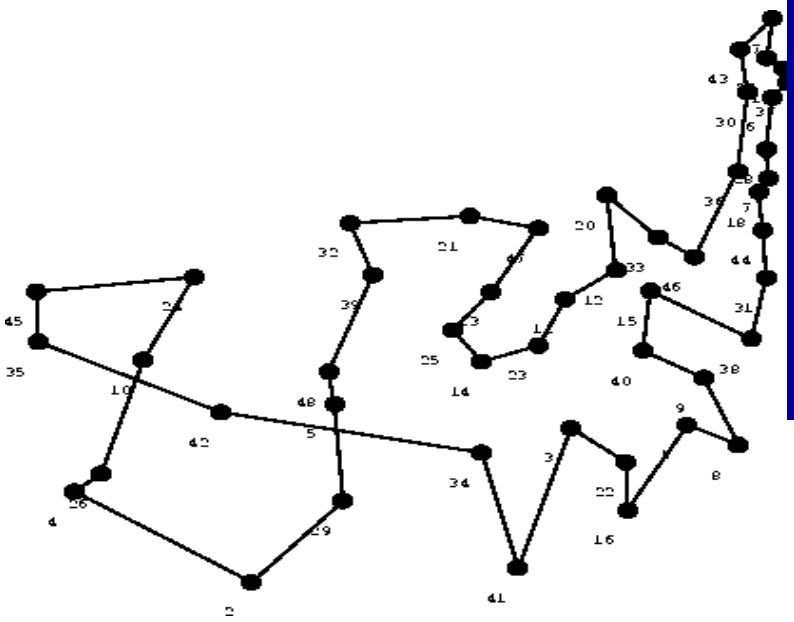
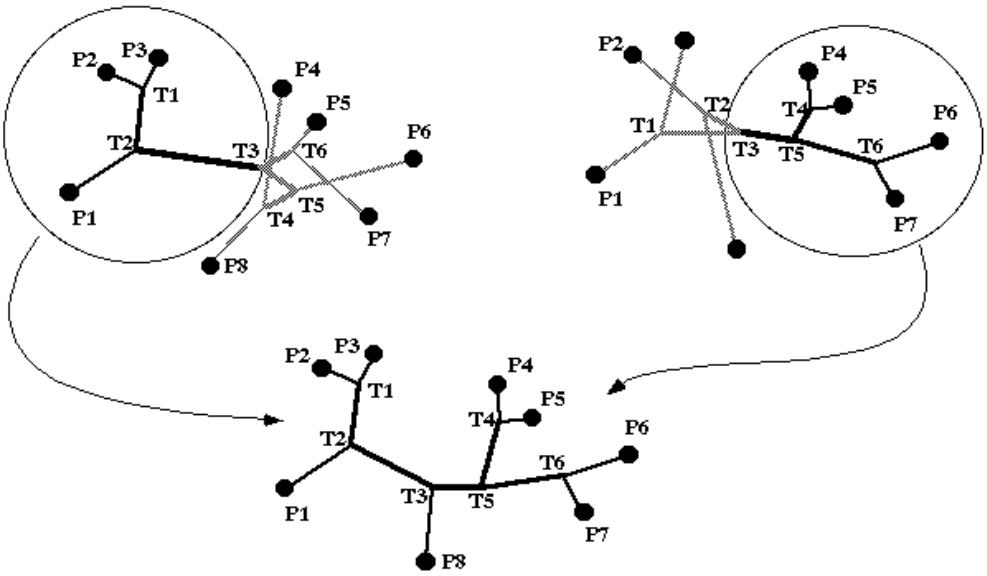
Heuristic Solution Methods



The University of Auckland



Used with permission; https://www.flickr.com/photos/30686429@N07/3953914015/

Heuristic Solution Methods

- used for “difficult” problems, typically NP-hard ones
- optimal solutions are very computationally expensive, or impossible to find
- use approximation algorithms — “heuristics” — to find “good” solutions
- many such problems are combinatorial in nature – the solution is a sequence or grouping of discrete objects

NP-Hard:

Introductory Example

The Problem:

Five not-so-good friends (named A=Andrew, B=Big Bird, C=Cathy, D=Daniel and E=Ernie) enter Kitty O’Briens to celebrate having just finished their Operations Research exam. After placing their orders for food, they seat themselves along the bar to enjoy the best of Irish brews (Figure 1). Unfortunately, they have strong preferences as to whom they sit beside. The ‘dislike factors’ are shown in Table 1. The friends want to organise themselves to minimise the total dislike.

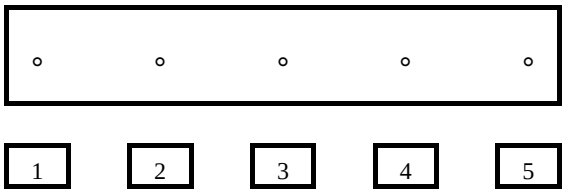


Figure 1: Bar Layout at Kitty O’Briens. Seat numbers are shown.

d(i,j)	A	B	C	D	E
A	-	13	4	13	6
B	13	-	5	7	3
C	4	5	-	5	2
D	13	7	5	-	6
E	6	3	2	6	-

Table 1: Friend dislike factors

A Possible Solution:

Consider the solution x=(A,B,C,D,E), meaning person A is in seat 1, person B is in seat 2, etc.

	Solution x									
	Seat 1 x_1	Seat 2 x_2	Seat 3 x_3	Seat 4 x_4	Seat 5 x_5	$d(x_1,x_2)$	$d(x_2,x_3)$	$d(x_3,x_4)$	$d(x_4,x_5)$	Objective f(x)
Starting Soln	A	B	C	D	E					

Mathematically:

- Let S be the set of all possible set of configurations.

In our case, S is the set of all possible orderings of the students;
S={(A,B,C,D,E), (A,B,C,E,D), (A, B, D, C, E), ..., (E, D,C, B,A) }.

How many different such solutions are there:

|S| =

But, if we consider solutions such as (A,B,C,D,E) and (E,D,C,B,A), we see that ...,

and so the total number of unique solutions is...

- Let $f(x) : x \in S \rightarrow \Re$ be a cost function that assigns a real number to each configuration $x \in S$.

In our case, let a solution be denoted by $x=(x_1, x_2, x_3, x_4, x_5)$, where x_q gives the person sitting in seat q. If d(i,j) is the dislike between person i and j, then the cost of solution x is

$$f(x) = d(x_1, x_2) + d(x_2, x_3) + d(x_3, x_4) + d(x_4, x_5)$$

Eg, for x=(A,B,C,D,E), f(x) = d(A,B)+ d(B,C)+ d(C,D)+ d(D,E) =13+5+5+6 = 29.

- Our goal is to find a configuration x for which f(x) achieves a minimum (or maximum) value, ie find some optimal configuration x^* satisfying:

$$f(x^*) = \min_{x \in S} f(x)$$

Solution Method 1:

Advantages:

Disadvantages:

Solution Method 2:

Advantages:

Disadvantages:

For a summary of integer programming, see Appendix 1.

Solution Method 3:

A greedy sequential algorithm constructs a solution by a series of steps that are '*greedy*'... they make a decision that looks like a good thing to do *now*, and '*sequential*'... we make a sequence of decisions, never reversing an earlier decision. In general, it does not guarantee optimality. This is an example of a ***construction heuristic.*** Construction heuristics are only useful if objective function values can be calculated for partial (as well as full) solutions.

Example: Develop a greedy sequential solution algorithm for our Kitty O'Briens problem.

d(i,j)	A	B	C	D	E
A	-	13	4	13	6
B	13	-	5	7	3
C	4	5	-	5	2
D	13	7	5	-	6
E	6	3	2	6	-

Table 1: Friend dislike factors

A possible Greedy Sequential Algorithm:

1. Let k=1
2. Form the first partial solution $\mathbf{x^1=(x^1_{left},x^1_{right})}$ by picking the best pair of people and sit them together (putting the person with the first name alphabetically on the left, ie as $\mathbf{x^1_{left}}$)
3. Repeat

3.1 Given the current partial solution $\mathbf{x^k = (x^k_{left},...,x^k_{right})}$, find the person y from the unused people who best fits (best d value) next to person x_{right}, forming a new solution $\mathbf{x^{k+1}=(x^k_{left},...,x^k_{right}, y)}$

3.2 Increment k

4. Until all people have been seated.

Solution $x^1=($

$x^2=($

$x^3=($

$x^4=($

Possible Improvements:

Note: A greedy sequential heuristic is only useful if some (at least approximate) objective function value can be calculated for a partial solution.

Using "Regret" Measures with Greedy Sequential Algorithms

It is often useful to calculate a 'regret' value which estimates how much worse the solution might get if we delayed some action (eg seating person A) until some later step. We then choose, as our next step, the maximum regret action. The particularly calculation of regret we use depends on the problem.

Example

Assume we are building a seating solution from left to right and have, at some current step:

Solution x^1=(C E)

We still have to seat 3 people A, B, D

d(i,j)	A	B	C	D	E
A	-	13	4	13	6
B	13	-	5	7	3
C	4	5	-	5	2
D	13	7	5	-	6
E	6	3	2	6	-

Friend dislike factors

Person *i*	**Add person *i* now, d_i^{now}:** Dislike for best option if we add person *i* (next to E)	**Add person *i* later, d_i^{later}:** Estimate of dislike incurred if person *i* is not seated now (i.e. not seated next to E). For example, we might choose the best dislike if seated next to one of the other unseated people)	**Regret** $r_i = d_i^{later} - d_i^{now}$ (being some estimate of extra dislike we incur if we don't do person *i* in this step)
i=A	d_A^{now}=	d_A^{later}=	r_A=
i=B	d_B^{now}=	d_B^{later}=	r_B=
i=D	d_D^{now}=	d_D^{later}=	r_D=

We choose the maximum regret action, i.e. we next add person ……. (next to E), giving x^2=(C E ….)

Note: In this example, the regrets are all positive; they could be negative if doing a person at a later step is better.

We can extend this to calculate a "regret-n" value that takes into account the difference between doing the best thing now for some person i and the second best, and the third best, and the 4th best, …, and the n'th best. However, regret values are just some estimate of the cost impact of not doing something now.

Solution Method 4:

Advantages:

Disadvantages:

d(i,j)	A	B	C	D	E
A	-	13	4	13	6
B	13	-	5	7	3
C	4	5	-	5	2
D	13	7	5	-	6
E	6	3	2	6	-

Table 1: Friend dislike factors

Example:

	Solution x									
	Seat 1 x_1	Seat 2 x_2	Seat 3 x_3	Seat 4 x_4	Seat 5 x_5	$d(x_1,x_2)$	$d(x_2,x_3)$	$d(x_3,x_4)$	$d(x_4,x_5)$	Objective f(x)
Solution 1	A	B	C	D	E	13	5	5	6	29
Solution 2										

etc.

Solution Method 5:

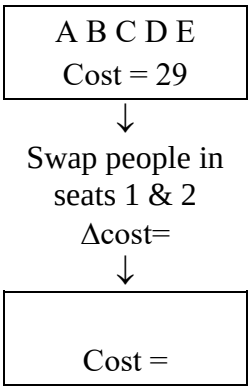
Advantages:

Disadvantages:

This is an example of an ***improvement heuristic***

Example: (Next Descent Iterations)

Our approach will be to try swapping adjacent people, accepting swaps if they make an improvement. (Other types of swaps are possible.) So, we will try swapping the people in positions (seats) 1 & 2 (ie swapping the values of x_1 and x_2), and if this doesn't improve things, then try the people in positions 2 & 3 (ie swapping x_2 and x_3), then 3 & 4 (swapping x_3 and x_4), and back to (x_1 and x_2) so on.



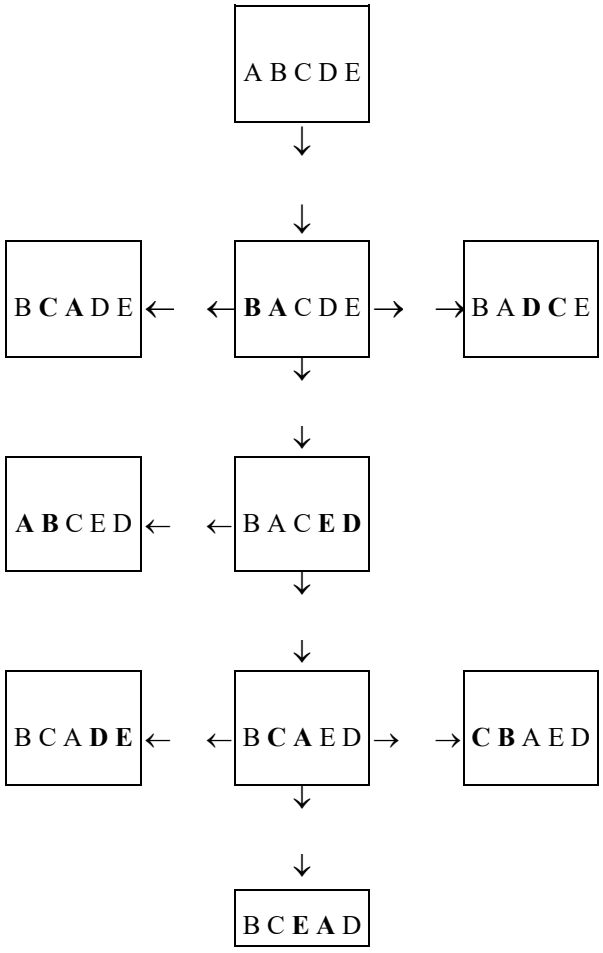
Current Solution x							Trial Solution y											
Seat 1 x_1	Seat 2 x_2	Seat 3 x_3	Seat 4 x_4	Seat 5 x_5	Objective f(x)		Seat 1 y_1	Seat 2 y_2	Seat 3 y_3	Seat 4 y_4	Seat 5 y_5	$d(y_1,y_2)$	$d(y_2,y_3)$	$d(y_3,y_4)$	$d(y_4,y_5)$	Objective f(y)	Change in Objective	Improves?
A	B	C	D	E	29	Swap x_1 & x_2												
						Swap x_2 & x_3												
						Swap x_3 & x_4												
						Swap x_4 & x_5												
						Swap x_1 & x_2												
						Swap x_2 & x_3												
						Swap x_3 & x_4												
						Swap x_4 & x_5												
						Swap x_1 & x_2												
						Swap x_2 & x_3												
						Swap x_3 & x_4												
						Swap x_4 & x_5												

Our final solution is: ……..

Is this the best possible solution (i.e. is this a global minimum)?

d(i,j)	A	B	C	D	E
A	-	13	4	13	6
B	13	-	5	7	3
C	4	5	-	5	2
D	13	7	5	-	6
E	6	3	2	6	-

Table 1: Friend dislike factors



Notes:
We want to stop when there are no more of our swaps that can give us any more improvements. How do we test this in practice?

Neighbourhoods

- This is an example of a local search/neighbourhood search algorithm.
- Each new solution generated above is found by examining *neighbours* of the current solution. We let y(**x**,i) denote a particular neighbour of x given some i
- Given a *neighbourhood rule*, the *neighbourhood* N(x) of some solution x is the set of all solutions that can be generated from x using the neighbourhood rule.
- We need to be able to quickly calculate the change in cost in moving to a neighbour. (If this isn't possible, then neighbourhood search may perform poorly.)

Given some solution $x=(x_1, x_2, x_3, x_4, x_5)$, our neighbourhood rule above is to:

Given some solution

$$x = (x_1, x_2, x_3, x_4, x_5),$$

the neighbourhood of x is the set of 4 neighbouring solutions,

$$N(x) =$$

where neighbouring solution y(x,i), $i \in \{1,2,3,4\}$ denotes the particular neighbour from N(x) formed by swapping the positions of x_i and x_{i+1} in x, i.e.

$$y(\mathbf{x},i)=(x_1, x_2,..., x_{i-1}, \mathbf{x_{i+1}, x_i}, x_{i+2}, x_{i+3}, ..., x_5),$$

or equivalently,

$$y(\mathbf{x}, i) = (y_1, y_2, y_3, y_4, y_5), \quad y_j = \begin{cases} \end{cases}$$

We need to quickly calculate the cost change moving to a neighbour. In our case, if

$$f(x) = d(x_1, x_2) + d(x_2, x_3) + d(x_3, x_4) + d(x_4, x_5)$$

then remembering that our 'dislike' matrix is symmetric (ie d(a,b)=d(b,a)), we can compute the changes in costs (new cost – old cost) associated with each neighbour using

$$f(y(x,1)) - f(x) = d(x_2, x_1) + d(x_1, x_3) + d(x_3, x_4) + d(x_4, x_5) - (d(x_1, x_2) + d(x_2, x_3) + d(x_3, x_4) + d(x_4, x_5))$$

$$=$$

$$f(y(x,2)) - f(x) =$$

$$f(y(x,3)) - f(x) =$$

$$f(y(x,4)) - f(x) =$$

In a general case (say with n people), and ignoring the special end cases (i=1, and i=n-1), we have

Solution x (before swapping x_i and x_{i+1}):

Soln	... x_{i-1}	x_i	x_{i+1}	x_{i+2} ...	
Costs	$d(x_{i-1}, x_i)$	$d(x_i, x_{i+1})$	$d(x_{i+1}, x_{i+2})$		

Solution y(x,i) after swapping x_i and x_{i+1}:

Soln	... x_{i-1}	x_{i+1}	x_i	x_{i+2} ...	
Costs	$d(x_{i-1}, x_{i+1})$	$d(x_{i+1}, x_i)$	$d(x_i, x_{i+2})$		

$f(x) = d(x_1,x_2)+...+d(x_{i-2},x_{i-1})$
$+d(x_{i-1},x_i)+d(x_i,x_{i+1})+d(x_{i+1},x_{i+2})$
$+d(x_{i+2},x_{i+3})+...+d(x_{n-1},x_n)$

$f(y(x,i)) = d(x_1,x_2)+...+d(x_{i-2},x_{i-1})$
$\mathbf{+d(x_{i-1},x_{i+1})+d(x_{i+1},x_i)+d(x_i,x_{i+2})}$
$+d(x_{i+2},x_{i+3})+...+d(x_{n-1},x_n)$

Change in cost

$\Delta f(x,i) = f(y(x,i)) - f(x)$
$=$

Swaps at the ends (i=1, and i=n-1) are special cases with easier formulae.

Typical Neighbourhood Search Algorithm

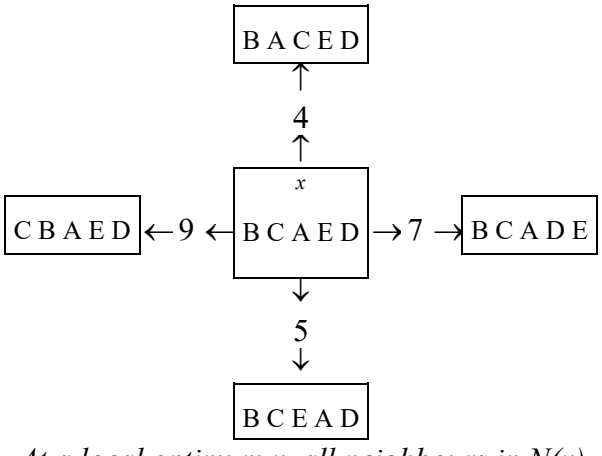
(We assume we want to find $x^* \in S$ that minimises f(x))

1) Let x, $x \in S$, be some initial configuration (for example, x can simply be chosen randomly, or maybe using a greedy sequential algorithm)
2) While not stopped do
3) Compare each neighbour $y \in N(x)$ with x, until we find one that is better, ie find $y \in N(x)$ with $f(y)<f(x)$, or determine that no neighbour y is better than x, i.e. $f(y) \geq f(x) \ \forall \ y \in N(x)$
4) If $f(y)<f(x)$ for some $y \in N(x)$ then
   Set x := y
   else $f(y) \geq f(x) \ \forall \ y \in N(x)$, so
   Stop

In step 3, we usually carry on searching the neighbourhood from where we got up to last. (This is what we did in the example above.)

Our final solution x will certainly be a …… ; none of x's neighbours will be better than x.

Hopefully x will also be a globally good solution, but we do not know if it is a global minimum.



At a local optimum x, all neighbours in N(x) are worse

Some Examples of Neighbourhood Rules

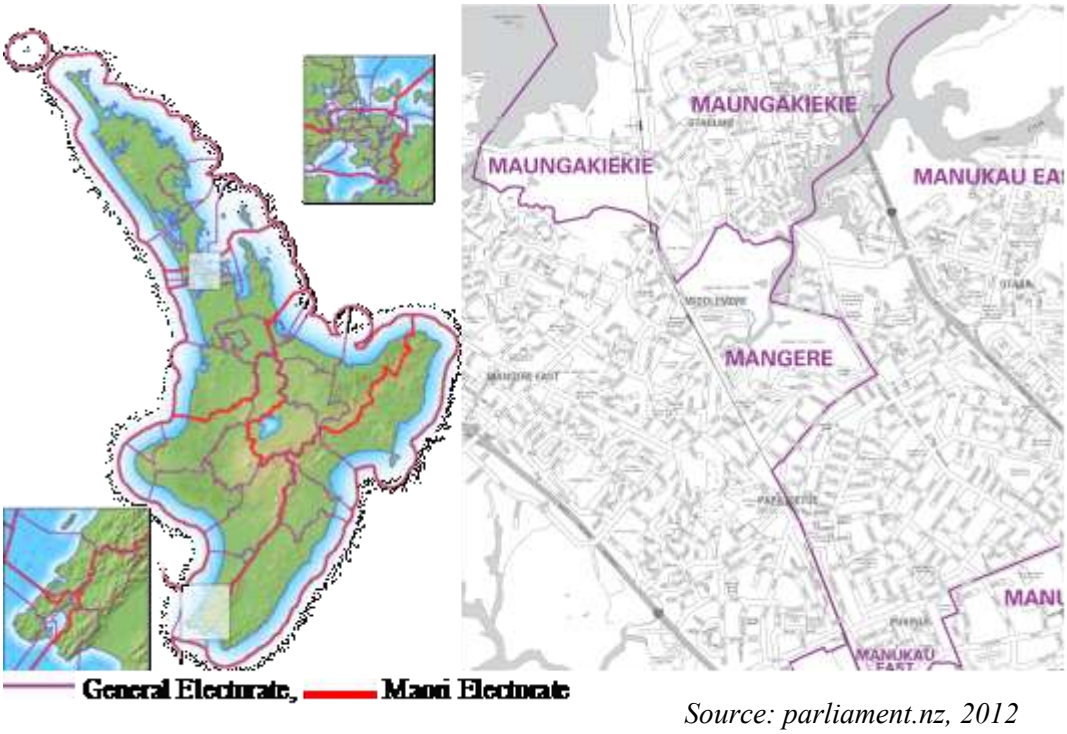
Problem: Given a fixed timetable, we wish to allocating lectures to rooms to minimise walking distance for students going from one lecture to the next.

Solution: Each lecture has an associated room.

Some Possible Neighbourhood Rules:

Problem: Divide NZ into voting electorates, so that each electorate contains roughly the same number of people, electorates are reasonably compact, and groups of adjacent houses tend to be in the same electorate

Solution: A map similar to that shown



Source: parliament.nz, 2012

Some Possible Neighbourhood Rules:

Multiple Neighbourhoods: Variable Neighbourhood Search

Variable neighbourhood search is simply neighbourhood search where multiple neighbourhood rules are used; see the turbine balancing example later.

