Linear Programming

Political Advertising problem:

* Four different platforms: building roads, gun control, farm subsidies, gasoline tax. Three types of ridings: urban, suburban, and rural. Assume that advertising can only be done "globally": all advertising is seen in all three types of ridings.

* Market research: for each advertising platform and riding type, number of voters gained or lost in ridings of that type, for each dollar of advertising spent on that platform.

	urb.	sub.	rur.
roads	−2	5	3
guns	8	2	−5
farm	0	0	10
gas	10	0	−2

* Leaders aim to gain at least 50,000 urban voters, 100,000 suburban voters, and 25,000 rural voters.

* Your task: spend as little as possible on advertising to gain the required number of votes in each type of riding.

* Here is one way to represent the problem.

  What are we looking for exactly? Amount to advertise on each platform. So introduce variables:

  x1 = $1000's to advertise on building roads
  x2 = $1000's to advertise on gun controls
  x3 = $1000's to advertise on farm subsidies
  x4 = $1000's to advertise on gasoline tax

  What's our goal? Spend as little as possible, in other words, minimize x1 + x2 + x3 + x4.

  What are our constraints? The need to gain some number of voters in each riding type, which can be expressed by linear inequalities:

```
−2 x1 + 8 x2 +  0 x3 + 10 x4 >= 50     (for urban ridings)
 5 x1 + 2 x2 +  0 x3 +  0 x4 >= 100    (for suburban ridings)
 3 x1 − 5 x2 + 10 x3  − 2 x4 >= 25     (for rural ridings)
```

  Anything else? Numerically, we cannot spend negative amounts, so

  x1 >= 0, x2 >= 0, x3 >= 0, x4 >= 0.

  This is known as a "linear program".

In general, linear program = optimization problem that consists of:

* variables: x_1, x_2, ..., x_n (real numbers)

* "objective function": c_1 x_1 + c_2 x_2 + ... + c_n x_n
  where c_i are real number constants

* "constraints": linear, i.e., of the form
      a_{i,1} x_1 + a_{i,2} x_2 + ... + a_{i,n} x_n <= / = / >= b_i
  for i = 1, 2, ..., m. Often written A x <= / = / >= b for matrix A,
  vector of variables x and vector of constants b.
  Example for political advertising (including non-negativity):

```
( -2   8   0  10 )  ( x1 )       (  50 )
(  5   2   0   0 )  ( x2 )       ( 100 )
(  3  -5  10  -2 )  ( x3 )       (  25 )
(  1   0   0   0 )  ( x4 )  >=   (   0 )
(  0   1   0   0 )               (   0 )
(  0   0   1   0 )               (   0 )
(  0   0   0   1 )               (   0 )
```

* problem: find real values of x_i's that maximize/minimize objective
  function and satisfy all constraints.

* "Integer programming": more restricted version where all constants and
  variables are integers. NP-hard (no efficient algorithm).

Solving linear programs:

* "Feasible region" = set of values of variables that satisfy all
  constraints. Feasible region can be:

  - Empty, e.g., constraints x_1 + x_2 <= -1 and 2 x_1 + 2 x_2 >= 5; no
    solution to linear program in this case.

  - Unbounded, e.g., constraints x_1 >= 0, x_2 >= 0, x_1 + x_2 >= 10; no
    solution to linear program in this case (value of objective function
    can be arbitrarily large so there is no maximum) -- if objective
    function is minimization, then treat this as bounded.

  - Bounded, e.g., constraints x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 10;
    either one or infinitely many solutions to linear program in this
    case, depending on objective function, e.g., with same constraints,
    unique solution to maximize objective function 2 x_1 - x_2 (x_1 =
    10, x_2 = 0) but infinitely many solutions to maximize objective
    function x_1 + x_2 (any two nonnegative values that add up to 10).

* Simplex method solves linear programs by, intuitively, moving from
  vertex to vertex along the boundary of the feasible region, using
  algebraic manipulations similar to Gaussian elimination. It runs in
  worst-case exponential time but in practice, this behaviour is very
  rarely encountered (most of the time it is quite efficient).

* "Interior point" methods solve linear programs in worst-case polynomial
  time but are just recently starting to be competitive with Simplex in
  practice.

Applications of Linear Programming:

* Network flows: Given network N=(V,E) with capacities c(e) for e in E,
  construct linear program with variables f_e (one for each e in E):
      maximize: \sum_{(s,u) in E} f_(s,u)
      subject to:

0 <= f_e <= c(e) for all e in E
\sum_{(u,v) in E} f_(u,v) - \sum_{(v,u) in E} f_(v,u) = 0
for all v in V

Direct re-statement of network flow problem: all valid flows in N yield values for f_e that satisfy all constraints ("feasible" values), and all feasible values for f_e are valid flow in N. So finding max flow in N equivalent to maximizing objective function.

Might this allow us to solve max flow problem more easily or faster? Unfortunately not: solving LP no more efficient than solving max flow (not surprisingly since problem is "more general"). One more catch: LP solution cannot guarantee integer flow on all edges (even when all edge capacities are integer) -- in contrast with Ford-Fulkerson algorithm (an its implementations) that guarantees integer flows in that case.

