Linear Programming

Slides by Carl Kingsford

Apr. 14, 2014

Linear Programming

Suppose you are given:

- A matrix A with m rows and n columns.
- A vector $\vec{b}$ of length m.
- A vector $\vec{c}$ of length n.

Find a length-n vector $\vec{x}$ such that

$$A\vec{x} \leq \vec{b}$$

and so that

$$\vec{c} \cdot \vec{x} := \sum_{j=1}^{n} c_j x_j$$

is as large as possible.

Linear Algebra

The matrix inequality:

$$A\vec{x} \leq \vec{b}$$

in pictures:

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \times \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix} \leq \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

Each **row** of A gives coefficients of a linear expression: $\sum_j a_{ij} x_j$.

Each row of A along with an entry of b specifies a linear inequality: $\sum_j a_{ij} x_j \leq b_i$.

A little more general

$$\text{maximize} \quad \sum_j c_j x_j$$
$$\text{subject to} \quad A\vec{x} \leq b$$

What if you want to minimize?

What if you want to include a "$\geq$" constraint $\vec{a_i} \cdot \vec{x} \geq b_i$?

What if you want to include a "$=$" constraint?

A little more general

$$\begin{aligned} \text{maximize} \quad & \sum_j c_j x_j \\ \text{subject to} \quad & A\vec{x} \leq b \end{aligned}$$

What if you want to minimize? Rewrite to maximize $\sum_j (-c_j) x_j$.

What if you want to include a "$\geq$" constraint $\vec{a_i} \cdot \vec{x} \geq b_i$?

What if you want to include a "$=$" constraint?

A little more general

$$\begin{aligned} \text{maximize} \quad & \sum_j c_j x_j \\ \text{subject to} \quad & A\vec{x} \le b \end{aligned}$$

What if you want to minimize? Rewrite to maximize $\sum_j (-c_j) x_j$.

What if you want to include a "$\ge$" constraint $\vec{a_i} \cdot \vec{x} \ge b_i$?
Include the constraint $-\vec{a_i} \cdot \vec{x} \le -b_i$ instead.

What if you want to include a "$=$" constraint?

A little more general

$$\begin{aligned} \text{maximize} \quad & \sum_j c_j x_j \\ \text{subject to} \quad & A\vec{x} \le b \end{aligned}$$

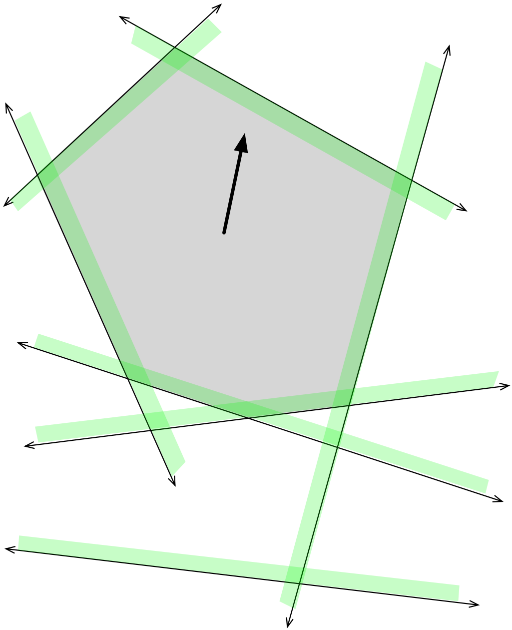
What if you want to minimize? Rewrite to maximize $\sum_j (-c_j) x_j$.

What if you want to include a "$\ge$" constraint $\vec{a_i} \cdot \vec{x} \ge b_i$?
Include the constraint $-\vec{a_i} \cdot \vec{x} \le -b_i$ instead.

What if you want to include a "$=$" constraint?
Include both the $\ge$ and $\le$ constraints.

Hence, we can use $=$ and $\ge$ constraints and maximize if we want.

A Geometric View of Linear Programming



History of LP Algorithms

The Simplex Method:

- One of the earliest methods.
- **Not** a polynomial time algorithm: for all proposed variants, there are example LPs that take exponential time to solve.
- Still very widely used because it is fast in practice.

The Ellipsoid Method:

- Discovered in the 1970s.
- First polynomial time algorithm for linear programming.
- Horribly slow in practice, and essentially never used.

Interior Point Methods:

- Polynomial.
- Practical.

In Practice?

There is *lots* of software to solve linear programs:

- CPLEX — commercial, seems to be the undisputed winner.
- GLPK — GNU Linear Programming Solver
- COIN-OR (CLP) — Another open source solver.
- ...
- NEOS server — http://www-neos.mcs.anl.gov/

Even Microsoft Excel has a built-in LP solver (though may not be installed by default).

What is Linear Programming Good For?