Solution Method 6:

d(i,j)	A	B	C	D	E
A	-	13	4	13	6
B	13	-	5	7	3
C	4	5	-	5	2
D	13	7	5	-	6
E	6	3	2	6	-

Friend dislike factors

Advantages:

Disadvantages:

Example: Next Descent (starting from random solution **CEADB**)

From before, we saw that the change in objective function (new – old) can be calculated using:

$f(y(x,1)) - f(x) = d(x_1, x_3) - d(x_2, x_3)$

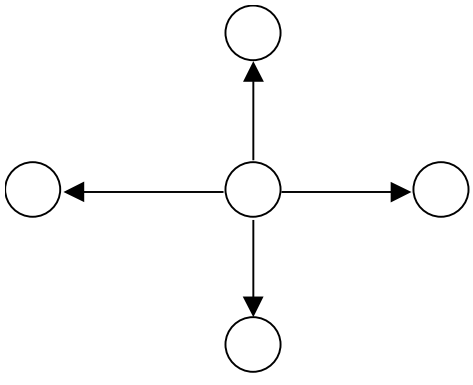
$f(y(x,2)) - f(x) = d(x_1, x_3) - d(x_1, x_2) + d(x_2, x_4) - d(x_3, x_4)$

$f(y(x,3)) - f(x) = d(x_2, x_4) - d(x_2, x_3) + d(x_3, x_5) - d(x_4, x_5)$

$f(y(x,4)) - f(x) = d(x_3, x_5) - d(x_3, x_4)$

Current Solution x					Objective f(x)		Change in Objective Function	Improves?
Seat 1 x_1	Seat 2 x_2	Seat 3 x_3	Seat 4 x_4	Seat 5 x_5				
C	E	A	D	B	28	Swap x_1 & x_2	f(y(x,1))-f(x))=	
						Swap x_2 & x_3	f(y(x,2))-f(x))=	
						Swap x_3 & x_4	f(y(x,3))-f(x))=	
						Swap x_4 & x_5	f(y(x,4))-f(x))=	
						Swap x_1 & x_2	f(y(x,1))-f(x))=	
						Swap x_2 & x_3	f(y(x,2))-f(x))=	
						Swap x_3 & x_4	f(y(x,3))-f(x))=	
						Swap x_4 & x_5	f(y(x,4))-f(x))=	
						Swap x_1 & x_2	f(y(x,1))-f(x))=	
						Swap x_2 & x_3	f(y(x,2))-f(x))=	
						Swap x_3 & x_4	f(y(x,3))-f(x))=	
						Swap x_4 & x_5	f(y(x,4))-f(x))=	

Notes: We have found another local minimum (with a much better objective function)



Solution Method 7:

Advantages: :

Disadvantages:

Notes:

- Can be used in single or repeated runs of the neighbourhood search
- Starting solution used below (ABCDE) has been picked randomly
- Changes in objective function are given by:

$f(y(x,1)) - f(x) = d(x_1, x_3) - d(x_2, x_3)$

$f(y(x,2)) - f(x) = d(x_1, x_3) - d(x_1, x_2) + d(x_2, x_4) - d(x_3, x_4)$

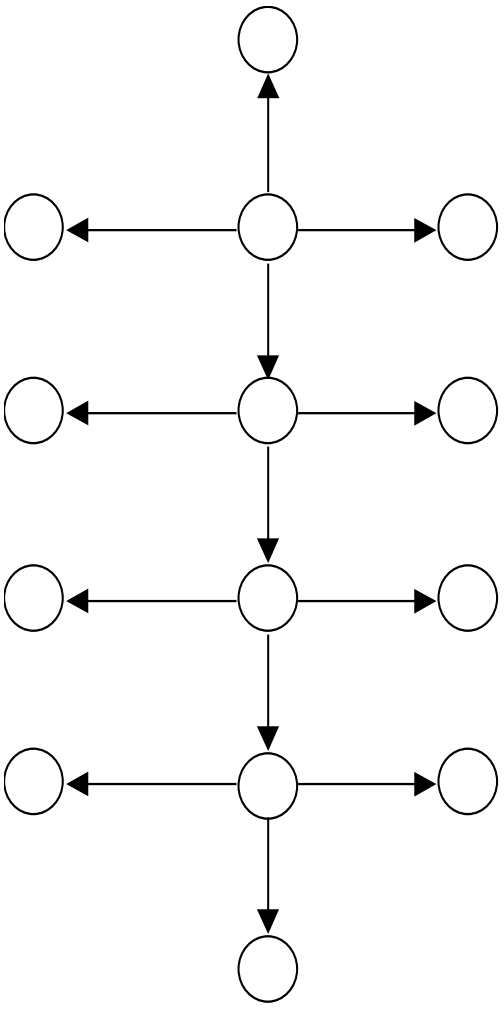
$f(y(x,3)) - f(x) = d(x_2, x_4) - d(x_2, x_3) + d(x_3, x_5) - d(x_4, x_5)$

$f(y(x,4)) - f(x) = d(x_3, x_5) - d(x_3, x_4)$

d(i,j)	A	B	C	D	E
A	-	13	4	13	6
B	13	-	5	7	3
C	4	5	-	5	2
D	13	7	5	-	6
E	6	3	2	6	-

Friend dislike factors

	Change in Objective	Best Change?	Current Solution x: Seat 1 x_1	Seat 2 x_2	Seat 3 x_3	Seat 4 x_4	Seat 5 x_5	Objective f(x)
Starting Soln			A	B	C	D	E	29
Swap x_1 & x_2	-1							
Swap x_2 & x_3	-7							
Swap x_3 & x_4	-2							
Swap x_4 & x_5	-3							
New solution								
Swap x_1 & x_2	8							
Swap x_2 & x_3	7							
Swap x_3 & x_4	-3							
Swap x_4 & x_5	-4							
New solution								
Swap x_1 & x_2	8							
Swap x_2 & x_3	8							
Swap x_3 & x_4	-2							
Swap x_4 & x_5	4							
New solution								
Swap x_1 & x_2	4							
Swap x_2 & x_3	4							
Swap x_3 & x_4	2							
Swap x_4 & x_5	3							
New solution								



Note: Sometimes we can find the best neighbour by solving an optimisation problem, allowing us to efficiently explore even very large neighbourhoods. We discuss this *optimised steepest descent* later; see 'Very Large Scale Neighbourhood Search'.

Different Neighbourhood Rules:

Some other possible neighbourhood rules for our bar seating problem:

Neighbourhood Rule Considerations

- A larger neighbourhood normally gives a better solution, but takes longer to find a local optimum
- With repeated local search runs, must trade off:

small neighbourhood		large neighbourhood
perhaps poor local optima	vs	better local optima
fast local search runs		slow local search runs
many such runs per minute		few such runs per minute

- A large neighbourhood will slow down each iteration of Steepest Descent & Tabu Search more than it will slow down Next Descent & Simulated Annealing
- We can use a mixture of neighbourhoods (*variable neighbourhood search*), e.g. start with a small fast neighbourhood, and then switch to a larger neighbourhood at end to improve solution

Constraints & Infeasible Solutions

In our examples, so far, have required all our solutions to be "feasible" in that they satisfy the constraint that:

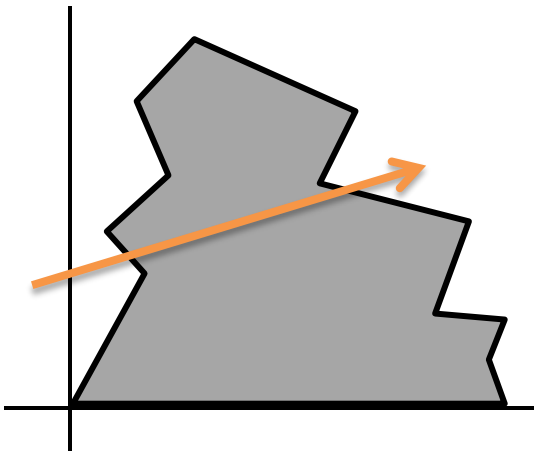
Sometimes we may have a list of constraints that are more difficult to satisfy, such as (for a larger version of our problem) "none of persons 1, 2, 3, or 4 are seated within 3 seats of each other".

If satisfying the constraints is difficult, then even finding a feasible solution may be hard. Thus, we may need to start with an infeasible solution, and perhaps also allow infeasible solutions to arise during the search. To encourage our final solution to be a feasible one, we need to penalise infeasible solutions by making their objective function value worse.

Quiz: In her local search for our bar seating problem, Susannah penalises any infeasible solutions (where persons 1, 2, 3 or 4 are seated too close to each other) by adding a fixed penalty of 1000 units whenever the solution is infeasible. Is this a good strategy?

Allowing Infeasible Solutions

We know from linear programming theory that good solutions often only "just feasible", i.e. they only just satisfy some constraints. This suggests that an effective local search will need to explore solutions that are on the boundary between feasible and infeasible. This search process may be more efficient if it is allowed to travel across the boundary into the infeasible space, i.e. we permit infeasible solutions to be generated during the search; this is illustrated in the figure below. There are some examples in the literature where doing this can improve the search process.



A representation of the feasible region (i.e. the set of all solutions that satisfy all the constraints) for a difficult local search problem. The arrow shows the direction of better solutions. Allowing the search to move through infeasible solutions may give better quality final solutions.

Different Ascent/Descent Strategies

We have examined *next descent* and *steepest descent*. Other possibilities include

- *partial steepest descent* - choose steepest of next 5 (say) descents encountered
- *random descent* - randomly accept or reject each descent as it is encountered
- *randomised hill climbing* – try random neighbours, keeping improvements
- any mixture of the above…
- *optimised steepest descent* – sometimes we can solve an optimisation problem to find the best neighbour even if the neighbourhood is huge; we discuss this later.

Termination Rules

When a single run of the local search is performed, it stops when no neighbour has a better objective function. If repeated runs are made, possible termination criteria are:

-
-

Did you know…

The word "heuristic" is derived from the Greek verb *heuriskein*, meaning “to find” or “to discover”. Archimedes is said to have run down the street shouting “*Heureka*” (I have found it) after discovering the principle of flotation in his bath. Later generations converted this to *Eureka*.

In 1957, George Polya wrote an influential book called *How to solve it* that used “Heuristic” to refer to the study of methods for discovering and inventing problem-solving techniques, particularly for the problem of coming up with mathematical proofs. Newell, Shaw and Simon stated in 1963, “A process that may solve a given problem, but offers no guarantees of doing so, is called a heuristic for that problem.” In applications of artificial intelligence, heuristics were viewed as “rules of thumb” that experts could use to generate good solutions without exhaustive search. These 2 ideas capture the modern meaning of heuristic.

Adapted from: Artificial Intelligence: A modern approach, Stuart Russel & Peter Norvig, Prentice Hall, 1995, p94

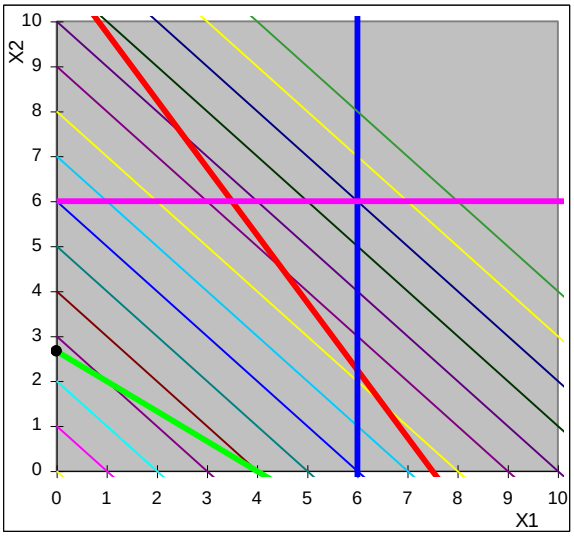
Linear Programming as Local Search

Linear programming is simply a local search algorithm with a bit of embedded intelligence in interpreting our solutions and choosing the neighbours.

Total Cost						-2.67
C	-1	-1	0	0	0	0

A Matrix							b	Pi
	1.2	0.8	1				9	0
	1	1.5		-1			4	-0.67
	1				1		6	0
		1				1	6	0
Bas	☐	☑	☑	☐	☑	☑		
Xn	0			0				
X	0	2.67	6.87	0	6	3.33		
rc	-0.3	0	0	-0.7	0	0		
	1	2	3	4	5	6		

Basis				Inverse Basis				Xb	
0.8	1	0	0	0	0.67	0	0	2.7	2
1.5	0	0	0	1	-0.53	0	0	6.9	3
0	0	1	0	0	0	1	0	6.0	5
1	0	0	1	0	-0.67	0	1	3.3	6
2	3	5	6						



Source: A J Mason EngSci 391 demonstration, 2017

$x_B=(x_2, x_3, x_5, x_6)$, $x_N=(x_1, x_4)$

A solution is:

The rule defining a neighbouring solution is:

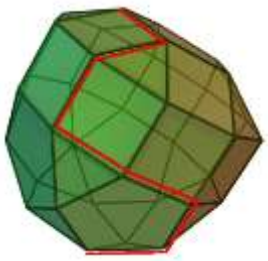
But...

-
-

Why do we find the optimal solution?

Figure (right): Visualisation of the simplex algorithm iterations in 3D (i.e. for a problem with 3 decision variables)

Source: https://creativecommons.org/licenses/by-sa/3.0/



Escaping Local Optima: Random Restarts vs Kicks

If we are to avoid the curse of local optimality, we need some way of escaping out of local optima when we find them. One approach is to add another layer – a *meta heuristic* – above the local search to actively control the direction of the search, allowing the local search to escape local optima.

Possible approaches include:

- escape from a local optimum by 'transporting' up to some new random solution
    - the 'beam me up, Scotty' option
    - We have already seen this as…
    - Advantage
        - Explores wide range of solutions
    - Disadvantage
        - Discards potentially useful information contained in good solutions
- 'kick' the solution to some nearby random solution

A Kick Move

Assume we have found some local optimum for our bar seating problem. We want to 'kick' our solution to some nearby random solution. How could we do this?

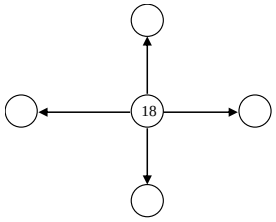
Example (assuming our neighbourhood rule is to swap adjacent people)

Current solution (a local optimum): ACDEB (objective function value = 18)

Our 'Kick' is 3 moves to a randomly chosen neighbour:

Starting solution: ACDEB: 18

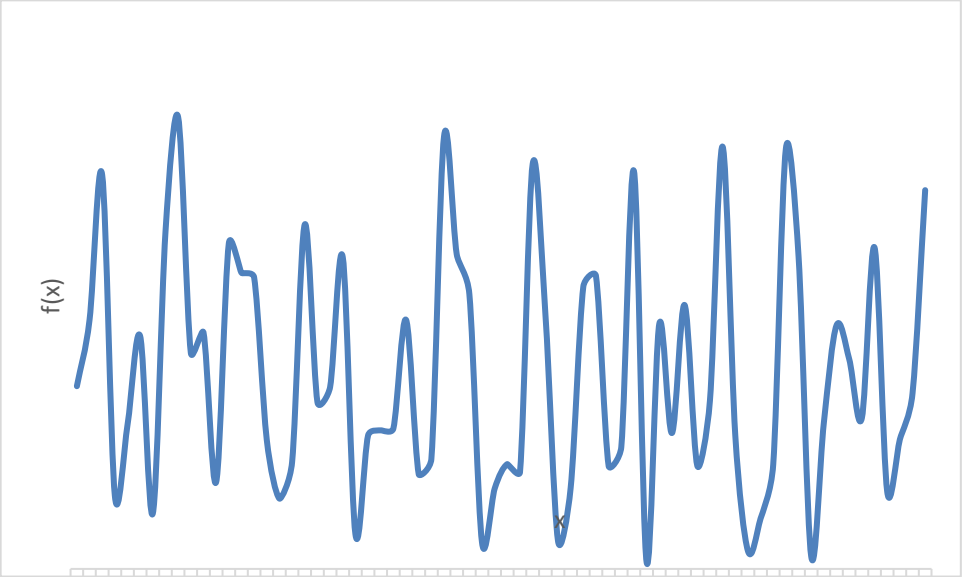
Solution after kick:



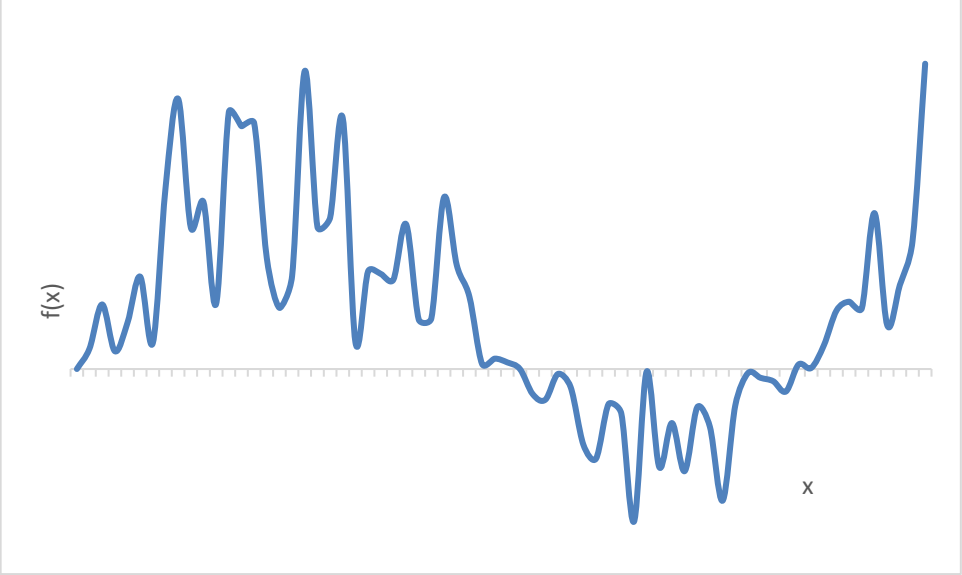
Comparison of Random Restarts vs Kicks

Consider the following two (simplified) problems. We assume we are using a neighbourhood rule under which solutions that are neighbours have a similar *x* value.

Plot of Objective Function for Problem 1 showing a highly random objective function where two solutions that are neighbours (have similar *x* values) can have very different objective function values.



Plot of Objective Function for Problem 2 where we see a more correlated function in which two solutions that are neighbours (have similar *x* values) tend to have similar objective function values.



Comment:

Meta Heuristics

There are a wide number of *meta heuristics* that allow the local search to escape local optima. Which is best for a particular problem typically needs to be determined using experimentation.

Tabu Search

- use memory to direct search out of local optima (see later)

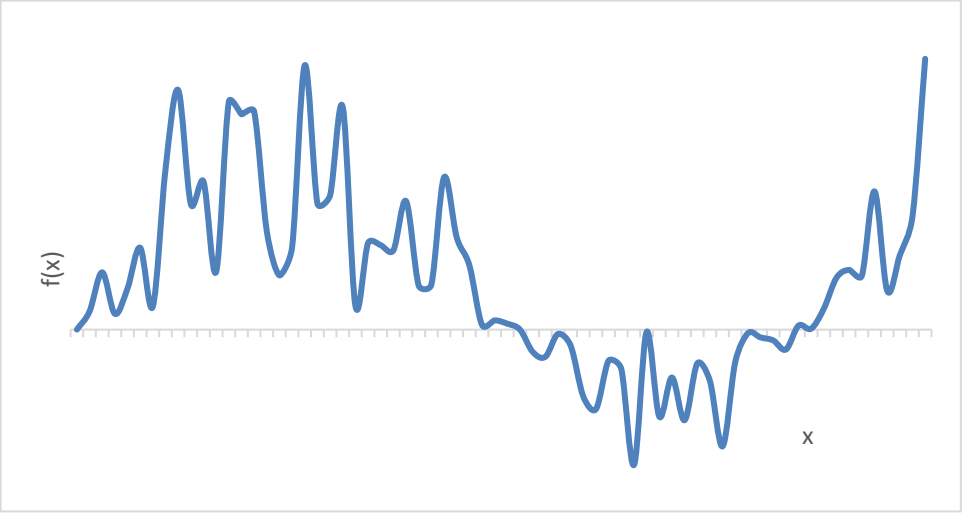
Simulated Annealing

- use randomness to accept changes that worsen the solution, thus escaping local optima (see later)

Threshold Accepting

- accept worse solutions if they are
    - 'not much worse' than the current solution , or
    - within some current global limit
- Example: Great Deluge algorithm
    - consider *maximisation* problem
    - assume rising global threshold ('water level')
    - accept any solutions better than threshold ('above water')
- References:
    - Gunter Dueck and Tobias Scheuer. Threshold Accepting: A General Purpose Optimization Algorithm Superior to Simulated Annealing. *Journal of Computational Physics*, 90:161--175, 1990.
    - Grant Telfar's (grant@isor.vuw.ac.nz) WWW pages (used in preparing this material)

Example:



Ant Colony Optimisation, Bee Optimization, …

- The latest in trendy optimization… Be sceptical!

Neural Nets

- A trendy way of fitting complex functions to complex data (called "learning"); typically poor at optimization

Tabu Search

Tabu search monitors the progress of the search, and uses this search *history* to actively modify the permitted neighbourhood solutions and/or their costs to direct the search into more promising areas of interest, and in particular, out of local optima.

In Tabu terminology, we say that a *move* is performed when we move from some solution x to a neighbour $y \in N(x)$, e.g. in going from x=(A,B,C,D,E) to y=(B,A,C,D,E), the move is a swap of 1 and 2 (written 1-2).

The simplest history we can remember is a list of the recently performed moves. In Tabu search, we direct the search by banning – making *tabu* – moves that have been performed recently (are in our list). Let H denote the current history, that is the current list of tabu moves. Now, when we generate the neighbouring solutions for x, some of these neighbours will be tabu because they will use a move in our list of tabu moves; these will not be candidate neighbours. Thus, our history gives a new smaller neighbourhood NC(H,x).

Under Tabu search, we typically choose the best neighbour from our reduced neighbourhood, and move to that neighbour (i.e. we perform a steepest descent). However, we make the move even it worsens the solution. This allows us to escape local optima.

The use of Tabu search is best shown using the following example. (In this example, we won't consider using history to change the costs of neighbouring solutions. However, the principle of directing the search through modifying costs is similar to modifying the neighbourhood, but also allows certain moves to be encouraged, perhaps because they haven't often been used but should be explored. Also note that the above description is a simplified introduction to Tabu search. Additional complexity can be introduced, such as rules to allow Tabu moves to be accepted if they give a significant improvement. See the references below for more details.)

References

Tabu search was developed by Fred Glover. References include

1. Fred Glover and Manuel Laguna, *Tabu Search*, in [4] pages 70-150, John Wiley and Sons, Inc.
2. Fred Glover, Future Paths for Integer Programming and Links to Artificial Intelligence, *Computers and Operations Research*, 5:533-549, 1986
3. Fred Glover. Tabu Search: A Tutorial. *Interfaces*, 20(4):74--94, 1990.
4. Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, Inc, 1993.

TabuRoute

TabuRoute is a popular Tabu search implementation for vehicle routing problems. See *A Tabu Search Heuristic for the Vehicle Routing Problem*, Michel Gendreau, Alain Hertz, Gilbert Laporte, Management Science, Vol. 40, No. 10 (Oct., 1994), pp. 1276-1290. Sample code is available online.

Tabu Search Example

- We always move to the best neighbour (i.e. use *steepest descent*, or *least ascent* if we cannot find an improving neighbour).
- Our *moves* are swaps of a pair of adjacent people
- We remember a *history* of the last three 'people pair' moves (eg A-B means we swapped A & B or, equivalently, B & A); these moves are considered *tabu*.
- We start from the solution B,C,A,D,E, and do steepest descent.

