CS5203 Optimisation

Paul Ahern

May 31, 2007

Abstract

Notes from lectures.

Contents

1 Lecture 11 January 2007

Lecturer Steve Prestwich

Subject Selected topics in Optimisation

Book Operations Research Models and Methods

1.1 Background

Constraint Based programming was developed in the 1980s to improve Logic Programing (LP, e.g. PROLOG).

This led to Constraint Logic Programming (CLP) which in turn begot Constraint Programming (CP).

In general CP is very good at solving feasibility problems, but is not the best approach for optimisation. Though there is considerable overlap between CP and Operations Research.

Operations Research (OR) was developed from 1936 by the RAF to process the data provided by radar stations. This proved so successful that it was extended to other fields throughout World War II.

After the war the Rand Corporation in the US developed OR further.

1.2 Linear Programming

Programming in the sense of planning or organising. Term predates computer programming.

A linear program is in the form: Maximize (or minimize) an Objective Function ($\Sigma_{j=1}^{n} c_j x_j$) subject to Linear Constraints ($\Sigma_{i=1}^{n} a_{ij} x_i \leq or = or \geq b_i$), Upper Bounds ($x_j \leq u_j$) and Non-negativity Restrictions ($x_j \geq 0$).

x_j are called Decision Variables and contain Real values. a_{ij}, b_i and c_j are constants.

1.2.1 Example

A company has three products P, Q and R. How much of each should be produced each week to maximize profits, given the following information:

	P	Q	R
Profit per unit	45	60	50
Maximum number of sales per week	100	40	60

The company has four machines. Each product has to pass through all four to be completed. Each machine is only available 2,400 minutes per week.

	Per unit processing time			
Machine	P	Q	R	Machine Availability
A	20	10	10	2,400
B	12	28	16	2,400
C	15	6	16	2,400
D	10	13	0	2,400

The factory costs 6,000 per week to run.

Linear program: $maximize\, 45P + 60Q + 50R$, subject to

$$P \leq 100, Q \leq 40, R \leq 60,$$

$$P \geq 0, Q \geq 0, R \geq 0,$$

$$20P + 10Q + 10R \leq 2,400,$$

$$12P + 28Q + 16R \leq 2,400,$$

$$15P + 6Q + 16R \leq 2,400,$$

$$10P + 13Q \leq 2,400.$$

Pass Linear Model to a standard solver and get the result: $P = 81.82, Q = 16.36, R = 60$. Thus Objective Function is $7,664$. $\therefore$

1. Profit is $1,664$.

2. Machine Usage:
   A is used for $20P + 10Q + 10R = 2,400$.
   B is used for $2,400$.
   C is used for $2,285$.
   D is used for $1,064$.
   Note that machines C and D have some idle time.

3. Products Made:
   P: 81.82 made but could sell 100.
   Q: 16.36 made but could sell 40.
   R: 60 made and could only sell 60.

Note that machines A and B run at full capacity so their constraints are only just satisfied. These are *tight* (or *active*) constraints. They are a bottleneck - they limit profits.

The market constraint on R is also tight. The upper bound 60 on R is also tight.

1.2.2 Abstractions, Assumptions, Simplifications

- We cannot make fractions of P, Q and R - they are units. We can round off to $P = 82, Q = 16$ and $R = 60$. For some problems this may violate a constraint or it may drastically change the objective function.[1]
- Everything is linear (the model takes no account of economies of scale).
- Exact market demand (this is surely unrealistic).
- Complications ignored (e.g. we assume that all combinations of P, Q and R can be scheduled through A, B, C and D with perfect efficiency - no queuing).

2 Lecture 15 January 2007

Jensen and Bard, Chapter 2.

By the end of the course we should be able to model new problems and find solutions.

2.1 Linear Programming

Understanding LPs by plotting graphs.

Taking the example from the previous lecture and assigning $R = 60$ to reduce the problem to one with two variables (which can be plotted on a two-dimensional graph) the problem becomes:

Maximize $Z = 45P + 60Q$,

subject to $20P + 10Q \leq 1800$ (A); $12P + 28Q \leq 1440$ (B); $15P + 6Q \leq 1440$ (C); $10P + 15Q \leq 2400$ (D); $P \leq 100$; $Q \leq 40$; $P \geq 0$; $Q \geq 0$.

In general, each constraint defined a different polyhedron.

Optimal feasible solution is always a vertex (corner) of the polyhedron. So we only need to check Z at each vertex (basically this is how linear solution software works). There is a finite number of vertexes.

[1] There is a nice method for solving linear problems, but it cannot be used if the variables have to be integers. Forcing integrity makes the problem much harder.

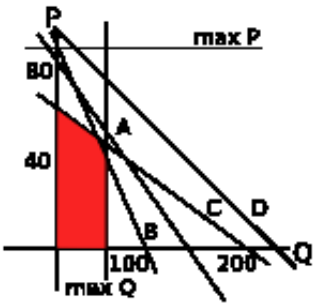

Figure 1: Polyhedron

2.2 Some more LP examples

2.2.1 Isovalue Contour parallel to Polyhedron side

Maximize $Z = 3X_1 - X_2$

Subject to $15X_1 - 5X_2 \leq 30$; $10X_1 + 30X_2 \leq 120$; $X_1 \geq 0$; $X_2 \geq 0$.

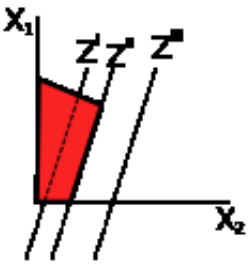

Figure 2: Many Solutions

In this case we have an infinite number of optimal feasible solutions. Algorithm checking vertexes still works and finds two optimal solutions.

2.2.2 Feasible Area Infinite

Maximize $Z = -X_1 + X_2$

Subject to $-X_1 + 4X_2 \leq 10$; $-3X_1 + 2X_2 \leq 2$; $X_1 \geq 0$; $X_2 \geq 0$.

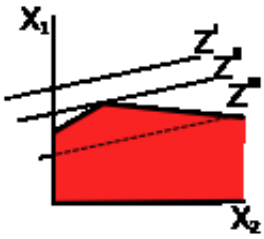

Figure 3: Infinite Area

Feasible area is infinite but there is only one optimal feasible solution.

2.2.3 No Optimal Feasible Solution

Maximize $Z = X_1 - X_2$

Subject to $3X_1 + X_2 \geq 10$; $X_1 \leq 4$; $X_1 \geq 0$; $X_2 \geq 0$.

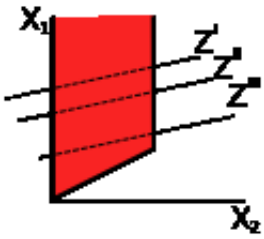

Figure 4: Infinite Solution

In this example there is no optimal feasible solution as the feasible solution is infinite.

2.2.4 No Feasible Region

Maximize $Z = X_1 + X_2$

Subject to $3X_1 + X_2 \geq 6$; $3X_1 + X_2 \leq 3$; $X_1 \geq 0$; $X_2 \geq 0$.

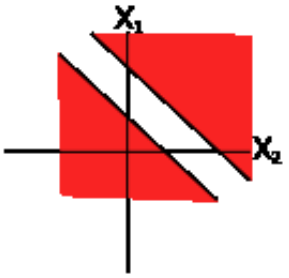

Figure 5: None Feasible

2.2.5 No Feasible Region 2

...$X_1 \geq 0$; $X_2 \geq 0$.

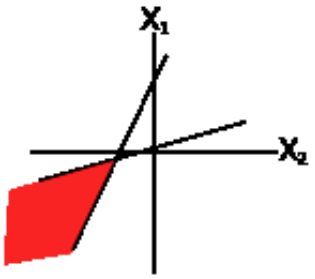

Figure 6: None Feasible 2

2.3 Blending Problem

Problem Blending ingredients to get a mixture with certain properties. In this example, animal feed.

The feed must contain between 0.8% and 1.2% of Calcium; At least 22% Protein and at most 5% Fibre. Minimize cost.

Ingredients	Calcium	Protein	Fibre	Unit Cost
Limestone (L)	0.380	0.00	0.00	10.0
Corn (C)	0.001	0.09	0.02	30.5
Soybean Meal (S)	0.002	0.50	0.08	90.0

What proportion of each ingredient should be in a unit of feed?

Minimize $Z = 10L + 30.5C + 90S$

Subject to $0.380L + 0.001C + 0.002S \geq 0.008$ (Calcium); $0.380L + 0.001C + 0.002S \leq 0.012$ (Calcium); $0.09C + 0.50S \geq 0.22$ (Protein); $0.02C + 0.08S \leq 0.05$ (Fibre); $L + C + S = 1$ (Proportion Constraint); $L \geq 0; C \geq 0; S \geq 0$.

Result: $L = 0.03$; $C = 0.65$; $S = 0.32$.

Substituting into Z, the cost is 0.49.

3 Lecture 22 January 2007

Jensen and Bard, Chapter 2.
More LP examples

3.1 Car Rental Problem

A company expects several visitors and must rent cars for all of them:

Day	Sat	Sun	Mon	Tue	Wed	Thu	Fri
Cars	2	5	10	9	16	7	11

A car rental company has different rental plans:

Plan	Description	Cost
1	Daily Sat-Sun	35
2	Daily Weekdays	50
3	3 Consecutive Weekdays	125
4	Weekend (Sat+Sun)	60
5	All Weekdays	180
6	All Week	200

Objective: Rent enough cars for all visitors; minimise cost.

Define Variables: X_j is the number of cars rented on day j ($j : 1 \ldots 7$). Y_k is the number of cars rented for three consecutive weekdays ($k : 3, 4, 5$). e.g. Y_3 is the number of cars rented Mon-Wed). W_E is the number of cars rented for whole weekend. W_D is the number of cars rented for all weekdays. W is the number of cars rented for the whole week.

