EL736 Communications Networks II: Design and Algorithms

Class6: Optimization Methods (II)

Yong Liu

10/17/2007

Outline

- AMPL/CPLEX Package
  - introduction
  - example
- Stochastic Methods
  - Local Search
  - Simulated Annealing
  - Evolutionary Algorithms
  - Simulated Allocation

Solving LP/IP/MIP with CPLEX-AMPL

- CPLEX is the best LP/IP/MIP optimization engine out there.
- AMPL is a standard programming interface for many optimization engines.
- Student version windows/unix/linux
  - 300 variables limit
- Full version on wan.poly.edu
  - single license, one user at a time

Essential Modeling Language Features

- Sets and indexing
  - Simple sets
  - Compound sets
  - Computed sets
- Objectives and constraints
  - Linear, piecewise-linear
  - Nonlinear
  - Integer, network
- . . . and many more features

  Express problems the various way that people do

  Support varied solvers

Introduction to AMPL

- Each optimization program has 2-3 files
  - optprog.mod: the model file
    - Defines a class of problems (variables, costs, constraints)
  - optprog.dat: the data file
    - Defines an instance of the class of problems
  - optprog.run: optional script file
    - Defines what variables should be saved/displayed, passes options to the solver and issues the solve command

Running AMPL-CPLEX

- Start AMPL by typing ampl at the prompt
- Load the model file
  - ampl: model optprog.mod; (note semi-colon)
- Load the data file
  - ampl: data optprog.dat;
- Issue solve and display commands
  - ampl: solve;
  - ampl: display variable_of_interest;
- OR, run the run file with all of the above in it
  - ampl: quit;
  - prompt:~> ampl example.run

AMPL Example

- minimizing maximal link utilization

minimize	r	
subject to	$\sum_{p=1}^{P_d} x_{dp} = h_d$	$d = 1, 2, ..., D$
	$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq c_e r$	$e = 1, 2, ..., E$
	x_{dp}, r continuous, non-negative.	

AMPL: the model (I)

parameters

```
param D  > 0 integer;
param E  > 0 integer;
param N  > 0 integer;
param Pd > 0 integer;
#//############################################################
set Nodes       := 1..N;
set link_nos    := 1..E;
set demand_nos  := 1..D;
set route_nos   := 1..Pd;
```

links

```
#Generation of links
param link_src {link_nos} within Nodes;
param link_dest {link_nos} within Nodes;
param link_capacity {link_nos} >= 0 integer;
```

demands

```
#Generation of Demands
param demand_src {demand_nos} within Nodes;
param demand_dest {demand_nos} within Nodes;
```

routes

```
#Generation of Routes
set Routes{demand_nos,route_nos} within link_nos;
```

```
param h {demand_nos} >= 0 integer;
#############################################################
#Generation of the variables required for optimization
#formulation
#############################################################
```

incidences

```
param delta {e in link_nos, d in demand_nos, p in route_nos}
        = if e in Routes[d,p] then 1 else 0;
#############################################################
#Variables for the problem
#############################################################
```

flow variables

```
var x {d in demand_nos, p in route_nos} >= 0; var r >= 0;
```

AMPL: the model (II)

Objective

Constraints

```
##################################################################
minimize MaxLinkUtil: r;
##################################################################
#Constraints
##################################################################
subj to all_demands {d in demand_nos}:
    sum{p in route_nos} x[d,p] = h[d];
subj to capacity_constraints {e in link_nos}:
    sum{d in demand_nos} ( sum{p in route_nos} (delta[e,d,p]
            *x[d,p]))
            - link_capacity[e]*r <= 0;
```

AMPL: the data

```
data;
param D  := 2;
param E  := 3;
param N  := 3;
param Pd := 2;


param: link_src    link_dest   link_capacity    :
    1       1           2           20
    2       2           3           10
    3       3           1           10      ;

param: demand_src    demand_dest   h        :=
    1       1            2          12
    2       2            3          10      ;

set Routes[1,1] := 1;
set Routes[1,2] := 2 3;
set Routes[2,1] := 2;
set Routes[2,2] := 1 3;

end;
```