Starting Soln x=	**B**	**C**	**A**	**D**	**E**	f(x)= 28
History:Tabu moves=			|		|	
Swap x_1 & x_2 : y(x,1)=	**C**	**B**	A	D	E	Δf=
Swap x_2 & x_3 : y(x,2)=	B	**A**	**C**	D	E	Δf=
Swap x_3 & x_4 : y(x,3)=	B	C	**D**	**A**	E	Δf=
Swap x_4 & x_5 : y(x,4)=	B	C	A	**E**	**D**	Δf=

Action: Swap x & x (&); Ban

New solution x=						f(x)=
History:Tabu moves=			|		|	
Swap x_1 & x_2 : y(x,1)=	**C**	**B**	A	E	D	Δf=
Swap x_2 & x_3 : y(x,2)=	B	**A**	**C**	E	D	Δf=
Swap x_3 & x_4 : y(x,3)=	B	C	**E**	**A**	D	Δf=
Swap x_4 & x_5 : y(x,4)=	B	C	A	**D**	**E**	Δf=

Action: Swap x & x (&); Ban

New solution x=						f(x)=
History:Tabu moves=			|		|	
Swap x_1 & x_2 : y(x,1)=	**A**	**B**	C	E	D	Δf=
Swap x_2 & x_3 : y(x,2)=	B	**C**	**A**	E	D	Δf=
Swap x_3 & x_4 : y(x,3)=	B	A	**E**	**C**	D	Δf=
Swap x_4 & x_5 : y(x,4)=	B	A	C	**D**	**E**	Δf=

Action: Swap x & x (&); Ban

New solution x=						f(x)=
History:Tabu moves=			|		|	
Swap x_1 & x_2 : y(x,1)=	**B**	**A**	C	E	D	Δf=
Swap x_2 & x_3 : y(x,2)=	A	**C**	**B**	E	D	Δf=
Swap x_3 & x_4 : y(x,3)=	A	B	**E**	**C**	D	Δf=
Swap x_4 & x_5 : y(x,4)=	A	B	C	**D**	**E**	Δf=

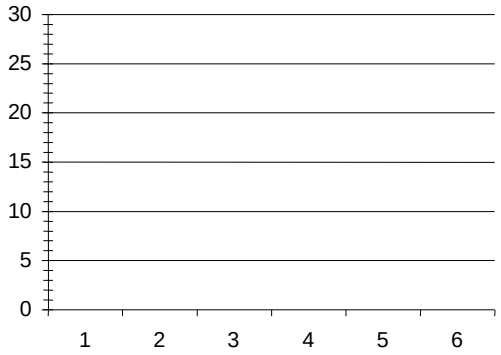
Action: Swap x & x (&); Ban ; Allow

New solution x=						f(x)=
History:Tabu moves=			|		|	
Swap x_1 & x_2 : y(x,1)=	**C**	**A**	B	E	D	Δf=
Swap x_2 & x_3 : y(x,2)=	A	**B**	**C**	E	D	Δf=
Swap x_3 & x_4 : y(x,3)=	A	C	**E**	**B**	D	Δf=
Swap x_4 & x_5 : y(x,4)=	A	C	B	**D**	**E**	Δf=

Action: Swap x & x (&); Ban ; Allow

New solution x=						f(x)=
History:Tabu moves=			|		|	

etc…



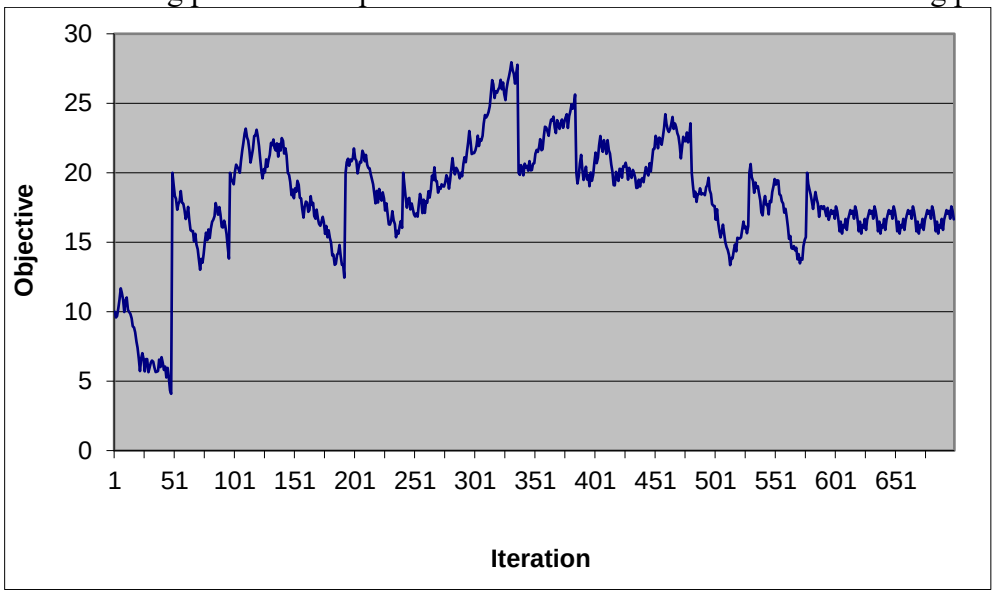
Note: We have found a new solution that is better than the last.

Typical Tabu Search Algorithm

- We assume we want to find $x^* \in S$ that minimises f(x).
- Let NC(H,x) denote the modified neighbourhood of x under history H.
- Let c(H,y) denote the modified objective function of solution y under the history H. For example, to encourage exploration of new solutions, c(H,y) may make y look less attractive if y has features that have occurred frequently in recent solutions.
- Let x denote the current solution, and x^* denote the best solution found so far.

1) Let x, $x \in S$, be some initial configuration (for example, x can simply be chosen randomly)
2) Let $x^*=x$
3) Let $H=\phi$
4) repeat
5) Calculate NC(H,x)
6) Choose x': $f(x')= \min_{y \in NC(H,x)} c(H,y)$, or a random neighbour if NC() is empty
7) x := x'
8) if $f(x') < f(x^*)$ then $x^* := x'$
9) update H
10) until termination rule is satisfied

A possible Tabu Search run:…
The following plot shows a possible Tabu search run for a vehicle routing problem.



What's going on?

How do we fix it?

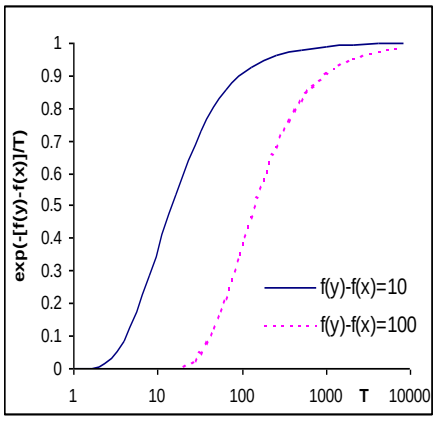
Simulated Annealing

- Developed by Kirkpatrick, Gelatt and Vecchi, in "Optimization by Simulated Annealing", IBM Research Report RC 9355 (1992)
- Explores neighbours in a random order
- Allow solutions to both worsen and improve
- Statistically controls acceptance of worse solutions
- Displays asymptotic convergence to the optimum (ie finds optimum with probability 1 if run forever).
- Based on annealing processes used, eg, in treating steel, where steel is heated and then cooled in a controlled fashion
- As temperature falls, particles arrange themselves in lower & lower energy states
- If temperature is dropped too quickly, particles get frozen into a high energy state
- Need to specify how the temperature T drops, ie specify a cooling schedule
- If y is worse than x, accept y with probability $e^{-[f(y)-f(x)]/T}$ – the *metropolis condition*.

Simulated Annealing - Inhomogeneous Algorithm

We assume we want to find $x^* \in S$ that minimises f(x).
Let $c_1, c_2, \ldots$ denote a series of decreasing temperatures

- Let x be some initial random configuration
- Let k = 1
- repeat
- let $y \in N(x)$ be a randomly chosen neighbour of x
- if $f(y)<f(x)$ then
  x := y
  else if random[0,1) < $\exp(-[f(y)-f(x)]/c_k)$ then
  x := y
- k := k +1
- until stopping criterion satisfied



Note: Convergence results require $c_k \to 0$ asymptotically as $k \to \infty$, and the convergence of the sequence $\{c_k\}$ is no faster than $O([\log k]^{-1})$. Eg, $c_{k+1}=\alpha c_k$, $\alpha<1$.

Simulated Annealing - Homogeneous Algorithm

Allows stabilising to a new equilibrium at each temperature.

- Let x be some initial random configuration
- Let k = 1
- repeat
- repeat
- let $y \in N(x)$ be a randomly chosen neighbour of x
- if $f(y)<f(x)$ then
  x := y
  else if random[0,1) < $\exp(-[f(y)-f(x)]/c_k)$ then
  x := y
- until equilibrium is approached sufficiently closely
- k := k +1
- until stopping criterion satisfied

Note: Convergence results require $c_k \to 0$ asymptotically as $k \to \infty$, and an infinite number of steps in approaching equilibrium.

Simulated Annealing Example

Notes:

- We are using an inhomogeneous temperature schedule $c_{k+1} = 0.8\ c_k$
- We picked a starting temperature of $c_1 = 12$.

Starting Soln	**x=**	**A**	**B**	**E**	**C**	**D**	f(x)= 23

Temperature c_1= 12

Pick a random neighbour y=y(x,i), i=1,2,3,4 from N(x):

Choose i=

Swap x & x	y= y(x,2)=						f(y)=

Is f(y) < f(x)?

Action:

New solution	**x=**						f(x)=

Temperature c_2=

Pick a random neighbour y=y(x,i), i=1,2,3,4 from N(x):

Choose i=

Swap x & x	y= y(x,3)=						f(y)=

Is f(y) < f(x)?

Metropolis Test: f(x) = f(y) =

$k = \exp(-[f(y)-f(x)]/c_2)$ =

r = random no. [0,1) =

Is r <= k?

Action:

New solution	**x=**						f(x)=

Temperature c_3=

Pick a random neighbour y=y(x,i), i=1,2,3,4 from N(x):

Choose i=

Swap x & x	y= y(x,1)=						f(y)=

Is f(y) < f(x)?

Metropolis Test: f(x) = f(y) =

$k = \exp(-[f(y)-f(x)]/c_3)$ =

r = random no. [0,1) =

Is r <= k?

Action:

Current soln	**x=**						f(x)=

Temperature c_4=

Pick a random neighbour y=y(x,i), i=1,2,3,4 from N(x):

Choose i=

Swap x & x	y= y(x,2)=						f(y)=

Is f(y) < f(x)?

Action:

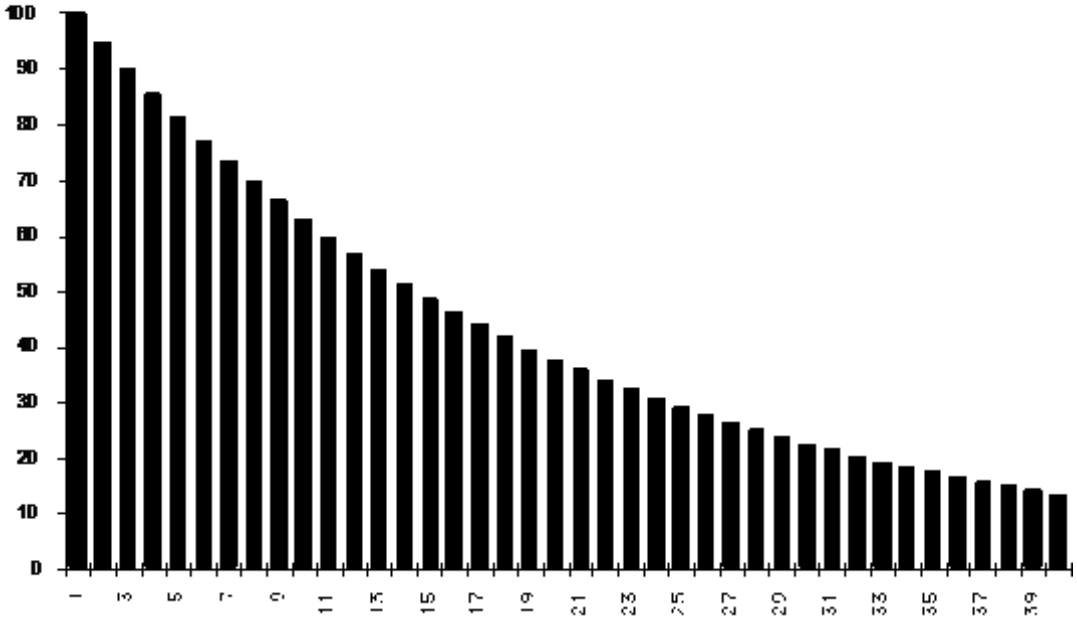
New solution	**x=**						f(x)=

Temperature c_5=

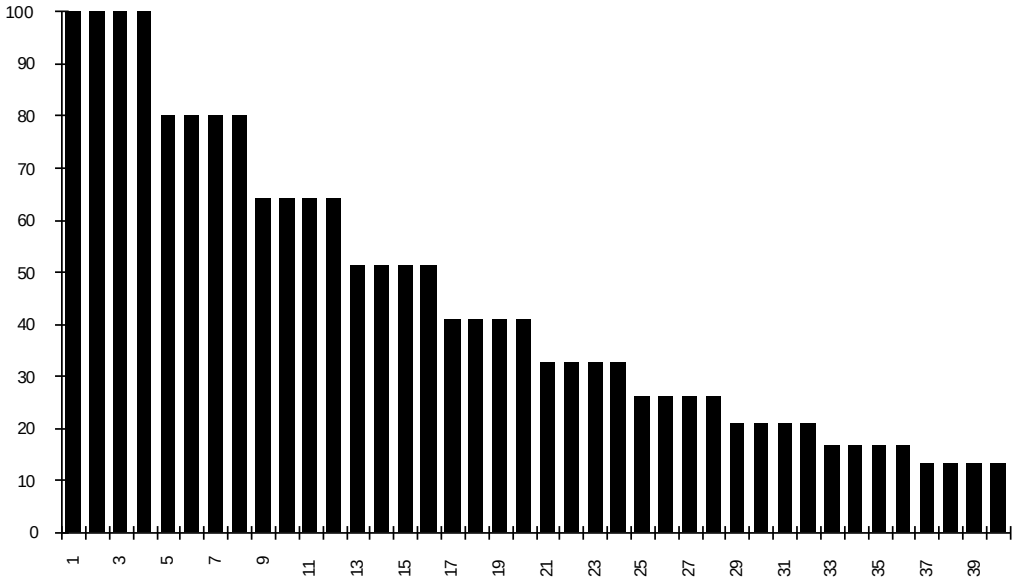
etc

Cooling Schedules:

- Experimentally choose c_1 so that neighbours are accepted at a rate of λ or greater, eg $\lambda = 0.8$ (Kirkpatrick et al)
- Let $c_{k+1} = \alpha\, c_k$, eg α=0.95 (Kirkpatrick et al)



Inhomogeneous Cooling Schedule



Homogenous Cooling Schedule (as seen during local search tests... not c_k)

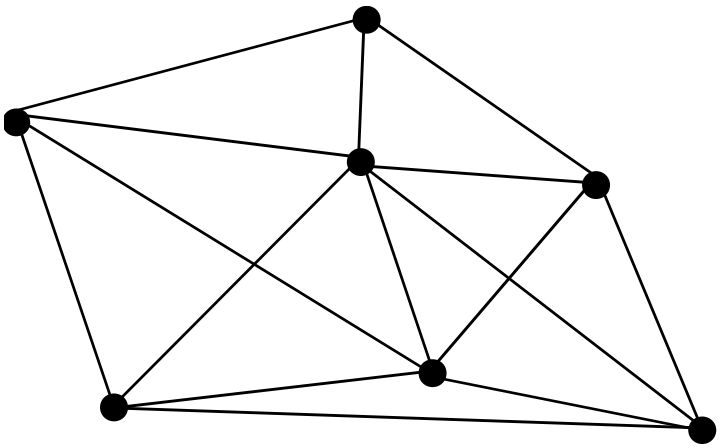
Approaching Equilibrium Sufficiently Closely:

- Run for some number of iterations dependent upon problem size, eg some multiple of number of variables
- Run for some number of iterations being a multiple of the neighbourhood size $|N(x)|$
- Run until variation in cost of accepted neighbours is small
- Run until at least $n_{accepted}$ new neighbours have been accepted, or more than n_{max} neighbours have been examined (n_{max} chosen using one of the above methods)

Stopping Rule:

- Stop when best solution not changed for many iterations, or
- Stop when $k=k_f$, or
- Stop when neighbours are being accepted at a rate of λ_f or less (eg $\lambda_f = 0.01$)

Travelling Salesperson Problem



d(i,j) = d(j,i) =distance from city i to j

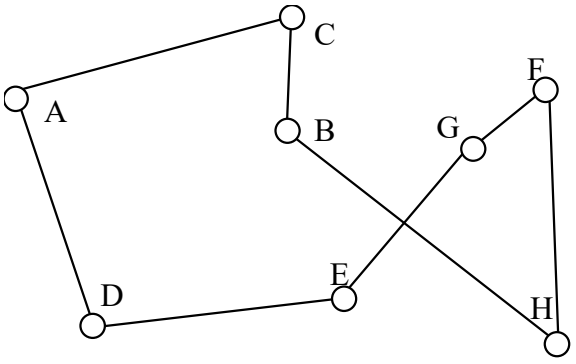
- Want to visit every city and travel the shortest distance (distances often symmetric)
- solution is the order in which we visit the cities
- Known to be NP-hard… polynomial solution algorithms unlikely
- a classic problem for neighbourhood search
- (also solved using integer programming with cuts)

2-opt Neighbourhood Definition

Assume some current solution $\mathbf{x}=(x_1, x_2, ..., x_n)$

e.g $\mathbf{x}=(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ =

where x_i is the i'th city visited in solution x, i=1, 2, ..., n.

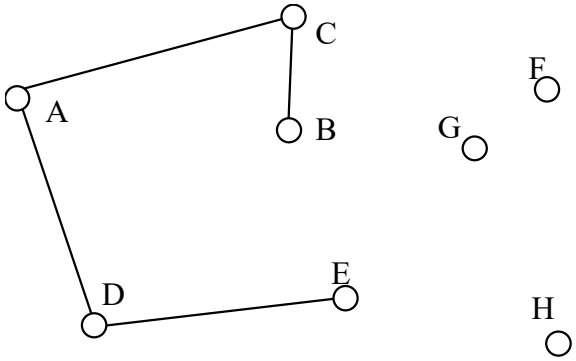


To generate a neighbour of **x**, choose two positions in the tour, say *p* and *q*, where *p*<*q*. e.g. we might choose *p*=4, *q*=6. The neighbour we will generate is designated y(**x**,*p*,*q*). To form this neighbour, we then reverse the order we visit the cities in positions *p* to *q* inclusive in **x**. This swap is known as a '2-opt' transition.

For example, with *p*=4, *q*=6, **x** =(A, C, B, H, F, G, E, D),

we have y(**x**,4,6) =

This gives us the tour...



The neighbour y(**x**,4,6)

In general, with n cities we have

$\mathbf{x}$= (x_1, x_2, ..., x_{p-1}, $\mathbf{x_p}$, $\mathbf{x_{p+1}}$, ..., $\mathbf{x_{q-1}}$, $\mathbf{x_q}$, x_{q+1} , x_{q+2}, ..., x_n)

$y(\mathbf{x},p,q)$= (x_1, x_2, ..., x_{p-1}, x_{q+1} , x_{q+2}, ..., x_n)

i.e.

$$y(\mathbf{x},p,q) = (y_1, y_2, \dots, y_n), y_j = \begin{cases} x_j & j < p \text{ or } j > q \\ x_{q-(j-p)} & \text{otherwise} \end{cases}$$

We need to determine how the total tour distance changes.We let d(i,j) be the distance between cities i and j, and we assume d(i,j)=d(j,i).

For our example above, the total distance, f(x), of the original tour

$\mathbf{x}$ = (A,C,B,H,F,G,E,D)

is given by

f($\mathbf{x}$) = d(A,C) + d(C,B) + d(B,H) + d(H,F) + d(F,G) + d(G,E) + d(E,D) + d(D,A)

while the distance for the new tour

y(x,4,6)=(A,C, B, G, F, H, E, D)

is given by

f(y($\mathbf{x}$,4,6)) =

d(A,C) + d(C,B) + d(B,G) + d(G,F) + d(F,H) + d(H,E) + d(E,D) + d(D,A)

Calculating the difference (new-old), and remembering that d(i,j)=d(j,i) gives

f(y(x,4,6))- f(x) = d(B,G) - d(B,H) + d(H,E) - d(G,E)

In general, the difference is given by

f(y($\mathbf{x}$,p,q))- f(x) =

For general k-opt transitions, see S. Lin & B.W. Kernighan, An effective heuristic algorithm for the Travelling Salesman Problem, Operations Research 21(1973) 498-516.

Next Descent using the TSP Neighbourhood

Our neighbourhood is formally given by

N(x) = {y(x,i,j),

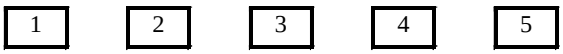
where y(x,i,j) is as defined above.
To explore our neighbourhood in a next-descent fashion we use the following algorithm:

```
For i = 1 to n-1
        For j=i+1 to n
                if f(y(x,i,j))-f(x) < 0 then x := y
```

where f(y(x,i,j))-f(x) is calculated efficiently as detailed above.

Our Bar Stool Seating Problem Revisited

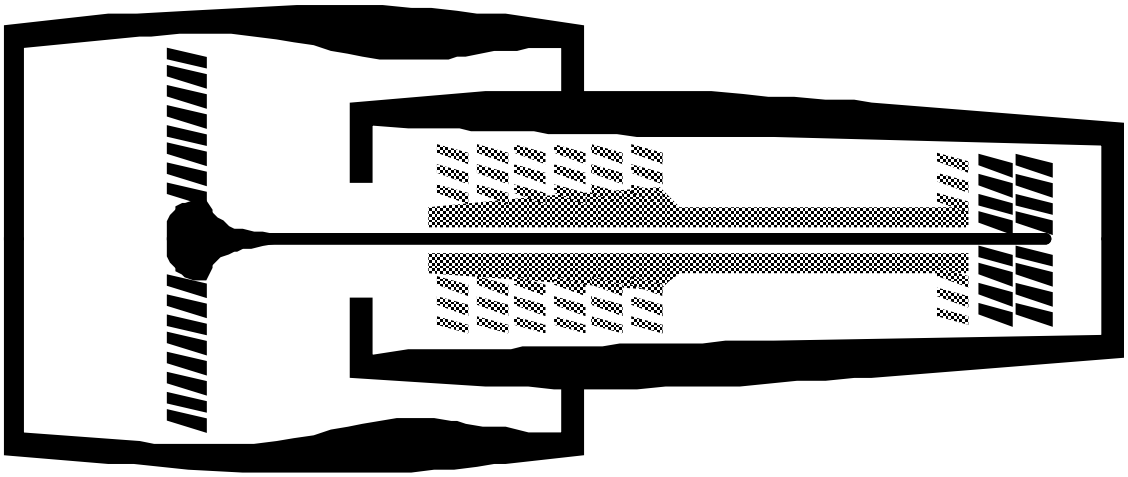
This is a good time to note that the bar-stool seating problem we introduced this material with is in fact an example of…



Fast Function Evaluations

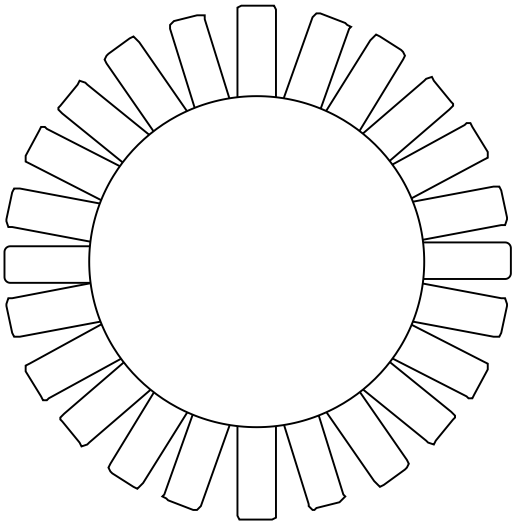
Heuristics rely primarily on speed, not cleverness. We saw how we could quickly compute the change in objective for a 2-opt swap for the travelling salesperson problem. Sometimes, to achieve quick calculation of the change when moving to a neighbour, we need to track not only the objective function of a solution, but also some underlying properties of the solution that get updated quickly and then used to calculate new objective function values. We see this in the next example.