One variable per plan, except for the first two which are both daily.
Covering Constraints, i.e. enough cars are rented each day:
(Sat) $X_1 + W_e + W \geq 2$;
(Sun) $X_2 + W_e + W \geq 5$;
(Mon) $X_3 + Y_3 + W_d + W \geq 10$;
(Tue) $X_4 + Y_3 + Y_4 + W_d + W \geq 9$;
(Wed) $X_5 + Y_3 + Y_4 + Y_5 + W_d + W \geq 16$;
(Thur) $X_6 + Y_4 + Y_5 + W_d + W \geq 7$;
(Fri) $X_7 + Y_5 + W_d + W \geq 10$.
Non-negativity Constraints: $X_j, Y_k, W_E, W_D, W \geq 0$.
Minimize $Z = 35X_1 + 35X_2 + 50X_3 + 50X_4 + 50X_5 + 50X_6 + 50X_7 + 125Y_3 + 125Y_4 + 125Y_5 + 60W_E + 180W_D + 200W$.
Result (Day-Cars): Sat-0, Sun-0, Mon-5, Tue-0, Wed-9, Thu-0, Fri-4. MonWed-0, TueThu-0, WedFri-2, Weekend-0, Weekday-0, Week-5.

3.2 Power Distribution Problem

Regional Power System, 3 generating stations (A, B, C) each serving its own area. Three outlying areas (X, Y, Z) with no stations. They have demands of 25, 50 and 30 MW.
Each station can supply:

Station	Capacity[2]	Unit Cost
A	100MW	500
B	75MW	475
C	100MW	400

Some stations can supply only some outlying areas: $X \to A\&C$; $Y \to B\&C$; $Z \to A\&B$.

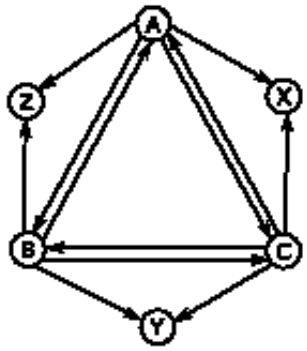

Figure 7: Powerflow Connections

Supplying from any of A, B, C to any X, Y, Z has 10% powerloss. Supplying from any station to another has 5% powerloss.

Find an optimal plan to supply X, Y, Z; minimize cost.

This is an example of the Generalised Network Flow model.

Define some variables:

Flows from Station to Station - $U_{AB}, U_{AC}, U_{BA}, U_{BC}, U_{CA}, U_{CB}$.

Flows from Station to area - $V_{AX}, V_{CX}, V_{BY}, V_{CY}, V_{AZ}, V_{BZ}$.

Power production - P_A, P_B, P_C.

Conservation Constraints (each station generates as much power as it outputs):

$U_{AB} + U_{AC} + V_{AY} + V_{AZ} - 0.95U_{BA} - 0.95U_{CA} = P_A$.

$U_{BA} + U_{BC} + V_{BY} + V_{BZ} - 0.95U_{AB} - 0.95U_{CB} = P_B$.

$U_{CA} + U_{CB} + V_{CX} + V_{CY} - 0.95U_{AC} - 0.95U_{BC} = P_C$.

$-0.9V_{AX} - 0.9V_{CX} = -25$.

$-0.9V_{BY} - 0.9V_{CY} = -50$.

$-0.9V_{AZ} - 0.9V_{BZ} = -30$.

Upper Bounds on output of stations: $P_A \leq 100; P_B \leq 75; P_C \leq 100$.

Non-negativity Constraints: $P_A \geq 0; P_B \geq 0; P_C \geq 0$.

Minimize $Z = 500P_A + 475P_B + 400P_C$.

Result: Station C, the least expensive resource, generates up to its limit of 100MW. This power serves the needs of areas X and Y, with the remaining 16.66MW being transshipped to station A. The latter quantity is then transmitted directly to area Z incurring an additional 10% loss on the way. Evidently, station A serves only its local area as a result of its high operating costs.

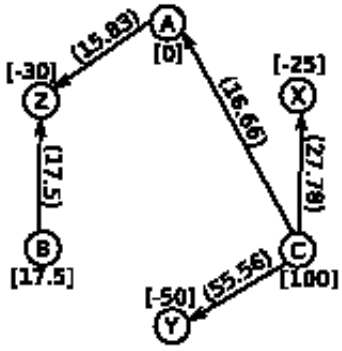

Figure 8: Powerflow Result

4 Lecture 25 January 2007

Jensen and Bard, Chapter 3 (Simplex) & 7 (Integer Programming).

4.1 Solving LP

Two approaches:

- Simplex (1947) works by exploring the feasible polyhedron's boundaries.
- Interior Point Methods starts inside the feasible region (guaranteed to finish in polynomial time).

Simplex is not proved to be polynomial but is usually the fastest method. Interior Point is particularly good at finding solutions efficiently for sparse patterns.

4.2 Integer Programming

In IP or ILP (Integer Linear Programming) variables can only take discrete values (usually integers - sometimes only 0 and 1).

If some variables are continuous and some discrete it is called a Mixed Integer Problem (MIP or MILP). A MIP is an LP and constraints, e.g. $V \in \{0, 1\}$.

MIP are harder to solve than LP. There is no known polynomial algorithm for MIP or IP. They are NP-hard problems.

MINLP - Mixed Integer Non-Linear Programming - is a MIP with some non-linear constraints, e.g. $X^2 + 3yz \leq 2$.

Quadratic Programming (QP): Constraints are linear but the Z function is a quadratic.

4.3 Site Selection Problem

A company will build new buildings at four sites (1,2,3,4). It has a choice of three designs (A, B, C).

Options	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3	C4
Income	6	7	9	11	12	15	5	8	12	16	19	20
Investment	13	20	24	20	39	45	12	20	30	44	48	55

Maximize total income; But total investment must be less than 100.

Notation:

- variable y_{ij}: $y_{ij} = 1$ means we use design i at site j.
  $y_{ij} = 0$ means we do not.

- constants Income P_{ij}, Investment a_{ij}.

The integer program: maximize $Z = \Sigma_i \Sigma_j P_{ij} y_{ij}$
Subject to $\Sigma_i \Sigma_j a_{ij} \leq 100$; $y \in \{0, 1\}$.
Solution $y_{A1} = 1; y_{A3} = 1; y_{B3} = 1; y_{B4} = 1; y_{C1} = 1$; others are zero.
But some sites have more than one design!
Need more constraints $\Sigma_i y_{ij} \leq 1$ - for each j. These are Logic Constraints.
Suppose the company come up with some additional constraints:

1. Site 2 must have a building.

2. Design A can only be used at 1,2 or 3 if it is also used at 4.

3. Only use two designs.

Add more constraints:

1. $\Sigma_i y_{i2} = 1$.

2. $y_{A1} + y_{A2} + y_{A3} \leq 3 y_{A4}$. (If $y_{A4} = 0$ then LHS also zero).

3. Third constraint is tricky to specify using the y variables. Use additional auxiliary variables (not needed to describe problem, but to help modelling) w_i; $w_i = 1$ means that design i is used; $w_i = 0$ means it isn't.
   The constraint is $\Sigma_i w_i \leq 2$.
   Also have to relate w and y: $\Sigma_i y_{ij} \leq 4 w_i$.

New solution: $y_{A1} = 1; y_{A4} = 1; y_{B2} = 1; y_{B3} = 1$; all others zero. $w_A = 1; w_B = 1$.

4.4 Logical Constraints

Modelling tricks to model logical relationships between variables.
Let the decision variables be $y_1 \ldots y_n \in \{0, 1\}$.

1. These decisions are mutually exclusive: $\Sigma_{i=1}^{n} y_i \leq 1$.

2. More generally, at most k decisions can be true: $\Sigma_{i=1}^{n} y_i \leq k$.

3. At least k decisions can be true: $\Sigma_{i=1}^{n} y_i \geq k$.

4. Exactly k decisions can be true: $\Sigma_{i=1}^{n} y_i = k$.

5. If any (one or more) of these are true then so is w: $\Sigma_{i=1}^{n} y_i \leq nw$.

5 Lecture 29 January 2007

Jensen and Bard, Chapter 7 (Integer Programming).

5.1 Logical Constraints

1. If all of these are true then so is w: $\Sigma_{i=1}^{n} y_i \leq (n-1) + w$.
2. If at least k of these are true then so is w: $\Sigma_{i=1}^{n} y_i \leq (k-1) + (n-(k-1))w$.
3. In a MIP consider $x \leq uy$ where x is real; y is binary; u is an upper-bound on x ($x \leq u$): If $x > 0$ then $y = 1$.
4. If the decision y_1 is true (1) and y_2 is false (0) then do w: $y_1 + (1 - y_2) \leq w + 1$.
5. **"Big-M" model** Suppose r is a positive Real; b is a binary. If $b = 1$ then $r = 100$, but if $b = 0$ then $r = 0$: $100 \leq r + M(1-b)$; $r \leq 100b$.
   Check: If $b = 0$, $100 \leq r + M$; $r \leq 0$ (i.e. $r = 0$). If $b = 1$, $100 \leq r$; $r \leq 100$ (i.e. $r = 100$).
   Big-M models are hard to solve.
6. Suppose we want x to take discrete values that are not consecutive integer, e.g. $\{1, 3, 4.5, 16\}$: Introduce four new binary variables y_1, y_2, y_3, y_4. Constraints $x = y_1 + 3y_2 + 4.5y_3 + 16y_4$; $y_1 + y_2 + y_3 + y_4 = 1$.

5.2 Maximum Density Still Life

In the game of Life, find an unchanging pattern, in a finite area, with the most living cells.

Game Rules:

1. A cell with two living neighbours does not change.
2. A cell with three living neighbours becomes living in the next iterations.
3. Any other cell becomes dead.

Model giving good results: For each cell e define a binary variable x_e (in a 10×10 area $e \ldots 100$).

Constraints

Death by Isolation: A cell with less than two live neighbours must be dead (otherwise it would change in the next iteration): $2x_e - \Sigma_{f \in N(e)} x_f \leq 0$ ($N(e)$ returns the set of e's neighbours).

Death by Overcrowding: A cell with more than three live neighbours must be dead: $3x_e + \Sigma_{f \in N(e)} x_f \leq 6$.

