1.204 Lecture 19

Continuous constrained nonlinear optimization:

Convex combinations 2:

Network equilibrium

Constrained optimization

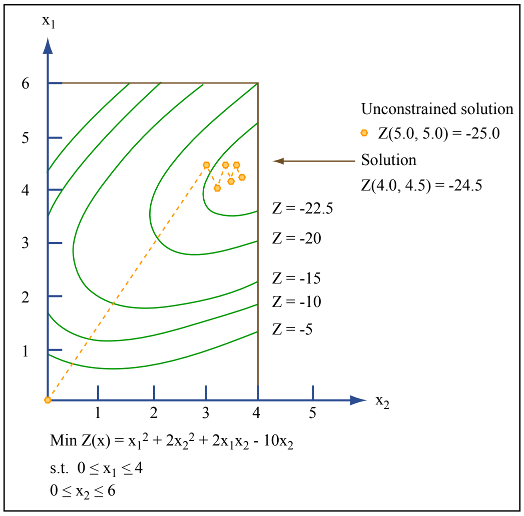


Figure by MIT OpenCourseWare.

Network equilibrium problem formulation

$$\min z(x) = \sum_{arcs\ a} \int_0^{x_a} t_a(\omega)\, d\omega$$

$$subject\ to$$

$$\sum_{paths\ k} f_k^{rs} = q_{rs} \quad \forall OD\ pairs\ r,s$$

$$f_k^{rs} \geq 0 \quad \forall k,r,s$$

$$x_a = \sum_i \sum_j \sum_k f_k^{rs} \quad \forall r,s,k$$

$$if\ \ a\ on\ path\ \ from\ r\ to\ s$$

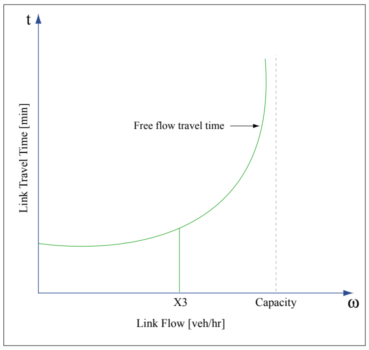


Figure by MIT OpenCourseWare.

Figures, examples from Sheffi

Convex combinations algorithm 1

- **Applying the convex combinations algorithm requires solution of a linear program at each step**

$$\min z^n(y) = \sum_{arcs\ a} \frac{\partial z(x^n)}{\partial x_a} \cdot y_a \quad \text{for all feasible y}$$

- **Gradient of z(x) is just the arc travel times:**

$$\frac{\partial z(x^n)}{\partial x_a} = t_a^n$$

- **The linear program becomes:**

$$\min z^n(y) = \sum_a t_a^n \cdot y_a$$

$$s.t. \quad \sum_k g_k^{rs} = q_{rs} \quad \forall r,s$$

$$g_k^{rs} \geq 0 \quad \forall k,r,s$$

$$y_a = \sum_{rs} \sum_k g_k^{rs} \quad \forall a\ in\ path$$

$$t_a^n = t_a(x_a^n)$$

Convex combinations algorithm 2

- The linear program minimizes travel times over a network with fixed, not flow-dependent times.
  - Total time is minimized by assigning each traveler to shortest O-D path
  - Thus, a shortest path algorithm, plus loading flow on the links used by each O-D pair, solves the linear program
- Line search step uses bisection method which, for a minimization problem, requires a derivative
  - It happens to be easy to compute. After a lot of algebra:

$$\frac{\partial}{\partial \alpha} z[x^n + \alpha(y^n - x^n)] = \sum_a (y_a^n - x_a^n)\, t_a(x_a^n + \alpha(y_a^n - x_a^n))$$

