It's search Jim,
but not as we know it

tsp

Traveling Salesman Problem - Windows Internet Explorer

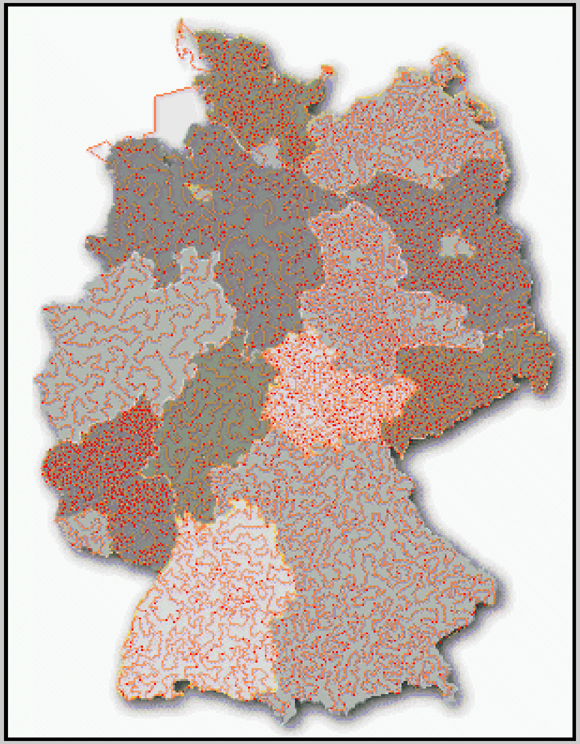
http://www.tsp.gatech.edu/

File Edit View Favorites Tools Help

Traveling Salesman Problem

The Problem
History
Applications
Methods
Optimal Tours
Concorde
Gallery
Test Data
Games
TSP Book

Search Site



The traveling salesman problem (TSP) asks for the shortest route to visit a collection of cities and return to the starting point. Despite an intensive study by mathematicians, computer scientists, operations researchers, and others, over the past 50 years, it remains an open question whether or not an efficient general solution method exists.

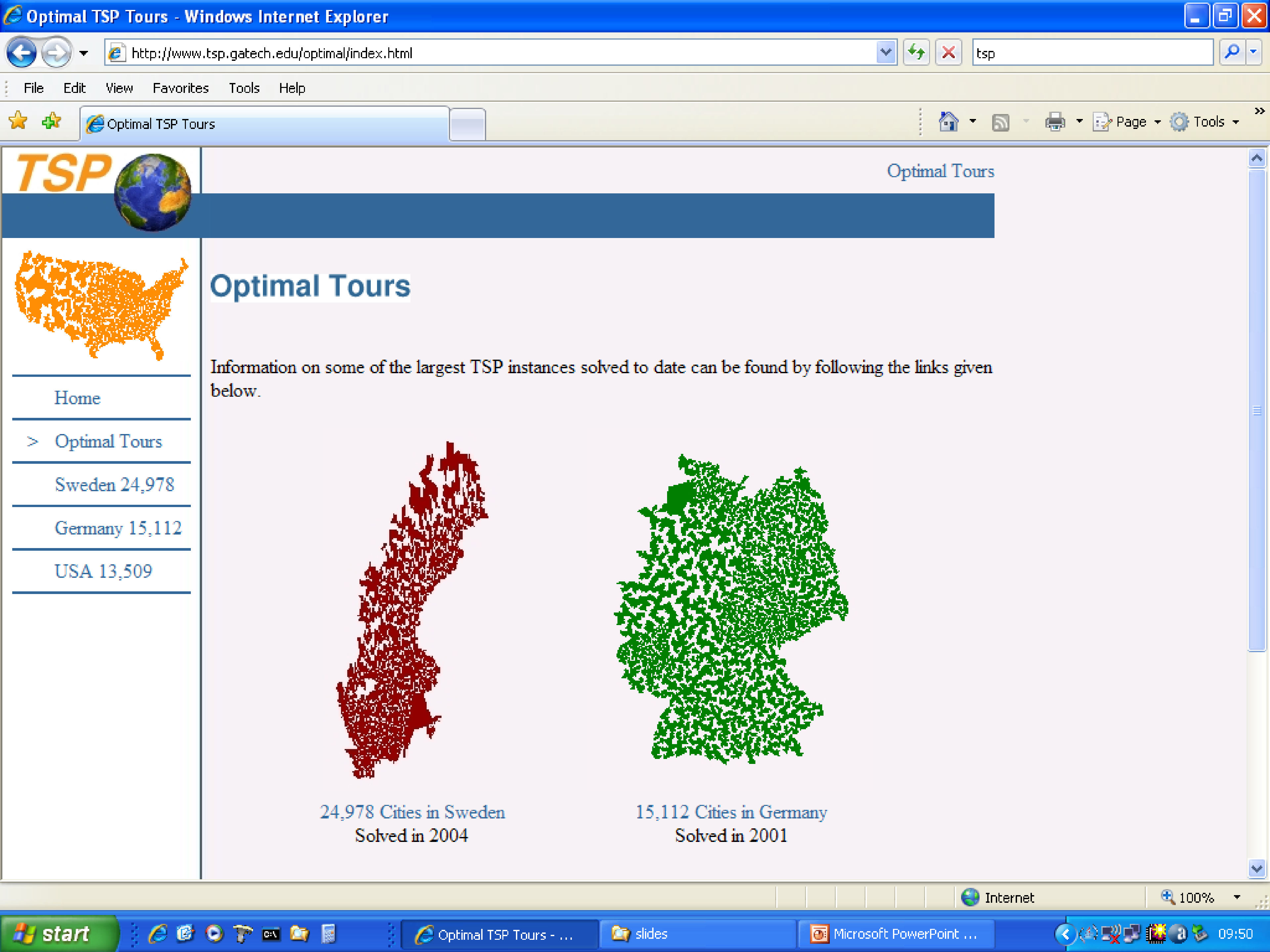
These pages are devoted to the history and applications of the TSP and to ongoing research towards the solution of large-scale problems. The work described here is supported by Office of Naval Research (N00014-03-1-0040) and National Science Foundation (DMI-0245609) grants, and by the School of Industrial and Systems Engineeringat Georgia Tech.

TSP Book

The Traveling Salesman Problem:

Lower Bounds

Description of the techniques we


Optimal TSP Tours - Windows Internet Explorer
http://www.tsp.gatech.edu/optimal/index.html
tsp
File Edit View Favorites Tools Help
Optimal TSP Tours
Page Tools
TSP
Optimal Tours
Home
> Optimal Tours
Sweden 24,978
Germany 15,112
USA 13,509
Optimal Tours
Information on some of the largest TSP instances solved to date can be found by following the links given below.
24,978 Cities in Sweden
Solved in 2004
15,112 Cities in Germany
Solved in 2001
Internet
100%
start
Optimal TSP Tours - ...
slides
Microsoft PowerPoint ...
09:50

Germany Home - Windows Internet Explorer

http://www.tsp.gatech.edu/d15sol/index.html

15,112

Solution of a 15,112-city Traveling Salesman Problem

On April 20th, 2001, David Applegate, Robert Bixby, Vašek Chvátal, and William Cook announced the solution of a traveling salesman problem through 15,112 cities in Germany. This was the largest TSPLIB instance that had been solved at the time, exceeding the 13,509-city tour through the United States that was solved in 1998.

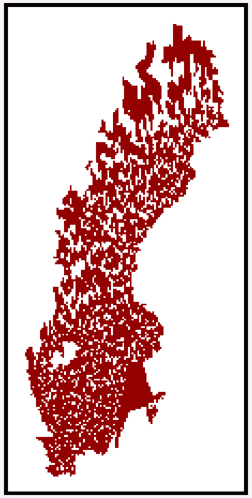
The optimal tour has length 1,573,084 in the units used in TSPLIB; this translates to a trip of about 66,000 kilometers through Germany. Pictures of the optimal tour can be found at Optimal Tour .

The computation was carried out on a network of 110 processors located at Rice University and at Princeton University. The total computer time used in the computation was 22.6 years, scaled to a Compaq EV6 Alpha processor running at 500 MHz. Details can be found in the Computation section.

In German Tours we display the 15,112-city tour together with other tours through Germany that have played a role in the history of the TSP.

Home
Optimal Tours
> Sweden Home
24,978 Cities
Sweden Tour
Finding the Tour
Computation
Data Sets

Optimal Tour of Sweden

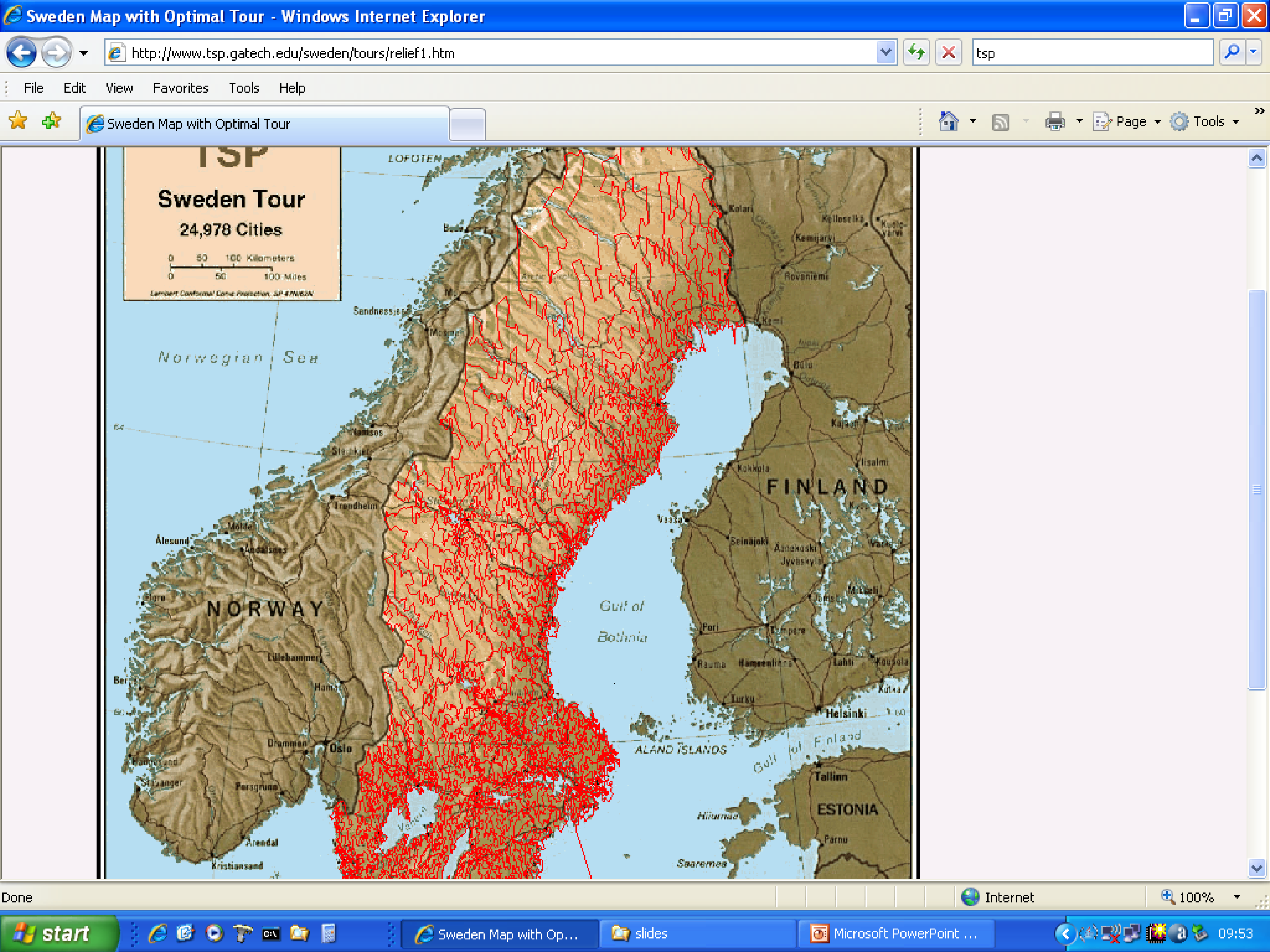


In May 2004, the traveling salesman problem of visiting all 24,978 cities in Sweden was solved: a tour of length 855,597 TSPLIB units (approximately 72,500 kilometers) was found and it was proven that no shorter tour exists. This is currently the largest solved TSP instance, surpassing the previous record of 15,112 cities through Germany set in April 2001.

- The 24,978 cities
- Pictures of the Sweden Tour
- How was the tour found?
- Details of the computation
- Data sets for Sweden TSP and tour

Research Team

- David Applegate, AT&T Labs - Research
- Robert Bixby, ILOG and Rice University
- Vašek Chvátal, Rutgers University
- William Cook, Georgia Tech


Sweden Map with Optimal Tour - Windows Internet Explorer
http://www.tsp.gatech.edu/sweden/tours/relief1.htm
File Edit View Favorites Tools Help
Sweden Map with Optimal Tour
TSP
Sweden Tour
24,978 Cities
0 50 100 Kilometers
0 50 100 Miles
Norwegian Sea
NORWAY
FINLAND
ESTONIA
Gulf of Bothnia
ALAND ISLANDS
Gulf of Finland
LOFOTEN
Trondheim
Ålesund
Lillehammer
Oslo
Drammen
Arendal
Kristiansand
Kemi
Vaasa
Pori
Turku
Helsinki
Tallinn
Done
Internet
100%
start
slides
Microsoft PowerPoint ...
09:53

But first, an example

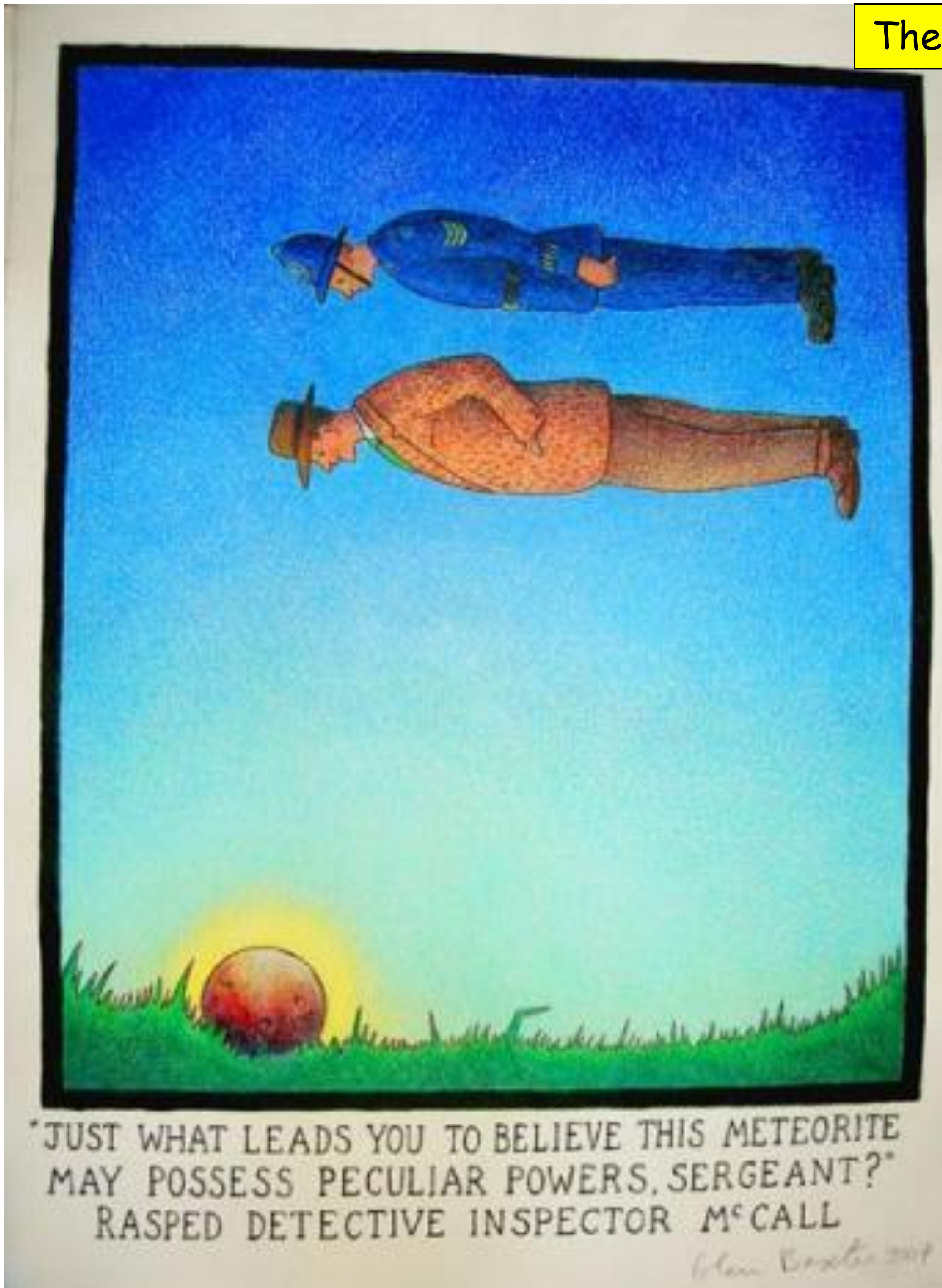
TSP
- given n cities with x/y coordinates
- select any city as a starting and ending point
- arrange the n-1 cities into a tour of minimum cost

Representation
- a permutation of the n-1 cities

A move operator
- swap two positions (let's say)
- 2-opt?
    - Take a sub-tour and reverse it
- how big is the neighbourhood of a state?
- how big is the state space?
- What dimensional space are we moving through?

