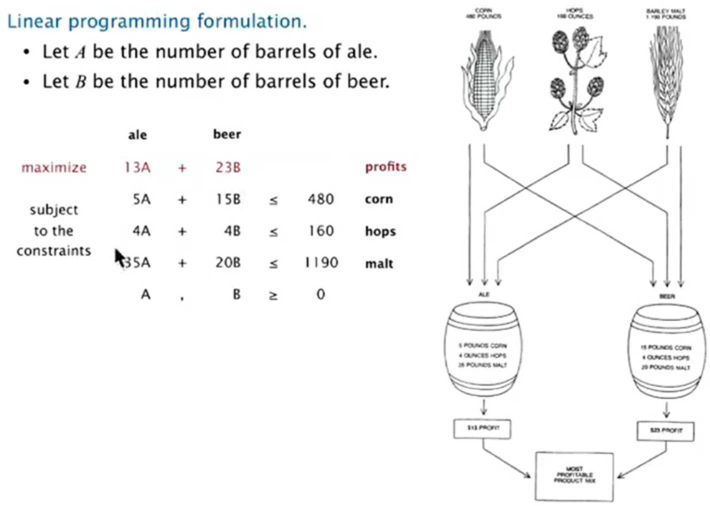
Linear Programming

Brewer's Problem

> Choose product mix to maximize profits



Geometry

- Inequalities define halfplanes
- feasible region is a convex polygon
- A set id convex if any two points a and b in the set, so is $\frac{1}{2}(a+b)$
- An extreme point of a set is a point in the set that can't be written as $\frac{1}{2}(a+b)$, where a and b are two distinct points in the set
  - Number of extreme points to consider is finite
  - Number of extreme points can be exponential
  - Greedy property---extree point optimal iff no better adjacent extreme point

Simplex Algorithm

Basic

- Start at some extreme point
- Pivot from one extreme point to an adjacent one
- Repeat until optimal

Jargon

- Basis--A basis is a subset of m of the n variables
- Basic feasible solution (BFS)
- Basic variable, non-basic variable

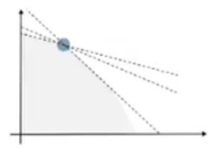
Property

- In typical practical applications, simplex algorithm terminates after at most $2(m+n)$ pivots

Degeneracy

Degeneracy. New basis, same extreme point.

"stalling" is common in practice



Cycling. Get stuck by cycling through different bases that all correspond to same extreme point.

- Doesn't occur in the wild.
- Bland's rule guarantees finite # of pivots.

choose lowest valid index for entering and leaving columns

Implementation

Issues

To improve the bare-bones implementation.

- Avoid stalling. ← requires artful engineering
- Maintain sparsity. ← requires fancy data structures
- Numerical stability. ← requires advanced math
- Detect infeasibility. ← run "phase I" simplex algorithm
- Detect unboundedness. ← no leaving row

Best practice. Don't implement it yourself!

Basic implementations. Available in many programming environments.
Industrial-strength solvers. Routinely solve LPs with millions of variables.

Min ratio

```java
    private int minRatioRule(int q) {
        int p = -1;
        for (int i = 0; i < m; i++) {
            // if (a[i][q] <= 0) continue;
            if (a[i][q] <= EPSILON) continue;
            else if (p == -1) p = i;
            else if ((a[i][m+n] / a[i][q]) < (a[p][m+n] / a[p][q])) p = i;
        }
        return p;
    }
```

Pivot

```java
    private void pivot(int p, int q) {

        // everything but row p and column q
        for (int i = 0; i <= m; i++)
            for (int j = 0; j <= m+n; j++)
                if (i != p && j != q) a[i][j] -= a[p][j] * (a[i][q] / a[p][q]);

        // zero out column q
        for (int i = 0; i <= m; i++)
            if (i != p) a[i][q] = 0.0;

        // scale row p
        for (int j = 0; j <= m+n; j++)
            if (j != q) a[p][j] /= a[p][q];
        a[p][q] = 1.0;
    }
```