* Shortest s-t path: Given graph G = (V,E) with weights w(e) for all e in E, construct linear program with variables d_v for each v in V:

```
maximize:    d_t
subject to:  d_v <= d_u + w(u,v)  for each (u,v) in E
             d_s = 0
             d_v >= 0  for each v in V
```

Minimizing d_t doesn't work because it allows settings of d_v smaller than true distances (e.g., d_v = 0 for all v in V). Maximizing works because constraints force d_v to be no more than shortest distance and maximization forces d_v to be at least shortest distance, for all v.

Example: Graph with vertices V = {s,a,b,t} and edges (s,a) with weight 1 (s,t) with weight 6, (a,b) with weight 2, (a,t) with weight 4, (b,t) with weight 1.
Linear program (each edge yields two constraints):

```
maximize:  d_t
subject to:
    d_s = 0
    0 <= d_a <= d_s + 1    0 <= d_s <= d_a + 1
    0 <= d_b <= d_a + 2    0 <= d_a <= d_b + 2
    0 <= d_t <= d_s + 6    0 <= d_s <= d_t + 6
    0 <= d_t <= d_a + 4    0 <= d_a <= d_t + 4
    0 <= d_t <= d_b + 1    0 <= d_b <= d_t + 1
```

* Minimum Vertex Cover: [NOT covered in evening section; please read!]
  Input: Undirected graph G = (V,E).
  Output: Subset of vertices C that "covers" every edge (i.e., each edge has at least one endpoint in C), with minimum size.

  - Representing as *integer* program:
    use variable x_i for each vertex v_i in V

```
minimize:    x_1 + x_2 + ... + x_n
subject to:  x_i + x_j >= 1    for all (v_i,v_j) in E
             0 <= x_i <= 1    for all v_i in V
```

    Note that last constraint equivalent to "x_i in {0,1}" when x_i restricted to integer values. Also, this 0-1 integer program is completely equivalent to the original problem, through correspondence: v_i in cover iff x_i = 1. In more detail:
    . Any vertex cover C yields feasible solution
      x_i = 1 if v_i in C, 0 if v_i not in C
      because each constraint x_i + x_j >= 1 satisfied (C must include one endpoint of each edge).
    . Any feasible solution to LP yields vertex cover
      C = { v_i in V : x_i = 1 }
      because for each edge (v_i,v_j), constraint x_i + x_j >= 1

ensures C contains at least one of v_i, v_j.

Unfortunately, Integer Programming (IP) is NP-hard, so problem canno be solved in polytime this way.

P and NP

Review and Overview:

* So far, we've seen:
  - problems solvable efficiently by "simple" algorithms (greedy);
  - problems that cannot be solved efficiently by simple techniques, but that can be solved efficiently by more complex algorithms (limited backtracking/network flows, dynamic programming);
  - problems that cannot be solved efficiently by any known algorithm (the only known solutions involve unlimited backtracking or exhaustive search).

* Wishlist (given a problem to solve):
  - General method for figuring out best algorithmic method to solve the problem. For certain types of problems, this is possible (e.g., "matroid theory" for greedy algorithms). For others, only trial-and-error and experience can help.
  - General method for proving problem has no efficient solution. For certain problems, this is possible. For others (the majority), no guarantees are known -- though we can get close, as we will see.

Running times and input size:

* To discuss problem complexity, need to be more precise about computation model (recursive functions, Turing machines, RAM model, etc.) and specific measure of input size.

* To be more precise, define "standard size":
  for any object, standard size = total number of bits required to write down the object (using reasonable encoding).
  - Example 1: graph G represented by adjacency matrix: n x n array of bits with total bit-size n^2 -- size(G) = n^2.
  - Example 2: integer x written down in binary notation requires floor(log_2(x+1)) many bits -- approx. log_2 x -- size(x) = log_2 x.
  - Example 3: list of integers [x_1,...,x_n] requires at least size(x_1) + size(x_2) + ... + size(x_n) bits to write down; if we use the same number of bits to write down each number, size([x_1,...,x_n]) = n * log_2(max(x_1,...,x_n)).

  With additional assumption that integers take up constant number of bits this is equivalent to n within a constant factor -- but that assumption does not always hold!

* As it turns out, different models of computation and different ways of measuring input size can affect running time by at most polynomial factors (e.g., doubling, squaring, etc.), e.g., problems solved in time \Theta(n) on one model may require time \Theta(n^3) on another model.

* Two exceptions:
  - Representing integers in "unary" notation (i.e., integer k represented using k many 1's) requires exponentially many more characters than binary (or any other base), so we rule this out.
  - Non-deterministic models of computation appear to solve certain problems exponentially faster than deterministic ones.

* How to handle differences of more than constant factor? Use coarser scale: ignore polynomial differences. Luckily, any polytime algo as a function of informal "size" is also polytime as a function of bitsize, a long as it uses only "reasonable" primitive operations -- comparisons, addition, subtraction -- other operations (multiplication, exponentiation) are _not_ considered "reasonable" because repeated application of the operation can make the result's size grow exponentially. This does not mean that we cannot use multiplication, jus that we cannot count it as a primitive operation.

Towards P and NP:

* Intuitively,

```
     P = {all problems solvable by some *deterministic* algorithm
           in polytime (worst-case)}
    NP = {all problems solvable by some *non-deterministic* algorithm
           in polytime (worst-case)}
```

* Formal definitions next week...

For Next Week

* Readings: Section 8.2
* Self-Test: Review algorithms from class, figure out exact running time a a function of standard size (as defined here).