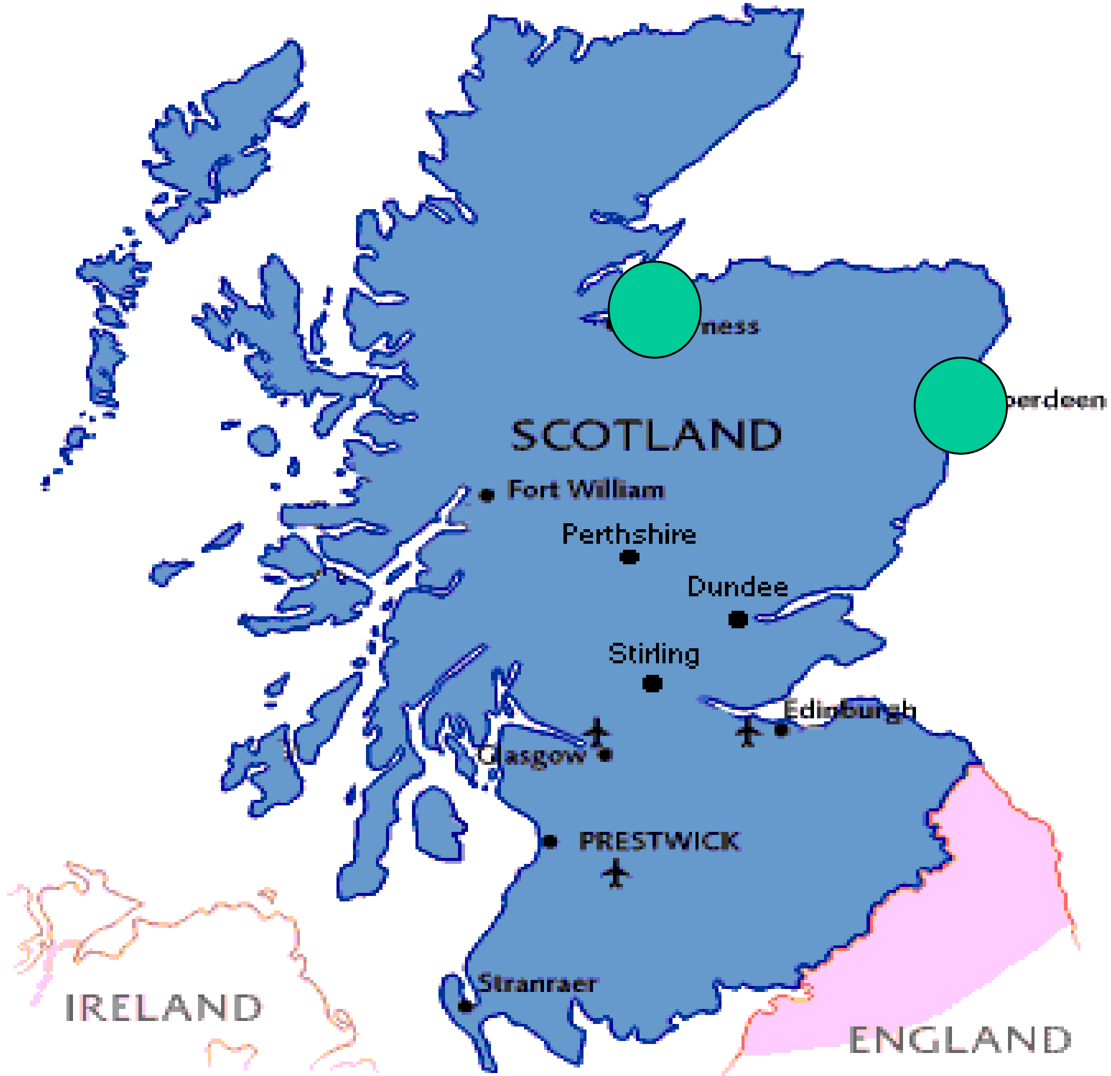
Evaluation Function
- cost/distance of the tour

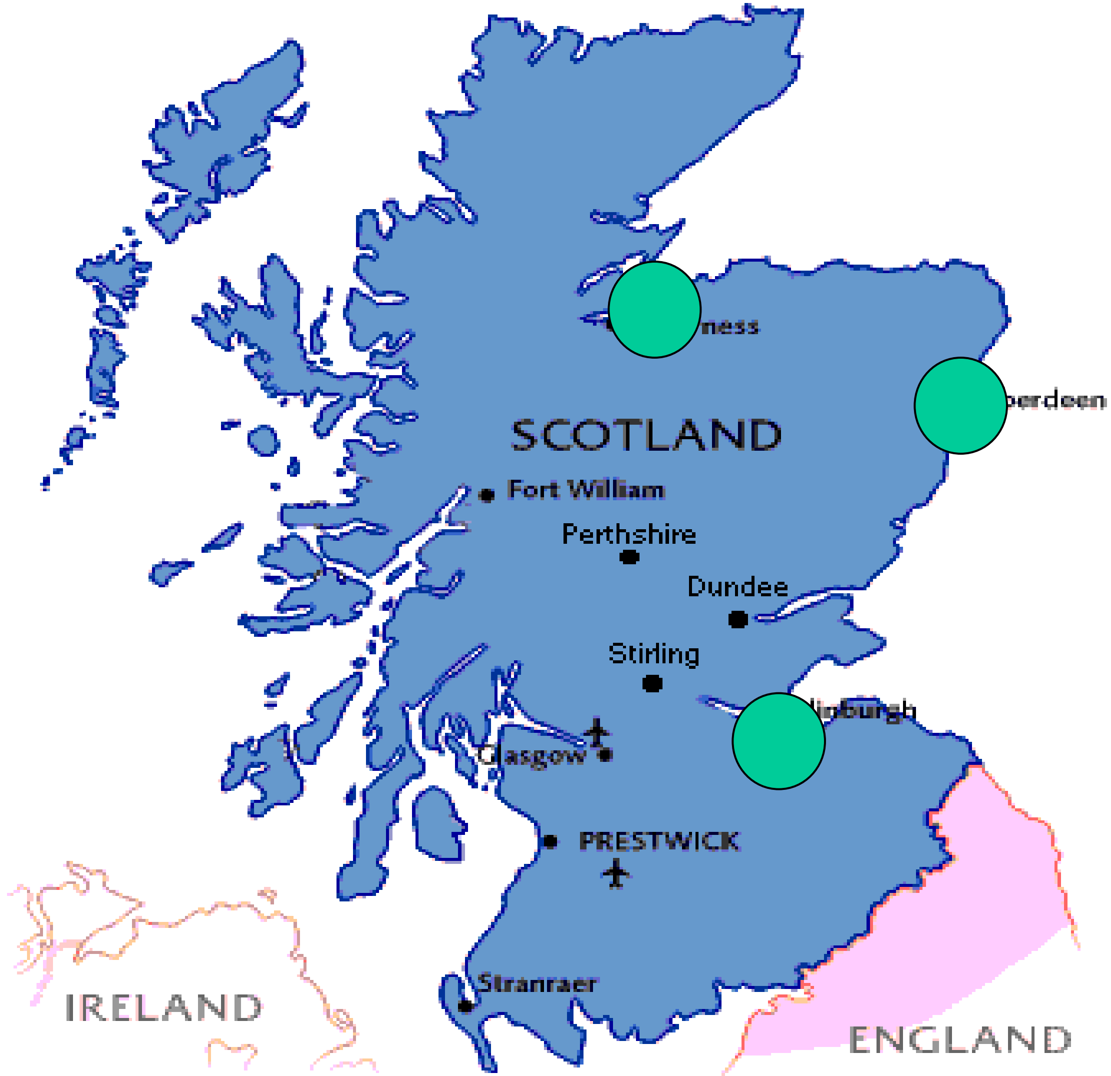
The dumbest possible algorithm

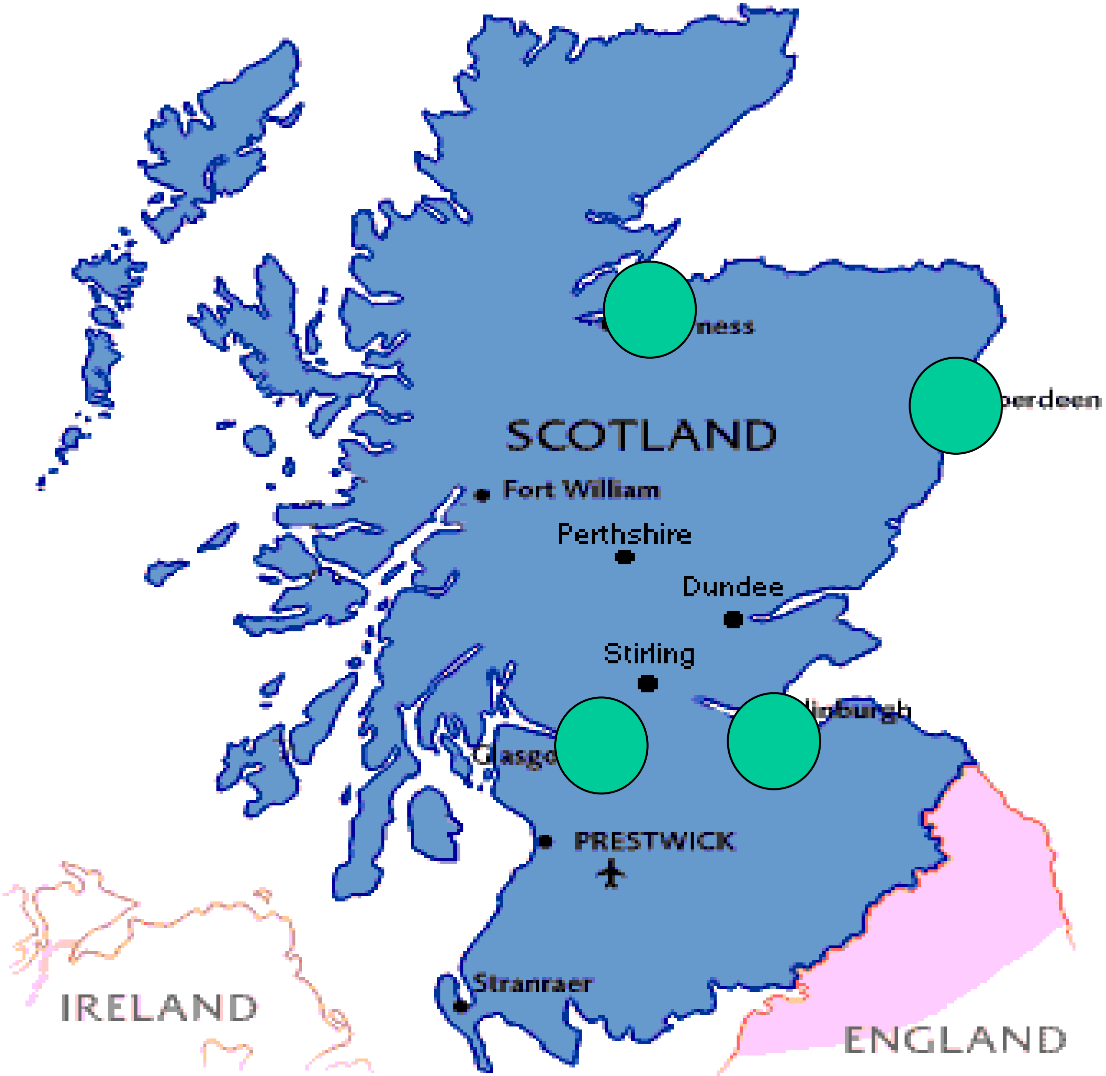


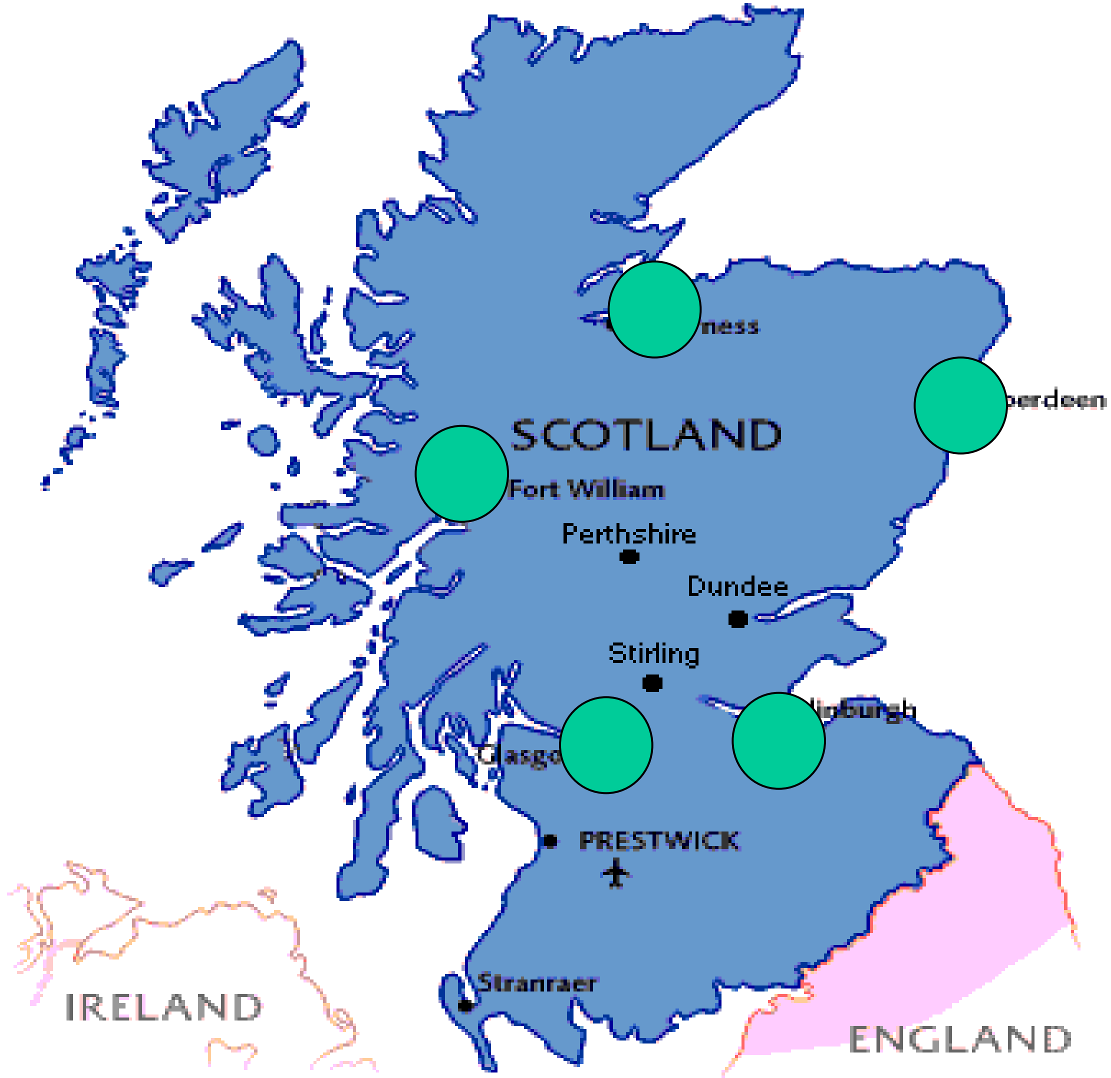

SCOTLAND
Inverness
Aberdeen
Fort William
Perthshire
Dundee
Stirling
Edinburgh
Glasgow
PRESTWICK
Stranraer
IRELAND
ENGLAND


ness
Aberdeen
SCOTLAND
Fort William
Perthshire
Dundee
Stirling
Edinburgh
Glasgow
PRESTWICK
Stranraer
IRELAND
ENGLAND

