomas M Liebling, Denis Naddef, George L Nemhauser, William R Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A Wolsey. *50 Years of integer programming 1958-2008: From the early years to the state-of-the-art*. Springer Science & Business Media, 2009.

[7] NZ Shor. An application of the method of gradient descent to the solution of the network transportation problem. *Materialy Naucnovo Seminara po Teoret i Priklad. Voprosam Kibernet. i Issted. Operacii, Nucnyi Sov. po Kibernet, Akad. Nauk Ukrain. SSSR, vyp*, 1:9–17, 1962.