Birth rule: A cell with three live neighbours must be live: $-x_e + \Sigma_{f \in S} x_f - \Sigma_{f \in N(e)-S} x_f \leq 2$ (S any three member set of $N(e)$).

No cell outside the area may become live: $x_g + x_h + x_i \leq 2$; where g, h, i are any three contiguous cells at the boundary of the area.

Maximize $Z = \Sigma_e x_e$.

A 13×13 area was solved in ten hours using integer programming software.

Example 8×8 solution:

1	1	0	1	1	0	1	1
1	1	0	1	1	0	1	1
0	0	0	0	0	0	0	0
1	1	0	1	1	0	1	1
1	1	0	1	1	0	1	1
0	0	0	0	0	0	0	0
1	1	0	1	1	0	1	1
1	1	0	1	1	0	1	1

5.3 Peaceable Armies of Queens

From the Optima Journal: $N \times N$ chessboard; Place two equal sized armies of queens so that no black queen can attack a white queen. Maximize size of armies.

Model giving good results: For each square (i, j) on the board define two binary variables: $b_{ij} = 1$ if there is a black queen in that square (otherwise zero); Likewise for $w_{ij} = 1$ and white queens.

Maximize $Z = \Sigma_i \Sigma_j b_{ij}$

Subject to $\Sigma_i \Sigma_j b_{ij} = \Sigma_i \Sigma_j w_{ij}$

For any two attacking squares (i, j) and (i', j'): $b_{ij} + w_{i'j'} \leq 1$.

This includes (i, j) and (i', j') lying on the same row, column, diagonal or being the same square.

9×9 boards have been solved in a reasonable amount of time by standard IP software. Specialist software has been used to solve 12×12 boards.

Example 12×12 solution:

.	.	.	b	b	b	.	.	.	.	.	b
.	.	.	b	b	b	.	.	.	.	b	.
.	.	.	b	b	b	.	.	.	b	.	b
.	.	.	.	b	.	.	.	.	.	b	b
.	.	.	.	.	.	.	.	.	b	b	b
.	.	.	.	.	.	.	.	.	b	b	.
.	.	w	.	.	.	w	.	.	.	.	.
.	w	w	.	.	.	.	.	.	.	.	.
w	w	w	.	.	.	.	.	w	.	.	.
w	w	.	.	.	.	.	w	w	.	.	.
w	.	w	.	.	.	w	w	w	.	.	.
.	w	.	.	.	.	w	w	w	.	.	.

6 Lecture 1 February 2007

6.1 Steel Mill Slab Design

Slabs of steel are produced in a few weights; then they are cut into pieces to fulfill orders. Given these orders, pack orders onto slabs, minimize the wastage (or total slab size).

Each order is given a property called "colour" representing the route it follows through the mill. No slab can be assigned more than p colours (e.g. $p = 2$).

We don't know how many slabs we need. This makes it hard to define variables. Choose a maximum number, d, of slabs, some might not be used. (e.g. divide the smallest slab size into the total order size).

Variables:

Weight of slab j is S_j.

$x_{ij} = 1$ if order i is assigned to slab j.

Numbers: $k_i$is colour of order i. w_i is weight of order i.

Constraints:

Each order is assigned to one slab - $\Sigma_i x_{ij} = 1$.

Slab capacity must be respected - $\Sigma_i w_i x_{ij} \leq S_j$.

For colour we need variables - $c_{kj} = 1$ if an order of colour k is assigned to slab j - $x_{ij} \leq c_{kj}$.

No more than p colours per slab - $\Sigma_k c_{kj} \leq p$.

Objective function: Minimize $Z = \Sigma_j S_j$.

6.2 Warehouse Location Problem

See section 7.4 in Jensen and Bard

A company wants to open warehouses to supply its stores and has a choice of locations. Each warehouse has the same maintenance cost and capacity (maximum number of stores it can supply). Each store is supplied by exactly one warehouse. The cost depends on the warehouse location.

Decide which warehouses to open and which warehouse should supply each store, minimize maintenance and supply costs.

Binary Variables: $O_i = 1$ if warehouse i is open (i.e. location i is used); $S_{ij} = 1$ if warehouse i supplies store j.

Constraints:

Each store supplied by one warehouse - $\Sigma_i S_{ij} = 1$.

Warehouse capacities - $\Sigma_j S_{ij} \leq C_i$(C is number of stores warehouse i can supply)

Only open warehouses can supply stores - $S_{ij} \leq O_i$ (alternatively, to use fewer constraints, $\Sigma_j S_{ij} \leq O_i$)

Objective function: Minimize $Z = k\Sigma_i O_i + \Sigma_i \Sigma_j S_{ij} k_i$ (first k is maintenance cost, second is supply costs)

6.3 Cutting Stock Problem

Jensen and Bard, page 243.

Paper company sells rolls of paper of fixed length and five possible widths: 5, 8, 12, 15, 17 (feet). These are cut from 25 foot wide rolls. All orders must be cut from these.

There are eleven ways of cutting:

Rolls	Wastage
17, 8	0
17, 5	3
15, 8	2
15, 5, 5	0
12, 12	1
12, 8, 5	0
12, 5, 5	3
8, 8, 8	1
8, 8, 5	4
8, 5, 5, 5	2
5, 5, 5, 5, 5	0

Demand:

Width	Demand
5	40
8	35
12	30
15	25
17	20

Meet demands and minimize wastage (or number of big rolls).

7 Lecture 5 February 2007

Cutting Stock Problem continued.

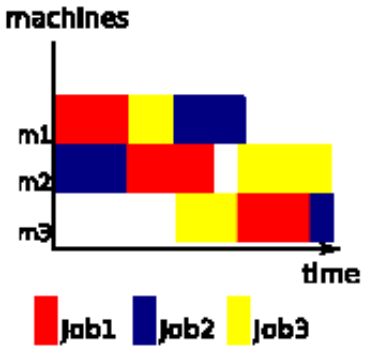

Figure 9: Sample 3x3 solution

Define eleven integer variables X_i to represent the number of rolls cut in each pattern.

Objective function: Minimize $Z = x_1 + \ldots + x_{11}$.

Subject to $x_2 + {}_4 + x_6 + 2x_7 + x_9 + 3x + 5x_{11} \geq 40$ (demand for width 5 rolls); $x_1 + x_3 + x_6 + 3x_8 + 2x_9 + x_{10} \geq 35$ (demand for width 8 rolls); $2x_5 + x_6 + x_7 \geq 30$ (demand for width 12 rolls); $x_3 + x_4 \geq 25$ (demand for width 15 rolls); $x_1 + x_2 \geq 20$ (demand for width 17 rolls).

There are better techniques than IP to solve these sorts of problems, as the numbers of variables can get very large.

7.1 Job-Shop Scheduling Problem (JSP)

An $n \times m$ JSP: n jobs, m tasks and resources. Each task has a duration D and has to be done on a particular resource (i.e. machine). Each resource can only be used by one task at a time. Each job's tasks must be done in order. Tasks, for the same job, cannot overlap in time.

Find a solution minimizing the total time taken: The MAKESPAN.

Model with IP (not necessarily the best way).

Represent time as integers $0 \ldots T-1$ (Guess a large T).

Define integer variables: S_{ij}: start time of task j in job i.

Non-negativity and upper-bounds: $S_{ij} \geq 0$ and $S_{ij} \leq T$.

Tasks in each job are in correct order: $S_{ij} - S_{ij-1} \geq D_{ij-1}$.

No resource is doing more than one task at once. Resource capacity constraint (remember that each task is on a specific machine): $(S_{ij}+D_{ij} \leq S_{i'j}) \oplus (S_{i'j}+D_{i'j} \leq S_{ij})$.

However $\oplus$ (XOR) is not in IP form, so define $0 \ldots 1$ variables $V_{ii'j}$ $(i < i')$ instead.

If $V_{ii'j} = 1$ then S_{ij} ends before $S_{i'j}$ begins; else vice-versa: $S_{ij} + D_{ij} - TV_{ii'j} \leq S_{i'j}$; $S_{i'j} + D_{i'j} - T(1 - V_{ii'j}) \leq S_{ij}$. (using T as a big-M)

Makespan integer variable m.

Objective function: Minimise m where $S_{ij} + D_{ij} \leq m$ for all tasks.

Additional (redundant) constraints sometimes help the software to find a solution more quickly. For example, in this case, the start times must be after the duration of the predecessors and the task and its successors must end before T: $\Sigma_{j'=1}^{j-1} D_{ij'} \leq S_{ij} \leq T - \Sigma_{j'=j}^{m} D_{ij'}$.

7.2 Travelling Salesman Problem (TSP)

Jensen & Bard, page 238.

Minimize total distance travelled to visit a set of cities (or minimize the cost of the journey: going from A to B may have a different cost than going from B to A - this is an asymmetric TSP).

Represent costs by a matrix:

	1	2	3	4
1	-	27	43	16
2	7	-	16	1
3	20	13	-	35
4	21	16	25	-

Many problems can be modelled using TSP, e.g. airplane takeoff order from an airport (big planes create more turbulence but are also more resistant to it; a light plane has to wait longer than a jumbo to take off after another jumbo).

Solution to the TSP is an optimal Hamiltonian cycle on a graph.

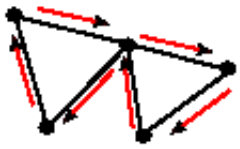

Figure 10: Graph

On a graph one can plan a tour which visits every vertex and returns. Hamiltonian path means that you don't have to return to your start point.

8 Lecture 12 February 2007

TSP continued.

G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. (1954)

For each trip between cities i and j define a binary variable: $x_{ij} = 1$ is trip is made from i to j; zero otherwise.

Distance matrix C_{ij}. Number of cities is n.

Objective function is to minimize $Z = \Sigma_{i=1}^{n} \Sigma_{j=1}^{n} C_{ij} x_{ij}$.