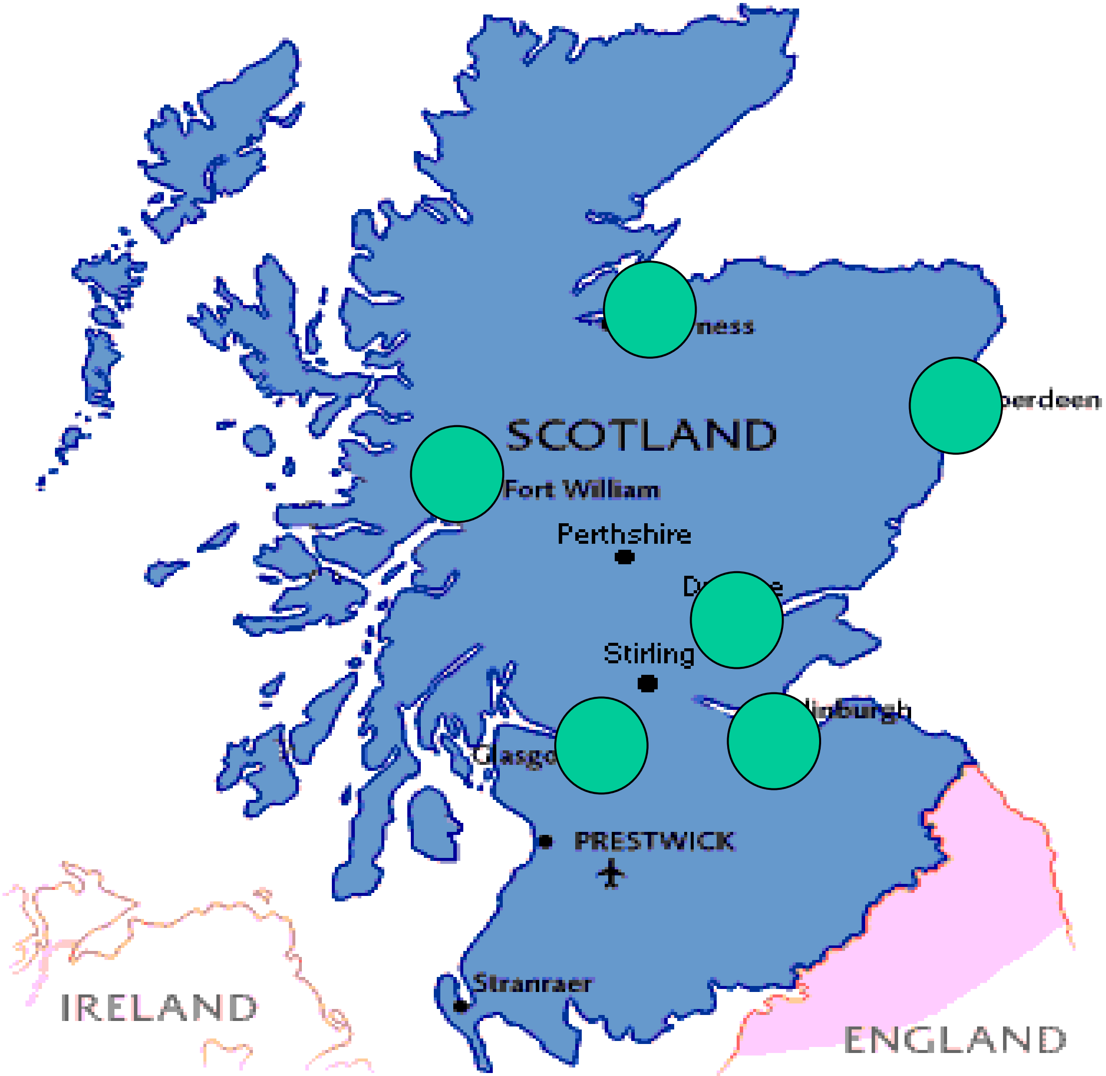

SCOTLAND
Fort William
Perthshire
Dundee
Stirling
Edinburgh
Glasgow
PRESTWICK
Stranraer
IRELAND
ENGLAND

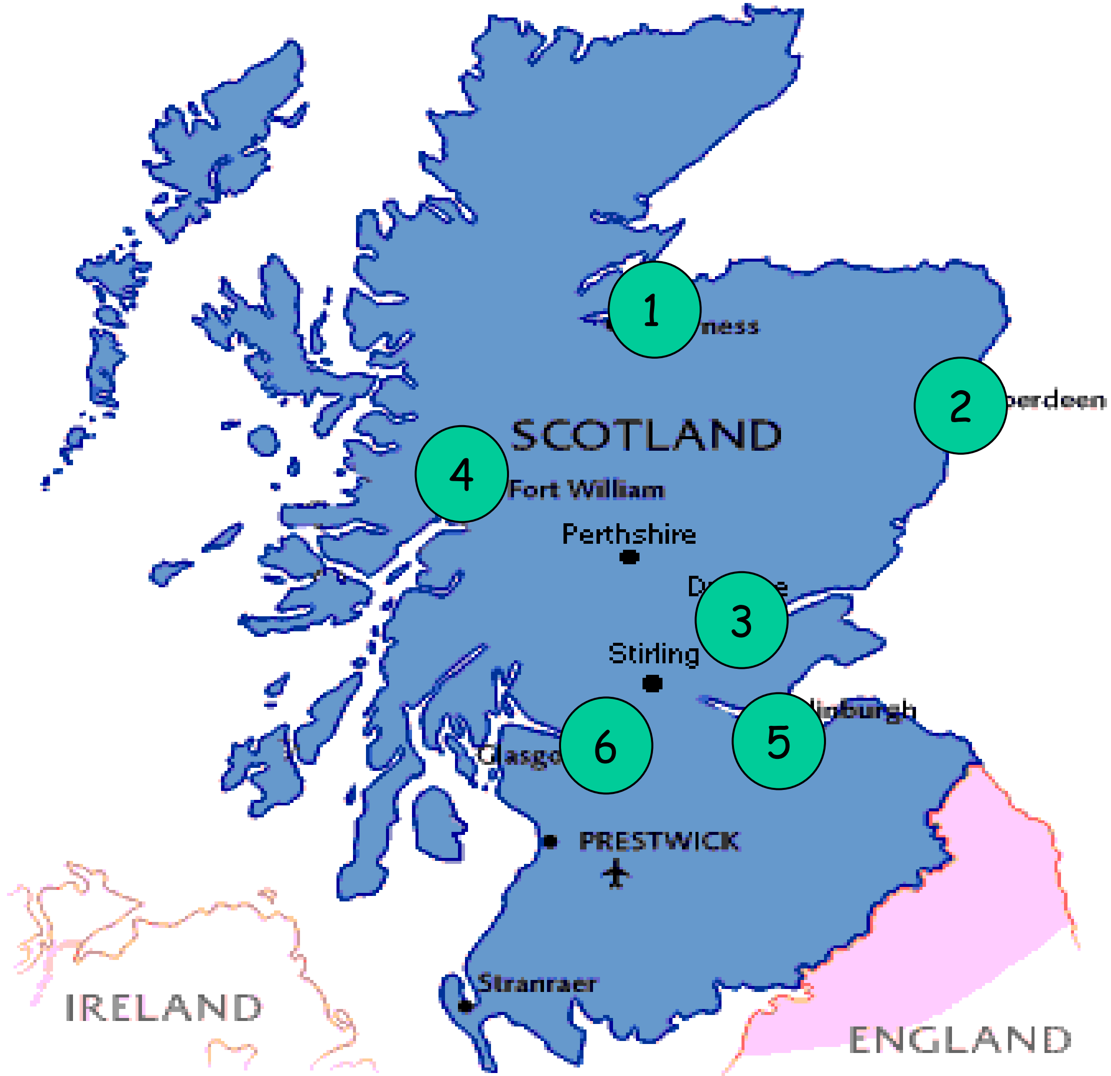

SCOTLAND
Fort William
Perthshire
Dundee
Stirling
Glasgow
PRESTWICK
Stranraer
IRELAND
ENGLAND

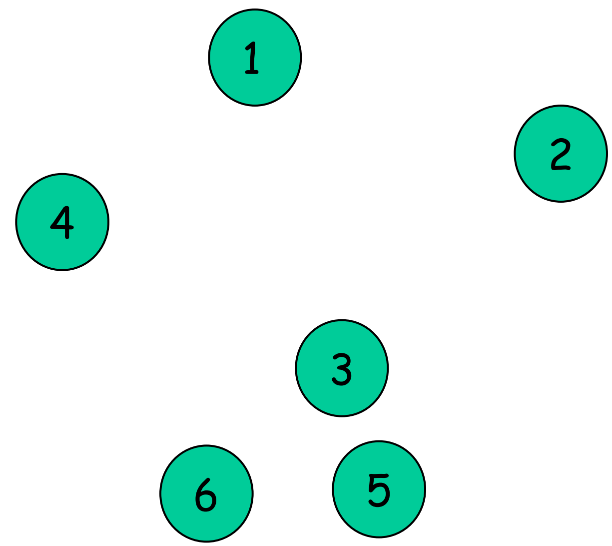

ness
perdeen
SCOTLAND
Fort William
Perthshire
Dundee
Stirling
inburgh
Glasgo
PRESTWICK
Stranraer
IRELAND
ENGLAND



ness
erdeen
SCOTLAND
Fort William
Perthshire
Dundee
Stirling
Glasgo
inburgh
PRESTWICK
Stranraer
IRELAND
ENGLAND

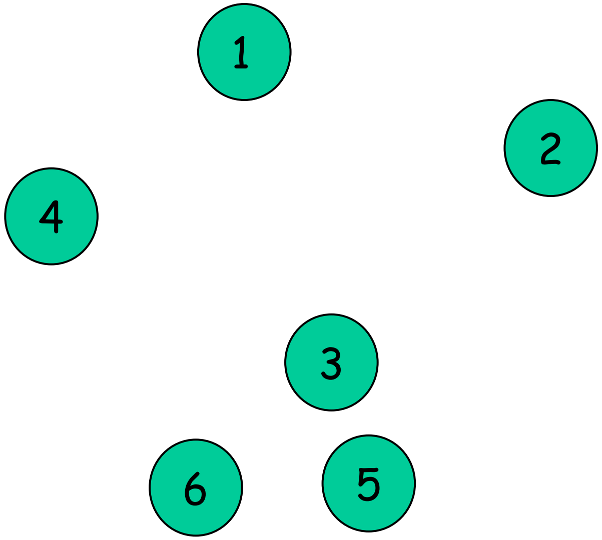

SCOTLAND
Fort William
Perthshire
Stirling
PRESTWICK
Stranraer
IRELAND
ENGLAND


1
2
SCOTLAND
4
Fort William
Perthshire
3
Stirling
6
5
PRESTWICK
Stranraer
IRELAND
ENGLAND


1
2
4
3
6
5

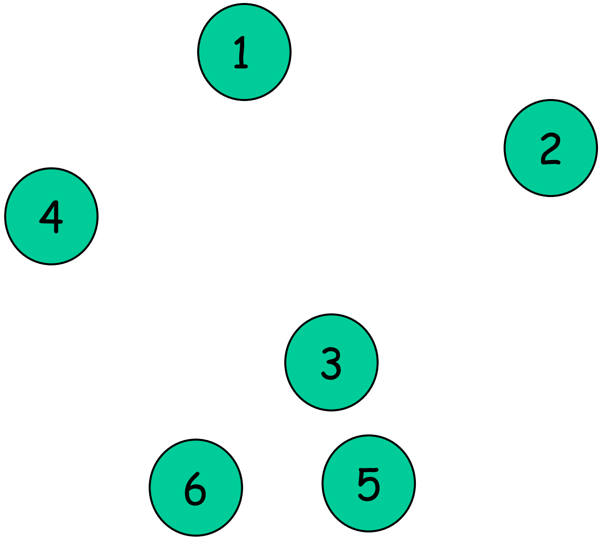
distance/cost table

	1	2	3	4	5	6
1	-	112	137	68	156	168
2		-	72	155	166	145
3			-	126	63	80
4				-	146	108
5					-	51
6						-



distance/cost table

	1	2	3	4	5	6
1	-					
2		-				
3			-			
4				-		
5					-	
6						-

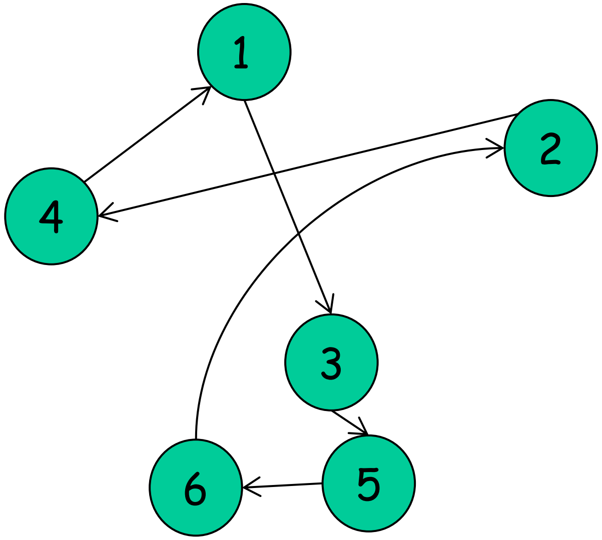


Permutation is a ***tour*** where we assume we start and end at 1st city in permutation

distance/cost table

	1	2	3	4	5	6
1	-					
2		-				
3			-			
4				-		
5					-	
6						-

tour: 1 3 5 6 2 4

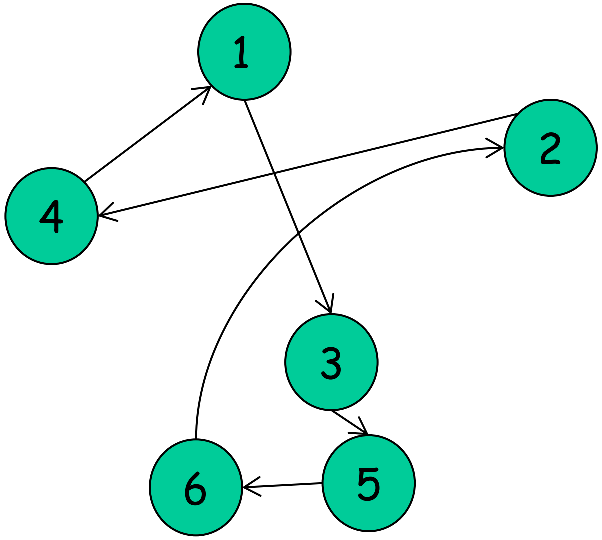


Permutation is a ***tour*** where we assume we start and end at 1st city in permutation

distance/cost table

	1	2	3	4	5	6
1	-					
2		-				
3			-			
4				-		
5					-	
6						-

tour: 1 3 5 6 2 4



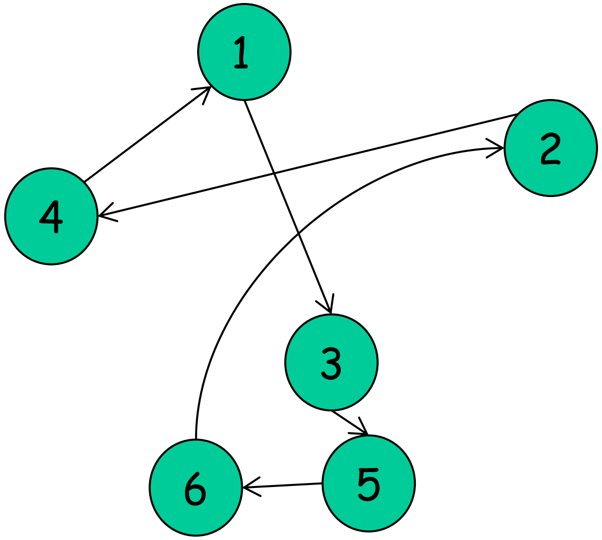
Permutation is a ***tour*** where we assume we start and end at 1st city in permutation

Use the distance/cost matrix to evaluate the tour

distance/cost table

	1	2	3	4	5	6
1	-	112	137	68	156	168
2		-	72	155	166	145
3			-	126	63	80
4				-	146	108
5					-	51
6						-

tour: 1 3 5 6 2 4



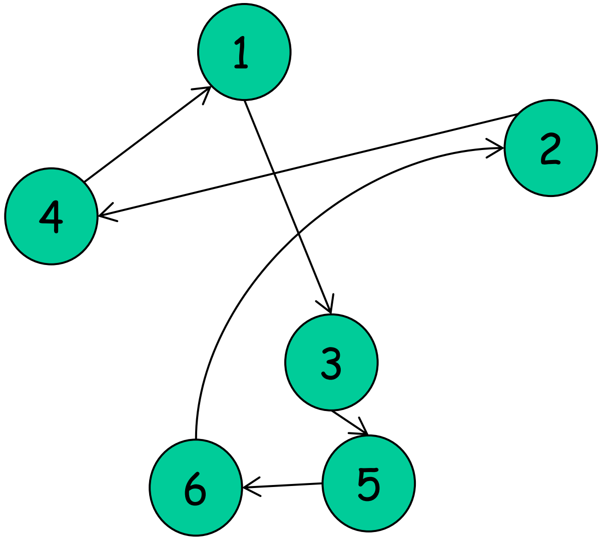
Permutation is a ***tour*** where we assume we start and end at 1st city in permutation

Use the distance/cost matrix to evaluate the tour

distance/cost table

tour: 1 3 5 6 2 4

	1	2	3	4	5	6
1	-	112	137	68	156	168
2		-	72	155	166	145
3			-	126	63	80
4				-	146	108
5					-	51
6						-



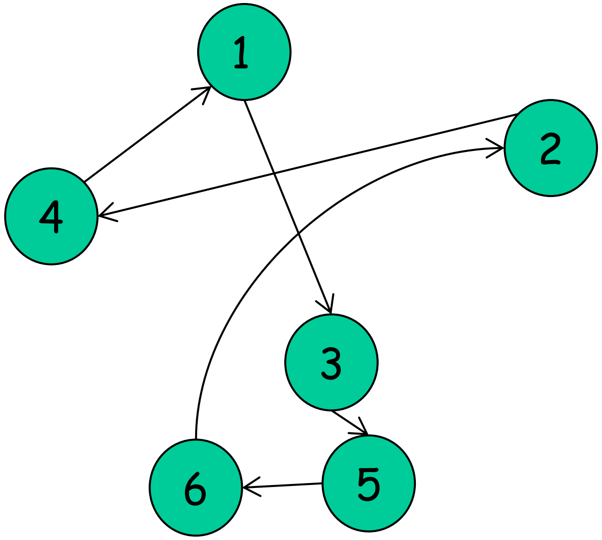
137 + 63 + 51 + 145 + 155 + 68 =

Use the distance/cost matrix to evaluate the tour

distance/cost table

tour: 1 3 5 6 2 4

	1	2	3	4	5	6
1	-	112	137	68	156	168
2		-	72	155	166	145
3			-	126	63	80
4				-	146	108
5					-	51
6						-



