SFWR ENG 4O03

Kemal Ahmed
Dr. Deza
Fall 2015

Contents

Linear Programming

Linear Program: an optimization problem in which the objective function is linear and each constraint is a linear inequality or equality

Decision variables: describe our choices that are under our control
Objective function: describes a criterion that we wish to max/minimize; doesn't have an in/equality
e.g. max 40x + 30y

Integer linear program: a linear program that only deals with integers

Constraints: describe the limitations that restrict our choices for our decision variables, always *inequalities*.
Free: no constraints

Basic variable: the variables corresponding to the identity matrix, usually have to be set to 0
Non-basic variable: …not basic variables

Converting constraints to equalities

Slack variable: basic variable greater than constraint, added to turn inequalities into equalities
Surplus variable: equation variable less than constraint, subtracted

Optimal Solution: either a maximum or minimum of the objective function based on constraints
Basic Solution: a solution which has as many slack variables as basic variables
Basic Feasible Solution: all basic variables are non-negative

- Unique
- obtained by setting the non-basic variables to 0

Standard form: when you take inequalities and use slack variables to turn them into equalities.

- Note: all variables need to be ≥ 0.
- All remaining constraints are expressed as equality constraints.

e.g.)

$2x_1 + 4x_2 - x_3 - x_4 \quad \geq 1$
$2x_1 + 4x_2 - x_3 - x_4 + s \quad = 1$

Polytopes

Convex set: all points lie on a common plane, i.e. no bulges! Intersections of convex sets are also convex sets

Hyperplane: a hyperplane in R^x is a shape in R^{x-1}, e.g. line in R^2

- cuts a space in half
- can't be R^{x-2}: you can't use a rope to cut a room in half

Half-Space: one of the halves of a space that has been split by a *hyperplane*. Each *half-plane* is represented by an inequality.

Open Half-Space: $a_1x_1 + a_2x_2 + \ldots + a_nx_n > b$

Closed Half-Space: includes the *hyperplane* separating the two *half-spaces*
$a_1x_1 + a_2x_2 + \ldots + a_nx_n \geq b$

Polyhedron: the intersection of finitely many *half-spaces*

Polytope: a bounded *polyhedron*, i.e. flat slides

[x_k^*]: optimal point to k^{th} LP, farthest point from the hyperplane the half-space (an LP) is associated with, i.e. the solution of the LP

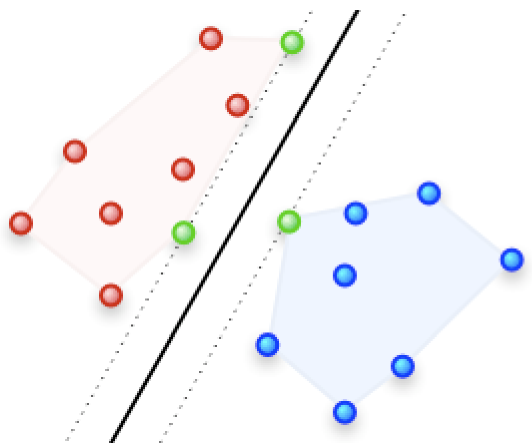
Full-dimensional: a polytope that is an n-dimensional object in R^n.

- One way to prove that P is not full dimensional is to exhibit a hyperplane $H = \{x \in \mathbb{R}^n: \alpha^T x = \beta\}$ satisfying $P \subset H$ (with $|\alpha| \neq 0$)
- One way to prove that P is full dimensional is to exhibit a point $\overline{x}$ satisfying $a_i^T \overline{x} < b_i$ for i = 1, 2,…, m.

Convex Polytope: a polytope consisting of flat planes, i.e. only *convex sets*

Support Vectors: data points that lie closest to the decision surface (or hyperplane); they support and define the edges of the decision surface, making them the most

Support Vector Machine (SVM): calculating a function based on two sets of points by determining the plane between them that differentiates them. This is determined by finding the optimal points of all possible *hyperplanes* separating the two data sets



https://www.youtube.com/watch?v=YsiWisFFruY

Graphical Method

1. Sketch the region corresponding to the system of constraints. The points inside or on the boundary of the region are the *feasible solutions*.
2. Find the vertices of the region.
3. Test the objective function at each of the vertices and select the values of the variables that optimize the objective function. For a bounded region, both a minimum and maximum value will exist. For an unbounded region, if an optimal solution exists, then it will occur at a vertex.

Simplex Method: Maximization

Simplex Method: useful for solving linear optimization problems cheaply

- Cannot be done with **strict inequalities**, i.e. when there is no possibility of being equal
- Can only work if your objective function is in *standard form*

Simplex Tableau: visual representation of stuff

- The *basic variables* can be identified if they have a column with one row of 1 and the rest of the rows are 0's. The value of the variable is at the row with the 1.
- The objective row is going to identify the constants for the new equation. You should see 0's in the columns that are non-basic.