Such that

- Arrive at city i from exactly one j: $\Sigma_{j=1}^{n} x_{ij} = 1$.
- From city i go to exactly one city j: $\Sigma_{i=1}^{n} x_{ij} = 1$.
- Visit each city only once. Prevent any tour that doesn't involve all cities. For a sub-tour S: $\Sigma_{i \in S} \Sigma_{j \in S} x_{ij} \leq |S| - 1$. (Need one of these constraints for every possible sub-tour. Not all trips in sub-tour can be made. Only need sub-tours

involving up to half the number of cities - the remaining cities would be in another small sub-tour. So there would be some $2^{n-1}-n-1$ such constraints).

This model only works for small problems - But constraints can be generated as needed and not put into a file.

There are at least eight different ways of modelling TSP as IP. Here is another, a MIP called the SEQUENTIAL FORMULATION.

Discovered in 1960 by Miller, Tucker and Zemlin.

Drop the sub-tour constraints. Define continuous variables u_i. If $u_i < u_j$ then we visit city i before city j: $u_i - u_j + nx_{ij} \leq n-1$.

Only need n^2 such constraints. Uses big-M technique to prevent sub-tours. Cases:

- If $x_{ij} = 0$ the $u_i - u_j \leq n-1$ (merely stops u_i from being much greater than u_j).
- If $x_{ij} = 1$ the $u_i - u_j \leq -1$ (forces u_i to be at least 1 less than u_j).

So the u_i's will be ordered and will tell us the order of the cities.

This model is far more compact, but harder to solve.

The specific distances between cities determine how difficult it is to find (or prove you've found) an optimal solution.

Some problems are equivalent to TSP. For example, scheduling aircraft takeoffs. Planes taking off create turbulence. The bigger the plane the more turbulence. Smaller planes are more sensitive to turbulence. This creates an ATSP (Asymmetric TSP).

Number of minutes each type of plane has to wait. Column plane takes off first; row plane takes off second:

	Jumbo	Airbus	Cessna
Jumbo	3	2	1
Airbus	4	3	1
Cessna	8	7	2

Objective function would be to minimize total waiting time.

8.1 Template Design

Of cat food labels.

Not all problems are easy to model as IP.

Template 1:

beef	beef	beef
beef	rabbit	rabbit
rabbit	tuna	tuna

Template 2:

tuna	beef	chicken
chicken	mixed	mixed
mixed	mixed	mixed

15,000 runs of template 1 and 23,000 runs of template 2; generate:

$15,000 \times 2$ tuna $+23,000 \times 1$ tuna etc. of each label.

The problem is to design a set of templates and runs satisfying demand (47,000 tuna, 80,000 beef, etc.) while minimizing the total number of runs.

We don't know the best number of templates, so fix it as a small number.

Each template has a number of slots (capacity) s ($s = 9$ in this case).

Define some integer variables: r_i is the number of runs of template i.

Call the demand for each type of variable i: D_i.

Objective function: Minimize $Z = \Sigma_i r_i$.

ST

Slots can't be empty: $\Sigma_i t_{ij} = S$.

Satisfy demand: $\Sigma_i r_i t_{ij} \geq D_j$.

This is the natural way to express the demand constraint, unfortunately $r_i t_{ij}$ is the product of two variables, so the problem is non-linear.

Some systems can handle non-linear constraints (probably not Lindo) but they can be harder to solve.

Here is a linear model of the same problem:

Define a binary variable a_{ijk} for each template i, label type j and each slot on the template k.

Also define integer variables f_{ijk} for how many labels of type j are created by slot k on template i.

Constraints:

A slot has only one type of label: $\Sigma_j x_{ijk} = 1$.

If label type j is in slot k of template i then that slot generates $r_i$of label j, otherwise zero: $f_{ijk} \leq r_i + M(1 - a_{ijk})$; $r_i \leq f_{ijk} + M(1 - a_{ijk})$; $f_{ijk} \leq M a_{ijk}$.

Now we can use a linear constraint for the demands: $\Sigma_i \Sigma_k f_{ijk} \geq D_i$.

This is not a very good model:

- It is a big-M model
- It has a lot of symmetry.

Symmetry: every solution has many different representations (can reorder slots on a template to get the same results). This makes the search space much bigger - there are more possibilities to explore.

9 Lecture 15 February 2007

IP in general is NP-hard (can take exponential time to solve) but some problems can be solved faster with Greedy Algorithms.

9.1 Greedy Algorithms

9.1.1 Minimum-Spanning Tree

Find a subset of edges such that there is a path between any two nodes, with minimum total path weight.

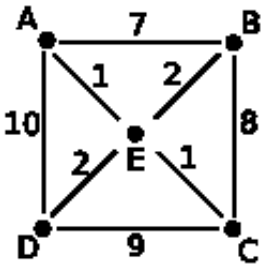

Figure 11: Graph

The solution is always a tree.

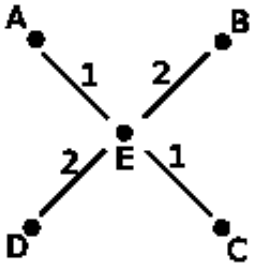

Figure 12: Minimum Spanning Tree (MST)

A Greedy Algorithm:

1. Let m be the number of nodes in the graph. Choose any node; Put it into a set S_1 and put all the other nodes into a set S_2.
2. From all the edges with ends in S_1 and S_2, choose the one with the smallest weight: (i, j) $i \in S_1$; $j \in S_2$.
3. Add edge (i, j) to the tree. Move j from S_2 to S_1.
4. If S_2 is now empty then halt, the tree is built
   Else repeat from step 2.

This algorithm always finds the MST. It take m^2 time (or better).

If we add a constraint, e.g. (A, E) and (A, B) cannot both be in the MST, the greedy algorithm may give an infeasible solution.

If the edges are directed then the greedy algorithm fails.

9.1.2 Machine Sequencing

One machine; n jobs; Job i takes P_i to complete; On completing job i there is a penalty of $C_i T_i$ (where C_i is a positive number and T_i is the time that job i completes). Find a schedule (permutation of jobs) with minimum total penalty.

Greedy algorithm: At each step choose job i with smallest $\frac{C_i}{P_i}$.

9.2 Heuristic Algorithms

Some problems have no Greedy Algorithm which can find optimum solutions, but some can be solved almost optimally in a similar way.

9.2.1 Knapsack Problem

Maximize $Z = \sum_{j=1}^{n} c_j x_j$

ST $\sum_{j=1}^{n} a_j x_j \leq b$

Where x_j are binary decision variables; c_j is the benefit of j being in the knapsack; a_j is the weight of object j; b is the capacity of the knapsack.

A greedy algorithm: add object of greatest $\frac{c_i}{a_i}$. This is not guaranteed to find the optimal solution, but is usually okay. It is called a Heuristic Algorithm.

It can sometimes be proven that a heuristic algorithm gives solutions which are close to optimal (say within a percentage).

9.3 Branch and Bound

IP solvers usually use the Branch and Bound algorithm. Lindo does.

The approach is similar in concept to that of propagation and backtracking in Constraint Programming. Branch and bound uses Relaxation and backtracking.

Say we have a MIP. Say we relax it by ignoring the integrality constraints: Replace $x \in \{0, 1\}$ with $0 \leq x \leq 1$.

Then we can apply Simplex to get a LP solution. Do this at every node in the search tree - when we set x to zero or to one. Backtrack if the Simplex solution is not better than the best solution found so far.

This approach relies on the fact that the optimal solution can never be improved by adding restrictions.

Solvers can also generate additional constraints (cutting planes) which can make a huge difference in efficiency. Compared to the situation in the 1980s, and taking hardware advances into account, branch and bound algorithms are now about two million times more efficient. So many previously intractable problems are now in reach.

Other relaxations: We may be able to ignore some constraints to get a relaxation that can be solved by a greedy algorithm.

For example in the TSP, if we ignore the sub-tour elimination constraints we get the assignment problem (assign each city to another, minimizing the cost). This can be solved in polynomial (n^3) time.

9.4 Dynamic Programming (DP)

Invented by Richard Bellman in 1950 at the Rand corporation.

Idea: In a recursive problem, we may make the same recursive calls many times. By remembering the results we may speed up finding the overall solution.

For example, finding the Fibonacci Numbers (intuitively we add the previous two numbers in the series to get the next one): $1, 1, 2, 3, 5, 8, 13, \ldots$

A recursive program:

```
int fib(int f) {
  if (f == 1 || f == 2) return 1;
  else fib(f - 1) + fib(f - 2);
}
```

This very slow (exponentially slow).

The intuitive method is much faster (linear).

9.4.1 The Change Problem

Find minimum number of coins adding up to some amount. For euro, dollar, etc. a greedy algorithm works:

- Choose biggest coin that is not greater than the amount and then subtract that from the amount; repeat until amount is zero.

For some currencies - with different coin denominations, this greedy algorithm fails.

For example, in 1875 the coins in the US currency were: 1c, 5c, 10c, 20c, 25c. How many coins needed to make up 40c?: Greedy algorithm gives an answer of (25c, 10, 5c). The best solution is (20c, 20c).

Suppose that the amount wanted is 77c and the coins available are worth 1c, 3c and 7c.

The answer must be one of these three possibilities:

1. 1c + best 76c solution.
2. 3c + best 74c solution.
3. 7c + best 70c solution.

These contain recursive calls. To calculate the number of calls:

$$number(m) = min \begin{cases} number(m-1) & +1 \\ number(m-3) & +1 \\ number(m-7) & +1 \end{cases}$$

The could be written directly as a program but it would be very slow. It will actually return the solution for $number(20)$, for example, billions of times. It is exponentially slow.

DP Approach: Solve the problem for all amounts up to 77.

First solve the problem for small amounts:

Amount	Coins	Number of Coins
1	1	1
2	1,1	2
3	3	1
4	3,1	2
5	3,1,1	3
6	3,3	2
7	7	1

Now for the amount 8 the answer is:

- either best 7c solution + 1c
- or best 5c solution + 3c
- or best 1c solution + 7c

Amount	Coins	Number of Coins
8	1,7	2

Continue calculating values up to 77c.

Complexity is linear in amount.