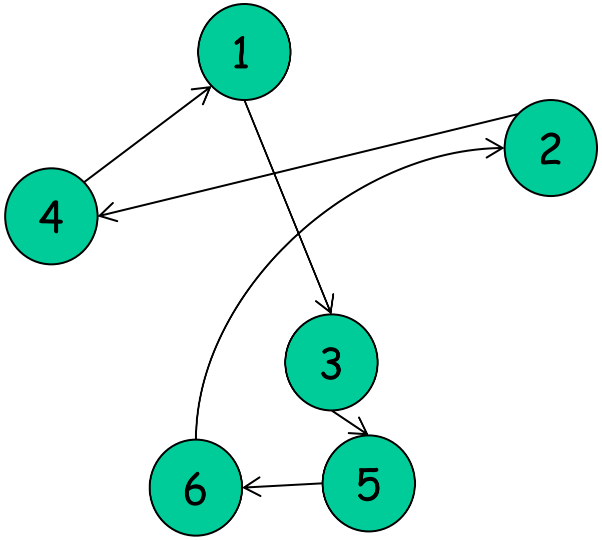
137 + 63 + 51 + 145 + 155 + 68 = 619

Use the distance/cost matrix to evaluate the tour

distance/cost table

	1	2	3	4	5	6
1	-					
2		-				
3			-			
4				-		
5					-	
6						-

tour: 1 3 5 6 2 4

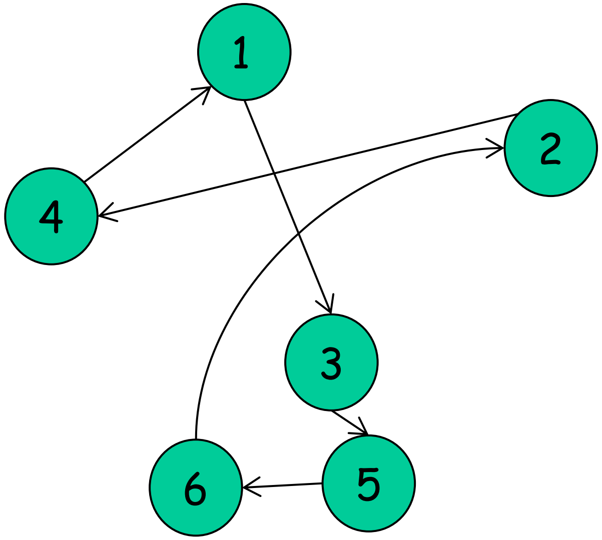


while time remains
do begin
randomly generate a tour
if it is better than the best
then save it
end

distance/cost table

	1	2	3	4	5	6
1	-					
2		-				
3			-			
4				-		
5					-	
6						-

tour: 1 3 5 6 2 4



1. How do I randomly generate a tour?
2. How do I evaluate tour?

Was that really that dumb?

Let's get smarter

Local Search (aka neighbourhood search)

We start off with a complete solution and improve it

or

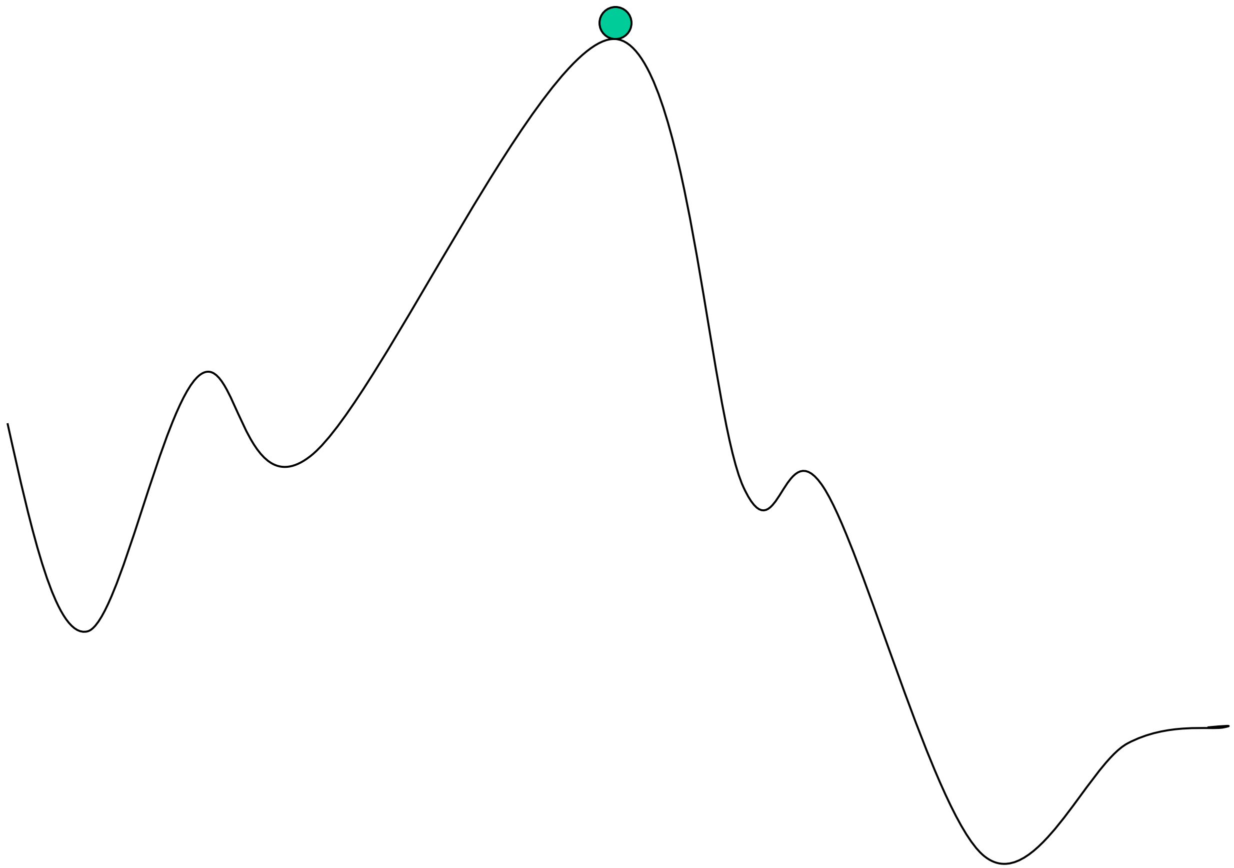
We gradually construct a solution, make our best move as we go

We need:
- a (number of) move operator(s)
    - take a state S and produce a new state S'
- an evaluation function
    - so we can tell if we *appear* to be moving in a good direction
    - let's assume we want to minimise this function, i.e. cost.

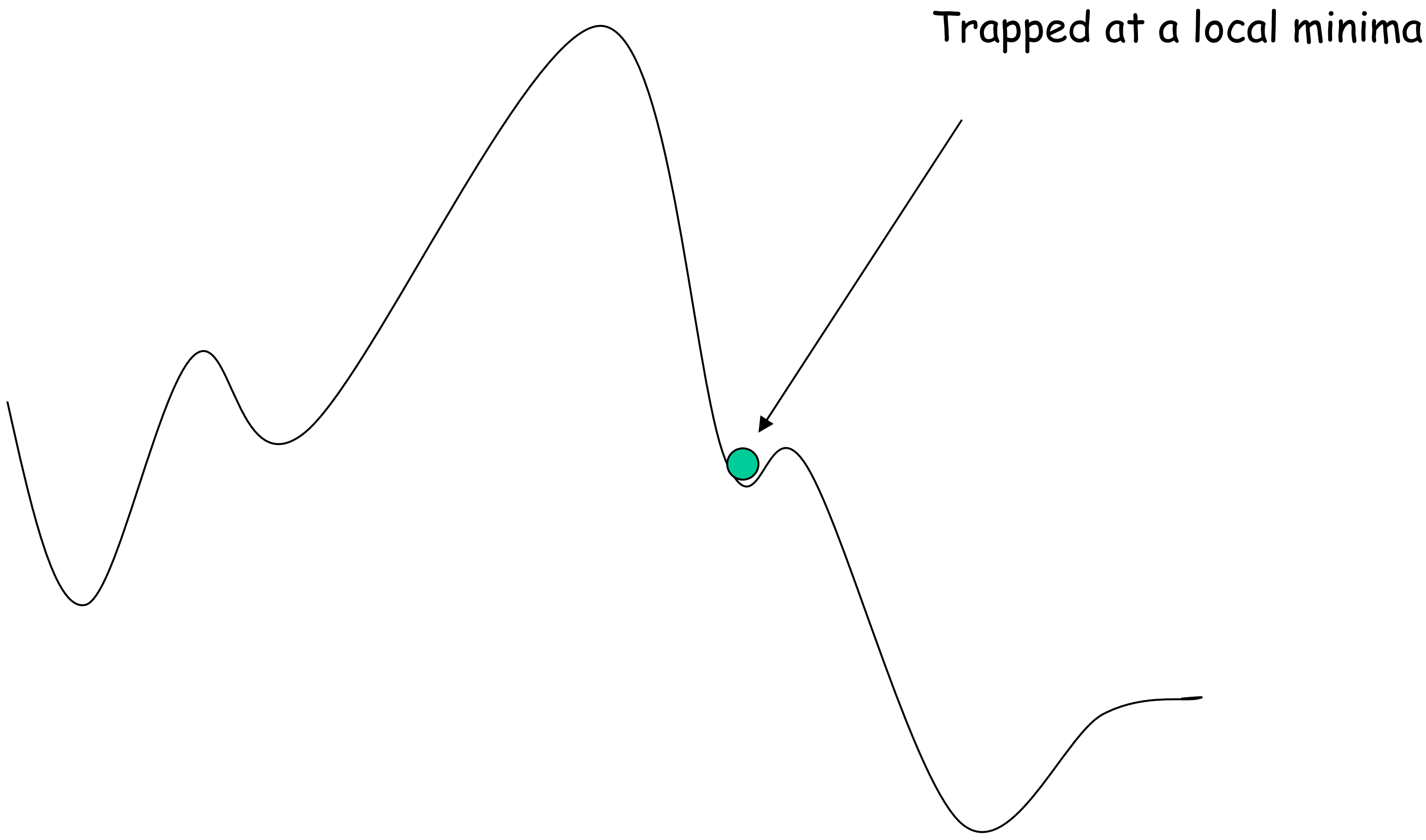
Find the lowest cost solution

Wooooooosh! Let's scream down hill.

Hill climbing/descending



Find the lowest cost solution



How can we escape?

How might we construct initial tour?

Nearest neighbour

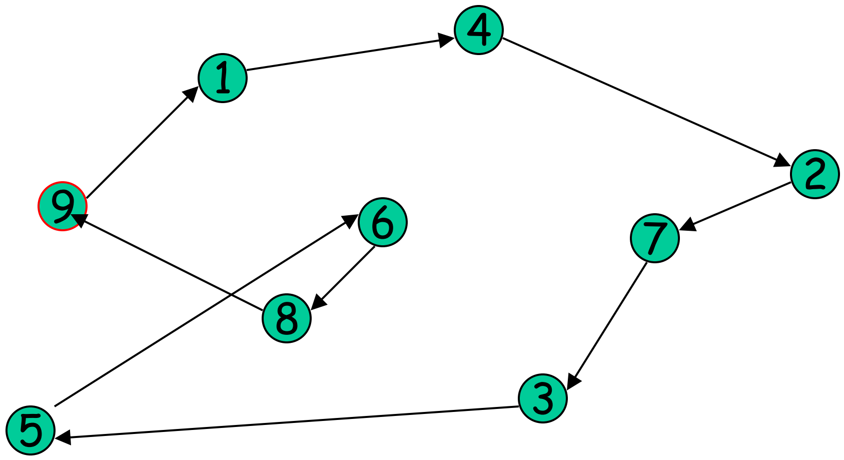
Furthest Insertion

Random

But first, an example

A move operator
- 2-opt?
    - Take a sub-tour and reverse it

A tour, starting and ending at city 9
9 1 4 2 7 3 5 6 8 9

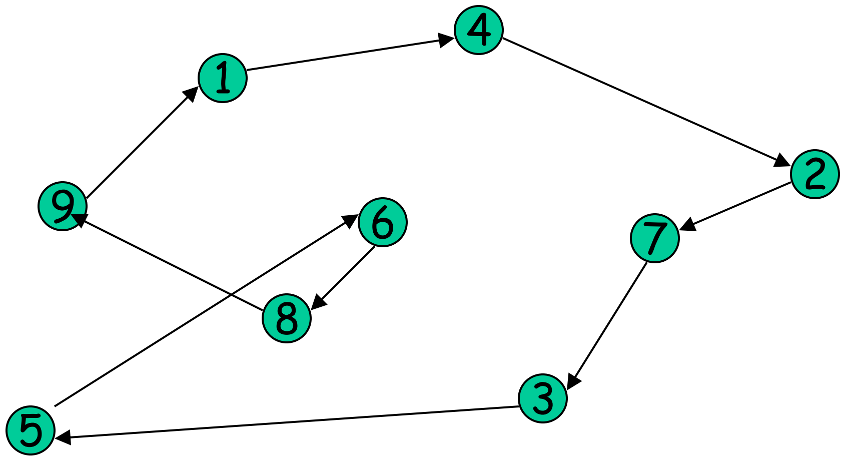


But first, an example

A move operator
- 2-opt?
    - Take a sub-tour and reverse it

9 1 4 2 7 3 5 6 8 9

reverse

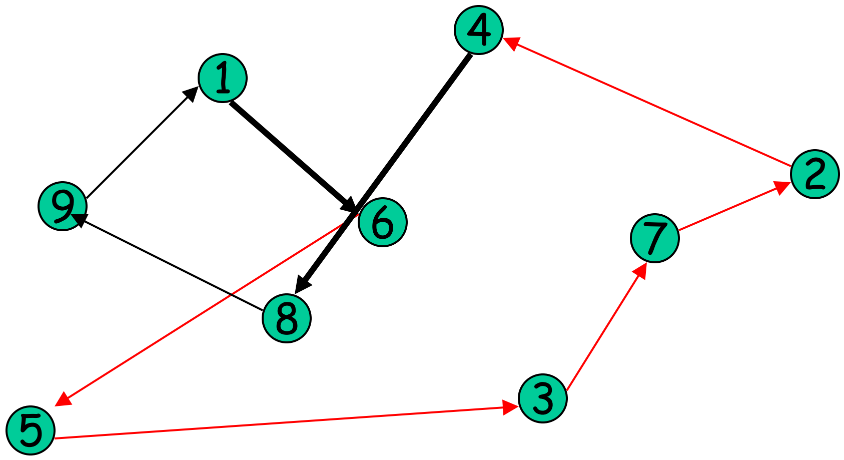


But first, an example

A move operator
- 2-opt?
  - Take a sub-tour and reverse it

9 1 4 2 7 3 5 6 8 9

9 1 6 5 3 7 2 4 8 9

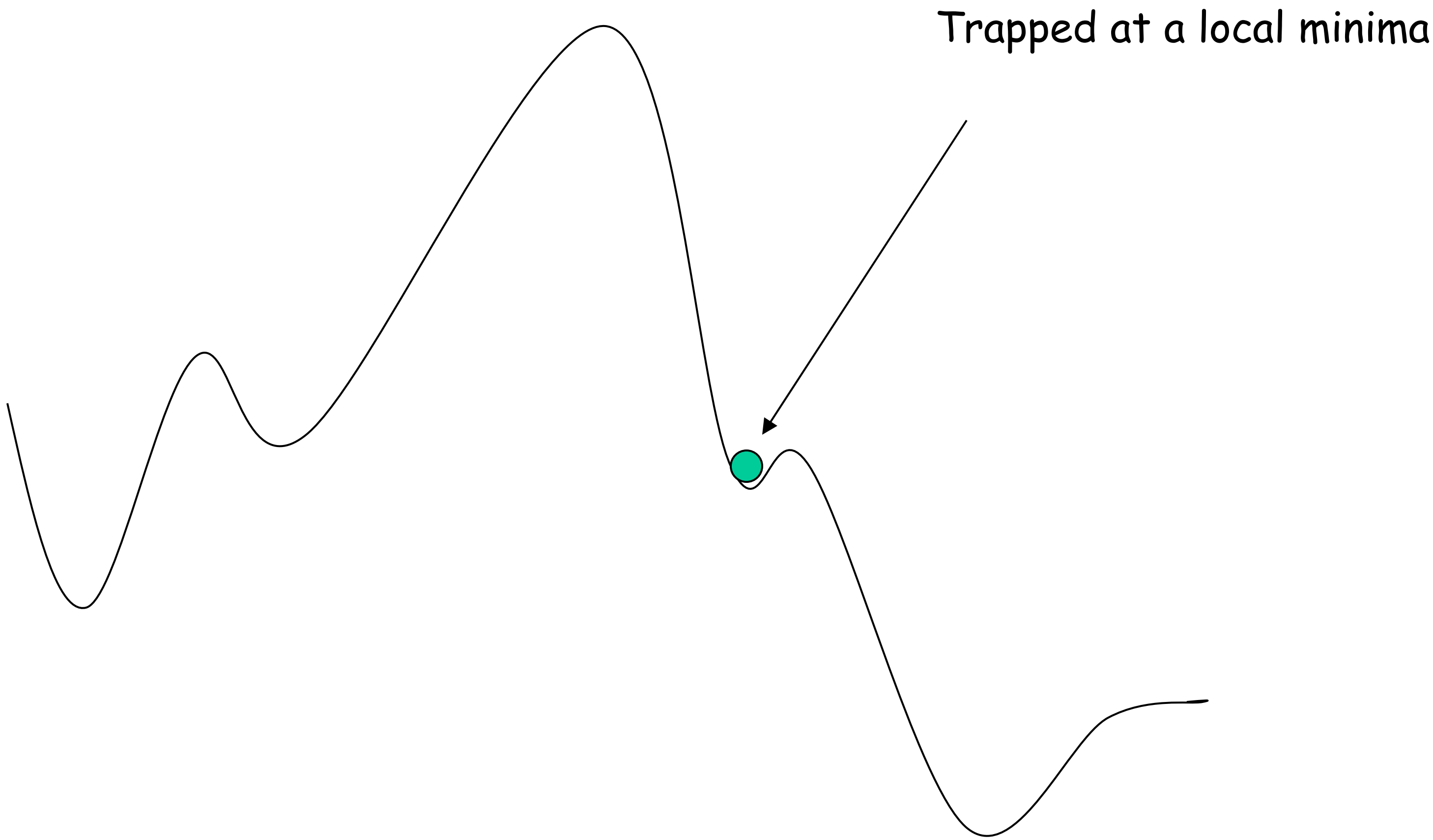


Steepest descent

```
S := construct(n)
improvement := true
while improvement
do let N := neighbourhood(S),
       S' := bestOf(N)
   in if cost(S') <= cost(S)
      then S := S'
           improvement := true
      else improvement := false
```

But … it gets stuck at a local minima

Find the lowest cost solution



How can we escape?