Stochastic heuristics

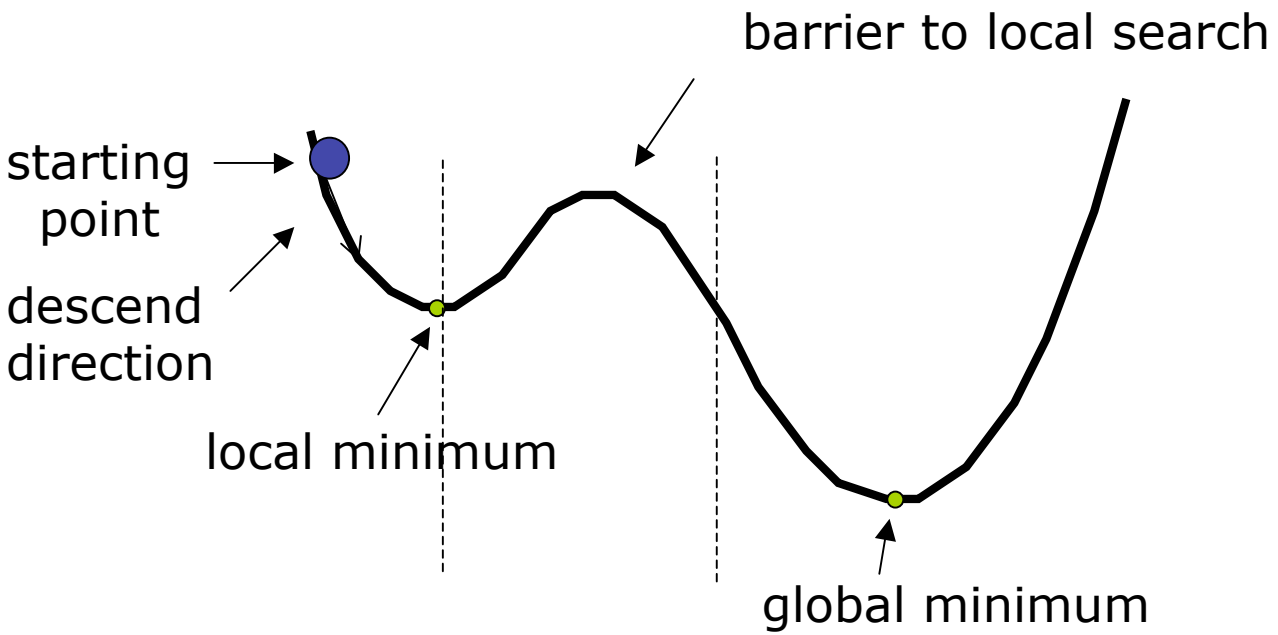
- Local Search
- Simulated Annealing
- Evolutionary Algorithms
- Simulated Allocation

Local Search: steepest descent

- minimize f(x)
- starting from initial point $x_c=x_0$
- iteratively minimize value $f(x_c)$ of current state x_c, by replacing it by point in its neighborhood that has lowest value.
- stop if improvement no longer possible.
- "hill climbing" when maximizing

Problem with Local Search

- may get stuck in local minima

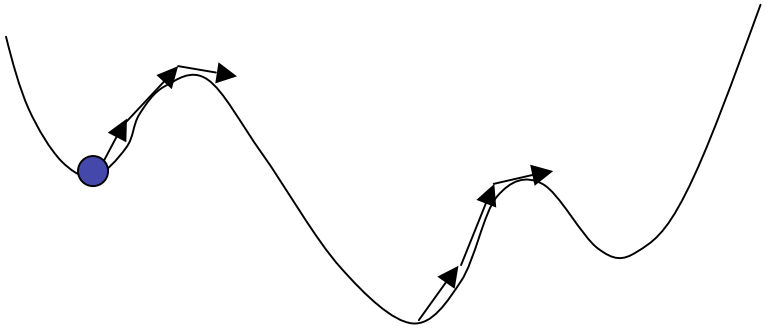


Question: How to avoid local minima?

What about Occasional Ascents?

desired effect

Help escaping the local optima.



adverse effect

Might pass global optima after reaching it

(easy to avoid by keeping track of best-ever state)

Simulated annealing: basic idea

- From current state, pick a random successor state;
- If it has better value than current state, then "accept the transition," that is, use successor state as current state;
- Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is lower as the successor is worse).
- So we accept to sometimes "un-optimize" the value function a little with a non-zero probability.

Simulated Annealing

Kirkpatrick et al. 1983:

- Simulated annealing is a general method for making likely the escape from local minima by allowing jumps to higher value states.
- The analogy here is with the process of annealing used by a craftsman in forging a sword from an alloy.