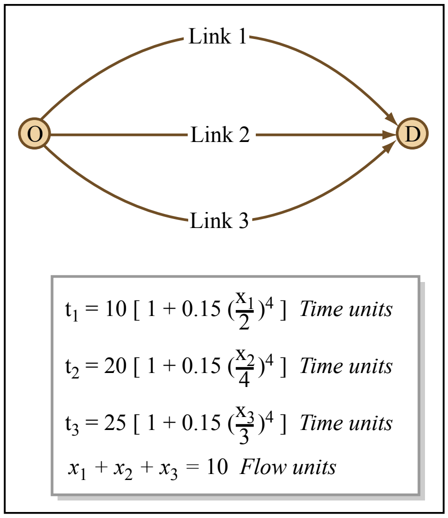
Convex combinations steps

- Step 0: Initialization.
  - Find shortest paths based on $t_a = t_a(0)$.
  - Assign flows to obtain $\{x_a^1\}$
- Step 1: Update times.
  - Set $t_a^n = t_a(x_a^n)$ for all a
- Step 2: Direction finding.
  - Find shortest paths based on $\{t_a^n\}$
  - Assign flows to obtain auxiliary flows $\{y_a^n\}$
- Step 3: Line search. Find α_n that solves
  - $$\min_{0 \le \alpha \le 1} \sum_a \int_0^{x_a^n + \alpha(y_a^n - x_a^n)} t_a(\omega)\, d\omega$$
- Step 4: Move. Set

$$x^{n+1} = x^n + \alpha_n (y^n - x^n)$$

- Step 5: Convergence test. If not converged, go to step 1

Example network



$$t_1 = 10 \left[1 + 0.15 \left(\frac{x_1}{2}\right)^4 \right] \textit{ Time units}$$

$$t_2 = 20 \left[1 + 0.15 \left(\frac{x_2}{4}\right)^4 \right] \textit{ Time units}$$

$$t_3 = 25 \left[1 + 0.15 \left(\frac{x_3}{3}\right)^4 \right] \textit{ Time units}$$

$$x_1 + x_2 + x_3 = 10 \textit{ Flow units}$$

Figure by MIT OpenCourseWare.

Example program output

```
Iter    Step      Link:  1     2     3   Objective fn  Alpha
0  Update     t:        10.0  20.0  25.0     0.00
   Move       x:       10.00  0.00  0.00

1  Update     t:       947.5  20.0  25.0  1975.00
   Direction  y:        0.00 10.00  0.00                0.597
   Move       x:        4.03  5.97  0.00

2  Update     t:        34.8  34.8  25.0   197.40
   Direction  y:        0.00  0.00 10.00                0.161
   Move       x:        3.38  5.00  1.61

3  Update     t:        22.3  27.3  25.3   189.99
   Direction  y:       10.00  0.00  0.00                0.036
   Move       x:        3.62  4.83  1.55

4  Update     t:        26.1  26.4  25.3   189.45
   Direction  y:        0.00  0.00 10.00                0.020
   Move       x:        3.55  4.73  1.73

5  Update     t:        24.8  25.9  25.4   189.36
   Direction  y:       10.00  0.00  0.00                0.007
   Move       x:        3.59  4.69  1.71
```

NetworkEquilibrium data members

```
public class NetworkEquilibrium implements MathFunction {
    public static final int EMPTY= Short.MIN_VALUE;
    private int nodes;
    private int arcs;
    private int[] head;              // Graph data structure
    private int[] to;                // Graph data structure
    private double[] timeBase;       // Zero flow time
    private double[] timeExponent;   // 4 in our example
    private double[] timeConst;      // .015/lanes in example
    private double[] D;     // Distance from root
    private int[] P;        // Predecessor node back to root
    private int[] Parc;     // Predecessor arc back to root
    private double[] xFlow; // Arc flows
    private double[] yFlow; // Auxiliary arc flows
    private double[] arcTime;        // t_a
    private double[][] ODtrips;      // q_rs

    // This Java code is tested only on our simple example
    // It should be basically correct for larger problems, but…
```

NetworkEquilibrium constructor

```
NetworkEquilibrium(int n, int a, int[] h, int[] t,
   double[] tBase, double[] tExponent, double[] tConst,
   double[][] od) {
       nodes= n;
       arcs= a;
       head= h;
       to= t;
       timeBase= tBase;
       timeExponent= tExponent;
       timeConst= tConst;
       ODtrips= od;
       arcTime= new double[arcs];
   }
```

Changes in shortest path

- Time is method, not data from array.
  - Replace `dist` in original version with `time()`
- Times are `double`, not `int` variables
  - Ints are faster, but doubles are easier
- Must keep track of predecessor arc as well as predecessor node in shortest path tree result
  - There may be multiple arcs
- Method is `private`