Consider 1-d Bin Packing

- how might we construct initial solution?
- how might we locally improve solution
  - what moves might we have?
  - what is size of neighbourhood?
  - what cost function (to drive steepest/first descent)?

Consider min-conflicts on an arbitrary csp

- how might we construct initial solution?
- how might we locally improve solution
    - what moves might we have?
    - what is size of neighbourhood?
    - what cost function (to drive steepest/first descent)?

Warning:
Local search does not guarantee optimality

 Kirkpatrick, Gelatt, & Vecci *Science 220*, 1983

Annealing, to produce a flawless crystal, a structure in a minimum energy state

- At high temperatures, parts of the structure can be freely re-arranged
    - we can get localised increases in temperature
- At low temperatures it is hard to re-arrange into anything other than a lower energy state
- Given a slow cooling, we settle into low energy states

Apply this to local search, with following control parameters

- initial temperature T
- cooling rate R
- time at temperature E (time to equilibrium)

Apply this to local search, with following control parameters

- initial temperature T (whatever)
- cooling rate R (typically R = 0.9)
- time at temperature E (time to equilibrium, number of moves examined)
- Δ change in cost (+ve means non-improving)

Accept a non-improving move with probability

$$e^{-\Delta/T}$$

Throw a dice (a uniformly random real in range 0.0 to 1.0), and if it delivers a value ***less*** than above then accept the non-improving move.

SA

K	Δ	t	$k^{-\Delta/t}$
5	1	1	0.2
5	1	10	0.85
5	1	100	0.95
5	1	10	0.85
5	2	10	0.72
5	3	10	0.62

Replaced e with k
As we increase temp t we increase probability of accept
As delta increases (cost is worse) acceptance decreases

Simulated Annealing Sketch (SA)

Kirkpatrick, Gelatt, & Vecci *Science 220*, 1983

```
S := construct(n)
while T > limit
do begin
    for in (1 .. E)
     do let N := neighbourhood(S),
              S' := bestOf(N)
              delta := cost(S') - cost(S)
         in if delta < 0 or (random(1.0) < exp(-delta/T))
             then S := S'
             if S' is best so far then save it
      T := T * R
      end
```

Scott Kirkpatrick's Home Page - Microsoft Internet Explorer

http://www.cs.huji.ac.il/~kirk/

Scott Kirkpatrick

School of Computer Science and Engineering
The Hebrew University of Jerusalem
Edmond Safra Campus, Givat Ram, Ross 81
Jerusalem 91904
Israel

email: kirk@cs.huji.ac.il
phone: +972-2-658-5838 (office)
+972-2-678-2148 (home)
+972-508-895893 (cell)
+972-545-590168 (GSM cell, used in Europe) fax: +972-2-658-5261 (office)

13 May 1983, Volume 220, Number 4598 SCIENCE

Optimization by Simulated Annealing

S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi

In this article we briefly review the central constructs in combinatorial optimization and in statistical mechanics and then develop the similarities between the two fields. We show how the Metropolis algorithm for approximate numerical simulation of the behavior of a many-body system at a finite temperature provides a natural tool for bringing the techniques of statistical mechanics to bear on optimization.

We have applied this point of view to a number of problems arising in optimal design of computers. Applications to partitioning, component placement, and wiring of electronic systems are described in this article. In each context, we introduce the problem and discuss the improvements available from optimization.

Of classic optimization problems, the traveling salesman problem has received the most intensive study. To test the power of simulated annealing, we used the algorithm on traveling salesman problems with as many as several thousand cities. This work is described in a final section, followed by our conclusions.

Combinatorial Optimization

The subject of combinatorial optimization (*1*) consists of a set of problems that are central to the disciplines of computer science and engineering. Research in this area aims at developing efficient techniques for finding minimum or maximum values of a function of very many independent variables (*2*). This function, usually called the cost function or objective function, represents a quantitative measure of the "goodness" of some complex system. The cost function depends on the detailed configuration of the many parts of that system. We are most familiar with optimization problems occurring in the physical design of computers, so examples used below are drawn from

Summary. There is a deep and useful connection between statistical mechanics (the behavior of systems with many degrees of freedom in thermal equilibrium at a finite temperature) and multivariate or combinatorial optimization (finding the minimum of a given function depending on many parameters). A detailed analogy with annealing in solids provides a framework for optimization of the properties of very large and complex systems. This connection to statistical mechanics exposes new information and provides an unfamiliar perspective on traditional optimization problems and methods.

that context. The number of variables involved may range up into the tens of thousands.

The classic example, because it is so simply stated, of a combinatorial optimization problem is the traveling salesman problem. Given a list of N cities and a means of calculating the cost of traveling between any two cities, one must plan the salesman's route, which will pass through each city once and return finally to the starting point, minimizing the total cost. Problems with this flavor arise in all areas of scheduling and design. Two subsidiary problems are of general interest: predicting the expected cost of the salesman's optimal route, averaged over some class of typical arrangements of cities, and estimating or obtaining bounds for the computing effort necessary to determine that route.

All exact methods known for determining an optimal route require a computing effort that increases exponentially with N, so that in practice exact solutions can be attempted only on problems involving a few hundred cities or less. The traveling salesman belongs to the large class of NP-complete (nondeterministic polynomial time complete) problems, which has received extensive study in the past 10 years (*3*). No method for exact solution with a computing effort bounded by a power of N has been found for any of these problems, but if such a solution were found, it could be mapped into a procedure for solving all members of the class. It is not known what features of the individual problems in the NP-complete class are the cause of their difficulty.

Since the NP-complete class of problems contains many situations of practical interest, heuristic methods have been developed with computational requirements proportional to small powers of N. Heuristics are rather problem-specific: there is no guarantee that a heuristic procedure for finding near-optimal solutions for one NP-complete problem will be effective for another.

There are two basic strategies for heuristics: "divide-and-conquer" and iterative improvement. In the first, one divides the problem into subproblems of manageable size, then solves the subproblems. The solutions to the subproblems must then be patched back together. For this method to produce very good solutions, the subproblems must be naturally disjoint, and the division made must be an appropriate one, so that errors made in patching do not offset the gains

S. Kirkpatrick and C. D. Gelatt, Jr., are research staff members and M. P. Vecchi was a visiting scientist at IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. M. P. Vecchi's present address is Instituto Venezolano de Investigaciones Cientificas, Caracas 1010A, Venezuela.

13 May 1983, Volume 220, Number 4598

SCIENCE

Optimization by Simulated Annealing

S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi

In this article we briefly review the ntral constructs in combinatorial optiization and in statistical mechanics and en develop the similarities between the o fields. We show how the Metropolis gorithm for approximate numerical mulation of the behavior of a manybody system at a finite temperature provides a natural tool for bringing the techniques of statistical mechanics to bear on optimization.

sure of the "goodness" of some complex system. The cost function depends on the detailed configuration of the many parts of that system. We are most familiar with optimization problems occurring in the physical design of computers, so examples used below are drawn from

with N, so that in practice exact s tions can be attempted only on probl involving a few hundred cities or l The traveling salesman belongs to large class of NP-complete (nonde ministic polynomial time comp problems, which has received exter study in the past 10 years (3). No me for exact solution with a computing fort bounded by a power of N has l found for any of these problems, b such a solution were found, it coul mapped into a procedure for solvin members of the class. It is not kn what features of the individual probl in the NP-complete class are the caus their difficulty.

Since the NP-complete class of p lems contains many situations of pra cal interest, heuristic methods have b developed with computational requ

Summary. There is a deep and useful connection between statistical mecha (the behavior of systems with many degrees of freedom in thermal equilibrium

original single-chip design), as is the new balance score, B, calculated as in deriving Eq. 7. The objective function analogous to Eq. 7 is

$$f = C + \lambda B \qquad (8)$$

where C is the sum of the number of external connections on the two chips and B is the balance score. For this example, $\lambda = 0.01$.

For the annealing schedule we chose to start at a high "temperature," $T_0 = 10$, where essentially all proposed circuit flips are accepted, then cool exponentially, $T_n = (T_1/T_0)^n T_0$, with the ratio $T_1/T_0 = 0.9$. At each temperature enough flips are attempted that either there are ten accepted flips per circuit on the average (for this case, 50,000 accepted flips at each temperature), or the number of attempts exceeds 100 times the number of circuits before ten flips per circuit have been accepted. If the desired number of acceptances is not achieved at three successive tempera- is considered "frozen" and annealing stops.

The finite temperature curves in Fig. 1 show the distribution of pins per chip for the configurations sampled at $T = 2.5$, 1.0, and 0.1. As one would expect from the statistical mechanical analog, the distribution shifts to fewer pins and sharpens as the temperature is decreased. The sharpening is one consequence of the decrease in the number of configurations that contribute to the equilibrium ensemble at the lower temperature. In the language of statistical mechanics, the entropy of the system decreases. For this sample run in the low-temperature limit, the two chips required 353 and 321 pins, respectively. There are 237 nets connecting the two chips (requiring a pin on each chip) in addition to the 200 inputs and outputs of the original chip. The final partition in this example has the circuits exactly evenly distributed between the two partitions. Using a more complicated balance score, which did not penalize imbalance of less than 100 circuits, we found partitions resulting in chips with 271 and 183 pins.

If, instead of slowly cooling, one were to start from a random partition and accept only flips that reduce the objective function (equivalent to setting $T = 0$ in the Metropolis rule), the result is chips with approximately 700 pins (several such runs led to results with 677 to 730 pins). Rapid cooling results in a system frozen into a metastable state far from the optimal configuration. The best result obtained after several rapid quenches is indicated by the arrow in Fig. 1.

Placement

Placement is a further refinement of the logic partitioning process, in which the circuits are given physical positions (*11, 12, 18, 19*). In principle, the two stages could be combined, although this is not often possible in practice. The objectives in placement are to minimize

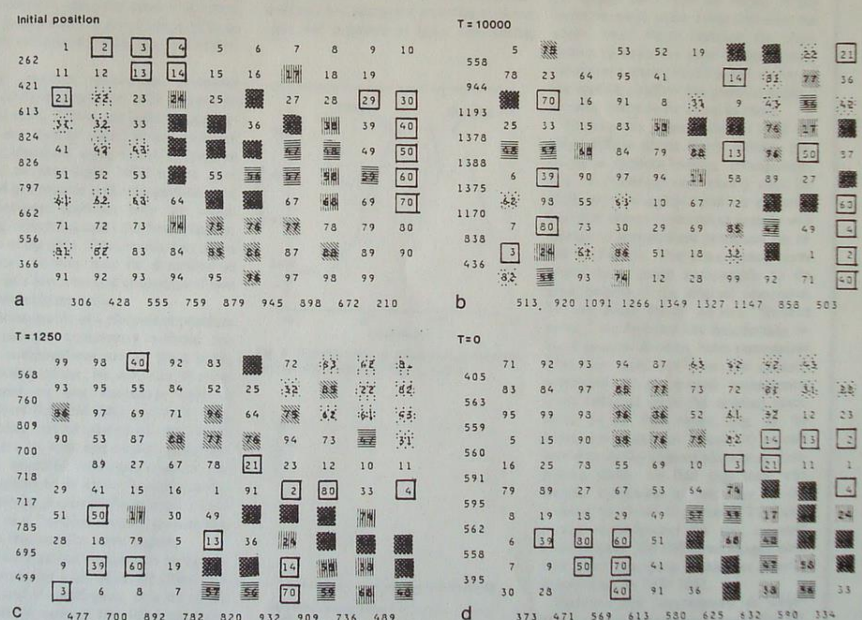
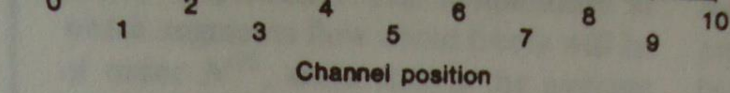


Fig. 3. Ninety-eight chips on a ceramic module from the IBM 3081. Chips are identified by number (1 to 100, with 20 and 100 absent) and function. The dark squares comprise an adder, the three types of squares with ruled lines are chips that control and supply data to the adder, the lightly dotted chips perform logical arithmetic (bitwise AND, OR, and so on), and the open squares denote general-purpose registers, which serve both arithmetic units. The numbers at the left and lower edges of the module image are the vertical and horizontal net-crossing histograms, respectively. (a) Original chip placement; (b) a configuration at $T = 10{,}000$; (c) $T = 1250$; (d) a zero-temperature result.



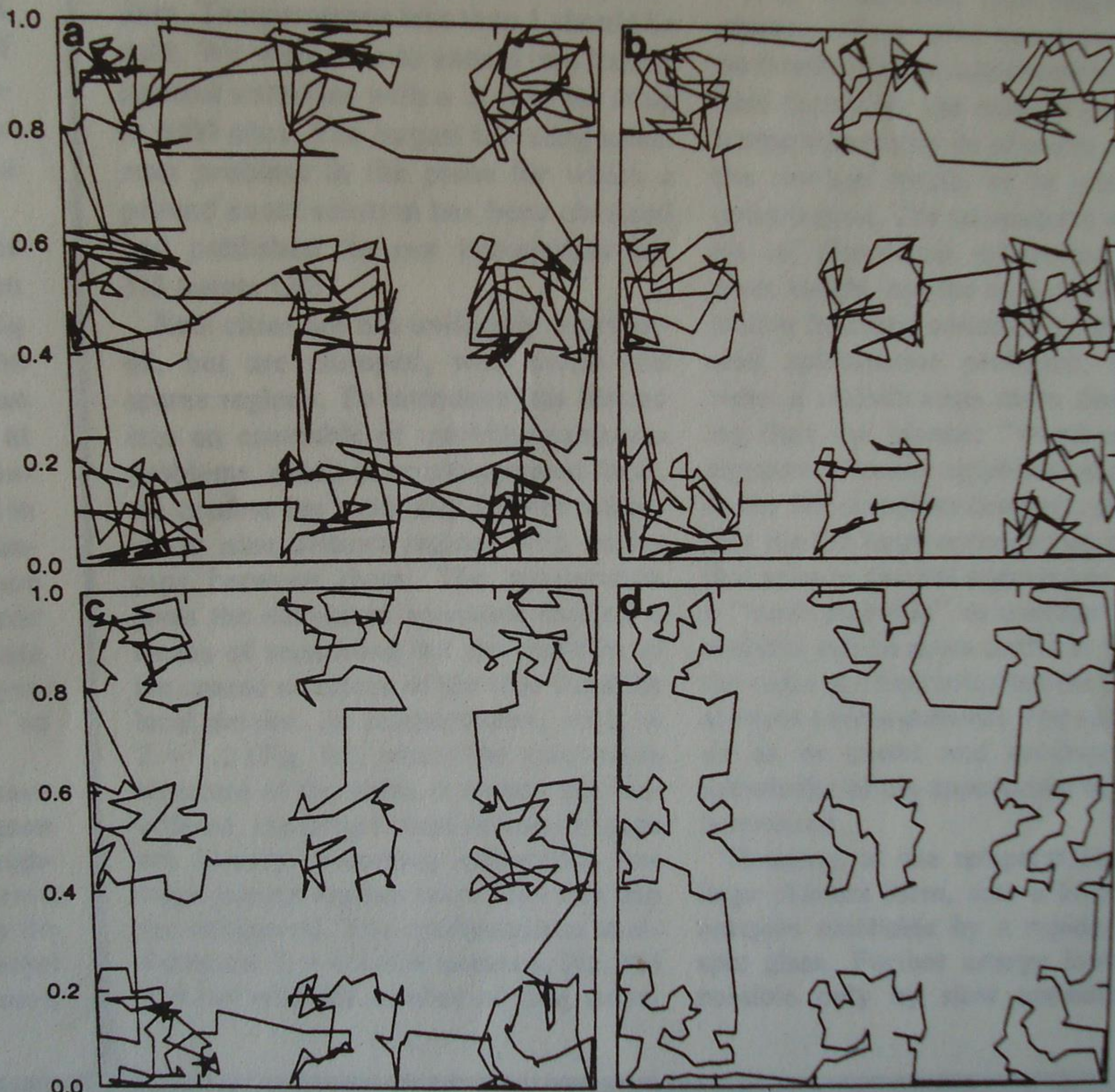


Fig. 9. Results at four temperatures for a clustered 400-city traveling salesman problem. The points are uniformly distributed in nine regions. (a) $T = 1.2$, $\alpha = 2.0567$; (b) $T = 0.8$, $\alpha = 1.515$; (c) $T = 0.4$, $\alpha = 1.055$; (d) $T = 0.0$, $\alpha = 0.7839$.