Turbine Balancing Problem



***Jet Engine** (Source: Balancing for Air NZ, Year 4 Project, Glenn Carter, 1993)*

- Air NZ Year 4 Project (Glenn Carter, 1993)
- Each blade i has a weight w_i.
- Each position j has coordinates x_j, y_j from centre
- How do we position the blades around the turbine so that the turbine is as well balanced as possible?



One Turbine to balance
Source: Balancing for Air NZ, Year 4 Project, Glenn Carter, 1993

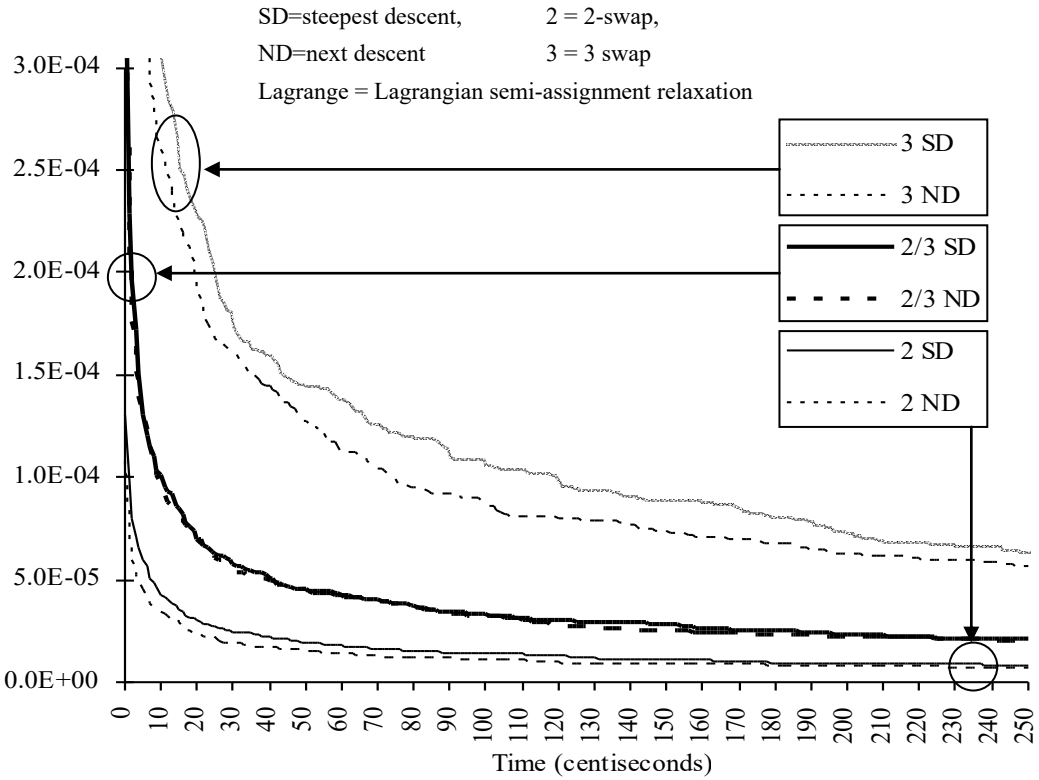
Problem Characterisation

Neighbourhood Strategy

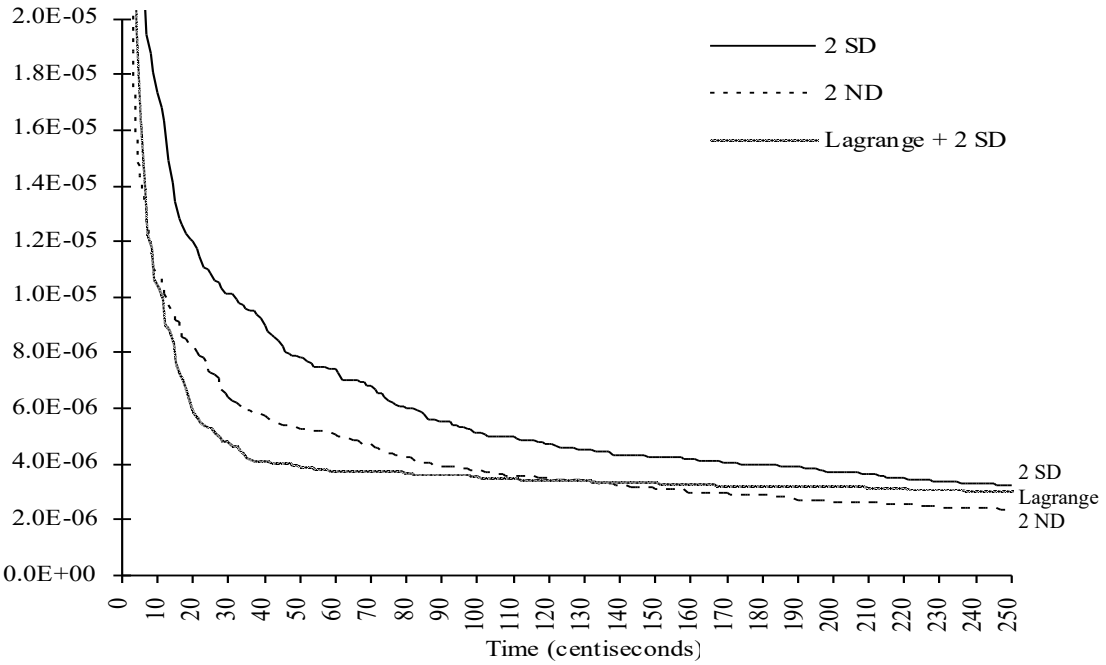
...
...

We need to calculate objective changes quickly. Therefore, for each solution, we need to remember its x and y co-ordinates, ie we store $x=\Sigma w_i x_{p(i)}$ and $y=w_i y_{p(i)}$). When moving to a neighbour, we first update x and y, and then compute the new objective z, and hence the change in objective.

Sample Results



Note: The 2/3 runs where we use 2-swaps until we cannot make any more improvements, and then switch to 3-swaps, are an example of a *variable neighbourhood search*.



Graphs Sourced from: Balancing for Air NZ, Year 4 Project, Glenn Carter, 1993

Satisfiability and Local Search in a Chip

(Adapted from http://www.owlnet.rice.edu/~tdanner/gsat/proposal.html)
(see also http://www.cs.duke.edu/~mlittman/courses/cps271/lect-03/)

Our project is build a chip to implement GSAT, a hill-climbing algorithm for satisfying large Boolean expressions (the 3-SAT problem). Given a Boolean expression (in a specific but general canonical form called 3CNF), the chip uses a randomized algorithm (GSAT) to find a truth assignment to the variables which satisfies the expression. The expression is kept in 4kB of off-chip SRAM, and the work variables (128) are kept in on-chip register space.

The 3-SAT Problem

Instance: a set of n Boolean variables a_1 to a_n, and an expression F of the form (a_i OR NOT a_j OR NOT a_k) AND (NOT a_l OR a_p OR NOT a_q) AND ... that is, consisting of a conjuntion of an arbitrary number of clauses, each of which are a disjunction of three variables, which may be complemented.

Problem Specification:

- binary expressions (listed in order of precedence): F_1 F_2 (conjunction: ``and''), $F_1+F_2$ (disjunction: ``or''), $\overline{F}$ (not F)
- Eg, find x_1, x_2, x_3, x_4, x_5, x_6 to make F= $(\overline{x_3}+x_2+x_1)(x_3+x_6+\overline{x_5})(\overline{x_2}+\overline{x_5}+\overline{x_1})(x_1+x_4+x_5)(x_3+\overline{x_1}+x_4)$=true
- 3-CNF (conjunctive normal form) formula: n variables, m clauses, k=3 literals per clause
  In above example, n=… variables, m=… clauses

Question: does there exist an assignment of truth values for each of the variables such that F is satisfied (that is, true)?

For more background, see http://www.cs.duke.edu/~mlittman/courses/cps271/lect-03/), or these lecture notes http://www.cs.duke.edu/~mlittman/courses/cps271/lect-03/. But consider: the solution-space for a 128-variable problem is 2^{128} - if a machine could try 1 billion solutions per second, it would still take *over 10 trillion years* to try every one.

The GSAT Algorithm

- Randomly initialize $a_{1..n}$
- Do
  - Select a_i at random
  - Complement(a_i)
  - If that increased the number of satisfied clauses
    - Continue

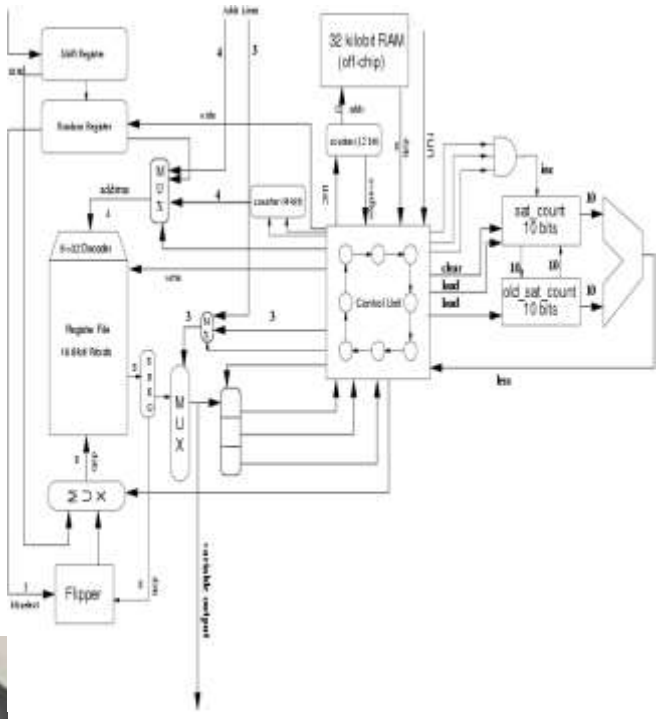
    Else
    - Complement a_i

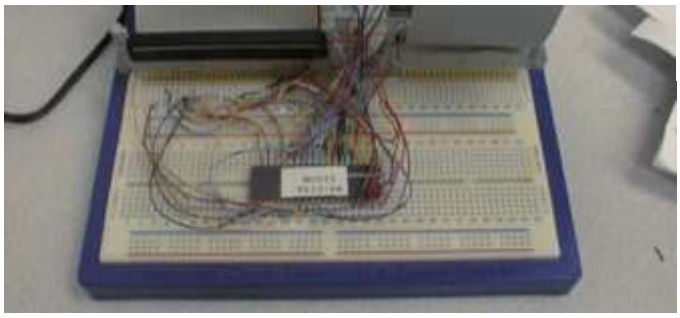
  While F is not satisfied
- Output $a_{1..n}$

See Paper: An Empirical Analysis of Search in GSAT

Chip Modules Needed

- On-chip register space for the variable bit-vector $a_{1..n}$, 128, 256, or 512 bits
- Off-chip RAM for storing the expression, on the order of 4 kilobits
- A source of randomness to select a variable to complement
- An accumulator for counting the number of satisfied clauses, and a register for storing the former value of this count
- A comparator to evaluate the effectiveness of the variable complement
- A controller to manage it all



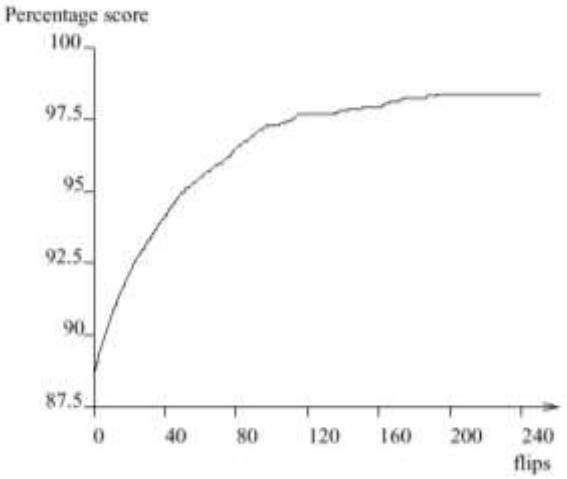


http://www.owlnet.rice.edu/~tdanner/gsat/proposal.html

Typical Results

Plot shows Score against number of iterations (flips). Score = number of satisfied clauses (eg (True)(True)(False) has score 2)

(http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume1/gent93a.pdf)



WSAT for Integer Local Search

WSAT(OIP) (http://www.ps.uni-sb.de/~walser/wsatpb/wsatpb.html) is a domain-independent local search method for linear integer constraints that can solve a range of hard realistic integer optimization problems. WSAT(OIP) has been demonstrated to be effective on large-scale constraint and integer optimization problems. Case studies have been carried out on capacitated production planning, airport gate allocation, sports scheduling (ACC Basketball League 97/98), radar surveillance (covering-type problems), course assignment, the Progressive Party Problem. Publications of the algorithm and of the case studies are available from this page, including two publications at AAAI97 and AAAI98. WSAT(OIP) V1.105 interfaces with AMPL, a modelling language for mathematical programming and the CPLEX callable library.

Genetic Algorithms

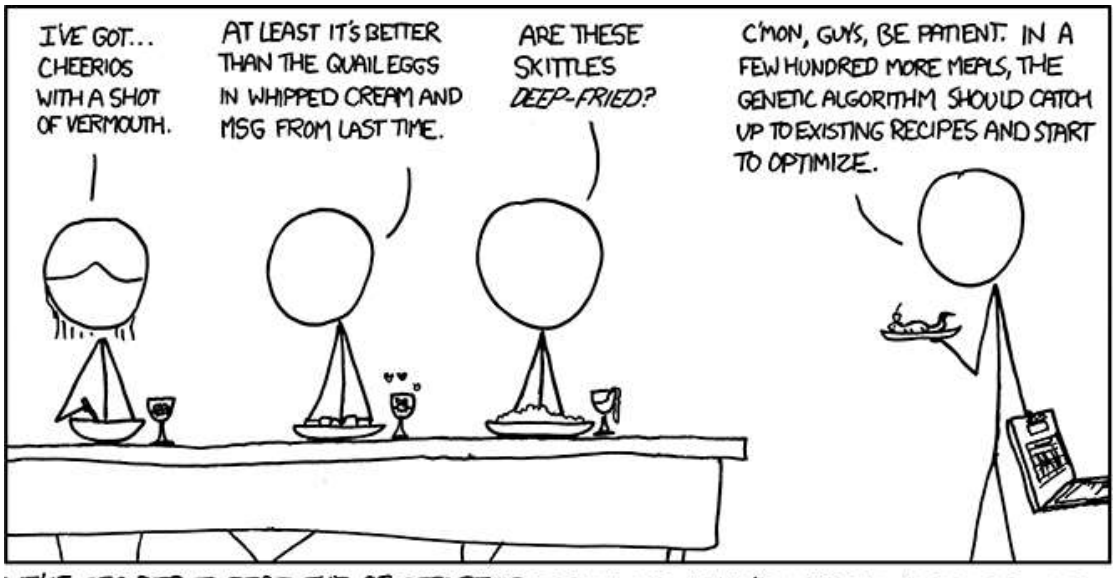
- Neighbourhood Search
- John Holland, University of Michigan, 1970's.
- Based on Evolution:
  - Searches with a Population of Individuals
  - Generations: Parents → Children
  - Mutation
  - Crossover - GA Sex
  - Selection - choose the best individuals
- Trendy!

Evolution...

"[Evolution] is all very natural and organic and in tune with mysterious cycles of the cosmos, which believes that there's nothing like millions of years of really frustrating trial and error to give a species moral fibre and, in some case, backbone.

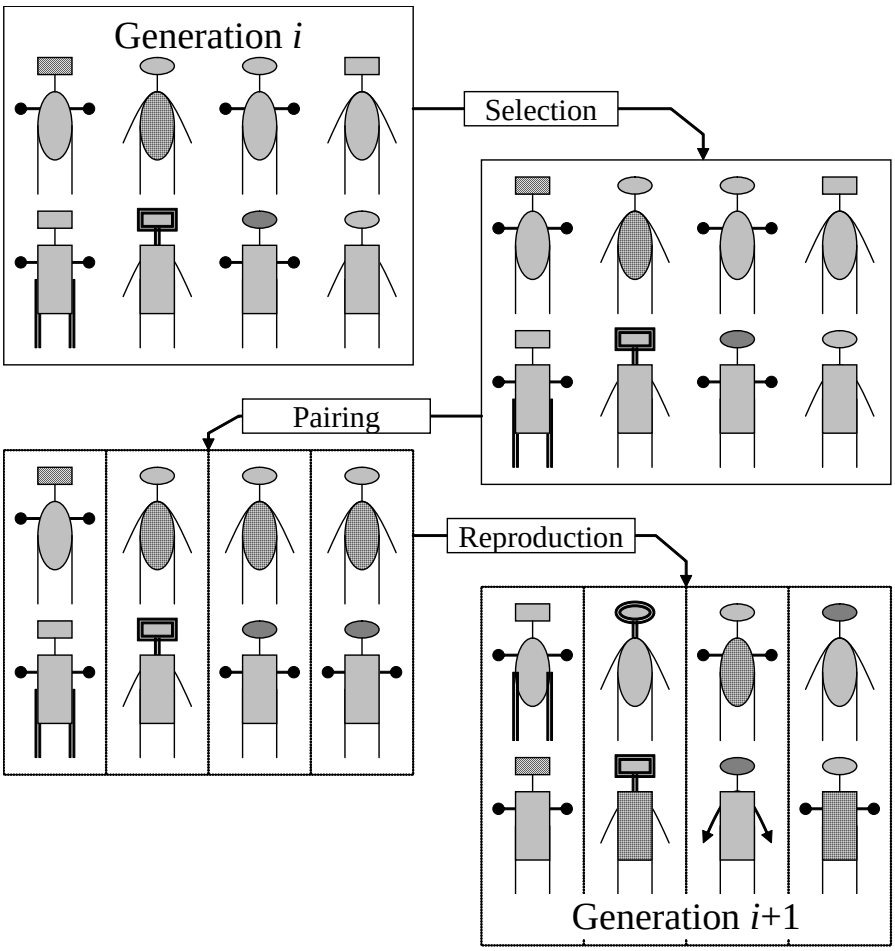
This is probably fine from the species' point of view, but from the perspective of the actual individuals involved, it can be a real pig, or at least a small pink root-eating reptile that might one day evolve into a real pig."

Terry Pratchett "Reaper Man," p11

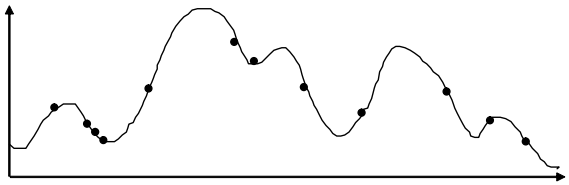


Source: http://xkcd.com/720/

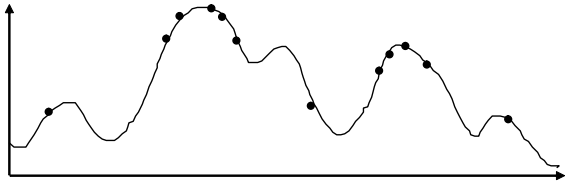
GA Iteration



GA Progress



Distribution of Individuals in Generation 0



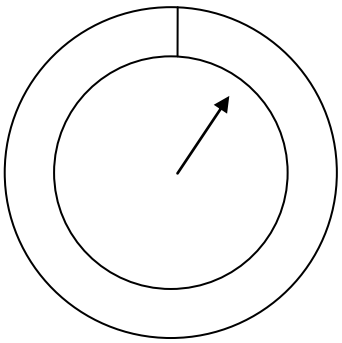
Distribution of Individuals in Generation N

Image from http://web.mst.edu/~ercal/387/slides/GATutorial.ppt

Selection

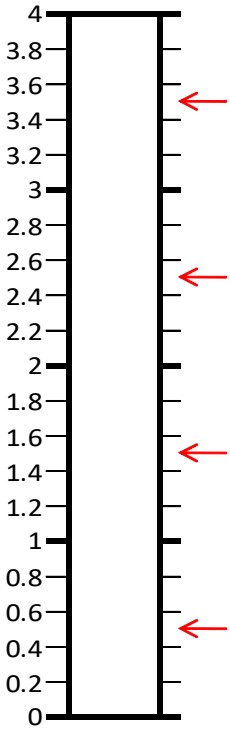
- Individual I has fitness f(I)
- Fitness Based Selection:
  - I appears an expected $f(I)/f_{avg}$ times in the mating pool on average
  - I_{best} appears an expected $\lambda=f(I_{best})/f_{avg}$ times; λ is called the..
- Fitness Scaling to control the selection pressure
  - I appears an expected $[f(I)+c]/[f_{avg}+c]$ times
  - I_{best} appears an expected $\lambda'=[f(I_{best})+c]/[f_{avg}+c]$ times

  Choose a c to give a reasonable λ'
- Rank Based
  - Best individual appears 2 times, second best 1.9 times etc.
- Tournament Selection
  - We build a mating pool of parent individuals by adding individuals one at a time
  - We must decide a tournament size $n_{tournament}$ <n and winner probability *p*
  - To select one more individual to add to the mating pool:
    - Choose $n_{tournament}$ random individuals from the current generation; these form a "tournament"
    - Choose the best individual in the tournament with probability p
    - Choose the second best individual in the tournament with probability p(1-p)
    - Choose the i'th best individual in the tournament with probability $p(1-p)^i$



Implementation of Roulette Wheel Selection for Fitness Based Selection:

For fitness based selection with $f(I_1)=0.9$, $f(I_2)=0.3$, $f(I_3)=1.5$, $f(I_4)=3.3$, $f_{avg}=1.5$, $\Sigma_k f(I_k)=6$. The wheel would be "spun" 4 times for a population of size 4. Each spin chooses I_j with probability $p_j=f(I_j)/\Sigma_k f(I_k)$



Implementation of Deterministic Sampling for fitness-based selection *using fitnesses above. Each segment has height* $f(I_j)/f_{avg}$. *Starting with a random offset, we step up the segments in a step size of 1 selecting an individual each time we touch its segment. In this example, we get* I_1 & I_3 *once, and* I_4 *twice in the mating pool.*

Reproduction

☐Solution Representation

Represent solutions as binary chromosomes

- done historically - no good reason for this! **eg I = 1 0 1 1 0 1 1 0 0 0 1 1**

☐ Produce new solutions using crossover & mutation

☐ 1-point Crossover:

Parent 1: P_1= *1 0 1 1 0 1 1 0 0 0 1 1*

Parent 2: P_2= 1 1 0 0 1 0 1 1 1 1 0 0

Child 1: C_1=

Child 2: C_2=

☐ 2-point Crossover:

Parent 1: P_1= *1 0 1 1 0 1 1 0 0 0 1 1*

Parent 2: P_2= 1 1 0 0 1 0 1 1 1 1 0 0

Child 1: C_1=

Child 2: C_2=

Mutation

☐Introduce random search using mutation

☐Example:

Before: C_1= 1 0 1 1 0 1 1 0 0 0 1 1

After: C_1=

GA Configuration

☐ Population Size
☐ Mutation Rate

☐ Crossover
- ☐Rate
- ☐Type, eg 1-point, 2-point, …

☐ Selection Pressure

☐ Termination after:
- ☐Convergence
- ☐No improvement
- ☐*x* generations

☐ Generational vs Steady State

☐ Replacement Policy

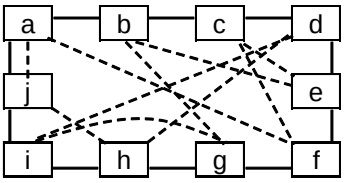
Problem Encoding

- How do we build chromosomes for our problem?
- How do we use crossover/mutation in our problem?
- Combinatorial Problems
  - TSP: Cross 1234 56789
    with 9876 54321
    getting 1234 54321, an invalid tour.
- Use special chromosome decodings and/or crossover:
  - For TSP, preserve edges? subtours?
  - Use known optimality conditions to decode chromosomes.