Maximum Flow

Problem (Maximum Flow). *Given a directed graph $G = (V, E)$, capacities $c(e)$ for each edge e, and two vertices $s, t \in V$, find a flow f in G from s to t of maximum value.*

What does a valid flow f look like?

- $0 \leq f(e) \leq c(e)$ for all e.
- $\sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w)$
  for all $v \in V$ except s, t.

Maximum Flow as LP

Create a variable x_{uv} for every edge $(u, v) \in E$. The x_{uv} values will give the flow: $f(u, v) = x_{uv}$.

Then we can write the maximum flow problem as a linear program:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{(u,t)\in E} x_{ut} & \\ \text{subject to} & 0 \le x_{uv} \le c_{uv} & \text{for every } (u, v) \in E \\ & \displaystyle\sum_{(u,v)\in E} x_{uv} = \sum_{(v,w)\in E} x_{vw} & \text{for all } v \in V \setminus \{s, t\} \end{array}$$

The first set of constraints ensure the capacity constraints are obeyed. The second set of constraints enforce flow balance.

Maximum Flow as MathProg

```
set V;                           # rep vertices
set E within V cross V;          # rep edges
param C {(u,v) in E} >= 0;       # capacities
param s in V;                    # source & sink
param t in V;

var X {(u,v) in E} >= 0, <= C[u,v];  # var for each edge

maximize flow: sum {(u,t) in E} X[u,t];

subject to balance {v in (V setminus {s,t})}:
  sum {(u,v) in E} X[u,v] = sum {(v,w) in E} X[v,w];

solve;
printf {(u,v) in E : X[u,v] > 0}: "%s %s %f", u,v,X[u,v];
end;
```

General MathProg Organization

Declarations

Objective Function

Constraints

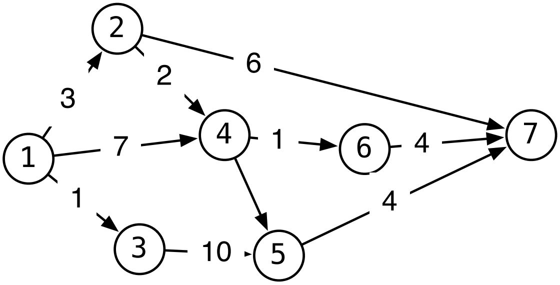
Output

Maximum Flow Data

The “model” on the previous slide can work any graph and capacities.

The “data” file of the MathProg program gives the specific *instance* of the problem.

Support your graph was this:



Maximum Flow Data

```
data;
set V := 1..7;
set E := (1,2) (1,3) (1,4) (2,4) (2,7) (3,5) (4,6)
         (4,5) (5,7) (6,7) ;
param C : 1 2 3 4 5 6 7 :=
        1 . 3 1 7 . . .
        2 . . . 2 . . 6
        3 . . . . 9 . .
        4 . . . . 0 1 .
        5 . . . . . . 4
        6 . . . . . . 4
        7 . . . . . . . ;
param s := 1;
param t := 7;
end;
```

Minimum-cost flow

Problem (Minimum-cost flow). *You are given a directed graph $G = (V, E)$ with capacities c_e on the edges and cost q_e on each edge so that sending α units of low on edge e costs αq_e dollars.*

Find a flow that gets r units of flow from s to t, and minimizes the cost.

Minimum-cost flow

Problem (Minimum-cost flow). *You are given a directed graph $G = (V, E)$ with capacities c_e on the edges and cost q_e on each edge so that sending α units of low on edge e costs αq_e dollars.*

Find a flow that gets r units of flow from s to t, and minimizes the cost.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{(u,v)} q_{uv} x_{uv} \\
\text{s.t.} \quad & 0 \leq x_{uv} \leq c_{uv} && \text{for all } (u,v) \in E \\
& \sum_{(u,v)\in E} x_{uv} = \sum_{(v,w)\in E} x_{vw} && \text{for all } v \in V \setminus \{s,t\} \\
& \sum_{(u,t)\in E} x_{ut} \geq r
\end{aligned}
$$

Shortest path

Problem (Shortest path). *Find the shortest path from s to t in a directed graph $G = (V, E)$ with positive edge lengths q_e.*

Shortest path

Problem (Shortest path). *Find the shortest path from s to t in a directed graph $G = (V, E)$ with positive edge lengths q_e.*

Variable x_{uv} records whether we use edge e or not.