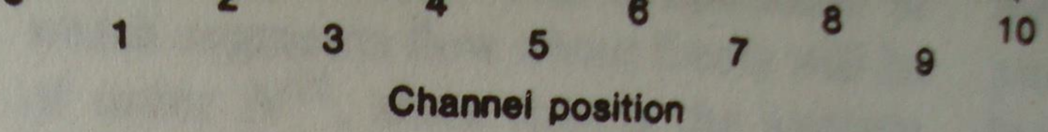
dom. All reduce the p
a link by more than 4
ed annealing with Z-m
random routing by 57
results for both x and

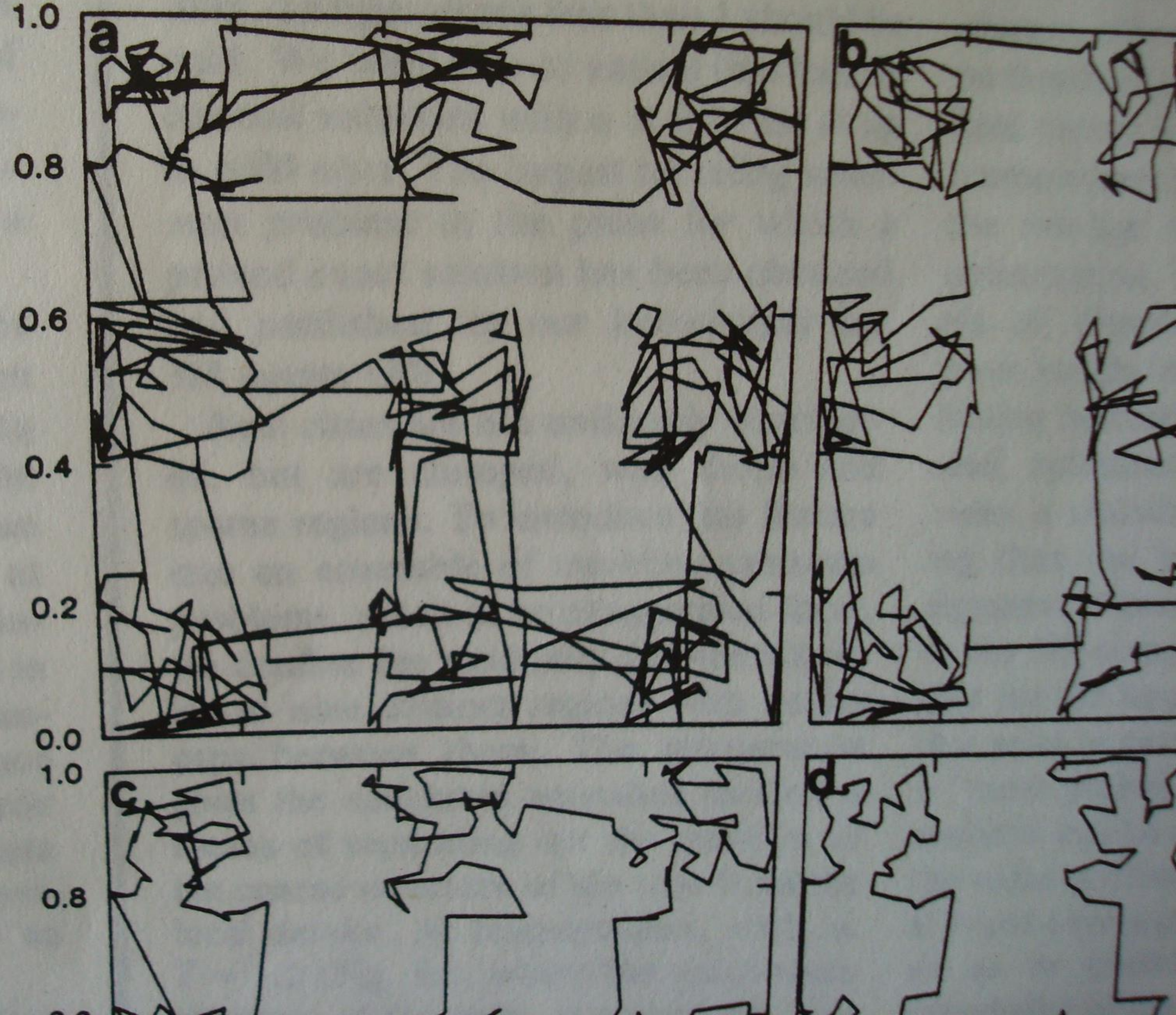
Traveling Salesmen

Quantitative analys
annealing algorithm
tween it and other
problems simpler than
computers. There is a
ture on algorithms for
man problem (*3*, *4*),
natural context for thi

If the cost of travel
is proportional to the
them, then each insta
salesman problem is s
positions of N cities.
arrangement of N po
random in a square
stance. The distance c
either the Euclidean m
hattan" metric, in w
between two points is
separations along the
axes. The latter is appr
cal design application
compute, so we will ad

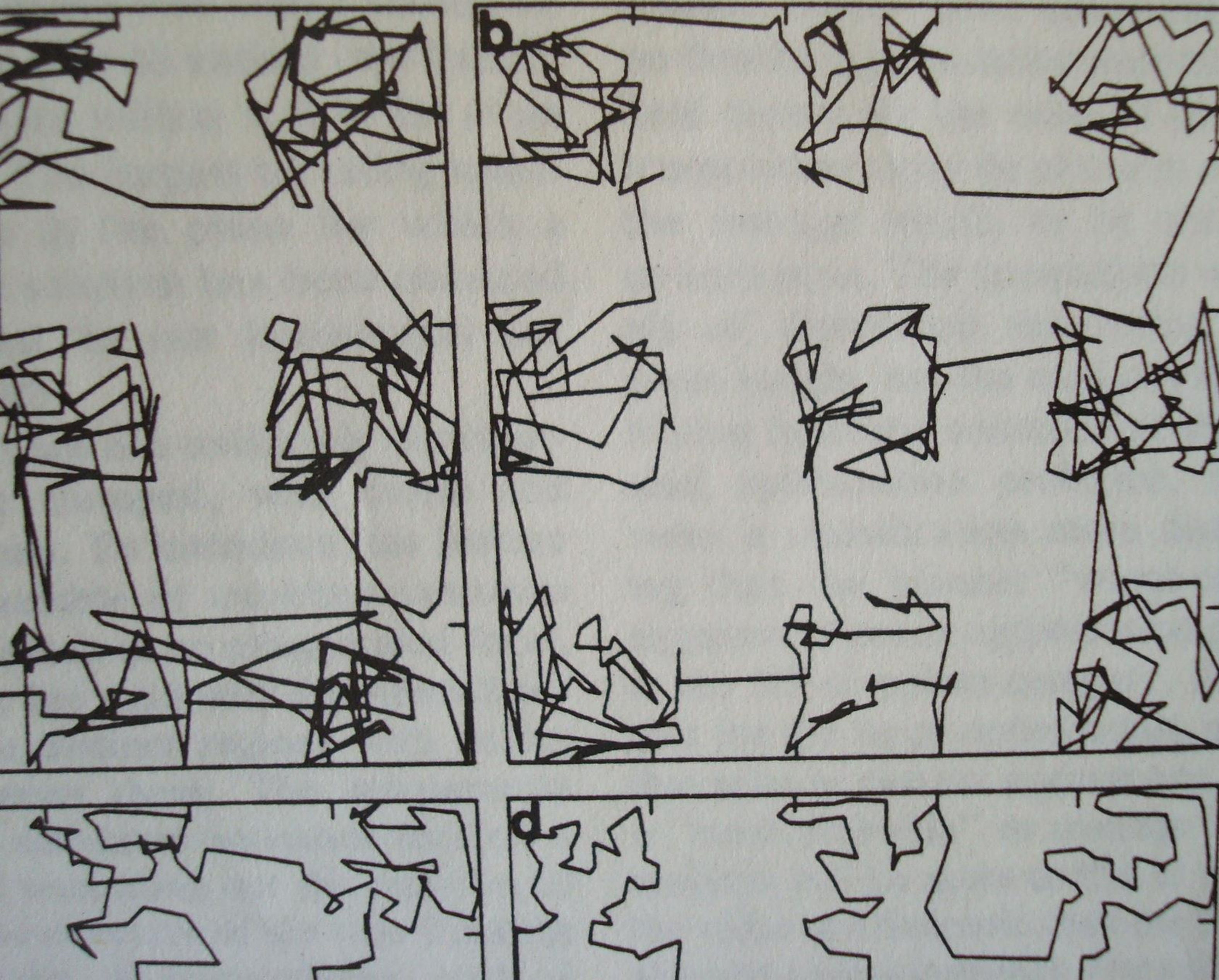
We let the side of
length $N^{1/2}$, so that the
between each city and
bor is independent of N
that this choice of length
optimal tour length per
of N, when one avera


1
2
3
4
5
6
7
8
9
10
Channel position


1.0
0.8
0.6
0.4
0.2
0.0
a
b
c
d

Channel position

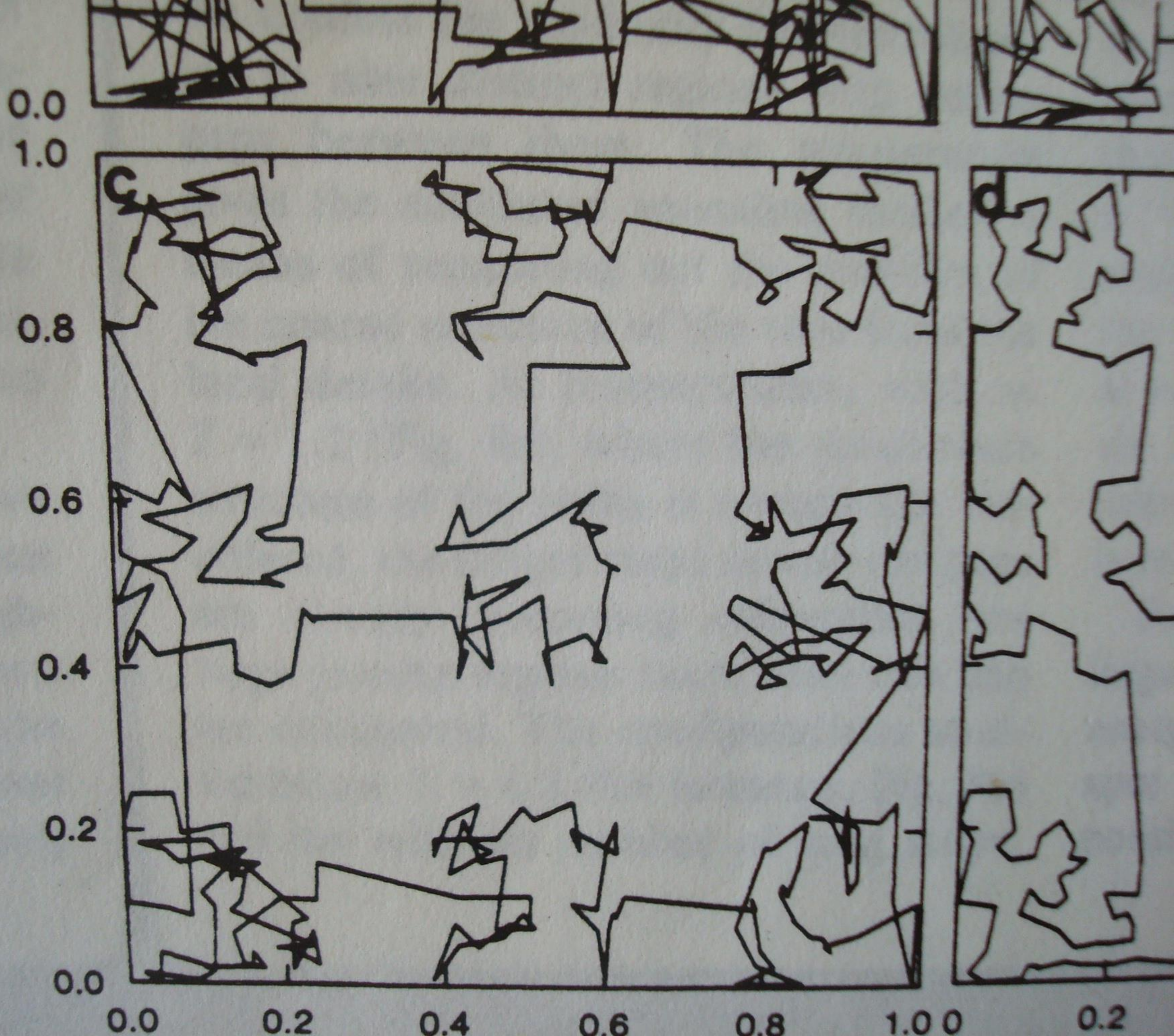
ed anneal
random ro
results for



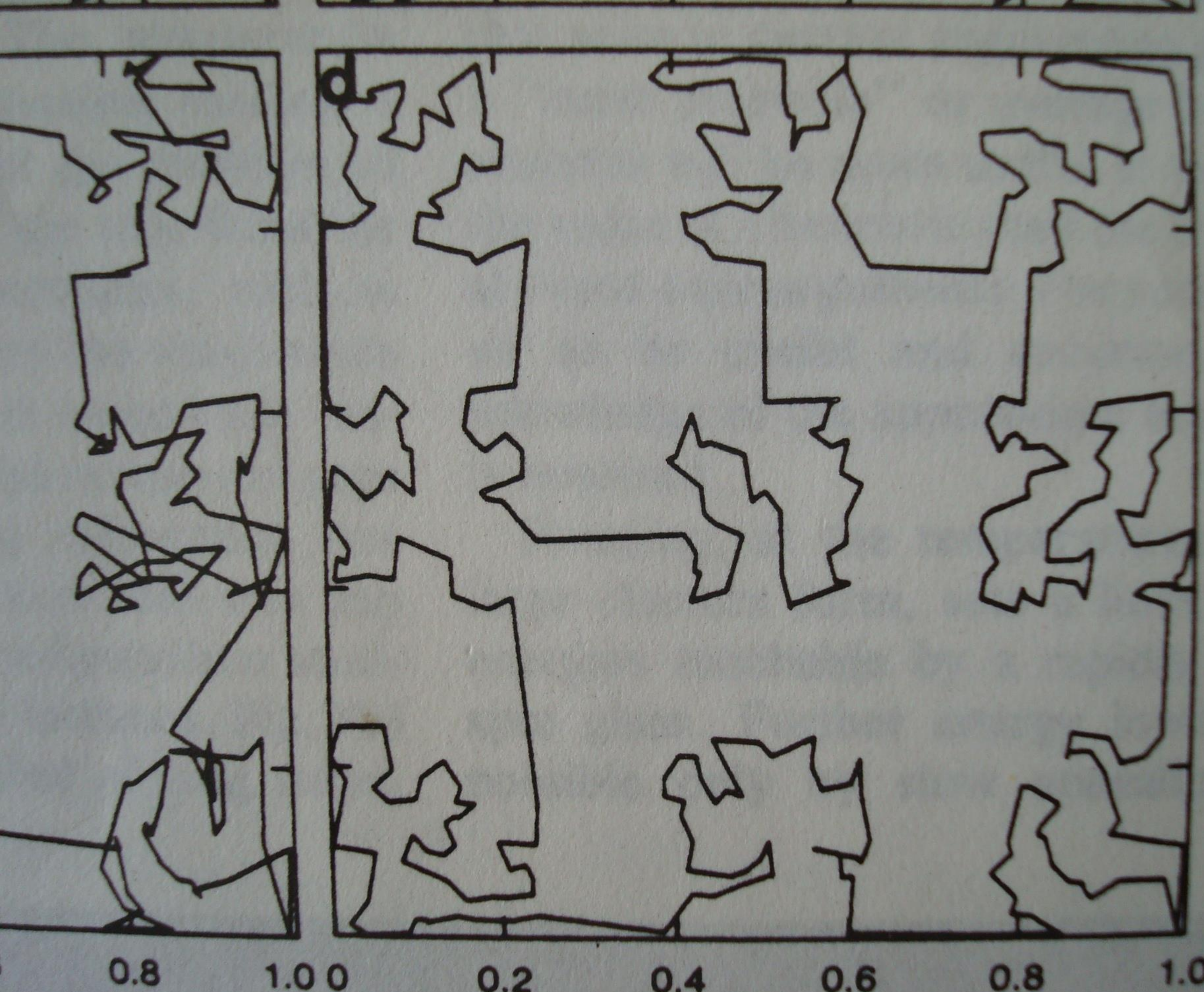
Traveling S

Quantita
annealing
tween it a
problems s
computers.
ture on alg
man probl
natural con

If the cos
is proporti
them, then
salesman p
positions o
arrangemen
random in
stance. The



0.0
1.0
c
d
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
0
0.2



the
sale
pos
arra
ran
sta
eith
hat
betv
sep
axe
cal
com
W
leng
betw
bor

SA can cycle!
- Escape a local minima
- Next move, fall back in!