Complete code

```java
public class LinearProgramming {
    private static final double EPSILON = 1.0E-10;
    private double[][] a;   // tableaux
    private int m;          // number of constraints
    private int n;          // number of original variables

    private int[] basis;    // basis[i] = basic variable corresponding to row i
                            // only needed to print out solution, not book

    public LinearProgramming(double[][] A, double[] b, double[] c) {
        m = b.length;
        n = c.length;
        for (int i = 0; i < m; i++)
            if (!(b[i] >= 0)) throw new IllegalArgumentException("RHS must be nonnegative");

        a = new double[m+1][n+m+1];
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                a[i][j] = A[i][j];
        for (int i = 0; i < m; i++)
            a[i][n+i] = 1.0;
        for (int j = 0; j < n; j++)
            a[m][j] = c[j];
        for (int i = 0; i < m; i++)
            a[i][m+n] = b[i];

        basis = new int[m];
```

```
        for (int i = 0; i < m; i++)
            basis[i] = n + i;

        solve();

        // check optimality conditions
        assert check(A, b, c);
    }

    // run simplex algorithm starting from initial BFS
    private void solve() {
        while (true) {

            // find entering column q
            int q = bland();
            if (q == -1) break;  // optimal

            // find leaving row p
            int p = minRatioRule(q);
            if (p == -1) throw new ArithmeticException("Linear program is unbounded");

            // pivot
            pivot(p, q);

            // update basis
            basis[p] = q;
        }
    }

    // lowest index of a non-basic column with a positive cost
    private int bland() {
        for (int j = 0; j < m+n; j++)
            if (a[m][j] > 0) return j;
        return -1;  // optimal
    }

   // index of a non-basic column with most positive cost
    private int dantzig() {
        int q = 0;
        for (int j = 1; j < m+n; j++)
            if (a[m][j] > a[m][q]) q = j;

        if (a[m][q] <= 0) return -1;  // optimal
        else return q;
    }

    // find row p using min ratio rule (-1 if no such row)
    // (smallest such index if there is a tie)
    private int minRatioRule(int q) {}

    // pivot on entry (p, q) using Gauss-Jordan elimination
```

```java
    private void pivot(int p, int q) {}

    public double value() {
        return -a[m][m+n];
    }

    public double[] primal() {
        double[] x = new double[n];
        for (int i = 0; i < m; i++)
            if (basis[i] < n) x[basis[i]] = a[i][m+n];
        return x;
    }

    public double[] dual() {
        double[] y = new double[m];
        for (int i = 0; i < m; i++) {
            y[i] = -a[m][n+i];
            if (y[i] == -0.0) y[i] = 0.0;
        }
        return y;
    }


    // is the solution primal feasible?
    private boolean isPrimalFeasible(double[][] A, double[] b) {
        double[] x = primal();

        // check that x >= 0
        for (int j = 0; j < x.length; j++) {
            if (x[j] < -EPSILON) {
                StdOut.println("x[" + j + "] = " + x[j] + " is negative");
                return false;
            }
        }

        // check that Ax <= b
        for (int i = 0; i < m; i++) {
            double sum = 0.0;
            for (int j = 0; j < n; j++) {
                sum += A[i][j] * x[j];
            }
            if (sum > b[i] + EPSILON) {
                StdOut.println("not primal feasible");
                StdOut.println("b[" + i + "] = " + b[i] + ", sum = " + sum);
                return false;
            }
        }
        return true;
    }

    // is the solution dual feasible?
    private boolean isDualFeasible(double[][] A, double[] c) {
```

```
        double[] y = dual();

        // check that y >= 0
        for (int i = 0; i < y.length; i++) {
            if (y[i] < -EPSILON) {
                StdOut.println("y[" + i + "] = " + y[i] + " is negative");
                return false;
            }
        }

        // check that yA >= c
        for (int j = 0; j < n; j++) {
            double sum = 0.0;
            for (int i = 0; i < m; i++) {
                sum += A[i][j] * y[i];
            }
            if (sum < c[j] - EPSILON) {
                StdOut.println("not dual feasible");
                StdOut.println("c[" + j + "] = " + c[j] + ", sum = " + sum);
                return false;
            }
        }
        return true;
    }

    // check that optimal value = cx = yb
    private boolean isOptimal(double[] b, double[] c) {
        double[] x = primal();
        double[] y = dual();
        double value = value();

        // check that value = cx = yb
        double value1 = 0.0;
        for (int j = 0; j < x.length; j++)
            value1 += c[j] * x[j];
        double value2 = 0.0;
        for (int i = 0; i < y.length; i++)
            value2 += y[i] * b[i];
        if (Math.abs(value - value1) > EPSILON || Math.abs(value - value2) > EPSILON) {
            StdOut.println("value = " + value + ", cx = " + value1 + ", yb = " + value2);
            return false;
        }

        return true;
    }

    private boolean check(double[][]A, double[] b, double[] c) {
        return isPrimalFeasible(A, b) && isDualFeasible(A, c) && isOptimal(b, c);
    }
```

```java

    // print tableaux
    private void show() {
        StdOut.println("m = " + m);
        StdOut.println("n = " + n);
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= m+n; j++) {
                StdOut.printf("%7.2f ", a[i][j]);
                // StdOut.printf("%10.7f ", a[i][j]);
            }
            StdOut.println();
        }
        StdOut.println("value = " + value());
        for (int i = 0; i < m; i++)
            if (basis[i] < n) StdOut.println("x_" + basis[i] + " = " + a[i][m+n]);
        StdOut.println();
    }


    private static void test(double[][] A, double[] b, double[] c) {
        LinearProgramming lp;
        try {
            lp = new LinearProgramming(A, b, c);
        }
        catch (ArithmeticException e) {
            System.out.println(e);
            return;
        }

        StdOut.println("value = " + lp.value());
        double[] x = lp.primal();
        for (int i = 0; i < x.length; i++)
            StdOut.println("x[" + i + "] = " + x[i]);
        double[] y = lp.dual();
        for (int j = 0; j < y.length; j++)
            StdOut.println("y[" + j + "] = " + y[j]);
    }

    private static void test1() {
        double[][] A = {
            { -1,  1,  0 },
            {  1,  4,  0 },
            {  2,  1,  0 },
            {  3, -4,  0 },
            {  0,  0,  1 },
        };
        double[] c = { 1, 1, 1 };
        double[] b = { 5, 45, 27, 24, 4 };
        test(A, b, c);
    }