10 Lecture 19 February 2007

10.1 Dynamic Programming

10.1.1 Manhattan Tourist Problem

Maximize total attractions in a walk across a grid like New York City.

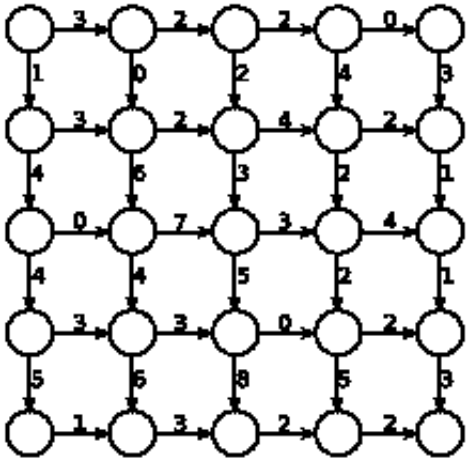

Figure 13: Manhattan Tourist Problem

Find longest path (in terms of edge weightings).

Could use IP, but it would take exponential time to run.

Could use a Greedy Algorithm (e.g. always take the biggest weighted edge from the current node - this gives a total path weight of 23, but there are longer).

DP solves it quickly and optimally: Find longest path from start to all other nodes.

Label every node with the length of the longest path to it.

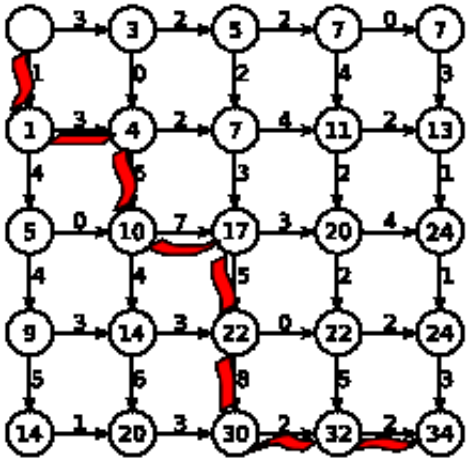

Figure 14: Manhattan Tourist Solution

At each step choose a node whose predecessors are already labelled with their longest path value.

Longest path problems on arbitrary graphs (Directed Acyclic Graphs) can always be solved in this fashion.

Calculate maximum of those labels and edge weight.

Principle of Optimality: The optimal solution at each stage does not depend on previous decisions (only on the current situation).

10.1.2 Edit Distance

We have two strings of symbols: How similar are they?

We could use the Hamming Distance: In how many places do they have a different symbol? E.g. ABCDE and AXCDY have distance 2.

But ATATATAT and TATATATA differ in every single position. So they are maximally different under Hamming Distance, but are really closely related.

How many "edits" does it take to transform one into the other? Find minimum Edit Distance between strings.

Edits:

Substitution Replace one symbol by another.

Deletion Delete a symbol.

Addition Add a symbol

E.g. TGCATAT and ATCCGAT.

5 steps			4 steps	
TGCATAT	Delete last		TGCATAT	Add front
TGCATA	Delete last		ATGCATAT	Delete T
TGCAT	Add front		ATGCAAT	Substitute
ATGCAT	Substitute		ATGCGAT	Substitute
ATCCAT	Add G		ATCCGAT	
ATCCGAT				

Algorithm:

Let [] be the empty string.

Let $[s|c]$ be String s followed by symbol c.

Base cases: $dist([][]) = 0$; $dist(s, []) = dist([], s) = |s|$.

Recursion:

$$dist([s_1|c_1], [s_2|c_2]) = min \begin{cases} dist(s_1, s_2) & +penalty(c_1, c_2) \\ dist([s_1|c_1], s_2) & +1 \\ dist(s_1, [s_2|c_2]) & +1 \end{cases}$$

$penalty(c_1, c_2) = 0$ if $c_1 = c_2$ and 1 if $c_1 \neq c_2$.

A recursive program in this form would take exponential time. DP just fills in a table from small strings to large ones and is very fast.

11 Lecture 22 February 2007

11.1 Dynamic Programming

11.1.1 Longest Common Sub-sequence

Given a sequence: ABCDE; a sub-sequence of it is: BDE. A sub-sequence is not necessarily continuous, but the symbols are in the same order.

Problem: Find longest sub-sequence in two strings.

E.g. ATCTGAT and TGCATA. A possible answer is TCTA.

Notation: Let the two strings be $V = V_1 \dots V_n$ and $W = W_1 \dots W_m$. Let S_{ij} be the length of the sub-sequence using $V_i \dots V_j$ and $W_i \dots W_j$.

Recursive method:

$$S_{ij} \quad = \quad max \begin{cases} S_{i-1,j} \\ S_{i,j-1} \\ S_{i-1,j-1} + 1 \quad if\, V_i = W_j \end{cases}$$

Base Cases: $S_{i0} = 0$; $S_{0j} = 0$.

Compute $S_{11}, S_{12}, S_{21}, S_{22}, \dots$

The diff command in Linux uses this method. It is very fast. A Text Editor which refreshes only those parts of the screen which it needs to following a small change can benefit from this calculation also.

11.1.2 Sequence Alignment

Align as much of two strings as possible. This is the same as the longest path problem. E.g.:

```
AB-CA-D...
```

```
A-DCAA-...
```

Terminology: A column with the same symbol is a MATCH; One with a different symbols is a MISMATCH; One with a space (gap) is an INDEL; One with a space n the top row is an INSERTION; One with a space on the bottom row is a DELETION.

Taking the strings

```
AT-GTTAT-
```

```
ATCGT-A-C
```

These can be represented by

```
0122345677
```

```
0123455667
```

These numbers are how many symbols from each string have been used so far in the alignment. It is merely a different way of representing the alignment. At each step take a symbol from the first string, or from the second, or both.

Think of the columns as coordinates (x, y) in a grid.

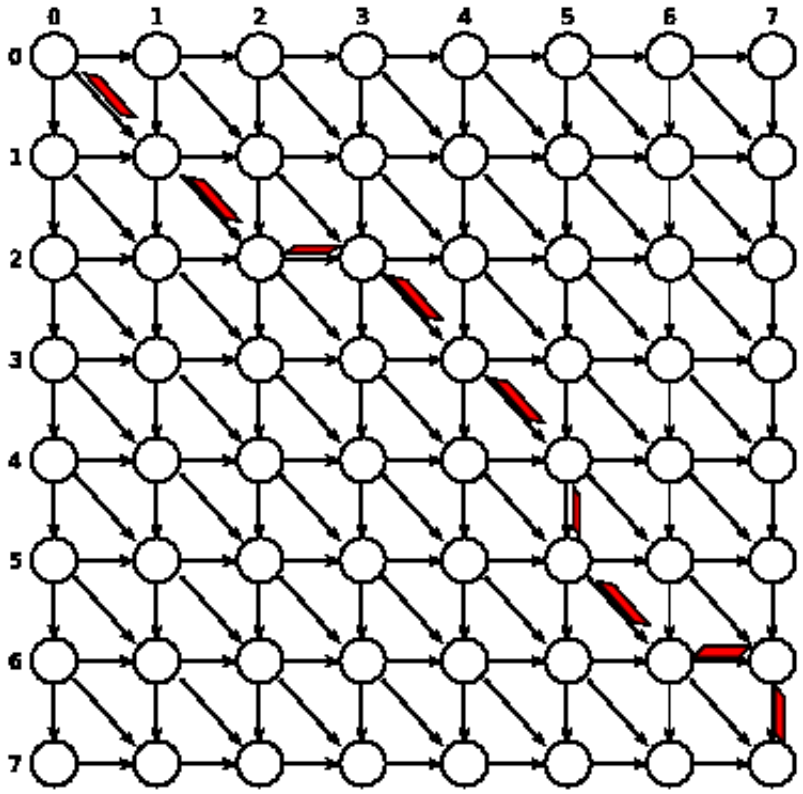

Figure 15: Edit Graph

Scoring diagonals with weight 1 and all others with weight zero. Apply a DP longest path algorithm. Could use negative weights to prevent gaps. This edit graph allows mismatches, but we could delete those edges.

Answer:

A	T	-	G	T	T	A	T	-
A	T	C	G	T	-	A	-	C
↘	↘	→	↘	↘	↓	↘	↓	→

Biologists often use a scoring matrix based on biological knowledge.

11.1.3 Local Alignment

Find the best alignment of the best substring.

A substring is part of a string taking consecutive symbols. For example in string ABCDE, a substring is CDE (BDE is a sub-sequence but not a substring).

Find two substrings that have the best possible alignment.

We could try all pairs of substrings and apply DP. This would be slow.

Better is to modify out Edit Graph: Add edges with weight zero from $(0, 0)$ to all other nodes and from all each node to (n, m).

The longest path in the new Edit Graph is the optimal local alignment.

11.1.4 Multiple Alignment

Align more than two strings.

In the Case of three strings the Edit Graph looks like a cube. The number of nodes increases exponentially and soon becomes too high to be practical. DP works up to seven or so strings. This problem is known as "The curse of dimensionality".

In practice we can use *progressive alignment*: Align the first two strings and then align the next with that result and so on. This will not give optimum results. Best results are obtained by aligning the most similar strings first and aligning remaining strings in order of similarity.

A product which does this (called CLUSTALW) was invented at UCC.

11.1.5 Longest and Shortest paths

Dijkstra algorithm finds the shortest paths.

In the following algorithm, u := Extract_Min(Q) searches for the vertex u in the vertex set Q that has the least d[u] value. That vertex is removed from the set Q and returned to the user.

```
 1 function Dijkstra(G, w, s)
 2 for each vertex v in V[G] // Initializations
 3    d[v] := infinity    // Known distance function from s to v
 4    previous[v] := undefined
 5 d[s] := 0              // Distance from s to s
 6 S := empty set         // Set of all visited vertexes
 7 Q := V[G]              // Set of all unvisited vertexes
 8 while Q is not an empty set // The algorithm itself
 9    u := Extract_Min(Q) // Remove best vertex from priority queue
10    S := S union {u}    // Mark it 'visited'
11    for each edge (u,v) outgoing from u
12       if d[u] + w(u,v) < d[v] // Relax (u,v)
13          d[v] := d[u] + w(u,v)
14          previous[v] := u
```

Is finding the shortest path no just the reverse of finding the longest path? Not quite.