TSP Crossover

- Chromosome is *order* of cities visited
- Edge Recombination - 1 of many options
  - Starting with parent 1's first city, add edges randomly chosen from either parent, or if stuck, go to a random unvisited city.
  - Parent 1 = abcdefghij
  - Parent 2 = cfajhdigbe
  - Child =
  - Edge Table

```
a  =  b   f   j
b  = a c e g
c  =  b def
d  =   c e  hi
e  =  bcd f
f  = a c e g
g  =  b   f hi
h  =    d  g ij
i  =    d  gh j
j  = a     hi
```

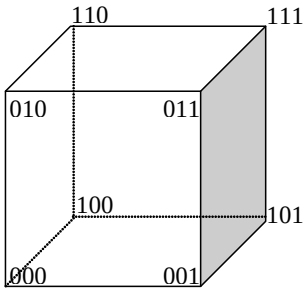


Holland's Theory of Schemata

"Sets of individuals with similar chromosomes."

eg 1** = { }

1*0 = { }



**1={001,011,101,111}

*01={001,101}

00*={000,001}

Schema Fitness

$$f(00*) = \frac{\qquad\qquad}{\qquad\qquad}$$

$$f(**1) \quad \frac{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

Schema Processing

☐ Schema Theorem (Holland, 1975)

$$m(H, g+1) \geq m(H, g)\frac{\bar{f}_g(H)}{\bar{f}_g}[1 - p_d(H)]$$

$m(H,g)$= number of instances of H in g'th generation.

$\bar{f}_g(H)$= average fitness of instances of H in the g'th generation.

$\bar{f}_g$ =average fitness of g'th generation.

$p_d(H)$= probability of crossover/mutation destroying an instance of H

☐Schema Disruption

Consider $p_d(H)$ for 1***1 vs 11***

Schema Processing

☐How many schemata are processed?

☐1011 represents 1011,101*,10*1,10**,
1*11,1*1*,1**1,1***,
*011,*01*,*0*1,*0**,
11,1*,***1,****
=2^k schemata

☐A population of size n represents between 2^k and $n2^k$ schemata.

☐A population of size n processes $o(n^3)$ schemata (where n is chosen large enough to ensure a given survival rate for specified schemata).

False Claims of Beneficial "Intrinisic Parallelism"

"Even though [GAs] try individuals one at a time, it is really schemata which are being tested and ranked. There are somewhere between 2^k and $n2^k$ schemata with instances in each generation. Each schema H changes its proportion in the generations at a rate largely determined by its observed performance f(H), and is largely uninfluenced by what is happening to other schemata. This is the foundation of the intrinsic parallelism of GAs."

- Holland 1975.

"High dimensionality of the search space creates no difficultites for genetic plans, in contrast to its effect on classical procedures, because of the intrinsic parallelism." - *Holland 1975*

No Free Lunch Theorem: These claims of Holland's are false (but often repeated by GA proponents). Specifically, they contradict the "no free lunch" theorem (Wolpert and Macready [1]) which (roughly speaking) says "that if an algorithm performs well

on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems." As an example, imagine trying to find the largest telephone number in a telephone book; there is no efficient way to do this as the problem has no structure that an algorithm can exploit

[1] Wolpert, D.H., Macready, W.G. (1997), "No Free Lunch Theorems for Optimization", *IEEE Transactions on Evolutionary Computation* **1**, 67.

Experimental GA Work

- Mostly non-comparative.
- Fine-tuning GA parameters, eg Schaffer et. al., 1989.
- Unsuitability of test suite demonstrated, Davis 1991.

GA Strengths

- Easy to use
- Black box' fitness
- Stochastic objective functions
- Good for parallel processors
- Only search technique that explicitly combines multiple solutions to form new solutions.

Example GA Application

Scientists at HP developed this AI to mix new music tracks for dancers based on biofeedback from the clubbers. The clubbers are each given a heart monitor, which sends information to the DJ through a wireless link. The DJ itself mixes music using genetic algorithms to find the tracks the audience likes best. The tracks are the "genes", and feedback from the audience determines the fitness levels of the genes.

Conclusions

- Past GA theory has been misleading & offers little guidance for the practitioner.
- Crossover is a unique optimisation tool that can exploit similarities between good solutions. If you can create a good crossover for your problem (and demonstrate this by comparing performance with a tuned "GA" in which crossover is turned off), then using a GA makes sense.
- More research is required to better understand crossover's potential.
- Efforts must be made to improve research in the field:
  - Rigorous analysis of crossover's contribution.
  - Comparison with other heuristics.

Links -

http://darwintunes.org/ listen and rate music being produced by a GA.
http://rednuht.org/genetic_cars_2/ or http://boxcar2d.com/ : Genetic algorithms to create "cars"

GRASP – Greedy Randomised Adaptive Search Procedures

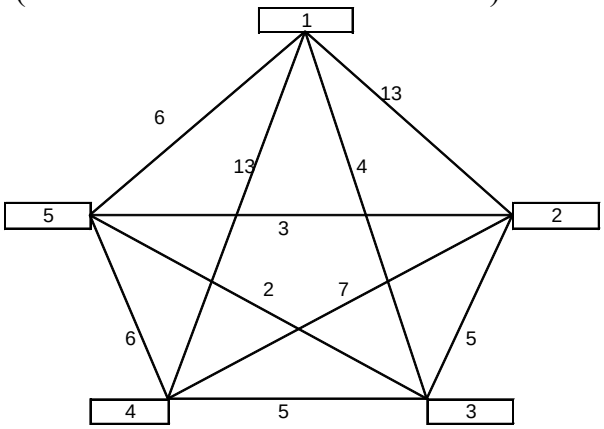
See handout "Greedy Randomized Adaptive Search Procedures" by Thomas A Feo and Mauricio GC Resendez

Key algorithm for creating our starting solutions for local search**:**

```
procedure ConstructGreedyRandomizedSolution
    Solution = {}
    while solution construction not done
        MakeRCL(RCL)                RCL=Restricted Candidate List
        s = SelectElementAtRandom(RCL)
        Solution = Solution ∪ s
        AdaptGreedyFunction(s)
```

Example:

Bar seating problem (modified TSP with no return to start)



A Possible Greedy Heuristic

Build a solution by starting at node 1, and then always going to the next unvisited closest node. (We always add new nodes at the end of the tour.) Stop when all nodes have been visited. (Do not return to the start.)

Greedy solution for our bar problem: Cost=

A Possible Grasp Algorithm for *ConstructGreedyRandomizedSolution*

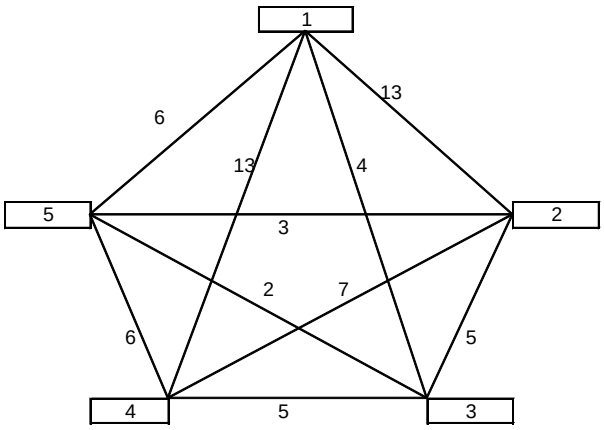
Start at node 1. To choose the next node, the closest *p* unvisited nodes are listed in ascending order. Generate a random number 0<=r<1.We then choose the 1st option if r< 1/p, otherwise we choose the 2nd option if r<2/p, etc.

Run 1 (using p=2)

Random numbers: 0.4, 0.7,0.9, 0.7

Current Node	Closest next p=2 un-visited nodes	Randomly chosen next node	New (Partial) Solution
1			

Solution Cost=



Run 2 (using p=2)

Start at node 1

Random numbers: 0.7, 0.8, 0.4, 0.7

Current Node	Closest next p=2 un-visited nodes	Randomly chosen next node	New (Partial) Solution
1			

Solution Cost=

etc

Note:
We see that our Grasp starting solutions are worse than the greedy sequential solution. However, we hope that occasionally our final Grasp solution (after performing local search) will be better.

Particle Swarm Optimisation (PSO)

PSO typically considers a problem with many continuous variables, e.g. w=weight of drawbridge, p=position of bridge, etc, and so a solution is a vector x=$\begin{pmatrix} w \\ p \end{pmatrix}$

The algorithm works with particles that move around in the solution space, and so the position of a particle specifies a possible solution. The algorithm processes a set of *n* solutions known as a "swarm". Each particle has a current direction and velocity that is updated (as described next) and then used to calculate the particle's position in the next iteration.

At some iteration, each particle i has a position (solution) x_i, some current velocity v_i, and its own best known position (solution) x_i^*. The best position (solution) ever found is given by x^* (being the best of x_i^*, i=1,2,…,n). Each particle i is attracted (with some randomness) towards its best solution x_i^* and the globally best solution x^*

The PSO algorithm for a minimisation problem with d decision variables (i.e. d dimensions) proceeds as follows:

PSO to minimise f(x)
Let n be the number of particles
For each particle i = 1, 2, …, n

Generate an initial (eg random) position $x_i=\begin{pmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,d} \end{pmatrix}$ & velocity $v_i=\begin{pmatrix} v_{i,1} \\ v_{i,2} \\ \vdots \\ v_{i,d} \end{pmatrix}$.

Assign $x_i^* \leftarrow x_i$ # x_i^* is particle *i*'s best ever solution

Let $x^* = \begin{pmatrix} x_1^* \\ x_2^* \\ \vdots \\ x_d^* \end{pmatrix}$ be the best x_i^* over i=1, 2, …, n # x^* is best solution ever found

While not terminated:
For each particle i = 1, 2, …, n:
Update particle's velocity randomly, dimension by dimension
For *k* = 1, 2, …, *d*
r_1=random value between 0 & 1
r_2=random value between 0 & 1
$v_{i,k} \leftarrow \alpha v_{i,k} + \beta r_1 (x_{i,k}^* - x_{i,k}) + \gamma r_2 (x_k^* - x_{i,k})$
Update particle's position
$x_i \leftarrow x_i + v_i$
Update particle's own best solution, and the global best
If $f(x_i) < f(x_i^*)$ then $x_i^* \leftarrow x_i$
If $f(x_i) < f(x^*)$ then $x^* \leftarrow x_i$

Notes:
α, β, γ are user-chosen parameters.
We terminate on a good enough solution or enough iterations.

The parameters can be tuned by hand or using some search heuristic.

For more information, see http://www.particleswarm.info/

Ant Colony Optimisation (ACO)

Ant Colony Optimisation was developed by Marco Dorigo in 1992 in his PhD thesis. His first algorithm sought shortest paths in a graph; based on the behavior of ants seeking a path between their colony and a source of food. Real ants leave pheromone trails that guide their fellow ants towards food. Short routes will have more ants per minute following the route than long routes, which means more pheromone will get laid on the short routes, which will then encourage more ants to follow short routes.

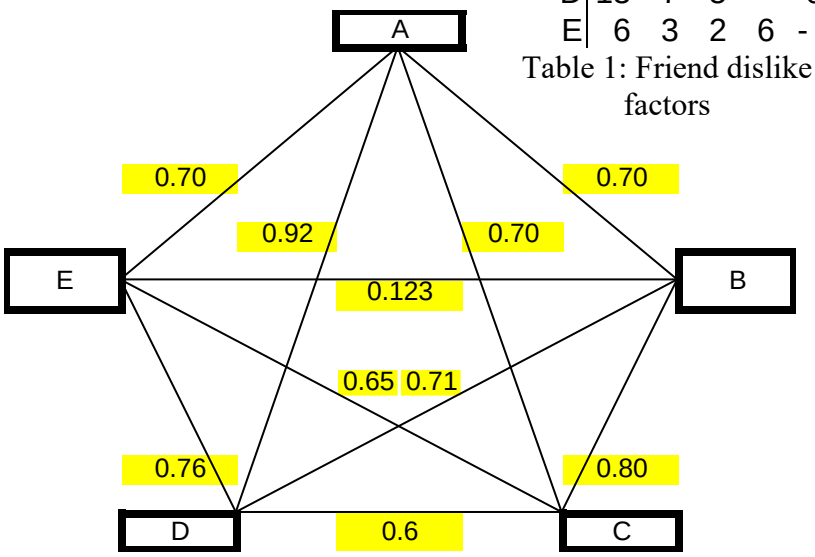
As in the Grasp approach, we build up the solution component by component. However, whereas Grasp chooses randomly between the 2 or 3 best next components, Ant Colony Optimisation can choose any next component, where each component is chosen with a probability dependent on how good that component has been in past solutions.

Bar Seating Example:
The highlighted values below show the 'quality' coefficients $\tau_{i,j}$ for each component (link) (i,j). A larger $\tau_{i,j}$ is better. These values were computed in previous unshown iterations.

d(i,j)	A	B	C	D	E
A	-	13	4	13	6
B	13	-	5	7	3
C	4	5	-	5	2
D	13	7	5	-	6
E	6	3	2	6	-

Table 1: Friend dislike factors

Solution Construction:
We construct the solution by randomly choosing solution components using probabilities based on the $\tau_{i,j}$ values.



Rnd Numbers
0.09
0.48
0.42
0.13
0.18

Starting Random Person:
Random value 0.09→...

Person	Coeffs for next person					Probabilities for next person					Probability Sums for Next Person						Next Person
	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E		
											0					1	0.48→
											0					1	0.42→
											0					1	0.13→
											0					1	0.18→

Thus, our solution is: x=……. with total dislike f(x)=
(Note: The first person is clearly important, and so it should really not be random, but controlled by a $\tau_{i,j}$ value, say $\tau_{0,j}$ where dummy person 0 is a fixed starting person.)

A Coefficient $\tau_{i,j}$ Update using just the Best Solution:
Let's assume we do this, say, 20 times (ie have 20 ants walking along routes), and get the following solutions.

1: E, D, A, C, B: f(x) = 26
2: B, C, A, D, E: f(x) = 33
…
20: A, B, D, C, E: f(x) = 29

with the best of these 20 solutions being
A, C, E, B, D: $f(x_{best}) = 16$

with components
$C(x_{best})$ = {A-C, C-E, E-B, B-D}

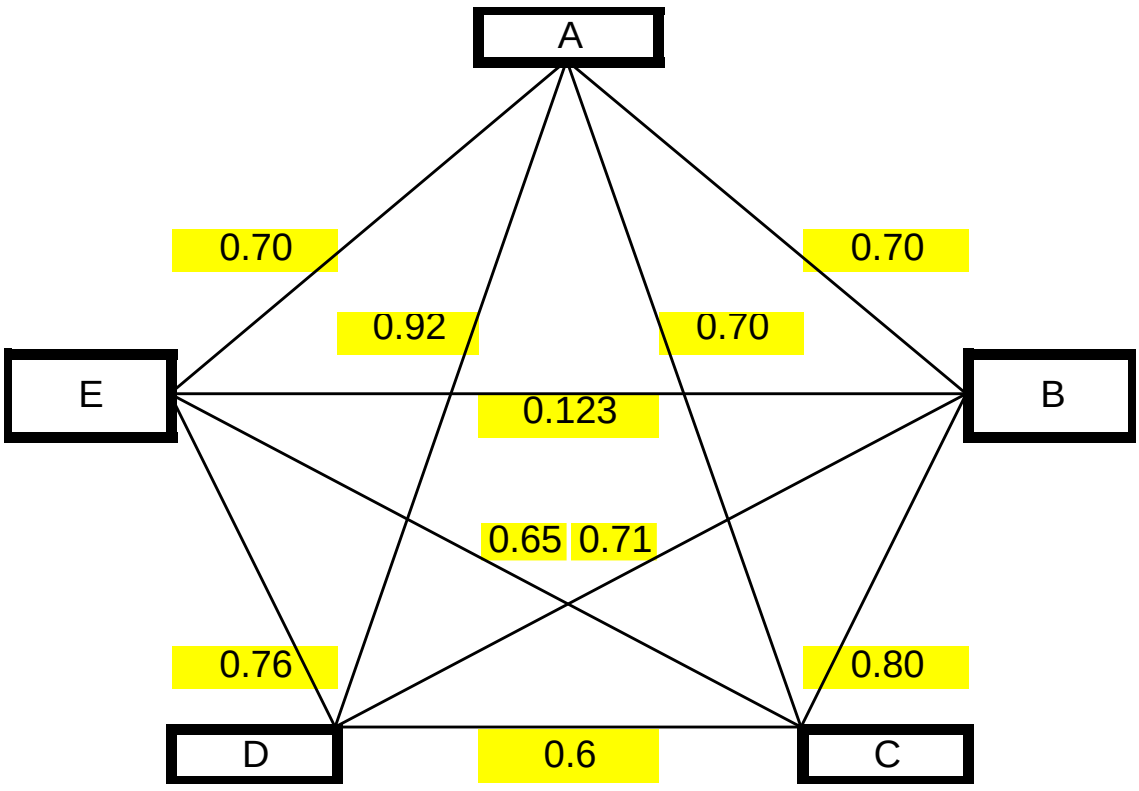
We then update the $\tau_{i,j}$ to reinforce the components in x_{best}:

$$\tau_{i,j} \leftarrow \rho\tau_{i,j} + (1-\rho)\begin{cases}1 & (i,j) \in C(x_{best}) \\ 0 & otherwise\end{cases}$$

where $\rho\tau_{i,j}$ gives the decay of the pheromone, and (1-ρ)*1 gives the reinforcement of the pheromone associated with the components of the best solution. The ρ determines the rate of update of the pheromone; ρ close to 1 implies low evaporation and slow updates, while ρ close to 0 implies high evaporation and fast updates.

To prevent very small values ($\tau_{i,j}$=0) or very large values, we may define limits, eg τ_{min}=0.1, τ_{max}=0.9, and put: if $\tau_{i,j}<\tau_{min}$ then $\tau_{i,j}\leftarrow\tau_{min}$; if $\tau_{i,j}>\tau_{max}$ then $\tau_{i,j}\leftarrow\tau_{max}$

Example: ρ=0.5, $C(x_{best})$ = {A-C, C-E, E-B, B-D}, no τ_{min} or τ_{max} limits



In the next iteration, the random solutions are more likely to be variants on the good A,C,E,B,D solution, and thus hopefully we will find an even better solution.

More Sophisticated Construction Algorithms

Let $\tau_{i,j}$ be the current pheromone value associated with adding component (person) j after component (person) i

Let $\eta_{i,j}$ be some problem specific measure of how good it is to add component (person) j after component (person) i. A larger $\eta_{i,j}$ indicates a better component to add. Eg, for our bar problem, we could put

$\eta_{i,j}$=… …

Then, the probability of adding component j to the partial solution ending with component i is given by

$p_{i,j}$ =

where α, β are constants indicating the relative importance of the two factors.

NB: For small α, and large β, we have a

More Sophisticated Pheromone Update Algorithms

Rather than just updating $\tau_{i,j}$ to reflect the single best solution, we can update $\tau_{i,j}$ to reflect all solutions, where the contribution of each solution depends on its quality.

Assume we are minimizing. If we generate m random solutions x_1, x_2, …, x_m, with objective function values $f(x_1)$, $f(x_2)$, …, $f(x_m)$ (eg total dislikes for our bar problem), and components $C(x_1)$, $C(x_2)$, …, $C(x_m)$ (eg the people who are adjacent in each solution), then we put

$\tau_{i,j} \leftarrow$

where f_{upper} is some upper bound on the f() values, and as before, $\rho\tau_{i,j}$ gives the decay of the pheromone, and $(1-\rho)*1$ gives the pheromone reinforcement.

Resources: Why not to mimic ants: https://www.youtube.com/watch?v=prjhQcqiGQc

Very Large Scale (VLS) Neighbourhood Search

For some problems, we can define very large neighbourhoods that can be explored very efficiently using, for example, shortest path, assignment or dynamic programming algorithms. For such problems, the algorithm produces the next solution to use in a steepest descent local search. Very large neighbourhoods mean good quality local optima. Efficient searching of these big neighbourhoods means we can quickly make big improvements in the objective function.

This approach can also be used in Tabu Search where the history can be incorporated into the search algorithm to prevent tabu moves being made.

An example of VLS neighbourhood search is local branching (covered shortly). Also see, for example, Dynasearch (by Chris Potts et al) where the neighbourhood is searched using dynamic programming.

Example: Multiple 'independent' 2-opts for TSP

Recall that a 2-opt neighbour was defined by $y(\mathbf{x},p,q)$ as follows

$\mathbf{x}=$ $(x_1, x_2, \ldots, x_{p-1}, \mathbf{x_p}, \mathbf{x_{p+1}}, \ldots, \mathbf{x_{q-1}}, \mathbf{x_q}, x_{q+1}, x_{q+2}, \ldots, x_n)$

$y(\mathbf{x},p,q)=$ $(x_1, x_2, \ldots, x_{p-1}, \mathbf{x_q}, \mathbf{x_{q-1}}, \ldots, \mathbf{x_{p+1}}, \mathbf{x_p}, x_{q+1}, x_{q+2}, \ldots, x_n)$

with

$\Delta_{pq}=f(y(\mathbf{x},p,q))- f(x) = d(x_{p-1},x_q) - d(x_{p-1},x_p) + d(x_p,x_{q+1}) - d(x_q,x_{q+1})$

If we have two independent (non-interacting) swaps that we do in sequence, then the total change in objective is the sum of the objective changes for the individual swaps.

Example 1:

$\mathbf{x}=$ $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$

Swap 1A: $y(x;2,3) =$ $(x_1, \underline{x_3, x_2}, x_4, x_5, x_6, x_7, x_8)$; $\Delta_{2,3}=f(y(x,2,3))-f(x)$

Swap 1B: $y(x;5,7) =$ $(x_1, x_2, x_3, x_4, \underline{x_7, x_6, x_5}, x_8)$; $\Delta_{5,7}=f(y(x,5,7))-f(x)$

Swap 1A followed by 1B:

$y(x;2,3;5,7)=$ $(x_1, \underline{x_3, x_2}, x_4, \underline{x_7, x_6, x_5}, x_8)$; $\Delta_{2,3;5,7}=\Delta_{2,3}+\Delta_{5,7}$

Example 2:

Swap 2A: $y(x;2,5) =$ $(x_1, \underline{x_5, x_4, x_3, x_2}, x_6, x_7, x_8)$

Swap 2B: $y(x;5,7) =$ $(x_1, x_2, x_3, x_4, \underline{x_7, x_6, x_5}, x_8)$

Are these independent in the sense above? That is, does $\Delta_{2,5;5,7}=\Delta_{2,5}+\Delta_{5,7}$?

Example 3:

Swap 3A: $y(x;2,3) =$ $(x_1, \underline{x_3, x_2}, x_4, x_5, x_6, x_7, x_8)$; $\Delta_{2,3}=f(y(x,2,3))-f(x)$

Swap 3B: $y(x;4,6) =$ $(x_1, x_2, x_3, \underline{x_6, x_5, x_4}, x_7, x_8)$; $\Delta_{6,8}=f(y(x,6,8))-f(x)$

Are these independent in the sense above? That is, does $\Delta_{2,3;4,6}=\Delta_{2,3}+\Delta_{4,6}$?