- Maintain a list of local moves that we have made
  - the tabu list!
  - Not states, but moves made (e.g. 2-opt with positions j and k)
- Don't accept a move that is tabu
  - ***unless it is the best found so far***
- To encourage exploration

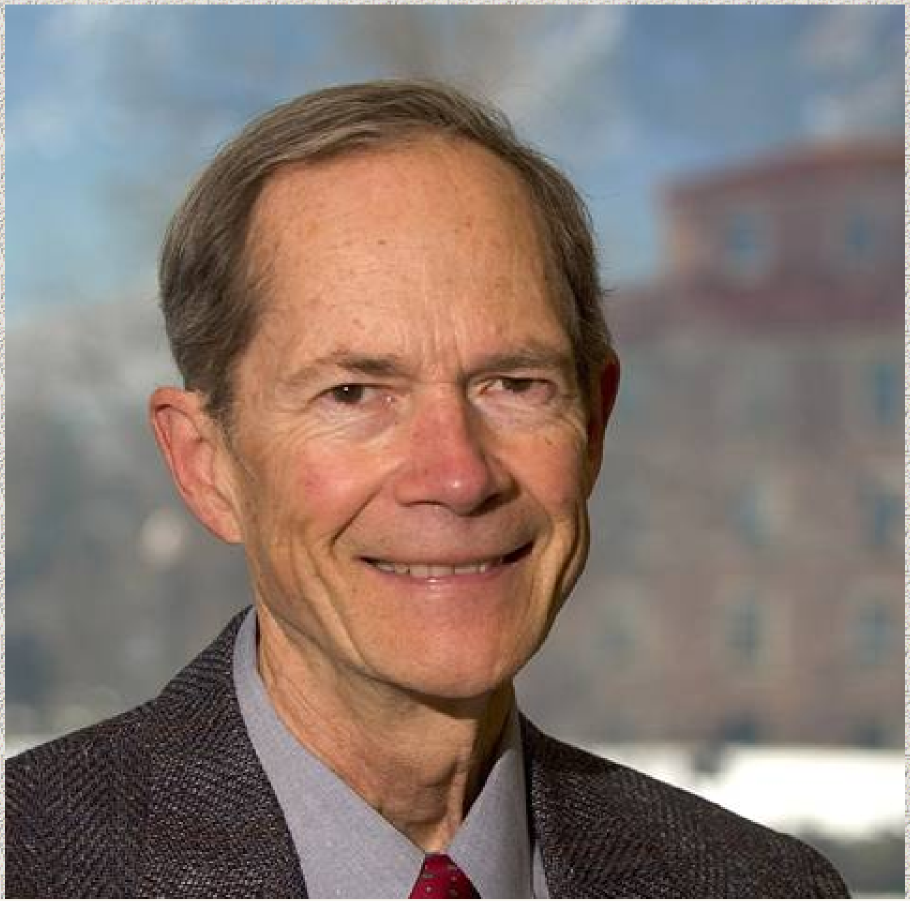
- Consider
  - size of tabu-list
  - what to put into the list
  - representation of entries in list
  - consider tsp and 1-d bp

Home Page for Glover, Fred - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://spot.colorado.edu/~glover/

Welcome to Fred Glover's Home Page



- Vita
- Publications
- Recent Papers Available for Downloading
- Tabu Search
- OptQuest

Done Internet

- (1) Construct a solution, going down hill, with steepest or 1st descent
- (2) analyse solution at local minima
    - determine most costly component of solution
        - in tsp this might be longest arc
- (3) penalise the most costly feature
    - giving a new cost function
- (4) loop back to (1) if time left

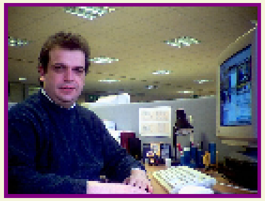
Constraint Programming and Constraint Optimization Group metaheuristic search - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Search Favorites Address http://cswww.essex.ac.uk/CSP/gls.html Go

Search Save to My Web Y! Mail Answers Personals Y! Mobile Sign in

The Guided Local Search Project

			
Chris Voudouris	**Edward Tsang**	**Patrick Mills**	**Tung-Leng Lau**

Guided Local Search (GLS) is a general meta-stochastic search strategy for solving constraint satisfaction and optimization problems. It is a control strategy designed to sit on top of hill-climbing algorithms to help them to escape local optima. It has been applied to a non-trivial number of problems, including: artificial problems, standard optimization problems and real life problems and achieved excellent results in both efficiency (in terms of speed) and effectiveness (in terms of quality of solutions).

Patrick Mills extended GLS by adding ***Aspiration*** and ***Randomness*** into the GLS. The resulting algorithm, EGLS (Extended GLS), is less less sensitive to the setting of the major parameter (Lamda-coefficient).

GLS_Solver is a solver that implements GLS in the SAT, Max-SAT and the QAP. A large number of measures are available (see details here) for the users to monitor and analyse the performance of GLS. This software enables the users to reproduce all the results that we have published in this project.

Some of the applications of GLS are:

- The travelling salesman problem (TSP) (demo available)
- The radio length frequency assignment problem (RLFAP)
- British Telecom's workforce scheduling problem
- The quadratic assignment problem

(1 item remaining) Opening page http://cswww.essex.ac.uk/CSP/gls.html... Internet

start 2 Internet Exp... 2 Windows Ex... emacs@BIAK localSearch 15:40

Genetic Algorithms (GA) John Holland, 1981

- Represent solution as a chromosome
- Have a population of these (solutions)
- Select the fittest, a champion
    - note, evaluation function considered measure of fitness
- Allow that champion to reproduce with others
    - using crossover primarily
    - mutation, as a secondary low lever operator
- Go from generation to generation
- Eventually population becomes homogenised

- Attempts to balance exploration and optimisation

- Analogy is Evolution, and survival of the fittest

It didn't work for me. I want a 3d hand, eyes on the back of my head, good looks , ...

GA sketch

- Arrange population in non-decreasing order of fitness
  - P[1] is weakest and P[n] is fittest in population
- generate a random integer x in the range 1 to n-1
- generate a random integer y in the range x to n
- Pnew := crossover(P[x],P[y])
- mutate(Pnew,pMutation)
- insert(Pnew,P)
- delete(P[1])
- loop until no time left

John Henry Holland

From Wikipedia, the free encyclopedia

Dr. **John Henry Holland** (February 2, 1929) , a pioneer in complex system and nonlinear science. He is known as the father of genetic algorithms. The recipient of the first computer science Ph.D from the University of Michigan, Holland is Professor of Psychology and Professor of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. He is also a member of The Center for the Study of Complex Systems (CSCS) at the University of Michigan, and a member of Board of Trustees and Science Board of the Santa Fe Institute.

John H Holland is the recipient of a MacArthur Fellowship and a fellow of the World Economic Forum. He is the author of a number of books about complex adaptive systems, including *Hidden Order: How Adaptation Builds Complexity* (1995), *Emergence: From Chaos to Order* (1998) and his ground-breaking book on genetic algorithms, *Adaptation in Natural and Artificial Systems* (1975,1992). Holland also frequently lectures around the world on his own research, and on current research and open ques Adaptation in Natural and Artificial Systems

External links

[edit]

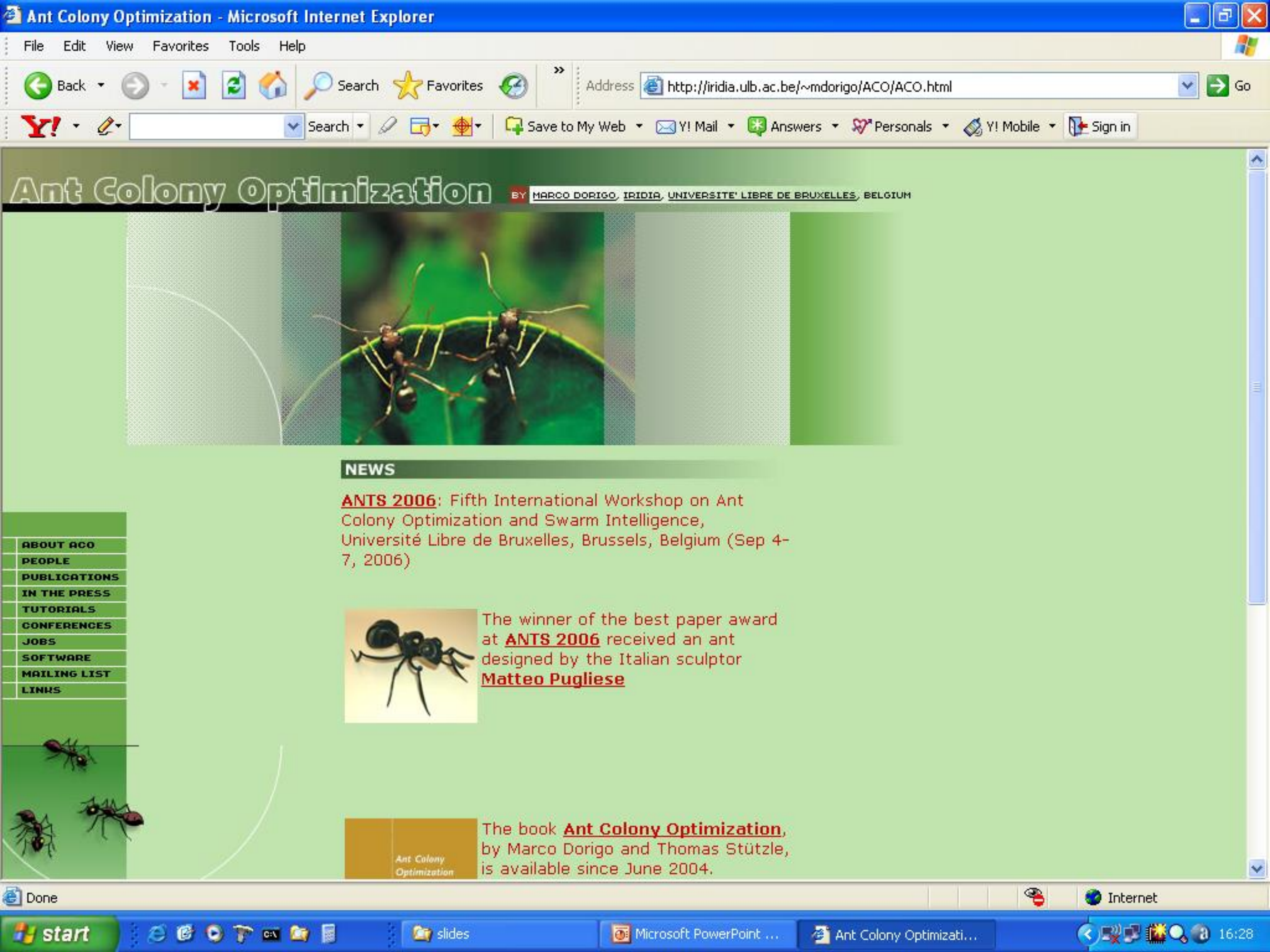
- Biography
- Susan Stepney: Bibliography of John Henry Holland
- John Holland's *Echo* project at the Santa Fe Institute

This article about a U.S. scientist is a stub. You can help Wikipedia by expanding it.

This article about a psychologist is a stub. You can help Wikipedia by expanding it.

This biographical article relating to a computer specialist in the United States is a stub. You can help Wikipedia by expanding it.

Categories: American scientist stubs | Psychologist stubs | United States computer specialist stubs | Cognitive scientists | American psychologists | 1929 births | Living people | University of Michigan faculty


Ant Colony Optimization - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://iridia.ulb.ac.be/~mdorigo/ACO/ACO.html
Go
Search Save to My Web Y! Mail Answers Personals Y! Mobile Sign in
Ant Colony Optimization
BY MARCO DORIGO, IRIDIA, UNIVERSITE' LIBRE DE BRUXELLES, BELGIUM
ABOUT ACO
PEOPLE
PUBLICATIONS
IN THE PRESS
TUTORIALS
CONFERENCES
JOBS
SOFTWARE
MAILING LIST
LINKS
NEWS
ANTS 2006: Fifth International Workshop on Ant Colony Optimization and Swarm Intelligence, Université Libre de Bruxelles, Brussels, Belgium (Sep 4-7, 2006)
The winner of the best paper award at ANTS 2006 received an ant designed by the Italian sculptor Matteo Pugliese
The book Ant Colony Optimization, by Marco Dorigo and Thomas Stützle, is available since June 2004.
Ant Colony Optimization
Done
Internet
start
slides
Microsoft PowerPoint ...
Ant Colony Optimizati...
16:28

WIKIPEDIA
The Free Encyclopedia

navigation
- Main page
- Community portal
- Featured content
- Current events
- Recent changes
- Random article
- About Wikipedia
- Contact us
- Make a donation
- Help

search
Go Search

toolbox
- What links here
- Related changes
- Upload file
- Special pages
- Printable version
- Permanent link
- Cite this article

Metaheuristic

From Wikipedia, the free encyclopedia

A **metaheuristic** is a heuristic method for solving a very general class of computational problems by combining user given black-box procedures — usually heuristics themselves — in a hopefully efficient way. The name combines the Greek prefix "meta" ("beyond", here in the sense of "higher level") and "heuristic" (from ευρισκειν, *heuriskein*, "to find").

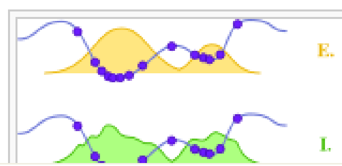
Metaheuristics are generally applied to problems for which there is no satisfactory problem-specific algorithm or heuristic; or when it is not practical to implement such a method. Most commonly used metaheuristics are targeted to combinatorial optimization problems, but of course can handle any problem that can be recast in that form, such as solving boolean equations.

In spite of overly-optimistic claims by some of their advocates, metaheuristics are not a panacea, and their indiscriminate use often is much less efficient than even the crudest problem-specific heuristic, by several orders of magnitude.

Contents [hide]

General concepts and nomenclature [edit]

The goal of combinatorial optimization is to find a discrete mathematical object (such as a bit string or permutation) that maximizes (or minimizes) an arbitrary function specified by the user of the metaheuristic. These objects are generically called *states*, and the set of all candidate states is the *search space*. The nature of the states and the search space are usually problem-specific.

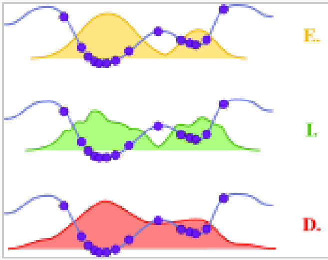


toolbox
- What links here
- Related changes
- Upload file
- Special pages
- Printable version
- Permanent link
- Cite this article

in other languages
- Deutsch
- Español
- Français
- 日本語

General concepts and nomenclature

[edit]



Legend: E=unequivocals, I=implicits, D=directs.

The goal of combinatorial optimization is to find a discrete mathematical object (such as a bit string or permutation) that maximizes (or minimizes) an arbitrary function specified by the user of the metaheuristic. These objects are generically called *states*, and the set of all candidate states is the *search space*. The nature of the states and the search space are usually problem-specific.

The function to be optimized is called the *goal function*, or *objective function*, and is usually provided by the user as a black-box procedure that evaluates the function on a given state. Depending on the meta-heuristic, the user may have to provide other black-box procedures that, say, produce a new random state, produce variants of a given state, pick one state among several, provide upper or lower bounds for the goal function over a set of states, and the like.

Some metaheuristics maintain at any instant a single *current state*, and replace that state by a new one. This basic step is sometimes called a *state transition* or *move*. The move is *uphill* or *downhill* depending on whether the goal function value increases or decreases. The new state may be constructed from scratch by a user-given *generator* procedure. Aternatively, the new state be derived from the current state by an user-given *mutator* procedure; in this case the new state is called a *neighbor* of the current one. Producers and mutators are often probabilistic procedures. The set of new states that can be produced by the mutator is the *neighborhood* of the current state.