Dijkstra allows cycles in the graph as well as working for undirected graphs, but cannot handle negative weights.

Dijkstra is very fast, but DP is faster if there are no cycles in the graph. DP algorithm can also be extended to other problems.

11.1.6 Knapsack problems by DP

Maximize benefit: $\Sigma_{j=1}^{n} c_j x_j$

ST $\Sigma_{j=1}^{n} a_j x_j \leq b$

To solve this by DP, solve it for items $1 \ldots j$ $(j \leq n)$ and also different Knapsack sizes $v \leq b$ (we assume b is an integer).

Find Z_{vj} for $j = 1 \ldots n$ and $v = 1 \ldots b$.

$$Z_{vj} = max \begin{cases} Z_{vj-1} & (don't\, pack\, item\, j) \\ Z_{v-a_i j} + c_j & (do\, pack\, item\, j) \end{cases}$$

Base cases: $Z_{v0} = 0$; $Z_{0j} = 0$.

Algorithm

```
For v = 0 to b
  Z_v0 = 0
For i = 0 to n
```

$Z_{0i} = 0$
For $v = 1$ to b
  For $j = 1$ to n
    $Z_{vi} = Max(Z_{vj-1}, Z_{v-a_ij} + c_i)$
Return Z_{bn}

Run-time is $O(bn)$ but Knapsack problem is NP-hard (or takes exponential time. It takes exponential time in its input b and n. To input b we need some bits. Size of b is exponential in its number of bits. This is known as pseudo-polynomial complexity: depends on a numerical value of input.

Best way of solving Knapsack problem?:

- If b is quite small (and an integer) the DP.
- If b is large (or continuous) use branch and bound (MIP).
- If time is short then a Greedy Algorithm is okay.
- A randomised algorithm can get better and better solutions as time increases (an ANYTIME algorithm).

Note In the 0/1 Knapsack problem each item is either packed or it is not (as in all examples hitherto). Using integer x_i allows several of each item. Can use DP.

11.1.7 Dynamic Programming notes

Very good, but suffers from the curse of dimensionality.

May be hard to find a recursion that obeys the principal of optimality.

Forward and backward recursion: we can start from either the source or sink in each problem.

There are approximate versions of DP, such as Progressive Alignment, Neuro-DP, etc.

Stochastic problems: Bioinfomatics, Control Theory etc.

The constraint Based approach of Memoisation is similar in principal to DP.

12 Lecture 26 February 2007

12.1 Metaheuristics

Metaheuristics includes: Randomised Algorithm; Greedy Algorithms; Evolutionary Computation; Ant Colony Optimisation; Simulated Annealing; Tabu Search; Swarm Intelligence...

Often the most practical way of solving problems.

12.1.1 Background and Nomenclature

Search space of states; Objective Function gives a real value for each state; Neighbourhood - the neighbours of a state can be reached by one local move. E.g. $(x = 1; y = 1; z = 0) \rightarrow (x = 0; y = 1; z = 0)$ is a local move.

We could let a local move be two changes or moves. In the Travelling Salesman Problem a local move could be changes routes between several cities.

Local Minimum: A state whose neighbours have higher objective function values (assuming we are minimizing).

Global Minimum: A state with the least objective function value.

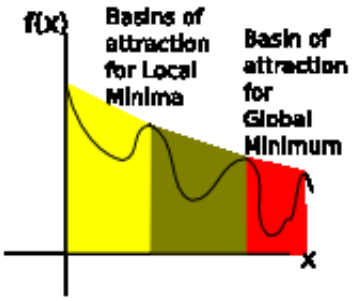

Figure 16: Example Objective Function for a single variable

Hill Climbing: Follow the local gradient. Good idea, but may become trapped in a local minimum.

Could use RANDOM RESTARTS: Start from a random state and use Hill Climbing (Gradient Descent) to a minimum; Keep a record of the best solution found so far.

Plateaus:

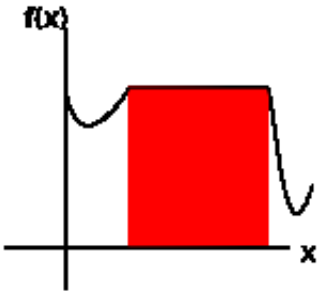

Figure 17: Plateau

Allow neutral moves (randomly pick among equally good choices).

Restarts can be inefficient.

Figure 18: Small target

There may be only a small chance of restart landing on the basin of attraction of the Global Minimum.

One answer is NOISE. For example, follow the local gradient with a probability of 0.9 otherwise make a random local move.

We need algorithms which can deal with different topologies.

12.1.2 Intensification v Diversification

Deterministic v Random

We need a compromise between these two extremes. Some algorithms:

Extreme Intensification Greedy Hill Climbing - always follow the local gradient.

Extreme Diversification Uninformed random picking - choose states at random.

Compromises:

Uninformed Random Walk (too diverse) - choose random neighbour.

Greedy Hill Climbing with random restarts or with Noise.

12.1.3 TSP Example

(Traveling Salesman Problem)

A state is a tour of the cities.

Objective function is the total distance travelled.

Local Moves: Take some subset of cities and rearrange the order of visits.

2-exchange: remove two edges and add two new ones.

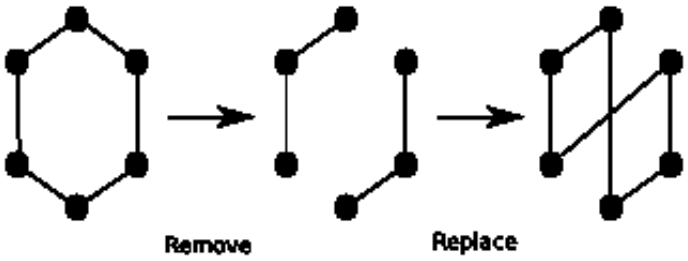

Figure 19: 2-exchange in TSP

k-exchange: remove k edges and add k new ones to form a new tour.

k-opt move: pick the best new edges.

Typically k is 2 or 3.

These methods can still reach a local minimum.

12.2 Stochastic Local Search

Stochastic usually just means random, but in Optimisation it often refers to maximising the probability of success.

Maintain a current state and make random moves to new states.

Randomized Hill Climbing:

```
S ← a random state
while terminate(s) == false
  U ← random(0,1)
  if (U ≤ P)
    S ← a random neighbour of S
  else
    S ← best neighbour of S
return S
```

P is the random walk parameter and controls Noise. E.g. $P = 0.1$

13 Lecture 1 March 2007

13.1 Probabilistic Hill Climbing

Probability of making a bad move depends on how bad the move is.

Algorithm (goal is to minimize $f(S)$):

```
S ← a random state
while terminate(S) = false
  u ← random(0,1) //random real between 0 and 1
  S' ← a random neighbour of S
  if f(S') ≤ f(S) or u ≤ e^((f(S)−f(S'))/T)  //Metropolis condition
    S ← S'
return S
```

T is called the temperature, greater temperature allows greater bad moves.

This algorithm is also called Constant Temperature Annealing.

Annealing (metallurgy), a heat treatment that alters the micro structure of a material causing changes in properties such as strength and hardness.

13.1.1 Simulated Annealing

Similar to previous method - allow temperature to vary.

Algorithm (goal is to minimize $f(S)$):

```
T = T0
S ← a random state
while terminate(S) = false
  do k times
    u ← random(0,1) //random real between 0 and 1
    S' ← a random neighbour of S
    if f(S') ≤ f(S) or u ≤ e^((f(S)−f(S'))/T)  //Metropolis condition
      S ← S'
  T = αT
return S
```

A termination condition might be based on T. E.g. $T \leq 0.00001$.

α is a number between 0 and 1 (say 0.95).

k is some integer which may increase at each iteration of T.

It has been proved that there is a cooling schedule for each problem that finds the optimum solution - but we don't know what it is. In practice we may allow reheating.

13.2 Tabu Search

Pick the best neighbour that is not forbidden.

Avoids small loops by using memory of its last few moves. But comparing entire states is expensive (Could try storing hashes of states). Instead we check *components* of states. For example if a local move changes a variable then don't reverse that change for several iterations. This is much cheaper.

However this Tabu condition prevents states which haven't been tried. Take the example of some binary variables:

Allowed	Var1	Var2	etc.
√	0	0	...
√	1	0	...
√	1	1	...
×	1	0	...

Components are stored in a Tabu List. This contains the last t moves. t is the Tabu Tenure.

Can add Tabu to other algorithms. A simple version is: make the best local move that isn't forbidden by the Tabu condition.

If the Tabu Tenure is too small then we won't escape from deep local minima. This is called Search stagnation and can be hard to spot.

If t is too large the search is too restricted and we might miss the optimal solution.

Variants:

- Robust Tabu Search - t is chosen randomly from a list of numbers; also forces moves not made for a long time.
- Reactive Tabu Search - t changed adaptively depending on recent search history.
  - If a state occurs twice then it assumes stagnation is occurring and increases t
  - else decreases t.
  - Can also revisit *elite solutions* (best ones found so far) and restart search from there but head in a different direction.

14 Lecture 5 March 2007

14.1 Dynamic Local Search

This is one of the most effective methods currently known.

Dynamically change the objective function during the search. Transform local minima to non-minima.

Attach a weight to each search component. A search component is, for example, a TSP edge, a single variable or constraint).

Modify the objective function to include all these weights.

When trapped in a local minimum, increase the weights of those components involved in the local minimum. Eventually the local minimum becomes a hill and the search moves off it.

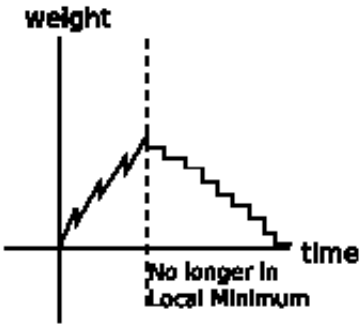

Figure 20: Weight Change

To avoid very large weights periodically decrease all weights. For example by multiplying them by 0.9. This is called *smoothing*.