Real annealing: Sword

- He heats the metal, then slowly cools it as he hammers the blade into shape.
  - if he cools the blade too quickly the metal will form patches of different composition;
  - if the metal is cooled slowly while it is shaped, the constituent metals will form a uniform alloy.

Simulated Annealing - algorithm

- uphill moves are permitted but only with a certain (decreasing) probability ("temperature" dependent) according to the so called Metropolis Test

```
begin
  choose an initial solution i∈ S;
  select an initial temperature T > 0;
  while stopping criterion not true
    count := 0;
    while count < L
      choose randomly a neighbour j∈N(i);
       ΔF:= F(j) - F(i);
      if ΔF ≤ 0 then i := j
                else if random(0,1) < exp (-ΔF / T) then i := j;
      count := count + 1
    end while;
    reduce temperature (T:= T×α)
  end while
end
```

Metropolis test

Simulated Annealing - limit theorem

- limit theorem: global optimum will be found
- for fixed T, after sufficiently number of steps:
  - Prob { X = i } = exp(-F(i)/T) / Z(T)
  - $Z(T) = \Sigma_{j \in S} \exp(-F(j)/T)$
- for $T \rightarrow 0$, Prob { X = i } remains greater than 0 only for optimal configurations $i \in S$

this is not a very practical result:

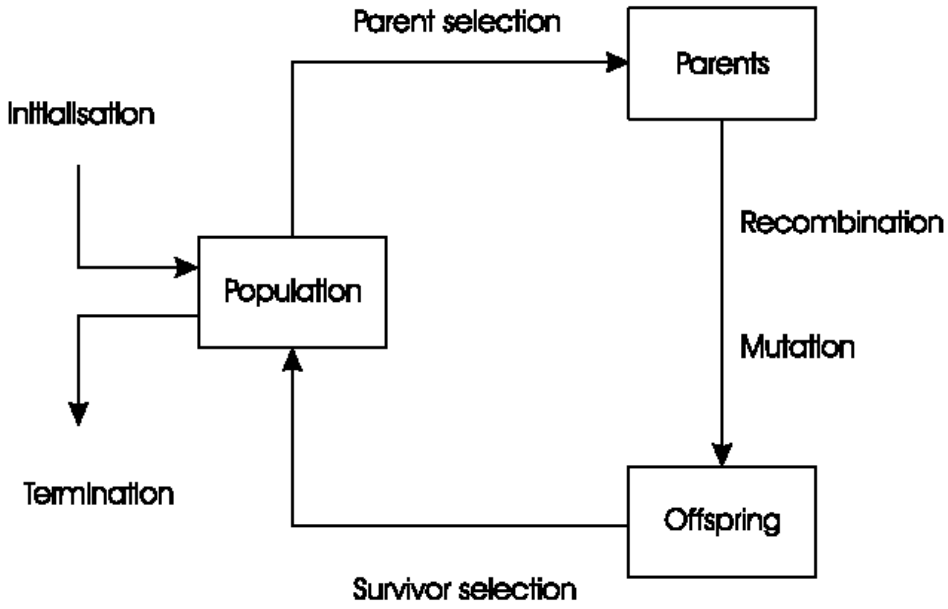
too many moves (number of states squared) would have to be made to achieve the limit sufficiently closely

Evolution Algorithm: motivation

- A population of individuals exists in an environment with limited resources
- Competition for those resources causes selection of those fitter individuals that are better adapted to the environment
- These individuals act as seeds for the generation of new individuals through recombination and mutation
- The new individuals have their fitness evaluated and compete (possibly also with parents) for survival.
- Over time Natural selection causes a rise in the fitness of the population

Evolution Algorithm: general schemes

- EAs fall into the category of "generate and test" algorithms
- They are stochastic, population-based algorithms
- Variation operators (recombination and mutation) create the necessary diversity and thereby facilitate novelty
- Selection reduces diversity and acts as a force pushing quality



Evolutionary Algorithm: basic notions

- **population = a set of μ chromosomes**
  - **generation = a consecutive population**
- **chromosome = a sequence of genes**
  - **individual, solution (point of the solution space)**
  - **genes represent internal structure of a solution**
  - **fitness function = cost function**

Genetic operators

- **mutation**
  - **is performed over a chromosome with certain (low) probability**
  - **it perturbs the values of the chromosome's genes**
- **crossover**
  - **exchanges genes between two parent chromosomes to produce an offspring**
  - **in effect the offspring has genes from both parents**
  - **chromosomes with better fitness function have greater chance to become parents**

In general, the operators are problem-dependent.