```

```

    // x0 = 12, x1 = 28, opt = 800
    private static void test2() {
        double[] c = {  13.0,  23.0 };
        double[] b = { 480.0, 160.0, 1190.0 };
        double[][] A = {
            {  5.0, 15.0 },
            {  4.0,  4.0 },
            { 35.0, 20.0 },
        };
        test(A, b, c);
    }

    // unbounded
    private static void test3() {
        double[] c = { 2.0, 3.0, -1.0, -12.0 };
        double[] b = {  3.0,   2.0 };
        double[][] A = {
            { -2.0, -9.0,  1.0,  9.0 },
            {  1.0,  1.0, -1.0, -2.0 },
        };
        test(A, b, c);
    }

    // degenerate - cycles if you choose most positive objective function
coefficient
    private static void test4() {
        double[] c = { 10.0, -57.0, -9.0, -24.0 };
        double[] b = {  0.0,   0.0,  1.0 };
        double[][] A = {
            { 0.5, -5.5, -2.5, 9.0 },
            { 0.5, -1.5, -0.5, 1.0 },
            { 1.0,  0.0,  0.0, 0.0 },
        };
        test(A, b, c);
    }

    public static void main(String[] args) {

        StdOut.println("----- test 1 --------------------");
        test1();
        StdOut.println();

        StdOut.println("----- test random ---------------");
        int m = Integer.parseInt(args[0]);
        int n = Integer.parseInt(args[1]);
        double[] c = new double[n];
        double[] b = new double[m];
        double[][] A = new double[m][n];
        for (int j = 0; j < n; j++)
            c[j] = StdRandom.uniformInt(1000);
        for (int i = 0; i < m; i++)
```

```
            b[i] = StdRandom.uniformInt(1000);
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                A[i][j] = StdRandom.uniformInt(100);
        LinearProgramming lp = new LinearProgramming(A, b, c);
        test(A, b, c);
    }
}
```