- The first column (if used) is only an indicator of the existence of the variable you're trying to min/maximize, i.e. 0's for all rows, except for the objective function
- RHS must be ≥ 0

Process:

1. You'll have as many slack variables as you have constraint equations.
2. Find the column with the smallest coefficient (< 0) in the objective function. That column is called the **pivot column**. The **entering variable** is the variable with the smallest coefficient.
3. **Minimum test**: find the row with the smallest **departing variable** or **exiting variable**, i.e. RHS/x_{pivot}. That row is called the **pivot row**. x_{pivot} must be ≥ 0
4. The intersection of the pivot row & column is called the **pivot point**.
5. If your pivot point ≠ 1, divide your row out by the value of your point
6. Use row operations, i.e. Gauss-Jordan to make the other elements in the pivot column 0.
7. Go to step 2, until objective function is all ≥ 0.

Simplex: Minimization

To minimize a function, we just oppositize the problem so we can use the maximization technique on it. You'll see. Just remember that we minimize [w] & maximize [z] AND minimize is (vars ≥ 0), while maximize is (vars ≤ 0). I'll explain using an example:

e.g.)

$w = 0.12x_1 + 0.15x_2$

$60x_1 + 60x_2 \geq 300$

$12x_1 + 6x_2 \geq 36$

$10x_1 + 30x_2 \geq 90$

1. Ignore slack variables for now. Make a matrix with just the variables you have.

w	x_1	x_2	
0	60	60	300
0	12	6	36
0	10	30	90
1	–0.12	–0.15	0

2. Find the transpose of this matrix

60	12	10	–0.12
60	6	30	–0.15
300	36	90	0

This gives us:

$z = 300y_1 + 36y_2 + 90y_3$

$60y_1 + 12y_2 + 10y_3 \leq 0.12$

$60y_1 + 6y_2 + 30y_3 \leq 0.15$

$300y_1 + 36y_2 + 90y_3 \leq 0$

Notice how the x's are now y's? Yeah I know you did. Well now, since you turned this into a maximization problem, what are you waiting for? Go to the maximization section!

Phase Simplex

This is useful for when you have a mix of constraints that are maximum and minimum constraints.

Artificial Variable [y]: since you can't have negative variables ($x_1, x_2 \geq 0$), you can't just use a regular slack variable

Phase I

1. Replace all negative slack variables with artificial variables
2. Replace objective function with $w = -\Sigma y_i$
3. Isolate your artificial variables in your constraint equations,
    a. e.g. $2x_1 + x_2 - x_3 - x_4 + y_2 = 10 \Rightarrow y_2 = 10 - 2x_1 - x_2 + x_3 + x_4$
4. Replace your y's in your objective function with the isolated artificial variables, then move the RHS's to the new RHS
    a. e.g. for $x_1 + x_2 - x_3 - x_4 + y_2 = 10$ & $-x_2 + x_4 + y_3 = 10$, $w - 2x_1 + x_3 = -20$
5. Treat as maximization.

Phase II

Oh no!

Bland's Rule

Bland's Rule: a way of guaranteeing that you don't repeat going over the same variables (a cycle) by picking the smallest (or most negative) number

Algorithms

See SFWR ENG 2C03 Summary.

Bellman-Ford vs Dijkstra's:

- Dijkstra's omits the possibility that past nodes can be improved.
- Bellman-Ford makes sure that old nodes have been covered. If you have already looked at a node, but the minimum path to the node changes, you have to re-look at the node as well as all nodes connected to it.

Dijkstra's: Shortest path

1. Use BFS to find, relax all paths connected to each node
2. At the end, show the path

Note: when doing Bellman-Ford:

1. Make a new node
2. If the value of a node changes, redo relaxations to that node
3. If still changing at N–1, it's a negative cycle

Sp

Knapsack

Can be represented as a linear program:

For a set of n items $x_{0..n}$, weights, $w_{0..n}$, and costs, $V_{0..n}$, max weight, k:

maximize $\sum \frac{V_i}{W_i} x_i$, where

$\Sigma w_i x_i \leq k$

items [i]:

knapsack [j]: current total capacity

V(i\j)	j=0	…	j=k
i=0			
…			
i=n			

1. Add one item at a time and see if it fits in each capacity range

[w_i]: item weight
[v_i]: item value
for i = 1.n
    w = knapsack capacity

$$OPT(k,X)=\left\{\begin{array}{ll} 0, & k=0 \\ OPT(k-1,X), & w_k > X \\ \max\left\{\begin{array}{l} OPT(k-1,X) \\ V_k + OPT(k-1,X-w_k) \end{array}\right\}, & else \end{array}\right\}$$

Constraint Graph

For each directed edge from b to a, there is a constraint
$a - b \leq d_{b\text{-}a}$

You can prove you have a negative weight cycle by putting your final Bellman-Ford values into the constraint graph and adding up the constraints

$-5 \leq -6$
$\underline{-3 \leq -1}$
$-8 \leq -7$
If that doesn't add up, you have a negative weights cycle.

Maximum Flow

Ford-Fulkerson algorithm

G(V,E)
Incoming flow = outgoing flow for each vertex

In the end, you're good when back edges are 0 and most forward edges are full

1. Draw graph
2. Show augmenting paths
3. Identify the limiting amount for each path
4. Draw final graph
5. Indicate max flow

After you identify a path, assume the edges already have the amount of the previous limiting amount.

Pairwise Distinct: every pair of elements consists of two different things (excluding the possibility that you selected the same element twice)