(μ + λ) - Evolutionary Algorithm

begin
    n:= 0; initialize(P_0);
        while stopping criterion **not** true
        O_n:= $\emptyset$;
        for i:= 1 **to** λ **do** O_n:= $O_n \cup$crossover(P_n):
        for $\varepsilon \in O_n$ **do** mutate(ε);
        n:= n+1,
        P_n:= select_best($O_n \cup P_n$);
    end while
end

Evolutionary Algorithm for the flow problem

- Chromosome: $\mathbf{x} = (x_1, x_2, \ldots, x_D)$
- Gene:

  $x_d = (x_{d1}, x_{d2}, \ldots, x_{dPd})$ - flow pattern for the demand d

5	2	3	3	1	4	
1	2	0	0	3	5	
1	0	2		1		chromosome
2		3				

Evolutionary Algorithm for the flow problem cntd.

❑ crossover of two chromosomes

- each gene of the offspring is taken from one of the parents
  - for each d=1,2,…,D: $x_d := x_d(1)$ with probability 0.5
    $x_d := x_d(2)$ with probability 0.5
- better fitted chromosomes have greater chance to become parents

❑ mutation of a chromosome

- for each gene shift some flow from one path to another
- everything at random

Simulated Allocation

- Modular Links and Modular Flows Dimensioning

***LP:* D/ML/MF** **Link-Path Formulation**

Modular Links and Modular Flows

indices

$d = 1, 2, ..., D,$	demands
$p = 1, 2, ..., P_d$	candidate paths for demand d
$e = 1, 2, ..., E$	links

constants

δ_{edp}	= 1, if link e belongs to path p realizing demand d; 0, otherwise
L	size of the demand flow capacity module
h_d	volume of demand d expressed as the number of demand modules
ξ_e	cost of one capacity module on link e
M	size of the link capacity module

variables

x_{dp}	number of demand modules allocated to path p of demand d
y_e	capacity of link e expressed in the number of modules

objective

$$\min \boldsymbol{F} = \sum_e \xi_e y_e \qquad (5.3.6a)$$

constraints

$$\sum_p x_{dp} = h_d \qquad d = 1, 2, \ldots, D \qquad (5.3.6b)$$

$$\sum_d \sum_p \delta_{edp} x_{dp} \leq (M/L) y_e \qquad e = 1, 2, \ldots, E \qquad (5.3.6c)$$

x_{dp} and y_e non-negative integers (5.3.6d)

SAL: general schemes

- Work with partial flow allocations
  - some solutions NOT implement all demands
- In each step chooses, with probability $q(x)$, between:
  - *allocate*(x) – adding one demand flow to the current state x
  - *disconnect*(x) – removing one or more demand flows from current x
- Choose best out of N full solutions

SAL: algorithm

ALGORITHM 5.6: Simulated Allocation (SAL)

```
procedure SAL
begin
  n := 0; x := 0; F^best := +∞;
  repeat
    if random(0, 1) < q(|x|) then allocate(x) else disconnect(x);
    if |x| = H then
      begin
        n := n + 1;
        if F(x) < F^best then
          begin
            F^best := F(x);
            x^best := x
          end
      end
  until n = N or F^best ≤ cost_lower_bound
end { procedure }
```

$$q(0) = 1; \quad q(H) = 0; \quad \frac{1}{2} < q(k) \leq 1, 0 < k < H$$

SAL: details

- allocate(x)
  - randomly pick one non-allocated demand module
    - $prob.(d) = (h_d - \sum_p x_{dp})/(H - |\mathbf{x}|) \quad d = 1, 2, \cdots, D.$
  - allocate demand to the shortest path
    - link weight 0 if unsaturated
    - link weight set to the link price if saturated
  - increase link capacity by 1 on saturated links
- disconnect(x)
  - randomly pick one allocated demand module
    - $prob.(d) = \sum_p x_{dp}/|\mathbf{x}| \quad d = 1, 2, \cdots, D.$
  - disconnect it from the path it uses
  - decrease link capacity by 1 for links with empty link modules