An algorithm based on this approach is called Guided Local Search. It can be used as an add on to local search algorithms described previously (e.g. Simulated Annealing).

There are various ways that the weights might be adjusted: Additive (where a constant is added); Multiplicative (where the weight is multiplied by a constant, e.g. 1.1).

14.2 Hybrid approaches

No algorithm is best for all problems. "No free lunch" theorem: For every algorithm there is a problem for which it is the worst.

14.2.1 Iterated Local Search

Uses two types of local move.

1. For reaching good states quickly (high intensification - quite greedy).
2. For escaping minima (high diversification).

14.2.2 GRASP

Acronym for Greedy Randomised Adaptive Search Procedure.

Steps:

1. Design a Greedy algorithm for the problem.
2. Use it to find a good first solution.
3. Use local search to improve the solution found.
4. Restart with a slightly randomised version of the Greedy algorithm.

It is quite expensive to restart but it often pays off (gets good results).

14.2.3 Adaptive Probing and Squeaky-wheel Optimisation

Like GRASP but use weights that are adjusted between restarts to pinpoint cause of failure to find optimal solution.

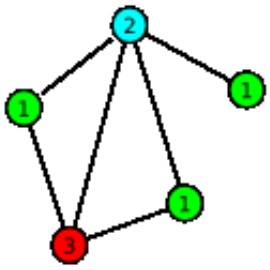

Figure 21: Graph Colouring

14.3 Graph Colouring

Assign "colours" to each vertex in a graph such that no two vertexes joined by an edge have the same colour.

This seemingly abstract problem has many real applications: Timetabling, Scheduling, Radio Frequency allocation, Pattern Matching in AI, etc.

Clique of size three needs at least three colours.

Several Local Search algorithms have been developed for the Graph Colouring problem.

14.3.1 Tabu Search

Start by colouring each vertex randomly.

Local move: change the colour of a single node.

Start with k colours, then make local moves until a k-colour solution is found. Then look for a $k-1$ solution. Repeat until no smaller solution is found.

14.3.2 Iterative Greedy

Start with no nodes coloured.

Colour nodes one by one, each time trying to reuse a colour.

If it not possible to reuse a colour then use a new one.

Restart using a node ordering that guarantees we don't need to use more colours - we may need fewer.

14.3.3 Impasse

A search state is a currently coloured subset of the nodes. The nodes that can't be correctly coloured is the impasse set.

Looks for k-colouring, then decreases k.

15 Lecture 8 March 2007

15.1 Probabilistic Approximate Completeness

PAC property: The longer a Local Search, with this property, is run the more likely it is to find the optimal solution.

The probability of finding the optimal solution tends towards one as time tends toward infinity. The search converges on the optimal solution.

Some algorithms may be trapped forever in a local minimum. These are called ESSENTIALLY INCOMPLETE.

15.1.1 Random Picking

Algorithm cannot be trapped. Has a probability $\frac{1}{s}$ (where s is the number of possible search states) of finding the optimum solution at each iteration.

$\frac{1}{s} > 0$ so random picking has the PAC property.

15.1.2 Randomised Hill Climbing

Algorithm:

```
s ← a random state
while termination(s) = false
  u ← random(0, 1)
  if u ≤ p
    s ← a random neighbour
  else
    s ← the best neighbour
return s
```

Proof of PAC:

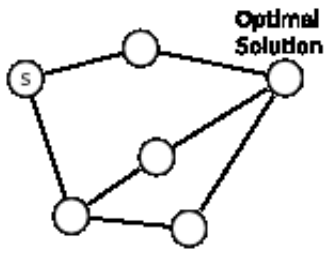

Figure 22: Distance from Optimum

Define distance between states as the number of local moves between them.

Consider any state which id some distance d from an optimum solution.

Prove that there is a non-zero chance that the next move reduces d.

At least one neighbour of s will be closer to the optimum.

With probability p the algorithm chooses a random neighbour, which could be one closer to the optimum.

15.1.3 An Essentially Incomplete algorithm

```
s ← a random state
while terminate(s) = false
  s ← a random choice of its two best neighbours
return s
```

This is not PAC.

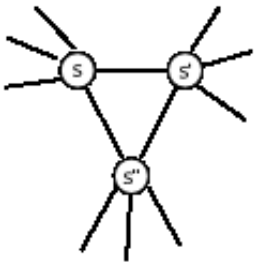

Figure 23: Essentially Incomplete

Suppose that the search space has three states s, s', s'' that are all neighbours of each other, but also have other neighbours. They are better than all their neighbours.

The algorithm will *always* choose one of s, s', s'' to move to from one of these.

15.2 Genetic Algorithms

Classic GA: Genes are bits.

More recent GAs: Use integers, reals, symbols or programs.

Mutation and Recombination: are the genetic operators.

Fitness function: operates on strings/genotypes/organisms.

16 Lecture 12 March 2007

16.1 Genetic Algorithms

If there are only a few genes then it is possible to try all combinations. For it to be worthwhile to use a GA there needs to be at least 30 genes involved in the problem.

GAs can solve huge problems (e.g. the human genome solving the problems of survival and reproduction).

Start by creating a population of chromosomes (strings of genes) or genotypes. Each chromosome has a fitness (defined by some function). If the evaluation function is f then the fitness is often defined as $\frac{f_i}{f_{mean}}$. Alternatively the algorithm could use fitness ranking (after sorting chromosomes into fitness order).

16.1.1 Recombination

1-point Crossover is the simplest (and seems to be what occurs in nature).

Parents		Children
00000000		11100000
11111111		00011111

2-point Crossover:

Parents		Children
00000000		00110000
11111111		11001111

Uniform Crossover: Randomly choose a parent for each gene.

Parents		Children
00000000		01101100
11111111		10010011

Good chromosome design puts closely related genes close together, this makes recombination less likely to break up good combinations of genes. But Uniform Crossover makes random decisions at each gene, so all designs are equally good. Uniform Crossover is more disruptive that the other methods.

An advantage of Uniform Crossover. Suppose parents are very similar, e.g.:

Parents
10011101
00010001

Consider gene differences only:

Parents
1***11**
0***00**

These are known as reduced surrogates.

If 1-point Crossover is used than there are four ways of generating the same offspring:

Parents
1***00**
0***11**

But only one way of generating:

Parents
1***10**
0***01**

So 1-point Crossover is biased. Uniform Crossover is not biased (it is more fair). However, we can use 1-point Crossover on surrogates to make it fair.

16.1.2 Mutation

Replace a single gene's value.

Premature Convergence

If all chromosomes have 0 in gene 15 (for example) the crossover can never produce a 1 in gene 15.

Mutation is necessary to avoid this.

Mutation is also a hill-climber. If we use only mutation we're doing local search on a population of states. This strategy may beat crossover for some problems, especially on a small population.

16.1.3 Non-binary alphabets

Early GAs used {0,1}. This is easier to analyse.

Bigger alphabets may be more natural. E.g. floating-point numbers can be represented by bits, but it may be easier to use floating-point genes.

16.2 Evolutionary Computation

Some Common methods adopted, under different names, in different fields of Science have been brought together as Evolutionary Computation.

1. Evolutionary Strategies (ES)
2. Evolutionary Programming (EP)

Common system:

$P \leftarrow$ `some initial population`
$Evaluate(P)$
`while termination == false`
  $P' \leftarrow Recombination(P)$
  $P'' \leftarrow Mutation(P')$
  $Evaluate(P'')$
  $P \leftarrow Select(P'')$

Different versions of this approach have been adopted.

- $(\mu + \lambda)$-ES: μ parents produce λ offspring, who are added to the population. Select the best μ to be the next generation.
- (μ, λ)-ES: Offspring replace their parents.
- The Genitor Algorithm: Selects 2 parents, generates 1 offspring which replaces the least fit member of the entire population. This is a steady-state GA. Keeps population ranked by fitness.
- The CHC Algorithm: Perform Recombination; Take n best unique chromosomes (e.g. $n = 100$) from the parents and offspring. Random parent selection. Uses Uniform Crossover. If copies (clones?) occur then apply cataclysmic mutation (a lot of mutation).

17 Lecture 15 March 2007

17.1 Parent Selection

Choose fittest and choose randomly are two strategies.

17.1.1 Tournament Selection

Choose a random subset of the population (of size k). k is called the tournament size.

1. Choose the best chromosome from this subset with probability p.
2. Choose the second-best with probability $p(1-p)$.
3. The third with probability $p(1-p)^2$.
4. and so on

This chooses parents quite randomly but with a bias toward the fittest.

Can choose k and p to alter the bias towards the fittest individuals.

This approach is useful for problems where fitness cannot be computed, but chromosomes can be compared (and ranked).

Tournament selection can be parallelised.

We may allow the fittest chromosome to survive into the next generation. This is called Elitism Selection.

17.2 Memetic Algorithms

The term Memetic is from The Selfish Gene by Richard Dawkins.

Apply local search or hill-climbing to a chromosome before placing it in the population. GAs are often quite slow and this can speed things up.

This approach is also called Hybrid GAs or Genetic Local Search.

Memetic algorithms can also use backtrack search.

17.2.1 Lamarkian and Baldwinian Learning

Lamark: Acquired characteristics can be inherited (this is false in nature). MAs use Lamarkian Learning.

Baldwin: Suggested that this happens indirectly. That there is evolutionary pressure for offspring to acquire characteristics. i.e. to learn more quickly.

In GAs - don't alter the chromosome, but reduce the fitness function more if greater effort is needed to improve it (Baldwin Effect).

17.3 Indirect GAs

A chromosome can represent instructions on how to find a solution. To find the fitness, we execute these instructions using a decoder.

17.4 Case Study: TSP

Design good operators: Offspring should usually have similar fitness to parents. Bad operators produce LETHALS - much less fit offspring.

TSP is a permutation problem $\Rightarrow$ find a good permutation.