Shortest path

Problem (Shortest path). *Find the shortest path from s to t in a directed graph $G = (V, E)$ with positive edge lengths q_e.*

Variable x_{uv} records whether we use edge e or not.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{(u,v)} q_{uv} x_{uv} \\
\text{s.t.} \quad & \sum_{(u,v)\in E} x_{uv} = \sum_{(v,w)\in E} x_{vw} \quad \text{for all } v \in V \setminus \{s, t\} \\
& \sum_{(u,t)\in E} x_{ut} = 1
\end{aligned}
$$

Shortest path

Problem (Shortest path). *Find the shortest path from s to t in a directed graph $G = (V, E)$ with positive edge lengths q_e.*

Variable x_{uv} records whether we use edge e or not.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{(u,v)} q_{uv} x_{uv} \\
\text{s.t.} \quad & \sum_{(u,v)\in E} x_{uv} = \sum_{(v,w)\in E} x_{vw} \quad \text{for all } v \in V \setminus \{s, t\} \\
& \sum_{(u,t)\in E} x_{ut} = 1
\end{aligned}
$$

This has a $0/1$ solution: equivalent to network flow with infinite capacities and a "dummy edge" from t to t' of lower bound 1; no capacity will be > 1 otherwise the cost could be reduced.

Maximum Bipartite Matching

Problem (Maximum Bipartite Matching).
Given a bipartite graph $G = (V, E)$, choose as large a subset of edges $M \subseteq E$ as possible that forms a matching.

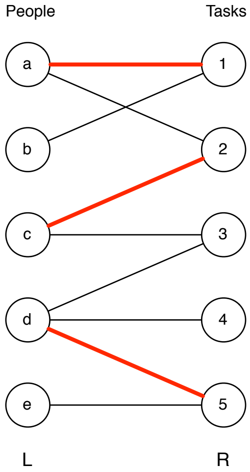
The red text gives an objective function.

The blue text gives constraints.

$$\text{maximize} \sum_{u,v} x_{uv}$$

$$\text{s.t.} \sum_{u} x_{uv} \leq 1$$

$$\sum_{v} x_{uv} \leq 1$$



Maximum Bipartite Matching

```
set A;
set B;
set E within A cross B; # a bipartite graph

var X {e in E} >= 0, <= 1; # variable for each edge

maximize numedges: sum {(u,v) in E} X[u,v];

s.t. matchA {u in A}: sum {(u,v) in E} X[u,v] <= 1;
s.t. matchB {v in B}: sum {(u,v) in E} X[u,v] <= 1;
end;
```

Bipartite Matching Data

```
data;
set A := a b c d e f;
set B := 1..5;
set E : 1 2 3 4 5 :=
      a + + - - -
      b - - + + +
      c + - + - +
      d - + - + -
      e - - - - +
      f + - + - - ;
end;
```

Integer Linear Programming

If we add one more kind of constraint, we get an **integer linear program** (ILP):

$$\begin{aligned} \text{maximize} \quad & \sum_j c_j x_j \\ \text{subject to} \quad & A\vec{x} \leq b \\ & x_i \in \{0, 1\} \quad \text{for all } i = 1, \ldots, n \leftarrow \end{aligned}$$

ILPs seem to be much more powerful and expressive than just LPs.

In particular, solving an ILP is NP-hard and there is no known polynomial time algorithm (and if P≠NP, there isn't one).

However: because of its importance, lots of optimized code and heuristics are available. CPLEX and GLPK for example provide solvers for ILPs.

Minimum Vertex Cover

Problem (Minimum Vertex Cover). *Given graph* $G = (V, E)$ *choose a subset of vertices* $C \subseteq V$ *such that every edge in* E *is incident to some vertex in* C.

Why is this useful?

- In a social network, choose a set of people so that every possible friendship has a representative.
- On what nodes should you place sensors in an electric network to make sure you monitor every edge?

Vertex Cover as an ILP

Create a variable x_u for every vertex u in V.

We can then model the vertex cover problem as the following linear program:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{v \in V} x_v \\
\text{subject to} \quad & x_u + x_v \geq 1 & \text{for every } \{u, v\} \in E \\
& x_u \in \{0, 1\} & \text{for all } u \in V
\end{aligned}
$$

The constraints "$x_u \in \{0, 1\}$" are called integrality constraints. They require that the variables be either 0 or 1, and they make the ILP difficult to solve.

Vertex Cover as MathProg

```
# Declarations
set V;
set E within V cross V;
var x {v in V} binary;    # integrality constraints.

# Objective Function
minimize cover_size: sum { v in V } x[v];

# Constraints
subject to covered {(u,v) in E}: x[u] + x[v] >= 1;

solve;

# Output
printf "The Vertex Cover:";
printf {u in V : x[u] >= 1}: "%d ", u;
end;
```

Summary

- Many problems can be modeled as linear programs (LPs).
- If you can write your problem as an LP, you can use existing, highly optimized solvers to give polynomial time algorithms to solve them.
- It seems even more problems can be written as integer linear programs (ILP).
- If you write your problem as an ILP, you won't have a polynomial-time algorithm, but you may be able to use optimized packages to solve it.