From these examples, we see that two swaps $y(x;p_1,q_1)$, $p_1 \le q_1$, and $y(x;p_2,q_2)$, $p_2 \le q_2$, with $p_1 \le p_2$ are independent if …….

We can represent these swaps as arcs in a graph with the following nodes:

$[x_1]$ x_2 x_3 x_4 x_5 x_6 x_7 x_8 $[x_1]$

Note: For simplicity, we fix x_1 as the first city (& by wrap around, as the 'last' city.)

Examples:

Swap 1A:y(x;2,3) = $(x_1, \underline{x_3, x_2}, x_4, x_5, x_6, x_7, x_8)$; $\Delta_{2,3}$=f(y(x,2,3))-f(x)

$[x_1]$ x_2 x_3 x_4 x_5 x_6 x_7 x_8 $[x_1]$

Swap 1B:y(x;6,8) = $(x_1, x_2, x_3, x_4, x_5, \underline{x_8, x_7, x_6})$; $\Delta_{6,8}$=f(y(x,6,8))-f(x)

$[x_1]$ x_2 x_3 x_4 x_5 x_6 x_7 x_8 $[x_1]$

Swap 1A+1B:

y(x;2,3;6,8)= $(x_1, \underline{x_3, x_2}, x_4, x_5, \underline{x_8, x_7, x_6})$; $\Delta_{2,3;6,8}=\Delta_{2,3}+\Delta_{6,8}$

$[x_1]$ x_2 x_3 x_4 x_5 x_6 x_7 x_8 $[x_1]$

We can now define a new very large neighbourhood as the set of all solutions reached by any combination of independent 2-opt swaps. To find the best neighbour for a steepest descent search, we formulate a shortest path in the following *improvement graph*.

$[x_1]$ x_2 x_3 x_4 x_5 x_6 x_7 x_8 $[x_1]$

Performance: Consider now, for this problem, how the solutions generated using the shortest path approach will compare with a standard 2-opt approach.

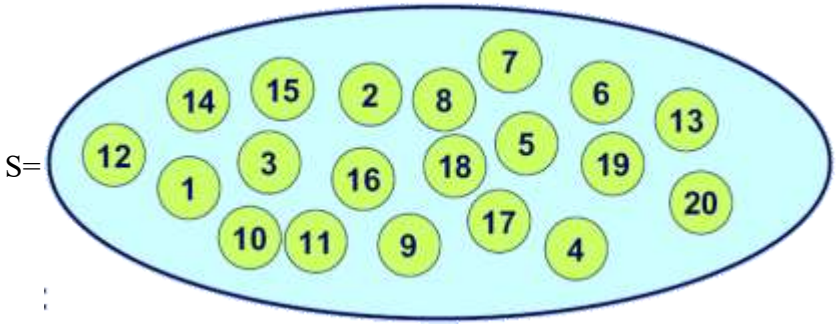
What can we say about the set of local optima that can be found under each approach?

So, will this very large neighbourhood give better quality local optima?

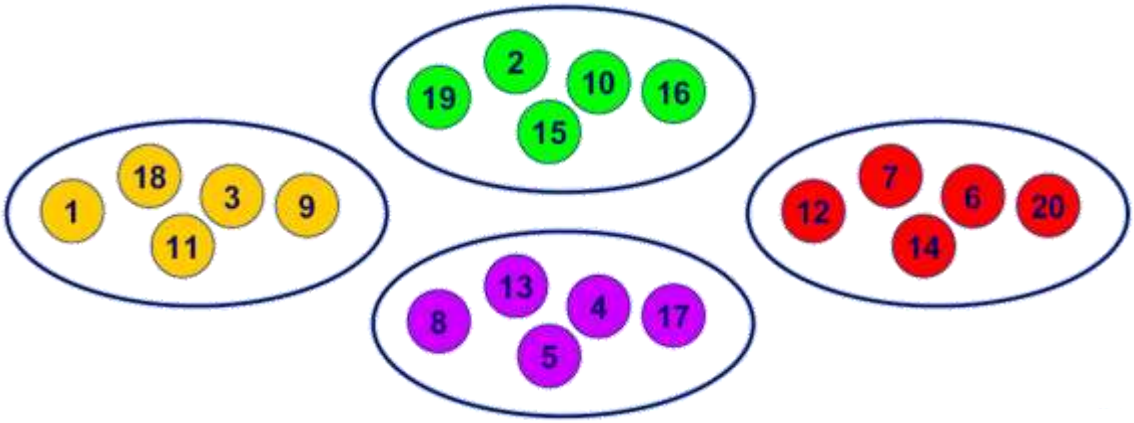
Will this very large neighbourhood give better local optima more quickly?

Example :Cyclic Exchange for Partitioning Problems
See http://ocw.mit.edu/NR/rdonlyres/Sloan-School-of-Management/15-082JNetwork-OptimizationSpring2003/243EFE06-A974-453A-B328-79B133697480/0/24vls_neighborhoodsearch.pdf

Given a set S objects, …



… partition S into p subsets (S_1, S_2, S_3, .. , S_p) …

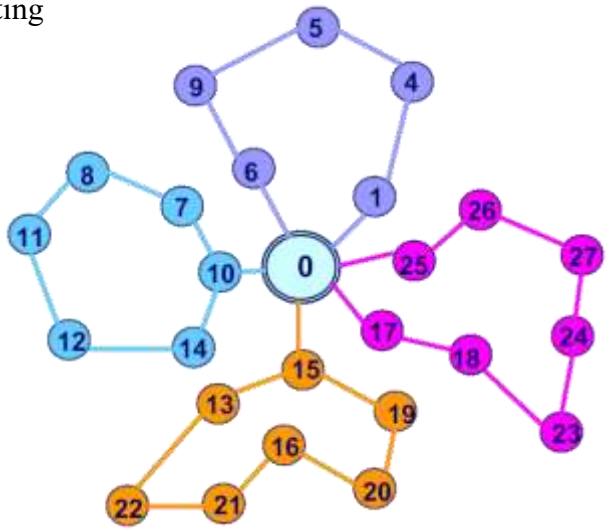


… so as to minimise $\Sigma f(S_i)$, where $f(S_i)$ is a (non-trivial) cost function defined for S_i.

Example Partitioning Problem: Vehicle Routing

S_A={1, 4, 5, 6, 9}
…
S_D={7, 8, 10, 11, 12, 14}

$f(S_i)$ is the distance obtained by solving a TSP for the cities in S_i.

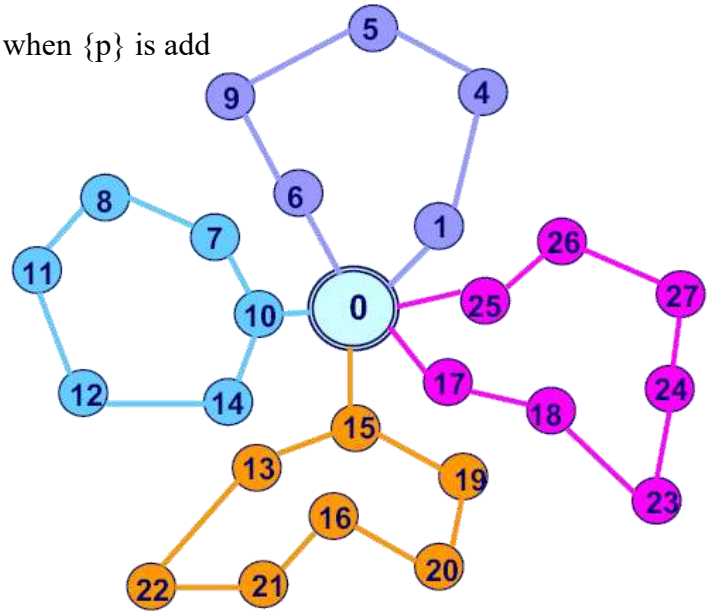


Typical neighbourhood rules for set partitioning might be…… …

Path Exchange Neighbourhood: A VLS Neighbourhood for Set Partitioning

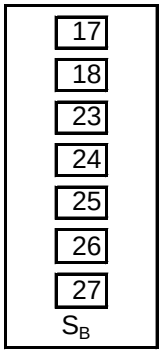
In a *path exchange* neighbourhood rule, we move one member from each set S_i into the next set S_{i+1}. In the *cyclic exchange* neighbourhood, we also allow a member of the last set (S_4 in the example below) to move into the first set (S_1).

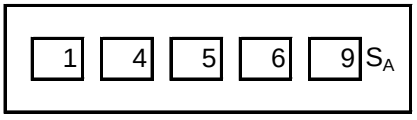
The change in objective for each set S_i when {p} is add and {q} removed is given by
$\Delta_i(p,q)$ = ……

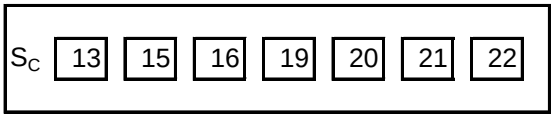


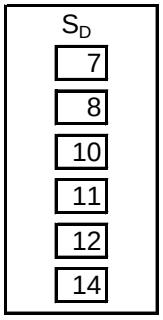
We can now represent this 'path exchange' neighbour as a cycle in a network defined as shown below. Note that only some arcs are shown; every node in set S_i should be connected to every node in the next set S_{i+1}.

Example Cycle & Objective Changes:
$S_A \leftarrow S_A+\{14\}-\{1\}$, $\Delta_A(14,1)$
$S_B \leftarrow S_B+\{1\}-\{17\}$, $\Delta_B(1,17)$
$S_C \leftarrow S_C+\{17\}-\{20\}$, $\Delta_C(17,20)$
$S_D \leftarrow S_D+\{20\}-\{14\}$, $\Delta_D(20,14)$









NB: Only some arcs shown;
every node in every set
should be connected to every
node in the next set.

Finding any cycle with a negative total distance in such a network containing all allowed changes for each set S_i gives a possible path exchange that improves the

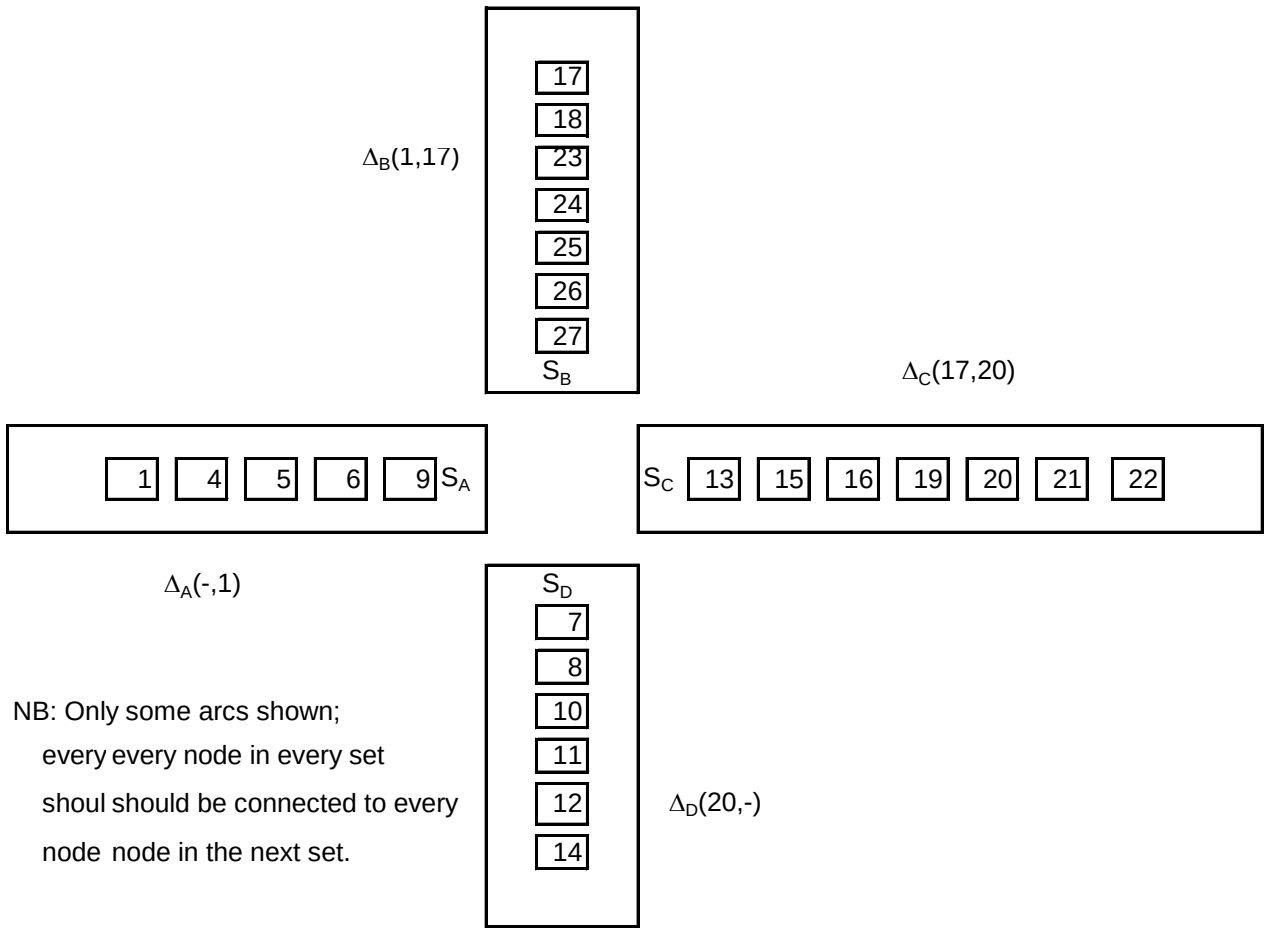
objective. (This path may make some sets worse to make others better.) Finding the most negative cycle in this network gives us the next solution to use in a steepest descent local search.

Note: There are several efficient algorithms for finding negative cycles in graphs.

Extensions: How would you modify the graph to allow an item to be added to a set from the previous set, without a corresponding item being removed? Eg, we want to move item 20 from S_C to S_D, and not move anything from S_D to S_A, giving the new set of changes:

$S_A \leftarrow S_A$ ~~+{14}~~ -{1},	objective function change = $\Delta_A(-,1)$
$S_B \leftarrow S_B$+{1}-{17},	objective function change = $\Delta_B(1,17)$
$S_C \leftarrow S_C$+{17}-{20},	objective function change = $\Delta_C(17,20)$
$S_D \leftarrow S_D$+{20} ~~-{14}~~	objective function change = $\Delta_D(20,-)$

We've used a -, as in $\Delta_D(20,-)$ and $\Delta_A(-,1)$ to denote the new changes in $f(S_D)$ & $f(S_A)$. We modify our graph by ....

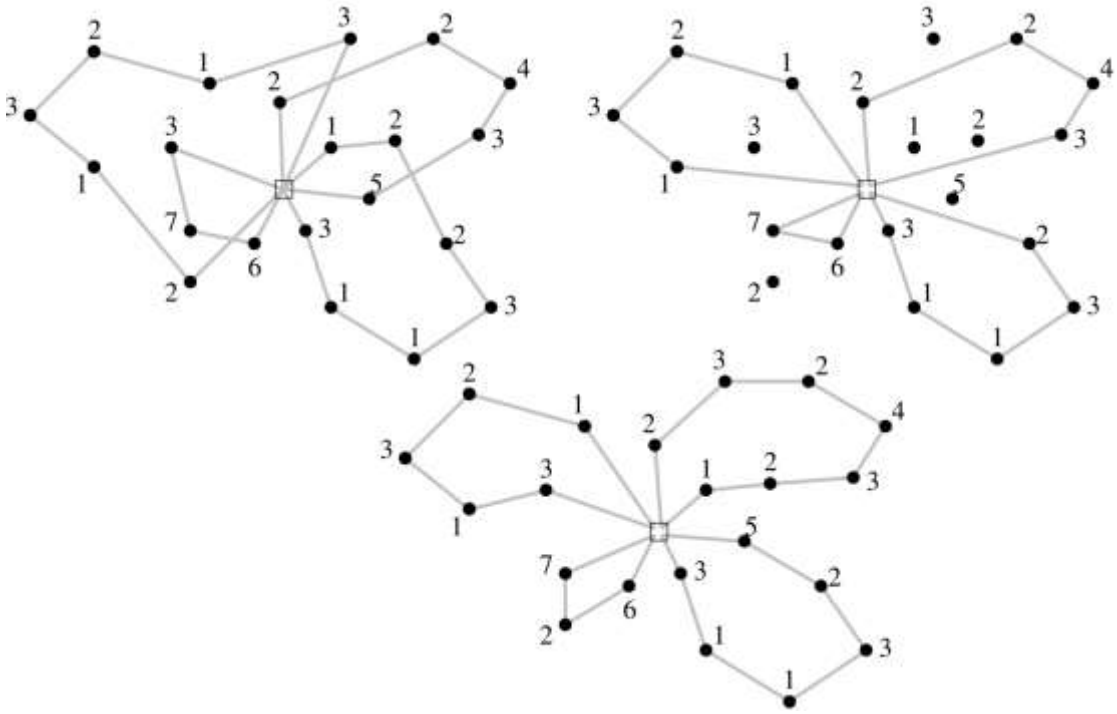


How would we allow some sets to be skipped, eg no change in S_C?

Destroy and Repair Heuristics: (Adaptive) Large Neighbourhood Search

A meta-heuristic known as Large Neighborhood Search (LNS) was proposed by Shaw[1] in 1998. In LNS, an initial solution is gradually improved by alternately destroying (i.e. making infeasible) and repairing the solution. Thus, the neighbourhood is defined by these "destroy" and "repair" operations.

We illustrate this with an example.



(Figure from technical report "Large neighborhood search" by David Pisinger and Stefan Ropke.)

Consider a capacitated vehicle routing problem in which customers have to be allocated to vehicle routes while satisfying some vehicle capacity. The top left figure, above, shows some capacitated vehicle routing solution in which 4 vehicle routes are used to serve a number of customers. This solution has routes that overlap, suggesting it may not be of good quality.

To improve this solution, we run a "destroy" algorithm which, in this case, targets those customers that appear to be causing the overlapped routes. The top-right figure shows the solution after 6 customers have been removed by this "destroy" operation.

This infeasible solution is then provided as input to a "repair" algorithm, such as a greedy sequential algorithm, which attempts to regain feasibility. This gives the improved solution shown in the bottom-most figure above.

[1] P. Shaw. Using constraint programmingand local search methods to solve vehicle routing problems. In CP-98 (Fourth InternationalConference on Principles and Practice of Constraint Programming), volume 1520 of Lecture Notes in Computer Science, pages 417–431, 1998.

The user has to choose appropriate destroy and repair algorithms. The destroy algorithm would often be a randomised heuristic that would produce different solutions each time it ran. The repair algorithm is then typically a (relatively simple) heuristic. However, in some cases, the repair problem may be solvable using a standard algorithm such as assignment or network flow.

Let: x be some current solution,
D(x) the set all "destroyed" solutions one can create from x,
y be some infeasible (destroyed) solution, and
R(y) be the solution obtained by repairing y

Then, our neighbourhood N(x) is given by

N(x) =

The destroy and repair algorithms often include randomness, and so LNS is randomly choosing a new solution from the neighbourhood. Because the neighbourhood is so large, it is typically not explored systematically, and so LNS cannot give any guarantees that a local optimum has been found.

Adaptive Large Neighbourhood Search (ALNS)

Adaptive Large Neighbourhood Search was introduced by Psinger and Ropke in 2006[2]. They describe it as follows:

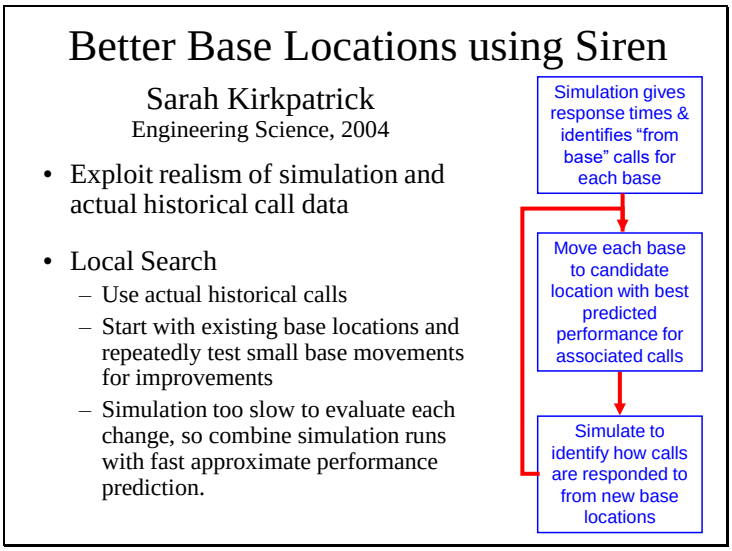
The Adaptive Large Neighborhood Search (ALNS) heuristic […] extends the LNS heuristic by allowing multiple destroy and repair methods to be used within the same search. Each destroy/repair method is assigned a weight that controls how often the particular method is attempted during the search. The weights are adjusted dynamically as the search progresses so that the heuristic adapts to the instance at hand and to the state of the search. Using neighbourhood search terminology, one can say that the ALNS extends the LNS by allowing multiple neighborhoods within the same search. The choice of neighborhood to use is controlled dynamically using recorded performance of the neighborhoods.

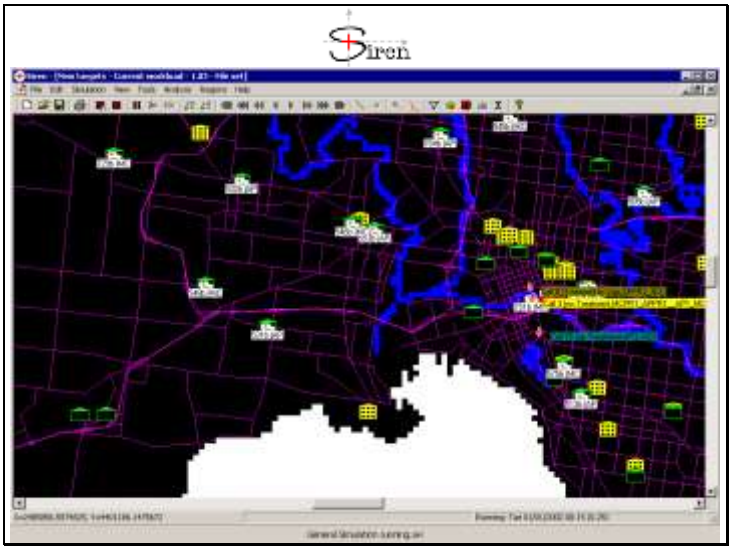
See Psinger and Ropke (2006) for more details.

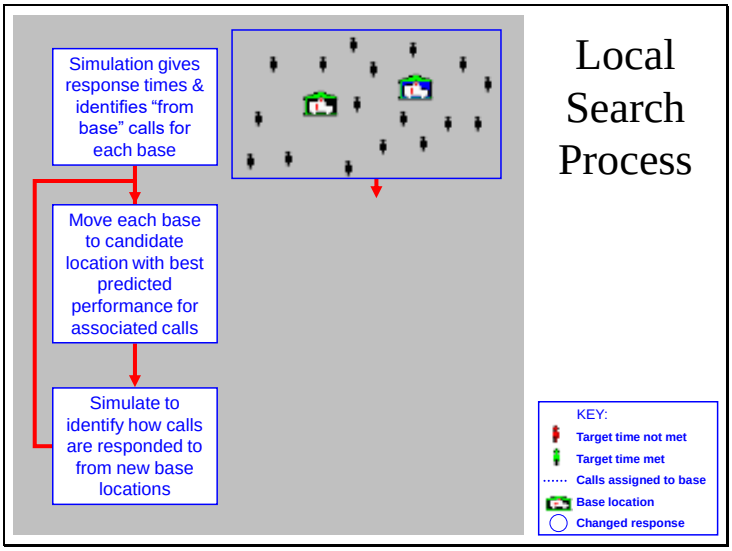
[2] S. Ropke and D. Pisinger. An adaptive large neighborhoodsearch heuristic for the pickup and delivery problem with time windows. Transportation Science, 40(4):455–472, 2006.