17.4.1 Representations of Tours

Path Representation Each city is a number e.g. (1..100) and a tour is a list of these numbers $(6, 8, 91, 3...)$. This is the most obvious method.

Binary Representation Replace integers by bits. e.g. $(4, 1, 3, 2, 5)$ becomes $(100, 001, 011, 010, 101)$. One problem with this approach is that invalid tours can emerge which include non-existent cities.

Ordinal Representation Again a list of integers $(1..n)$, where n is the number of cities. The i^{th} number on the list is in the range $1 \ldots n - (i + 1)$. Start with a list to use as a reference $L = (1, 2, 3, 4, 5)$.

To represent $(4, 1, 3, 2, 5)$ use the position in L to represent the city. The first city is 4 which is in position four in L so represent it by 4 and remove it from L.

The tour $(4, 1, 3, 2, 5)$ is represented by $(4, 1, 2, 1, 1)$.

18 Lecture 22 March 2007

18.1 Case Study: TSP, continued

Classical operators work badly on the path representation. E.g. $(4, 1, 3, 2, 5)$ may mutate to $(4, 2, 3, 2, 5)$, which is not a permutation.

The Ordinal representation works with the classical operators: Never gives an illegal chromosome, but it gives poor results.

The best results have been obtained by using new operators on the straightforward path representation.

18.1.1 New Crossover Operators

PMX (Partially Mapped Crossover)

Choose two cutpoints randomly:

$$\frac{1,2,3,|4,5,6,|7,8}{3,7,5,|1,6,8,|2,4}$$

Swap contents between those points:

$$\frac{1,2,3,|1,6,8,|7,8}{3,7,5,|4,5,6,|2,4}$$

These are the mapping section. Define the mappings $1 \leftrightarrow 4$, $6 \leftrightarrow 5$, $8 \leftrightarrow 6$.

Note invalid values outside of mapping section:

$$\frac{x,2,3,|1,6,8,|7,x}{3,7,x,|4,5,6,|2,x}$$

Replace values at x positions according to the mappings. May need to iterate this process several times for each value:

$$\frac{4,2,3,|1,6,8,|7,5}{3,7,8,|4,5,6,|2,1}$$

The positions of many cities are unchanged: Offspring may have similar fitness to parents.

CX (Cycle Crossover)

Create offspring in which about half the cities take their position from one of the parents.

Basic idea:

Each gene comes from one parent together with its position.

Informal procedure:

1. Make a cycle of genes from P1 in the following way.
   (a) Start with the first gene of P1.
   (b) Look at the gene at the same position in P2.
   (c) Go to the position with the *same* gene in P1.
   (d) Add this gene to the cycle.
   (e) Repeat step b through d until you arrive at the first gene of P1.
2. Put the genes of the cycle in the first child on the positions they have in the first parent.

Worked Example:

$$\frac{1,2,3,4,|5|,6,7,8}{2,4,6,8,7,5,3,1}$$

Choose random starting point (position five in the first parent in this case). Add it to the offspring $(.,.,.,.,5,.,.,.)$.

The city at position five in the second parent is 7, take value from that position in the first parent and add it to the offspring $(.,.,.,.,5,.,7,.)$.

The city at position seven in the second parent is 3, take value from that position in the first parent and add it to the offspring $(.,.,3,.,5,.,7,.)$.

The city at position three in the second parent is 6, take value from that position in the first parent and add it to the offspring $(.,.,3,.,5,6,7,.)$.

The city at position six in the second parent is 5, which has already been included, this completes the cycle. The rest of the cities can be taken from the second parent $(2, 4, 3, 8, 5, 6, 7, 1)$.

Form other offspring from complement of each gene, subtract each gene value from $n + 1$ giving $(7, 5, 6, 1, 4, 3, 2, 8)$.

OX1 (Order Crossover)

There is also an OX2 variant.

Try to preserve the relative ordering of cities in offspring.

Choose a subtour of one of the parents and preserve the order of that subtour in the other parent.

Choose two cutpoints randomly:

$$\frac{1, 2, |3, 4, 5, |6, 7, 8}{2, 4, |6, 8, 7, |5, 3, 1}$$

Copy subtours into offspring:

$$\frac{., ., |3, 4, 5, |., ., .}{., ., |6, 8, 7, |., ., .}$$

Starting from the second cutpoint, copy the cities from the other parent (in order), leave out any cities already in that offspring:

$$\frac{8, 7, |3, 4, 5, |1, 2, 6}{4, 5, |6, 8, 7, |1, 2, 3}$$

18.1.2 New Mutation Operators

DM (Displacement Mutation)

Move a subtour to a new position:

$$\frac{1, 2, |3, 4, 5, |6, 7, 8}{1, 2, 6, 7, |3, 4, 5, |8}$$

EM (Exchange Mutation)

Pick two cities and exchange them.

SIM (Simple Inversion Mutation)

Pick two cutpoints randomly and reverse the order between those points.

18.2 Case Study: Knapsacks

Maximize $Z = \Sigma_{j=1}^{n} c_j x_j$ (c_j = benefit)

ST $\Sigma_{j=1}^{n} a_j x_j \leq b$ (a_j = weight; b = capacity).

Method 1 (2002)

Solutions are n bits. e.g. 1001 means pack items 1 & 4.

Z is not enough to measure fitness as some chromosomes are illegal (e.g. 1111).

Penalty functions: modify the fitness function to penalise illegal chromosomes for their infeasibility.

Fitness: $\Sigma_{j=1}^{n} c_j x_j - penalty(x_1, \ldots, x_n)$

The penalty is

$$\begin{cases} 0 & if\ \Sigma_{j=1}^{n} a_j x_j \leq b \\ \sqrt{(\Sigma_{j=1}^{n} a_j x_j) - b} & if\ \Sigma_{j=1}^{n} a_j x_j > b \end{cases}$$

The Genetic Algorithm uses: 1-point crossover, elitism, point mutation, population size 30, mutation rate 0.01 and crossover rate 0.8.

19 Lecture 26 March 2007

19.1 Case Study: Knapsacks (continued)

Method 2 (1994)

Multiple Knapsack problem. Same chromosome representation as method 1. Different penalty function.

GA: Population 50; Mutation Rate $\frac{1}{n}$; crossover rate 0.6; proportional selection; 1-point crossover.

If too many chromosomes are illegal, replace some by legal ones found by a greedy heuristic.

Method 3 (2003)

Integer values, many copies of an item may be packed. Reduce to binary genes, bit representation of integers. Instead of a penalty function they *repair* illegal chromosomes by removing items until capacity is not exceeded. Remove items based on their profitability (benefit divided by weight).

GA: Stead state; Uniform crossover; Tournament selection; Population 100; Mutate 2-bits per chromosome; Remove duplicate chromosomes.

Method 4 (2001)

Transposition, move a substring from one parent and copy it to a position in the other. Can be done with only one parent - asexual reproduction.

Method 5 (1998)

Problem-space exploration. GA generates a starting point for a greedy algorithm - choose items with greatest profitability first. To get different solutions add small numbers to the profitabilities.

Use GA to generate these numbers. Numbers are in a range $\pm e$, where e is the difference between the smallest and largest profitabilities.

GA: Elitist; Generational; Population 100; Crossover rate 0.9; Mutation rate 0.01; Proportional Selection.

Method 6 (Steve's own)

Evolve a permutation as for TSP (order of adding items). Fitness computed by adding items in that order, skipping items that would exceed capacity.

19.2 Other topics

Not covered in this course:

- Newton's Method.
- Sensitivity Analysis - Tight and Slack constraints.
- Neural Networks.
- Massive Parallelism.
- DNA Computing.
- Quantum Computing

20 Lecture 29 March 2007

20.1 Exam

Five questions, answer them all.

There were six main topic areas in the course. Here they are broken down into what needs to be known for the exam and what does not.

20.1.1 Linear Programming

Know Terminology; How to model problems; Graphs (of solutions - how to draw); When is LP an approximation to the real world; Approximation (variables should be integer, coefficients may be estimates, non-Linear).

Don't need Unimodularity; Simplex; Specific models.

20.1.2 Integer Programming

Know Terminology; Modelling (logical constraints, extra variables, big-M).

Don't need Specific Models; Relaxation; Branch and Bound.

20.1.3 Dynamic Programming

Know Fibonacci; Change problem; Longest Path; 2-way alignment; Longest Common Subsequence; Knapsack problems; DP and Dijkstra.

Don't need Edit distances; Local alignment; Multiple alignment, Stochastic problems.

20.1.4 Local Search

Know Terminology; Escaping Local Minima (restart, noise, Tabu, Dynamic Local Search - changes objective function); Algorithms (Greedy Hill Climbing etc.); Probabilistic Approximation Completeness.

Don't need Robust Tabu; Reactive Tabu; Guided Local Search; GRASP; Graph Colouring Algorithm.

20.1.5 Genetic Algorithms

Know Terminology; Crossover (1-point, 2-point, uniform, PMX, CX, OX); Mutation (Classical, displacement, exchange, simple inversion); Representations of problems (e.g. TSP - binary, path, ordinal).

Don't need Genitors; CHC; Evolutionary Strategies; Evolutionary Programming; Mimetic Algorithms; Tournament Selection, Indirect GA (where decoder is used to transform chromosome into solution); Knapsack Case Study.

20.1.6 Greedy Algorithm

Know How do they work; Why are they useful (fast and give reasonable solutions); Can they solve all optimisation problems (no).

Don't need Proofs; Particular algorithms.

[Skip OR]

20.1.7 Example Questions

LP Model a problem. Show graphically how it works and the solution found. Explain why LP may be unrealistic.

IP Model a problem. Discuss big M constraints. Define logical constraints.

DP Model a problem as a longest path. Use dynamic programming to solve by hand (change problem?). Remember that graphs may not be regular grids.

LS Describe several algorithms (Probabilistic Hill Climbing, Simulated annealing etc.). Prove if PAC or not.

GA Model problem is different ways. Compare different operators (e.g. crossover, mutation); Alphabet (values genes can take); Illustrate different operators with examples.

Greedy Invent a Greedy algorithm for a problem. It doesn't have to be very good, just greedy.