Reductions

Modelling

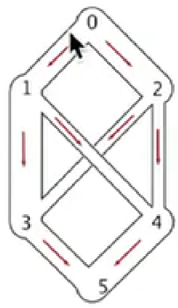
- Linear "programming" (1950s term) = reduction to LP (modern term)
    - Process of formulating an LP model for a problem
    - Solution to LP for a specific problem gives solution to the problem
- Steps
    - identify variables
    - define constraints
    - define objective function
    - convert to standard form
- examples
    - shortest path
    - maxflow
    - bipartite matching
    - assignment problem
    - 2-person zero-sum games

Maxflow (revisited)

Variables. x_{vw} = flow on edge $v \rightarrow w$.
Constraints. Capacity and flow conservation.
Objective function. Net flow into t.

maxflow problem

```
6    ← V
8    ← E
0 1  2.0
0 2  3.0
1 3  3.0
1 4  1.0
2 3  1.0
2 4  1.0
3 5  2.0
4 5  3.0
     ↑
  capacities
```



LP formulation

Maximize $x_{35} + x_{45}$
subject to the constraints

capacity constraints:

$$\begin{aligned} 0 \le x_{01} \le 2 \\ 0 \le x_{02} \le 3 \\ 0 \le x_{13} \le 3 \\ 0 \le x_{14} \le 1 \\ 0 \le x_{23} \le 1 \\ 0 \le x_{24} \le 1 \\ 0 \le x_{35} \le 2 \\ 0 \le x_{45} \le 3 \end{aligned}$$

flow conservation constraints:

$$\begin{aligned} x_{01} &= x_{13} + x_{14} \\ x_{02} &= x_{23} + x_{24} \\ x_{13} + x_{23} &= x_{35} \\ x_{14} + x_{24} &= x_{45} \end{aligned}$$

Maximum cardinality bipartite matching

LP formulation. One variable per pair.
Interpretation. x_{ij} = 1 if person i assigned to job j.

maximize

$$x_{A0} + x_{A1} + x_{A2} + x_{B0} + x_{B1} + x_{B5} + x_{C2} + x_{C3} + x_{C4} + x_{D0} + x_{D1} + x_{E3} + x_{E4} + x_{E5} + x_{F2} + x_{F4} + x_{F5}$$

subject to the constraints

at most one job per person	at most one person per job
$x_{A0} + x_{A1} + x_{A2} \le 1$	$x_{A0} + x_{B0} + x_{D0} \le 1$
$x_{B0} + x_{B1} + x_{B5} \le 1$	$x_{A1} + x_{B1} + x_{D1} \le 1$
$x_{C2} + x_{C3} + x_{C4} \le 1$	$x_{A2} + x_{C2} + x_{F2} \le 1$
$x_{D0} + x_{D1} \le 1$	$x_{C3} + x_{E3} \le 1$
$x_{E3} + x_{E4} + x_{E5} \le 1$	$x_{C4} + x_{E4} + x_{F4} \le 1$
$x_{F2} + x_{F4} + x_{F5} \le 1$	$x_{B5} + x_{E5} + x_{F5} \le 1$

all $x_{ij} \ge 0$

Theorem. [Birkhoff 1946, von Neumann 1953]
All extreme points of the above polyhedron have integer (0 or 1) coordinates.
Corollary. Can solve matching problem by solving LP. ← not usually so lucky!