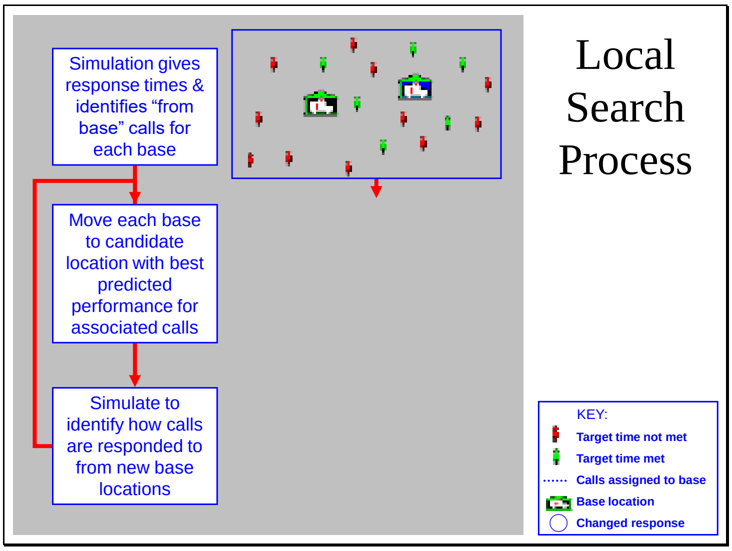
Case Study: Local Search and Simulation

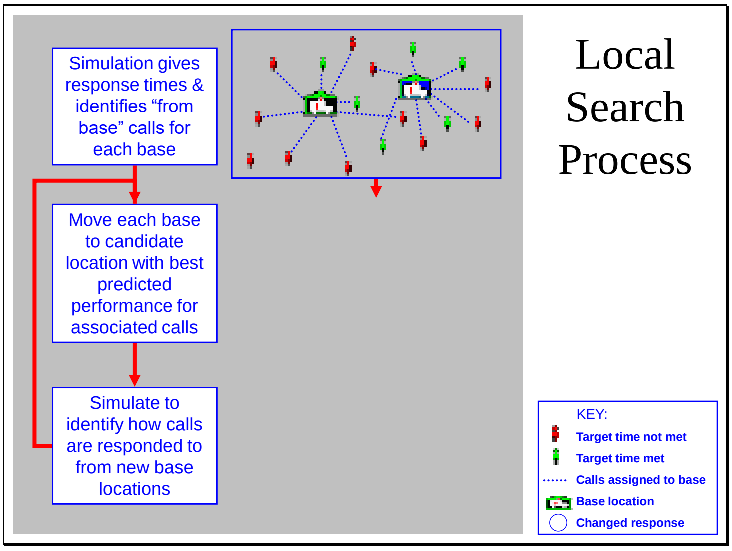
Image Source: AJ Mason © 2004

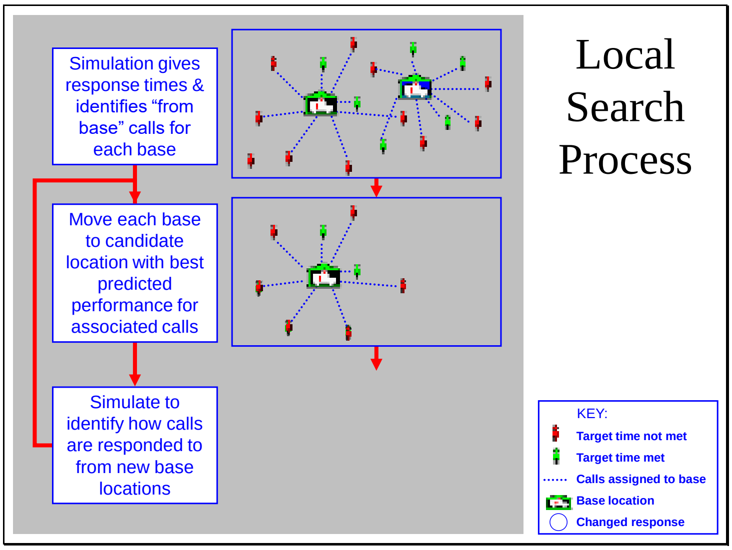


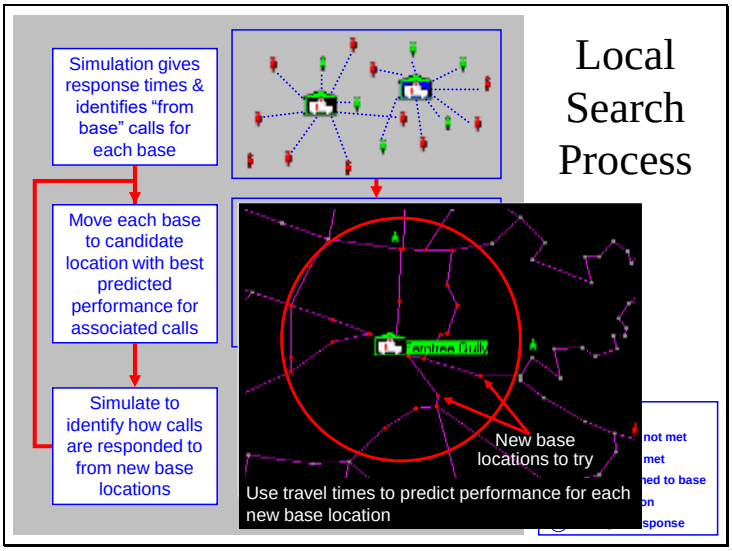


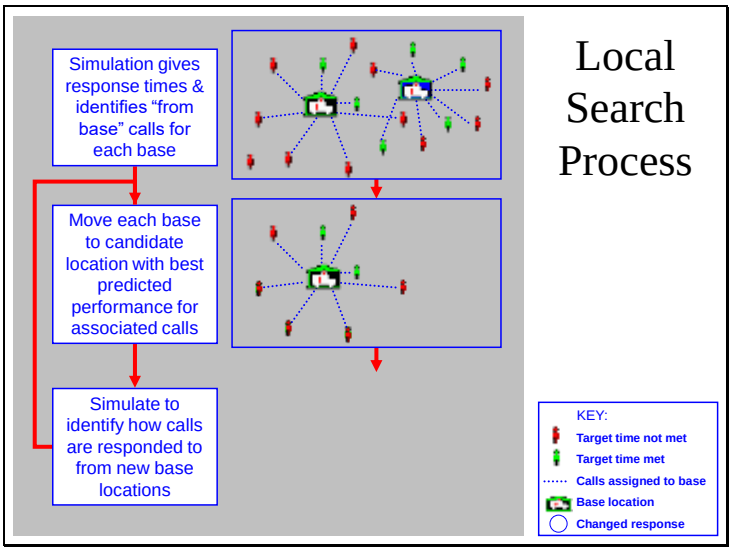


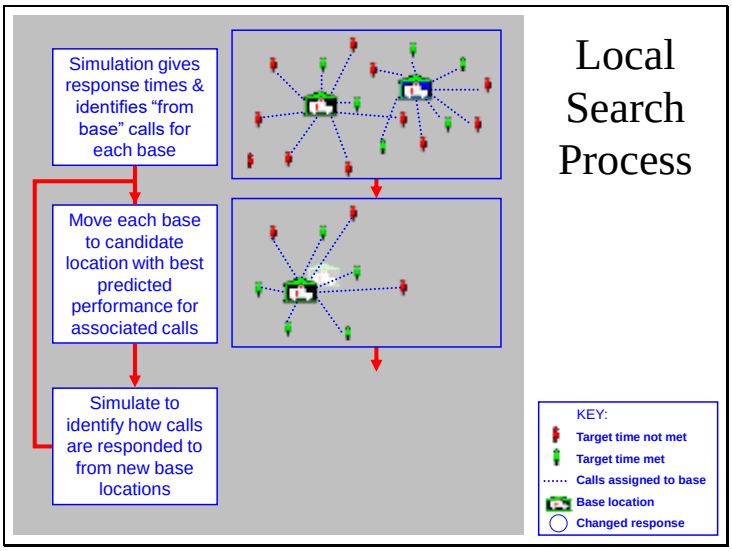

Local Search Process
Simulation gives response times & identifies “from base” calls for each base
Move each base to candidate location with best predicted performance for associated calls
Simulate to identify how calls are responded to from new base locations
KEY:
Target time not met
Target time met
Calls assigned to base
Base location
Changed response

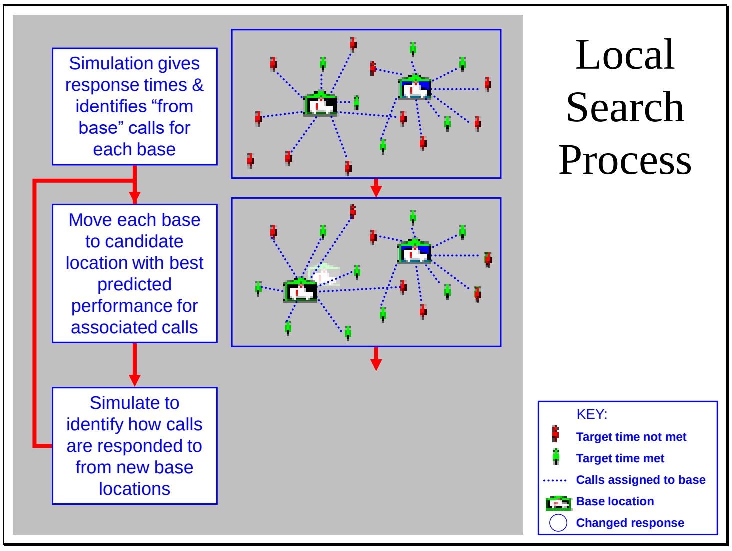

Local Search Process
Simulation gives response times & identifies “from base” calls for each base
Move each base to candidate location with best predicted performance for associated calls
Simulate to identify how calls are responded to from new base locations
KEY:
Target time not met
Target time met
Calls assigned to base
Base location
Changed response


Local Search Process
Simulation gives response times & identifies “from base” calls for each base
Move each base to candidate location with best predicted performance for associated calls
Simulate to identify how calls are responded to from new base locations
KEY:
Target time not met
Target time met
Calls assigned to base
Base location
Changed response

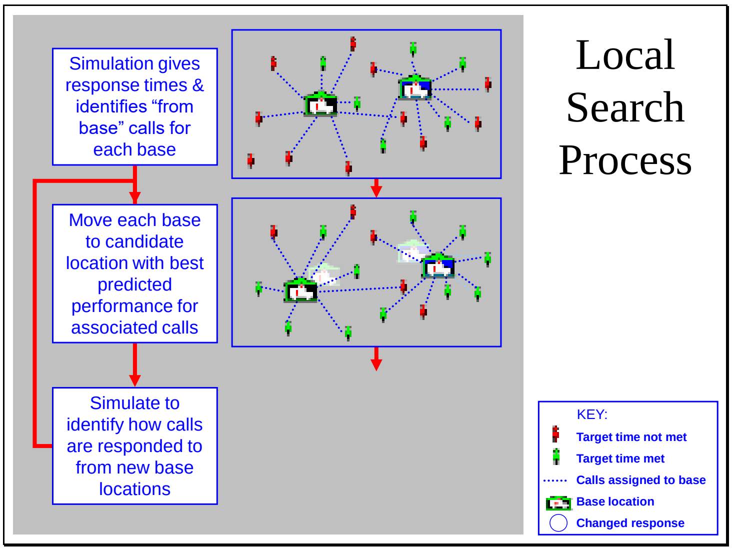

Local Search Process
Simulation gives response times & identifies "from base" calls for each base
Move each base to candidate location with best predicted performance for associated calls
Simulate to identify how calls are responded to from new base locations
New base locations to try
Use travel times to predict performance for each new base location


Local Search Process
Simulation gives response times & identifies "from base" calls for each base
Move each base to candidate location with best predicted performance for associated calls
Simulate to identify how calls are responded to from new base locations
KEY:
Target time not met
Target time met
Calls assigned to base
Base location
Changed response

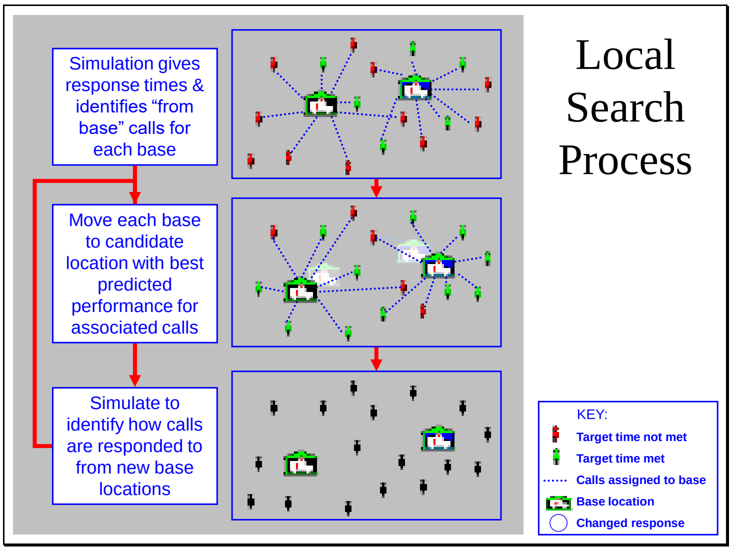

Local Search Process
Simulation gives response times & identifies "from base" calls for each base
Move each base to candidate location with best predicted performance for associated calls
Simulate to identify how calls are responded to from new base locations
KEY:
Target time not met
Target time met
Calls assigned to base
Base location
Changed response



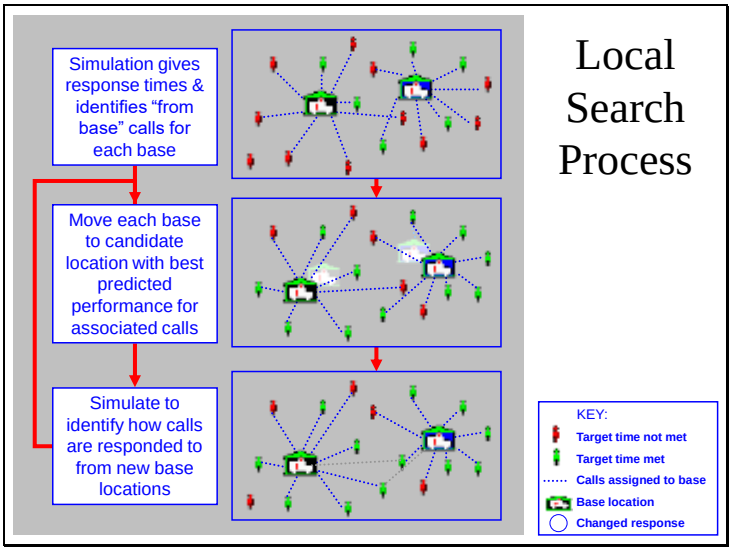
Simulation gives response times & identifies “from base” calls for each base
Move each base to candidate location with best predicted performance for associated calls
Simulate to identify how calls are responded to from new base locations
Local Search Process
KEY:
Target time not met
Target time met
Calls assigned to base
Base location
Changed response

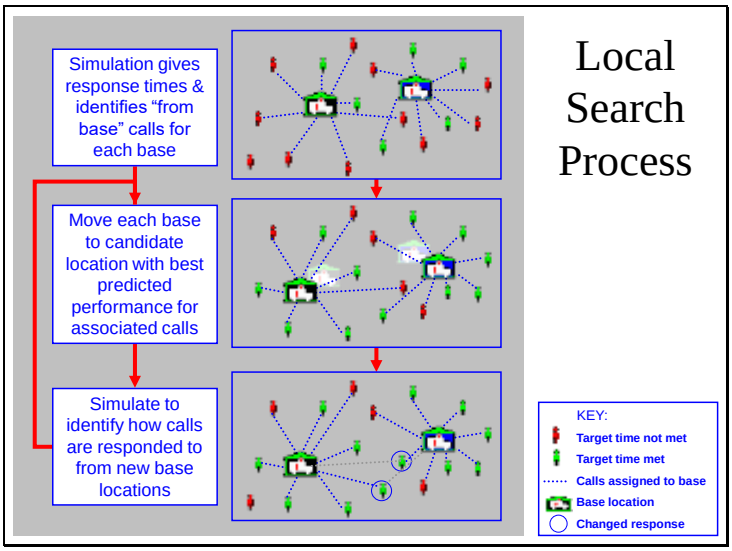


Simulation gives response times & identifies “from base” calls for each base
Move each base to candidate location with best predicted performance for associated calls
Simulate to identify how calls are responded to from new base locations
Local Search Process
KEY:
Target time not met
Target time met
Calls assigned to base
Base location
Changed response



Simulation gives response times & identifies “from base” calls for each base
Move each base to candidate location with best predicted performance for associated calls
Simulate to identify how calls are responded to from new base locations
Local Search Process
KEY:
Target time not met
Target time met
Calls assigned to base
Base location
Changed response

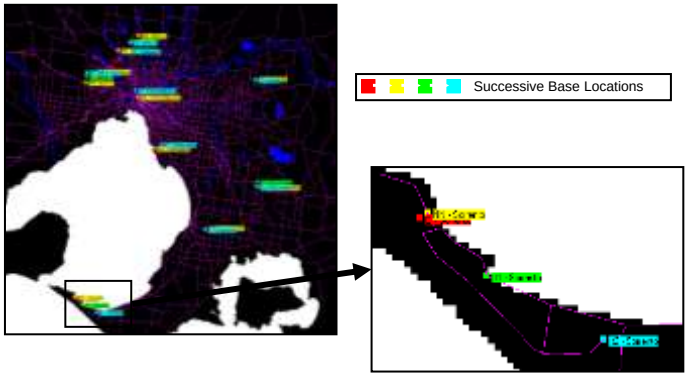


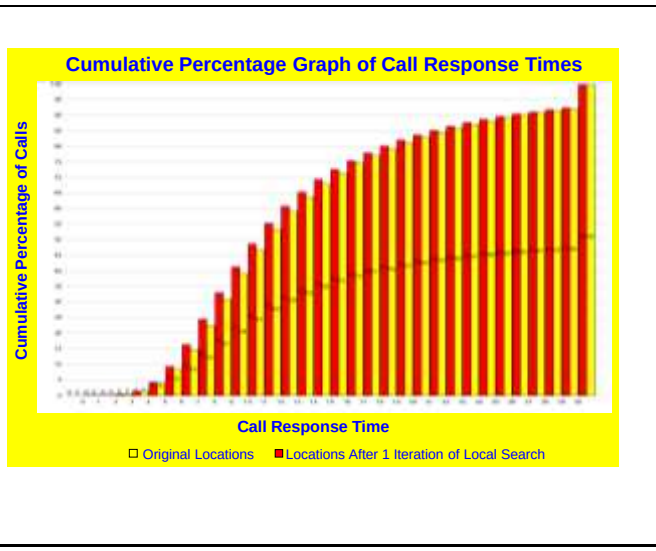


Local Search Results

- All call data was randomised so the results presented do not reflect the actual performance of the Melbourne Ambulance Service
- New base locations were tested using a second independent set of calls
  - Improvements not specific to data used in Local Search
- Improvements made in all response targets
  - 9.0% more priority code 1 calls reached within 8 minutes
  - 2.0% more priority code 1 calls reached within 13 minutes
  - 1.5% more priority code 2 calls reached within 25 minutes
  - 0.5% more priority code 3 calls reached within 60 minutes

New Base Locations





But Wait… There's More…

Source: http://www.igi-global.com/publish/call-for-papers/call-details/2306

Ant Colony Search Algorithm,
Artificial Bee Colony Algorithm,
Artificial Neural Networks,
Bee-Hive Algorithm,
Binary Differential Evolution Algorithm,
Biogeography-Based Optimization,
B-Snake Algorithm,
Collaborative Tabu Search and Decomposition Method,
Simulated Annealing Method,
Cuckoo Optimization Algorithm,
Cultural algorithms,
Differential Evolution,
Differential Harmony Search Algorithm,
Estimation of Distribution Algorithms,
Evolutionary programming Based Tabu Search Method,
Evolutionary Strategies and Evolutionary Programming Methods,
Extended Neighbourhood Search algorithm (ENSA),
Fast Snake Algorithm,
Fire-Fly Algorithm,
Fuzzy Logic Algorithm,
Generalized Reduced Gradient Method,
Genetic Algorithm,
Gravitational Search Method,
Greedy Snake Algorithm,
Guided Local Search,
Harmony Search Algorithms,
Honey Bees Mating Optimization Algorithm,
Hopfield Method,
Hopfield Neural Networks,
Iterated Local Search,
Lagrangian Firefly Algorithm,
Memetic Algorithm,
Meta-Heuristic Algorithm,
Particle Swarm Optimization,
Quantum-Inspired Binary PSO,
Scatter Search Methods,
Seeded Memetic Algorithm,
Self-Realized Differential Evolution Algorithm,
Shuffled Frog Leaping Algorithm,
Simulated Annealing Methods,
Simulated Annealing,
Single and Multivariable Constrained Methods,
Snake Curve Extraction Algorithm,
Stochastic Local Search,
Tabu-Search Methods,
Teaching- and Learning-Based Optimization,
Variable Neighborhood Search,
Waggle Dance Algorithm

…and even more…

Hosseini, E., Ghafoor, K.Z., Emrouznejad, A. *et al.* Novel metaheuristic based on multiverse theory for optimization problems in emerging systems. *Appl Intell* (2020). https://doi.org/10.1007/s10489-020-01920-z
https://link.springer.com/article/10.1007/s10489-020-01920-z#citeas
Published: 11 November 2020

Novel metaheuristic based on multiverse theory for optimization problems in emerging systems

Abstract: Finding an optimal solution for emerging cyber physical systems (CPS) for better efficiency and robustness is one of the major issues. Meta-heuristic is emerging as a promising field of study for solving various optimization problems applicable to different CPS systems. In this paper, we propose a new meta-heuristic algorithm based on Multiverse Theory, named MVA, that can solve NP-hard optimization problems such as non-linear and multi-level programming problems as well as applied optimization problems for CPS systems. MVA algorithm inspires the creation of the next population to be very close to the solution of initial population, which mimics the nature of parallel worlds in multiverse theory. Additionally, MVA distributes the solutions in the feasible region similarly to the nature of big bangs. To illustrate the effectiveness of the proposed algorithm, a set of test problems is implemented and measured in terms of feasibility, efficiency of their solutions and the number of iterations taken in finding the optimum solution. Numerical results obtained from extensive simulations have shown that the proposed algorithm outperforms the state-of-the-art approaches while solving the optimization problems with large feasible regions.

….

Meta-heuristics have main concepts, which have been simulated from treatment of animals, insects or natural events. The most important concept of ant colony is pheromone of ants, particle swarm optimization has been based on the global best, while main concept of genetic algorithm is combination, warming of egg in *laying chicken algorithm (LCA)* and explosion in big bang algorithm are the most important concepts of these algorithms. In this paper, we have inspired MVA algorithm from mainly from the existence of several worlds and big bangs.