Shortest path

```
private void shortHKNetwork(int root) {   // root is argument
  final int MAX_COST= Integer.MAX_VALUE/2;
  final int NEVER_ON_CL= -1;
  final int ON_CL_BEFORE= -2;
  final int END_OF_CL= Integer.MAX_VALUE;
  D= new double[nodes];        // double, not int
  P= new int[nodes];
  Parc= new int[nodes];        // May be >1 arc between nodes
  int[] CL= new int[nodes];
  for (int i=0; i < nodes; i++) {
    D[i]= MAX_COST;
    P[i]= EMPTY;
    Parc[i]= EMPTY;
    CL[i]= NEVER_ON_CL;
  }
  // Initialize root node
  D[root]= 0;
  CL[root]=  END_OF_CL;
  int lastOnList= root;
  int firstNode= root;
```

Shortest path 2

```
do {
  double Dfirst= D[firstNode];
  for (int link= head[firstNode]; link < head[firstNode+1]; link++) {
    int outNode= to[link];
    double DoutNode= Dfirst+ time(link);  // Compute time()
    if (DoutNode < D[outNode]) {
      P[outNode]= firstNode;
      Parc[outNode]= link;                // Record new predecessor arc
      D[outNode]= DoutNode;
      int CLoutNode= CL[outNode];
      if (CLoutNode == NEVER_ON_CL || CLoutNode == ON_CL_BEFORE) {
        int CLfirstNode= CL[firstNode];
        if (CLfirstNode != END_OF_CL && (CLoutNode == ON_CL_BEFORE ||
            DoutNode < D[CLfirstNode])){
          CL[outNode]= CLfirstNode;
          CL[firstNode]= outNode; }
      else {
        CL[lastOnList]= outNode;
        lastOnList= outNode;
        CL[outNode]= END_OF_CL; }
    }
  }
}
int nextCL= CL[firstNode];
CL[firstNode]= ON_CL_BEFORE;
firstNode= nextCL;
} while (firstNode < END_OF_CL); }
```

equilibrium()

```
public void equilibrium() {
  final double CRITERION= 0.001;      // Convergence criterion
  final int MAX_ITERATIONS= 10;       // Set higher in real code
  // Step 0: Initialization
  xFlow= new double[arcs];            // Initialize xFlow = 0
  update();
  xFlow= directionFind();
  double convergence;
  int iterations= 0;
  do {
    // Step 1: Set times based on initial flows
    update();
    // Step 2: Direction finding
    yFlow= directionFind();
    // Step 3: Line search
    double alpha= lineSearch(this, 0.0, 1.0); // 0 <= alpha <= 1
    // Step 4: Move
    convergence= move(alpha);
    // Step 5: Check convergence
    iterations++;
  } while (convergence > CRITERION && iterations < MAX_ITERATIONS);
}
```

update(), time()

```
private void update() {
  for (int i= 0; i < arcs; i++) {
    arcTime[i]= time(i);
  }
}

private double time(int link) {
  final double TOLERANCE= 1E-8; // Sqrt of machine precision
  double time;
  if (xFlow[link] < TOLERANCE)
    time= timeBase[link];
  else {
    double delay= 1.0 + timeConst[link]*
        Math.pow(xFlow[link], timeExponent[link]);
    time= timeBase[link]*delay;
  }
    return time;
}
```

directionFind()

```
private double[] directionFind() {
  double[] flow= new double[arcs];
  // Assign trips on shortest path (set of arcs)
  for (int i= 0; i < nodes; i++) {  // Loop thru origin nodes
    shortHKNetwork(i); // Shortest path from i to all nodes
    for (int j= 0; j < nodes; j++) {
      if (i != j) {
        int pred= P[j];
        while (pred != EMPTY) {      // While not back at root
          // Add this flow to arcs in shortest path from O to D
          flow[Parc[j]] += ODtrips[i][j];
          // Find previous arc on path, until we reach the root
          pred= P[pred];
         }
      }
    }
  }
  return flow;
  // We have flows on all arcs, based on current travel times
}
```

lineSearch()

```
// Uses d.derivative—see next slide

private double lineSearch(MathFunction d, double a, double b) {
  final double TOLERANCE= 1E-8;  // Square root of machine precision
  final double MAX_ITERATIONS= 1000;
  double m;                            // Midpoint
  int counter= 0;
  for (m= (a+b)/2.0; Math.abs(a-b) > TOLERANCE; m= (a+b)/2.0) {
    counter++;
    if  (d.derivative(m)< 0.0)       // If derivative negative,
      a= m;                            // Use right subinterval
    else
      b= m;                            // Use left subinterval
    if (counter > MAX_ITERATIONS)
      break;
  }
  return m;
}
// There are better line searches such as Brent’s method (see
// Numerical Recipes 10.2-10.4) but bisection is simple and stable
```

derivative()

```
public double derivative(double alpha) {
  final double TOLERANCE= 1E-8;
  double deriv= 0.0;
  for (int i= 0; i < arcs; i++) {
    double time;
    double newFlow= xFlow[i]+ alpha*(yFlow[i]- xFlow[i]);
    if (newFlow < TOLERANCE)
      time= timeBase[i];
    else {
      double delay= 1.0 + timeConst[i]*
               Math.pow( newFlow, timeExponent[i]);
      time= timeBase[i]*delay;
    }
    deriv += (yFlow[i] - xFlow[i])*time;
  }
  return deriv;
}
```

```
public interface MathFunction {
   double derivative(double alpha);
}
```

move(), main()

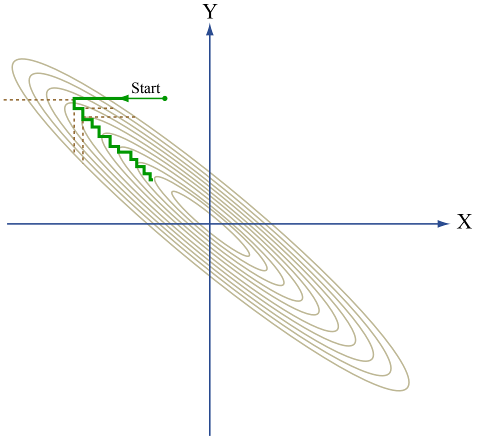
```
private double move(double alpha) {
   double sumFlows= 0.0;
   double sumRootMeanDiffFlows= 0.0;
   for (int i= 0; i < arcs; i++) {
          double flowChange= alpha*(yFlow[i] - xFlow[i]);
          xFlow[i]+= flowChange;
          sumFlows+= xFlow[i];
          sumRootMeanDiffFlows+= flowChange*flowChange;
   }
   // Compute convergence criterion here, since we have
   // current and previous xFlow
   return Math.sqrt(sumRootMeanDiffFlows)/sumFlows;
}

public static void main(String[] args) {
    NetworkEquilibrium g= new NetworkEquilibrium();
    g.equilibrium();
}
```

Summary

- **Convex combinations method (also known as Frank-Wolfe decomposition) solves network equilibrium problem**
  - **Flow taken from more congested paths, assigned to less congested paths, until flow changes are small**
  - **Process equalizes travel times on all paths for O-D pair**
  - **Uses shortest path code to solve linear program subproblem**
    - **Shortest path is very efficient even for large networks**
  - **Convex combinations method converges slowly**
    - **Faster methods exist but require more data storage and don't use shortest path subproblem as naturally**
- **Note the building blocks:**
  - **Shortest path algorithm, which uses graph, queue (candidate list) and tree data structures**
  - **Line search is a bisection (divide and conquer) algorithm**
  - **NetworkEquilibrium is the master problem, solved by reusing the building blocks listed above**
  - **Having a library of data structures and core algorithms is key to problem solving and design**

Performance of convex combinations



Successive minimizations along coordinate directions in a long, narrow valley

Figure by MIT OpenCourseWare.

Numerical Recipes

Performance of convex combinations

- Convergence is slow, even in our 3 link example
  - Objective function value changes little, but flows and times are accurate to only 2 places after 9 iterations
- Complexity is measured in two dimensions
  - Typical performance ~O(n), where n is number of arcs:
    - Running time increases linearly in usual experience
  - Linear convergence is claimed:
    - $\varepsilon_{n+1} = k\,\varepsilon_n^m$, m = 1
    - It depends on definition of ε
      - Objective function converges linearly
      - Arc flows and times appear to converge sublinearly
    - Linear convergence means each iteration gains one more significant figure
      - Sublinear means less than one more significant figure
      - Superlinear means more than one more significant figure

Congestion

- Highway networks have diseconomies of scale in urban areas
  - Congestion is major element in urban form, environmental quality, urban economics (agglomeration), and travel behavior
  - More trips mean lower service quality
- Transit networks on separate rights of way are often regarded as having economies of scale
  - More trips mean higher service frequency, which gives higher service quality
  - Eventually congestion occurs in transit also, as capacity is approached
- Extensions of network equilibrium formulations handle highway-transit demand equilibrium, variable demand, …
  - Sheffi covers many of them
  - Regional trade, etc. can also be modeled this way

1.204 Computer Algorithms in Systems Engineering
Spring 2010