Comparison of MVA and existing metaheuristic methods

F	MVA		MVO		GWO		PSO		GA	
	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
F1	1.0589	0.4698	2.08583	0.648651	2319.19	1237.109	3.552364	2.85373	27,187.58	2745.82
F2	5.3647	3.4279	15.92479	44.7459	14.43166	5.923015	8.716272	4.929157	68.6618	6.062311
F3	123.4689	85.7954	453.2002	177.0973	7278.133	2143.116	2380.963	1183.351	48,530.91	8249.75
F4	1.9361	1.3689	3.123005	1.582907	13.09729	11.3469	21.5169	6.71628	62.99326	2.535643
F5	836.279	756.148	1272.13	1479.477	3425,462	3304,309	1132.486	1357.967	65,361,620	29,714,021
F6	1.5824	0.6843	2.29495	0.630813	5009.442	3028.875	86.62074	147.3067	49,574.1	8545.149
F7	0.01478	0.0115	0.051991	0.029606	0.408082	0.119544	0.577434	0.318544	18.72524	4.935256

Local Search with Branch and Bound

Based on http://www.crt.umontreal.ca/cpaior/article-danna-36.pdf

For a summary of branch and bound and integer programming, see Appendix 1.

Standard Branch and Bound algorithms are very good at finding lower bounds, and (eventually) proving that we have found the best solution. However, if the LP and IP objective function values are very different, then it can be very hard to get an integer solution from the LP relaxation, and so branch and bound can perform badly.

Fix and Dive Heuristic (a Guided Dive)

When a problem is very hard for Branch and Bound, we can give up on proving optimality, and instead use a modified Branch and Bound as a 'fix and dive' heuristic. In this approach, we solve an LP, fix all the variables that are naturally integer (ie add constraints forcing these variables to stay at their current values), and then add a constraint to force some fractional variable to become integer. The new LP is solved, and we then repeat this process until all the variables are integer. (If the problem becomes infeasible, then we have to back track.)

Example

Consider some problem with 5 binary variables. We solve the LP and get

$x_1=1, \quad x_2=0.1, x_3=0.6, x_4=0.2, x_5=1, \quad x_6=0.7$

We will branch by rounding the 'least fractional' variable to its closest integer (0 or 1)

Add 'fixing' constraints: and branching constraint

Solve the LP and get

$x_1=1, \quad x_2=0, \quad x_3=0.9, x_4=0.3, x_5=1, \quad x_6=1$

Add fixing constraints: and branch constraints

Solve the LP and get

$x_1=1, \quad x_2=0, \quad x_3=1, \quad x_4=0.6, x_5=1, \quad x_6=1$

Add 'fixing' constraints: (none) and branching constraint

Solve the (trivial!) LP and get

$x_1=1, \quad x_2=0, \quad x_3=1, \quad x_4=1, \quad x_5=1, \quad x_6=1$

Stop as all variables are integer.

If we know something about the problem, we can 'guide' the dive by using this information to help choose which variables to branch on next, and what values to force them to.

Heuristic Helpers for Branch and Bound

If finding integer solutions is hard, then we can often give the branch and bound a helping hand by using a heuristic to construct good integer solutions. (Remember that good integer solutions help the branch and bound process by providing a tight bound, thus allowing us to skip searching poor sections of the branch and bound tree.) The

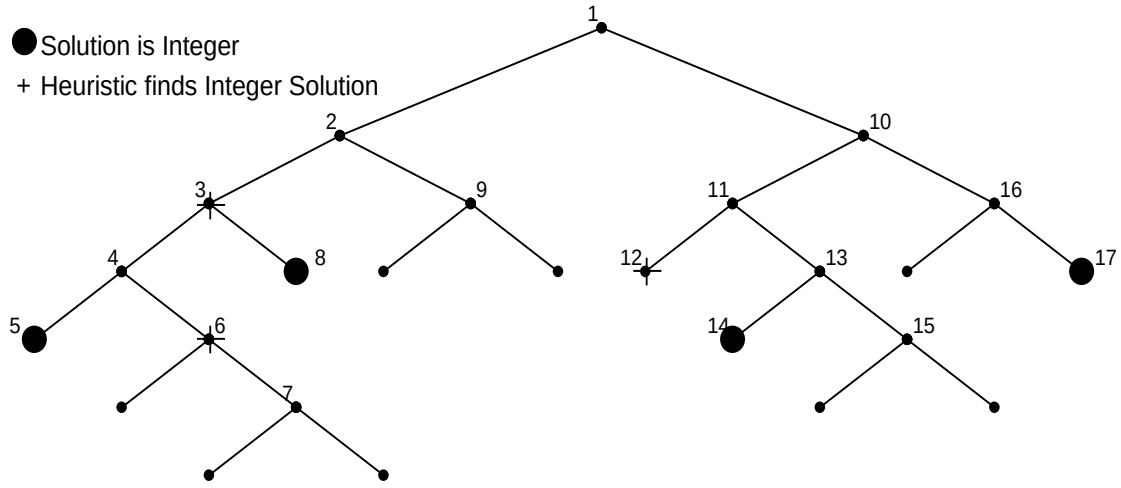
helper heuristic can be called at regular intervals (say after every 10'th node is added to the tree). The heuristic will typically take some LP solution and use this solution in some way to help construct an integer solution. For example, one such heuristic we could use is ....

The CPLEX branch and bound solver makes good use of helper heuristics; the nodes marked '+' in the output below show where a new improved integer solution was found using one of Cplex's heuristics.

	Node	Nodes Left	Objective	IInf	Best Integer	Best Node	Cuts/ ItCnt	Gap
	0	0	346.0000	536		346.0000	551	
*	0+	0		0	6262.0000	346.0000	551	94.47%
			560.0000	490	6262.0000	Cuts: 700	1131	91.06%
			560.0000	592	6262.0000	Cuts: 688	1561	91.06%
*	260+	256		0	3780.0000	560.0000	3566	85.19%
*	300+	296		0	2992.0000	560.0000	3703	81.28%
*	300+	296		0	2626.0000	560.0000	3703	78.67%
*	393	368		0	2590.0000	560.0000	4405	78.38%
*	680+	628		0	2576.0000	560.0000	6928	78.26%
*	690+	638		0	2538.0000	560.0000	6946	77.94%
*	710+	656		0	2478.0000	560.0000	7011	77.40%
*	720+	658		0	2448.0000	560.0000	7027	77.12%
*	720+	645		0	2402.0000	560.0000	7027	76.69%
*	730+	639		0	2360.0000	560.0000	7070	76.27%
*	820+	726		0	2340.0000	560.0000	8134	76.07%

...

A possible branch and bound tree with a heuristic helper might be as follows:



Using Incumbent Solutions

If we have found some good solution during the branch and bound, it would be nice to use this to help create better solutions within the branch and bound process. This introduces the idea of local search to branch and bound. So called "local branching" is one way to do this.

Local Branching

Matteo Fischetti and Andrea Lodi. Local branching. Programming Conference in honor of Egon Balas, 2002.

Consider some problem with binary (0/1) variables x_1, x_2, ..., x_n, and assume we have some current best-so-far solution, known as an 'incumbent' solution, x_1^*, x_2^*, ..., x_n^*. Local branching creates a MIP (mixed integer programming) model that describes a neighborhood of that solution. It does so by adding a "local branching constraint" to the model, stating that at most r binary variables will be allowed to differ in the incumbent and in the subsequent solutions:

As an example, if we have n=5, r=2, and some current incumbent integer solution

$x_1^*=1$, $x_2^*=0$, $x_3^*=0$, $x_4^*=1$,$x_5^*=1$

then we would add the local branching constraint:

.....

The new problem with this constraint is known as a **sub-MIP**, and will hopefully give a new better integer solution when solved.

This sub-MIP is then solved in the normal way. In other words, the MIP solver itself is used to explore a small neighbourhood of the current incumbent. This exploration is truncated with a time or node limit.

Local branching is based on the same assumptions as local search, namely that there exist other good solutions in the neighbourhood of a good solution.

The key points in this scheme are the following:

- The search is concentrated on a reduced part of the search space defined by a secondary MIP model which we call the sub-MIP.
- This sub-space is explored with a MIP solver (i.e., using a branch-and-bound algorithm) to find the **best** neighbour (i.e. we are using steepest descent). If the sub-MIP problem is too difficult to fully explore using branch and bound, then we may not find the best neighbour, but instead just a good neighbour.

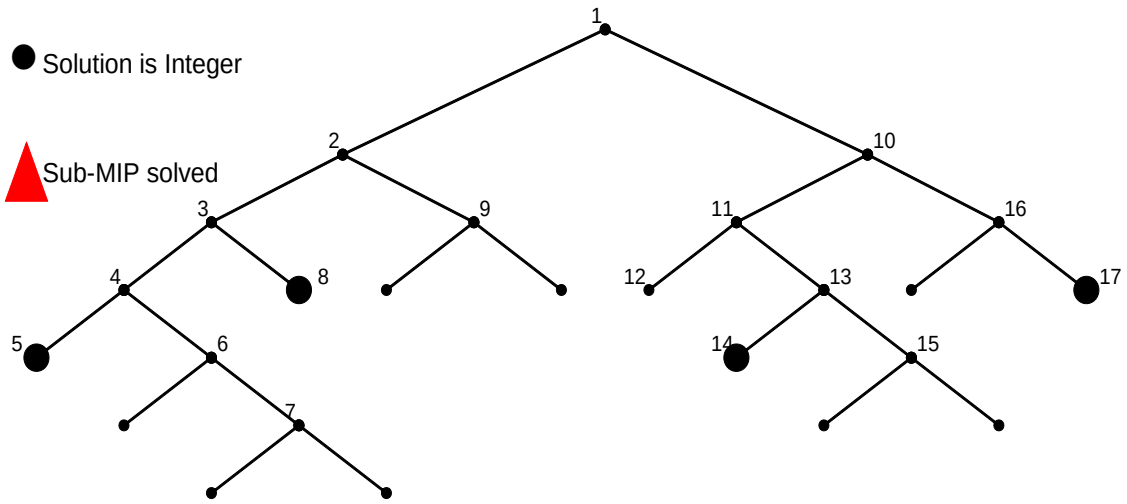
Local branching is a good approach to take if there is a large gap between the objective function of the best integer solution and the objective function of the LP relaxation. Adding the local branching constraint will (hopefully) create a sub-MIP model which is easier to solve because the sub-MIP has an LP objective that is closer to the true IP objective than to the LP relaxation objective function of the original problem.

There are several approaches we can take to making use of local branching.

Local Branching to define a Neighbourhood for Neighbourhood Search: The first is to perform a standard local search where, given some current solution, we define and solve the associated sub-MIP (using a standard branch and bound solver) to find the best neighbouring solution (or at least a better neighbouring solution). This solution then becomes our new current solution, and the process repeats until a local optimum is found.

Local Branching as a Heuristic within Branch and Bound: Another approach is to perform a normal branch and bound, but trigger a sub-MIP search whenever we find a

new integer solution, as shown in the following diagram. This second approach is available in the standard Branch and Bound solver Cplex; the figure below shows this in action. We hope to find new better integer solutions in the Sub-MIP solves.



Solving sub-MIPs for each new incumbent

Math-Heuristics = Matheuristic = MIP Heuristics

The above approaches embed local search as part of the Branch and Bound process being used to solve a Mixed Integer Programme (MIP). An alternative approach is to incorporate an integer programming model as part of a heuristic procedure, perhaps for finding a good neighbour, for example. This blended approach gives what is termed a “Math-Heuristic” or “MIP Heuristic”. Because modern MIP solvers are so good, you can often create a very powerful heuristic in this way. For example, crossover in a GA may solve a MIP to get the best possible child.

Relaxation Induced Neighborhood Search (RINS)

Adapted from: Emilie Danna, Edward Rothberg, and Claude Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. Technical report, ILOG, 2003.

A neighbourhood search can be thought of as defining a sub-problem by adding an extra constraint to ensure we explore just those solutions that are neighbours to some current solution. Another way of defining neighbouring solutions is to fix a subset of the variables to the values they take in some incumbent (best so far) solution. In other words, some decisions (that is some variable assignments) form a partial solution that is perceived as satisfactory; if these decisions are not reconsidered, then we have defined a subset of problems that form a neighbourhood. It may be a useful search strategy to spend some time searching this neighbourhood for good solutions, i.e. to focus the search for some limited time on the extension of this partial solution.

The question is how to choose the variables to fix. In a MIP, the incumbent and the continuous relaxation each achieve one and fail to achieve one of the following conflicting objectives: integrality and optimality. Relaxation Induced Neighborhood Search (RINS) is based on the intuition that decisions (values of variables) that are common to the incumbent (best integer solution) and the current LP relaxation form a partial solution that could be extended (relatively) easily to give an integer solution that is (near-) optimal.

Example:
As an example, if we have some current incumbent integer solution

$x_1^*=1, \quad x_2^*=0, \quad x_3^*=0, \quad x_4^*=1, \quad x_5^*=1$

and some current LP solution

$x_1=1, \quad x_2=1, \quad x_3=0.5, \quad x_4=0.2, \quad x_5=1$

then searching for an integer solution with…

might give us a good new integer solution.

A typical RINS algorithm proceeds as follows. At some (fractional or integer) node of the global branch-and-bound tree, the following steps are performed:

1. Fix the integer variables that have the same values in the incumbent and in the current continuous relaxation;
2. Solve a sub-MIP on the remaining variables. Exploration of the sub-MIP is typically truncated through a time or node limit.

Example 2:
As an example, if we have some current incumbent integer solution

$x_1^*=1, \quad x_2^*=0, \quad x_3^*=0, \quad x_4^*=1, \quad x_5^*=1$

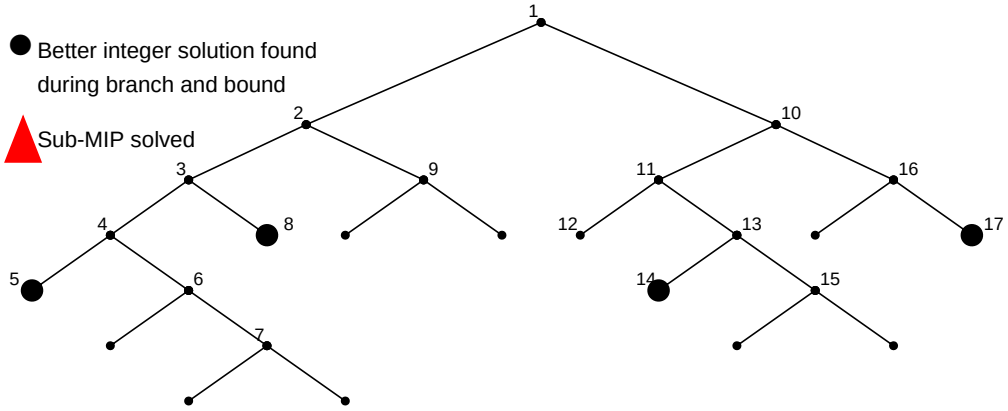
and some current LP solution

$x_1=0.6, \quad x_2=0, \quad x_3=0.5, \quad x_4=1, \quad x_5=0$

then our sub-MIP is the original MIP plus the following additional RINS constraints:

The technical report notes that: "... unlike local branching, which requires a new incumbent to explore a new neighbourhood, RINS can be called at each node of the global branch-and-bound tree. However, experience suggests that it is preferable to apply RINS less frequently to achieve more diversity across consecutive sub-MIPs."

The branch and bound tree below shows how a RINS implementation might work. (The number beside each node shows the order in which the nodes were explored.) Here we run RINS for every second node (starting once we find a first integer solution). We hope to find new better integer solutions in the Sub-MIP solves.



"A major strength of RINS is that it explores a neighbourhood of both the incumbent and the continuous relaxation. The incumbent and the continuous relaxation thus play perfectly symmetrical roles: if one is of poor quality, the other will automatically help to define a fruitful neighbourhood, and vice versa."

The following plot shows the results for some problems of intermediate difficulty. A small Gap indicates good solutions. Using RINS is clearly good for these problems!

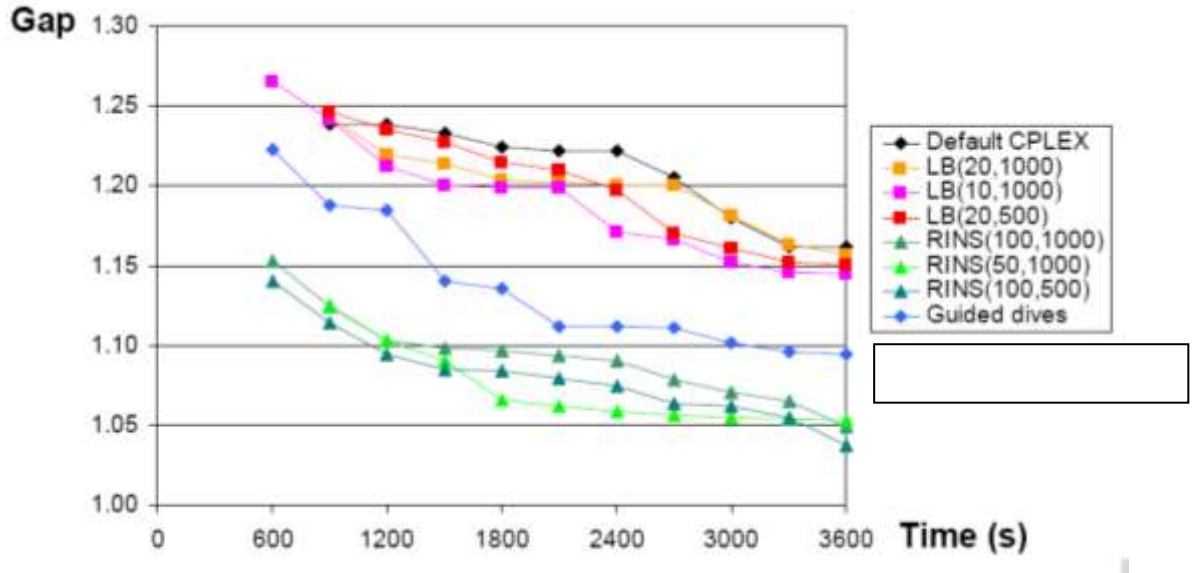


Figure from http://www.mpi-sb.mpg.de/conferences/adfocs-03/Slides/Rothberg_3.pdf The RINS runs are the bottom 3 lines. The top 4 lines are default CPLEX and Local Branching. The gap is the relative difference between the upper and lower bounds averaged over the test problems; a smaller gap is better.

Appendix 1: Integer Programming Overview

We often represent (i.e. formulate) optimization problems as integer programmes (IP's), such as:

$$\begin{array}{lll} \text{IP:} \quad \min & -3x_1 - 3x_2 - x_3 & \\ \text{s.t.} & 2x_1 + 4x_2 + 2x_3 + x_4 = 5, & \\ & x_1, x_2, x_3, x_4 \in \{0,1\} & \end{array}$$

Solving this involves finding values for x_1, x_2, x_3 and x_4 that are binary (in this case, or integer more generally), that satisfy our 'constraint' $2x_1 + 4x_2 + 2x_3 + x_4 = 5$ and minimise our objective function $-3x_1 - 3x_2 - x_3$.

Note that if we allow some variables to be fractional, then we have a 'mixed integer programme' (a MIP).

An Integer Program (IP) is hard to solve. But, we can instead `relax' the binary (or integer) requirement, and get an easier 'linear program' problem where fractional values are allowed:

$$\begin{array}{lll} \text{LP:} \quad \min & -3x_1 - 3x_2 - x_3 & \\ \text{s.t.} & 2x_1 + 4x_2 + 2x_3 + x_4 = 5, & \\ & x_1, x_2, x_3, x_4 \in [0,1] & \end{array}$$

Note: $x_1, x_2, x_3, x_4 \in [0,1]$ means $0 \le x_1 \le 1$, $0 \le x_2 \le 1$, $0 \le x_3 \le 1$, $0 \le x_4 \le 1$.

A linear programme (LP) is easy to solve using the Simplex Algorithm:

$$\begin{array}{lll} \text{LP:} \quad \min & -3x_1 - 3x_2 - x_3 & \\ \text{s.t.} & 2x_1 + 4x_2 + 2x_3 + x_4 = 5, & \\ & x_1, x_2, x_3, x_4 \in [0,1] & \end{array}$$

↓

Simplex Algorithm

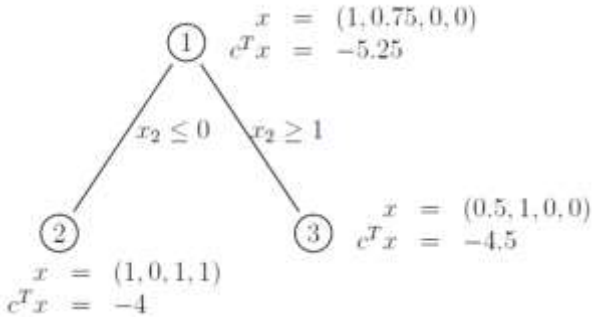
↓

$$x^{(1)} = \begin{bmatrix} 1 & \frac{3}{4} & 0 & 0 \end{bmatrix}$$

We don't like this solution as x_2 is not 0 or 1. We don't know which it should be, and so we must explore both options can creating two new linear programmes with an extra constraint on x_2, either $x_2=0$ (or, equivalently $x_2<=0$), or $x_2=1$ (or equivalently $x_2>=1$)

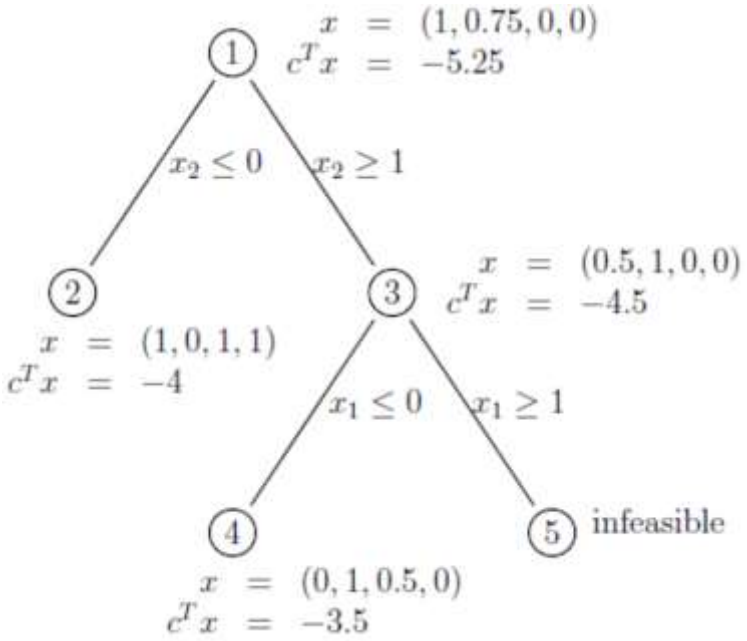
LP: min $-3x_1 - 3x_2 - x_3$ s.t. $2x_1 + 4x_2 + 2x_3 + x_4 = 5$, $x_1, x_2, x_3, x_4 \in [0,1]$ $x_2=0$ (or $x_2<=0$) ↓ Simplex Algorithm ↓ $x^{(2)} = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}$ Objective $c^Tx = -4$	LP: min $-3x_1 - 3x_2 - x_3$ s.t. $2x_1 + 4x_2 + 2x_3 + x_4 = 5$, $x_1, x_2, x_3, x_4 \in [0,1]$ $x_2=1$ (or $x_2>=1$) ↓ Simplex Algorithm ↓ $x^{(3)} = \begin{bmatrix} \frac{1}{2} & 1 & 0 & 0 \end{bmatrix}$

We show this as tree:



So far, we have found one integer solution, being node (2); this best integer solution so far is called the "incumbent solution". Note that we have not yet proven it to be the best solution, and so we have to explore more solutions.

Node 3 is not integer (it has fractional values), and so we do this again:



Note that node (5) is infeasible, meaning there is no choice of x_1, x_2, x_3, x_4 that simultaneously satisifies all the constraints.

We keep following this process, updating the best solution as we find better solutions, until we can prove we have the best solution. We use a technique called 'bounding' (which operates like A*) to avoid exploring nodes that cannot give a solution that is better than the best found so far. However, this process can still give very large trees.

For more details, see https://en.wikipedia.org/wiki/Integer_programming