More sophisticated meta-heuristics maintain, instead of a single current state, a *current pool* with several candidate states. The basic step then may add or delete states from this pool. User-given procedures may be called to select the states to be discarded, and to generate the new ones to be added. The latter may be generated by *combination* or *crossover* of two or more states from the pool.

A metaheuristic may also keep track of the *current optimum*, the optimum state among those already evaluated so far.

Since the set of candidates is usually very large, metaheuristics are typically implemented so that they can be interrupted after a client-specified *time budget*. If not interrupted, some *exact metaheuristics* will eventually check all candidates, and use heuristic methods only to choose the order of enumeration; therefore, they will always find the true optimum, if their time budget is large enough. Other metaheuristics give only a weaker probabilistic guarantee, namely that, as the time budget goes to infinity, the probability of checking every candidate tends to 1.

Common meta-heuristics

[edit]

Since the set of candidates is usually very large, metaheuristics are typically implemented so that they can be interrupted after a client-specified *time budget*. If not interrupted, some *exact metaheuristics* will eventually check all candidates, and use heuristic methods only to choose the order of enumeration; therefore, they will always find the true optimum, if their time budget is large enough. Other metaheuristics give only a weaker probabilistic guarantee, namely that, as the time budget goes to infinity, the probability of checking every candidate tends to 1.

Common meta-heuristics [edit]

Some well-known meta heuristics are

- Random optimization
- Local search
- Greedy algorithm and hill-climbing
- Random-restart hill climbing
- Best-first search
- Simulated annealing
- Ant colony optimization
- Tabu search
- Genetic algorithms
- GRASP
- Swarm intelligence
- Stochastic Diffusion Search
- Generalized extremal optimization

Innumerable variants and hybrids of these techniques have been proposed, and many more applications of metaheuristics to specific problems have been reported. This is an active field of research, with a considerable literature, a large community of researchers and users, and a wide range of applications.

General criticisms [edit]

While there are many computer scientists who are enthusiastic advocates of metaheuristics, there are also many who are highly critical of the concept and have little regard for much of the research that is done on it.

Innumerable variants and hybrids of these techniques have been proposed, and many more applications of metaheuristics to specific problems have been reported. This is an active field of research, with a considerable literature, a large community of researchers and users, and a wide range of applications.

General criticisms [edit]

While there are many computer scientists who are enthusiastic advocates of metaheuristics, there are also many who are highly critical of the concept and have little regard for much of the research that is done on it.

Those critics point out, for one thing, that the general goal of the typical metaheuristic — the efficient optimization of an arbitrary black-box function — cannot be solved efficiently, since for any metaheuristic *M* one can easily build a function *f* that will force *M* to enumerate the whole search space (or worse). Further, the No-free-lunch theorem has proven that over the set of all mathematically possible problems, each optimization algorithm will do on average as well as any other. Thus, at best, a specific metaheuristic can be efficient only for restricted classes of goal functions (usually those that are partially "smooth" in some sense). However, when these restrictions are stated at all, they either exclude most applications of interest, or make the problem amenable to specific solution methods that are much more efficient than the meta-heuristic.

Moreover, the more advanced metaheuristics rely on auxiliary user-given black-box producers, mutators, etc.. It turns out that the effectiveness of a metaheuristic on a particular problem depends almost exclusively on these auxiliary functions, and very little on the heuristic itself. Given any two distinct metaheuristics *M* and *N*, and almost any goal function *f*, it is usually possible to write a set of auxiliary procedures that will make *M* find the optimum much more efficient than *N*, by many orders of magnitude; or vice-versa. In fact, since the auxiliary procedures are usually unrestricted, one can submit the basic step of metaheuristic *M* as the generator or mutator for *N*. Because of this extreme generality, one cannot say that any metaheuristic is better than any other, not even for a particular class of problems — in particular, better than brute force search, or the following "banal metaheuristic":

```
1. Call the user-provided state generator.
2. Print the resulting state.
3. Stop.
```

Finally, all metaheuristic optimization techniques are extremely crude when evaluated by the standards of (continuous) nonlinear optimization. Within this area, it is well-known that to find the optimum of a smooth function on *n* variables one must essentially obtain its Hessian matrix, the n by n matrix of its second derivatives. If the function is given as a black-box procedure, then one must call it about $n^2/2$ times, and solve

```
  2. Print the resulting state.
  3. Stop.
```

Finally, all metaheuristic optimization techniques are extremely crude when evaluated by the standards of (continuous) nonlinear optimization. Within this area, it is well-known that to find the optimum of a smooth function on n variables one must essentially obtain its Hessian matrix, the n by n matrix of its second derivatives. If the function is given as a black-box procedure, then one must call it about $n^2/2$ times, and solve an n by n system of linear equations, before one can make the first useful step towards the minimum. However, none of the common metaheuristics incorporate or accommodate this procedure. At best, they can be seen as computing some crude approximation to the local gradient of the goal function, and moving more or less "downhill". But gradient-descent is a terrible non-linear optimization method, because the gradient of a "typical" function is usually almost orthogonal to the direction towards the minimum.

Pragmatics [edit]

Independently of whether those criticisms are valid or not, metaheuristics can be terribly wasteful if used indiscriminately. Since their performance is critically dependent on the user-provided generators and mutators, one should concentrate on improving these procedures, rather than twiddling the parameters of sophisticated metaheuristics. A trivial metaheuristic with a good mutator will usually run circles around a sophisticated one with a poor mutator (and a good problem-specific heuristic will often do much better than both). In this area, more than in any other, a few hours of reading, thinking and programming can easily save months of computer time. On the other hand, this generalization does not necessarily extend equally to all problem domains. The use of genetic algorithms, for example, has produced evolved design solutions that exceed the best human-produced solutions despite years of theory and research. Problem domains falling into this category are often problems of combinatorial optimization and include the design of sorting networks, and evolved antennas, among others.

Bibliography [edit]

C. Blum and A. Roli (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys* 35 (3) 268–308.

Links [edit]

DGPF A distributed framework for randomized, heuristic searches like GA and Hill Climbing which comes with a specialization for Genetic Programming and allows to combine different search algorithms.

HC, SA, TS, GLS are *point based*

GA is *population based*

All of them retain the best solution found so far (of course!)

Local Search: a summary

- cannot guarantee finding optimum (i.e. incomplete)
- these are "meta heuristics" and need insight/inventiveness to use
- they have parameters that must be tuned
- tricks may be needed for evaluation functions to smooth out landscape
- genetic operators need to be invented (for GA)
    - example in TSP, with PMX or order-based chromosome
    - this may result in loss of the spirit of the meta heuristic
- challenge to use in CP environment (see next slides)

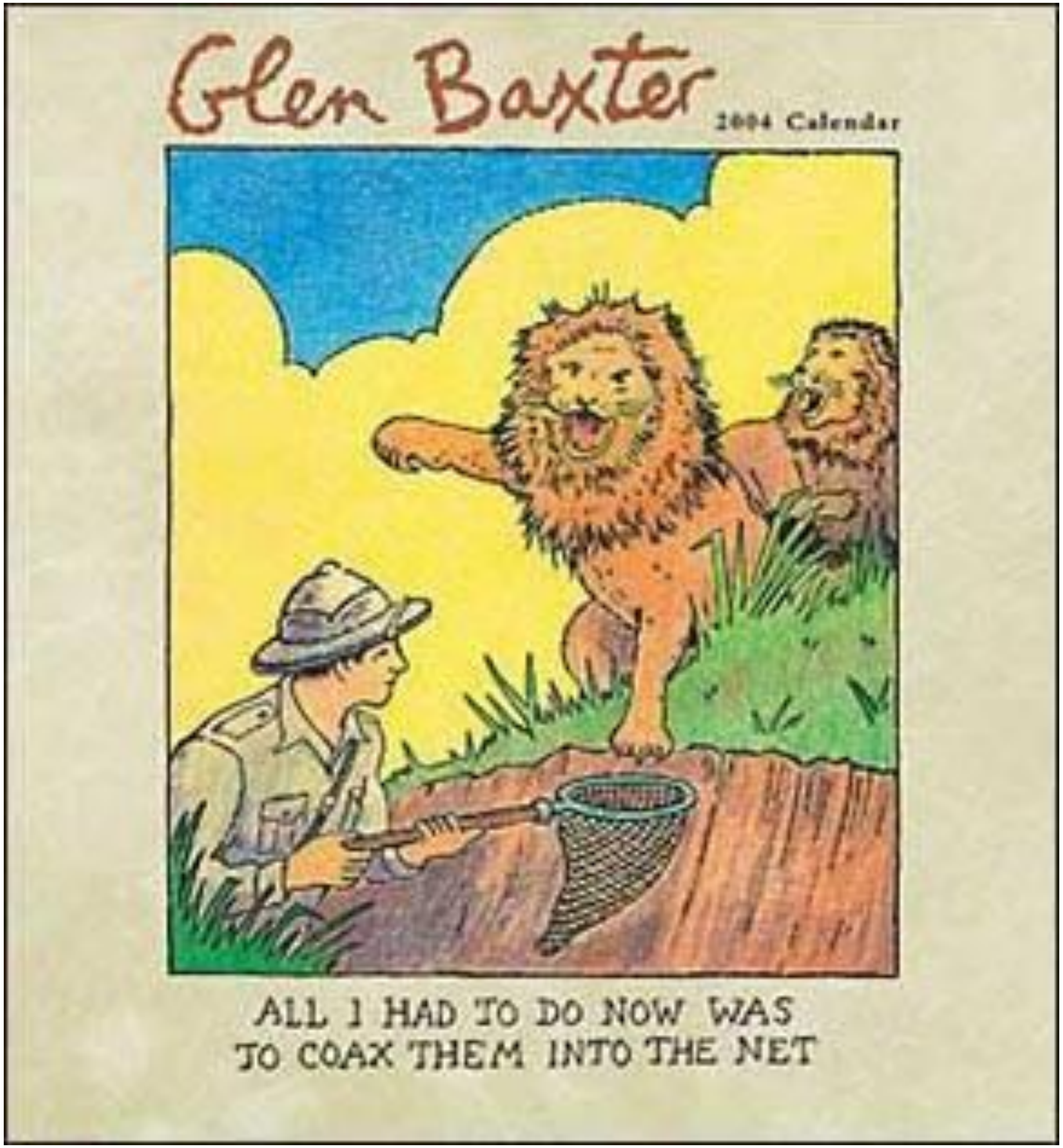
Local search for a csp (V,C,D)

let cost be the number of conflicts

Therefore we try to move to a state with less conflicts

Warning
Local search cannot prove that there is no solution
neither can it be guaranteed to find a solution

Problems with local search and CP?



Problems with local search and csp?

- how do we do a local move?
- how do we undo the effects of propagation?
- can we use propagation?
- maybe use 2 models
    - one active, conventional
    - one for representing state
        - used to compute neighbourhood
        - estimate cost of local moves

Dynadec / Solution Videos

A Constraint-Based Architecture for Local Search

Laurent Michel
Brown University, Box 1910,
Providence RI 02912
ldm@cs.brown.edu

Pascal Van Hentenryck
Brown University, Box 1910,
Providence RI 02912
pvh@cs.brown.edu

ABSTRACT

Combinatorial optimization problems are ubiquitous in many practical applications. Yet most of them are challenging, both from computational complexity and programming standpoints. Local search is one of the main approaches to address these problems. However, it often requires sophisticated incremental algorithms and data structures, and considerable experimentation. This paper proposes a constraint-based, object-oriented, architecture to reduce the development time of local search algorithms significantly. The architecture consists of declarative and search components. The declarative component includes *invariants*, which maintain complex expressions incrementally, and *differentiable objects*, which maintain properties that can be queried to evaluate the effect of local moves. Differentiable objects are high-level modeling concepts, such as constraints and functions, that capture combinatorial substructures arising in many applications. The search component supports various abstractions to specify heuristics and meta-heuristics. We illustrate the architecture with the language COMET and several applications, such as car sequencing and the progressive party problem. The applications indicate that the architecture allows for very high-level modeling of local search algorithms, while preserving excellent performance.

1. INTRODUCTION

Combinatorial optimization problems are ubiquitous in practical applications such as logistics, scheduling, resource allocation, and computational biology to name only a few. Most of these problems are NP-complete and challenging both from computational and software engineering standpoints. Indeed, reasonable solutions to these problems often involve complex algorithms and data structures, as well as significant experimentation.

The last two decades have witnessed the emergence of many languages and libraries for combinatorial optimization (e.g., [6, 9, 14, 25, 26]). These languages may dramatically reduce development time for these applications, while inducing small overheads in efficiency. However, most tools focus on global search which includes branch & bound and constraint satisfaction algorithms. They offer little or no support for local search.

Local search is one of the most widely used approaches to combinatorial optimization because it often produces high-quality solutions in reasonable time. Local search tackles combinatorial optimization problems by moving from a configuration to one of its neighbors until a feasible configurationers to focus on high-level modeling issues and to relieve them from many tedious and error-prone aspects of local search. The resulting programs are often intuitive and natural, yet they are also efficient since they encapsulate years of research on incremental algorithms. Another important benefit of the architecture is its compositionality. It is easy to add new constraints, and to modify or remove existing ones, without having to worry about the global effect of these changes. In the same spirit, the architecture clearly separates the modeling and search components and makes it easy to experiment with different meta-heuristics without affecting the problem modeling. Finally, the architecture is non-intrusive and lets programmers choose their own modeling and meta-heuristic, thus supporting a wide variety of local search algorithms.

It is also worth emphasizing that the architecture builds on fundamental research in programming languages. It integrates one-way constraints pioneered in the Sketchpad and ThingLab object-oriented systems [24, 4] and generalizes them to accommodate finite differencing techniques on algebraic and set expressions [19, 29]. It also encapsulates efficient incremental graph algorithms [21, 1], and uses polymorphism heavily to obtain its compositional nature. Observe also that the resulting architecture has some flavor of aspect-oriented programming [13], since constraints represent and maintain properties across a wide range of objects. Finally, note also that many of the concepts introduced in COMET are much more widely applicable and would benefit many other applications, where incremental algorithms and data structures are heavily used. This is typical in many greedy algorithms, as well as in many heuristic and approximation algorithms.

This paper illustrates the architecture using COMET, a Java-like programming language under development at Brown University. COMET supports the local search architecture with a number of novel concepts, abstractions, and control structures. However, it is important to point out that the architecture could be implemented equally well as a library or on top of an existing language. COMET simply enables us to experiment easily with a variety of designs and implementation techniques, and to choose a syntax reflecting the semantics of the architecture.

them. Moreover, some of these applications are rather sophisticated and, in one case, COMET enabled us to close an open problem. We also report that the efficiency of a JIT implementation of COMET is comparable to special-purpose algorithms. It is also useful to mention that COMET helped us to design the fastest algorithm for warehouse location, which will be reported in another paper. As a consequence, we believe that the architecture brings significant benefits for the implementation of local search algorithms, which are so fundamental in numerous application areas.

The rest of the paper is organized as follows. Section 2 reviews the architecture in more detail. Section 3, 4, and 5 present three applications to give some of the flavor of the architecture. Section 6 describes the implementation, which generalizes finite-differencing techniques, and Section 7 concludes the paper.

2. THE ARCHITECTURE

Our architecture for local search, depicted in Figure 1, consists of a declarative and a search component organized in three layers. The kernel of the architecture is the concept of *invariants* (or one-way constraints) over algebraic and set expressions [15]. Invariants are expressed in terms of incremental variables and specify a relation which must be maintained under assignments of new values to its variables. For instance, the code fragment

```
inc{int} s(m) <- sum(i in 1..10) a[i];
```

declares an incremental variable `s` of type `int` (in a model `m`) and an invariant specifying that `s` is always the summation of `a[1],...,a[10]`. Each time, a new value is assigned to an element `a[i]`, the value of `s` is updated accordingly (in constant time). Note that the invariant specifies the relation to be maintained incrementally, not how to update it. Incremental variables are always associated with a model (`m` in this case), which makes it possible to use a very efficient implementation which dynamically determines a topological order in which to update the invariants. As we will see later in the paper, COMET supports a wide variety of algebraic and set invariants. It also contains a number of graph invariants (to incrementally maintain shortest and longest paths) but these are not discussed in this paper.

```
int n = 512;
range Size = 1..n;
Model m();
UniformDistribution distr(Size);

inc{int} queen[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen,neg));
S.post(new AllDifferent(queen,pos));
inc{set{int}} conflictSet(m);
conflictSet <- argMax(q in Size) S.violations(queen[q]);
m.close();

while (S.violationDegree())
  select(q in conflictSet)
    selectMin(v in Size)(S.getAssignDelta(queen[q],v))
      queen[q] := v;
```

Figure 3: The Queens Problem in Comet.

Figure 3 depicts the COMET program for the queens problem. As can easily be seen, the COMET program is extremely compact and high-level. Since this is our first application, we describe the program in detail. The first instructions declare the size of the board and a range containing all the board positions. The instruction

```
Model m();
```

declares a model m. As mentioned earlier, models are container objects which store incremental variables, invariants, and constraints. They make it possible to have an efficient planning/execution implementation based on dynamic topological orderings [16]. The above instruction is, in fact, syntactic sugar for the more traditional (Java-like) code

```
Model m = new Model();
```

The instruction

```
UniformDistribution distr(Size);
```

declares a uniform distribution which can be used to generate pseudo-random numbers in range Size. This random distribution will be used to generate the initial positions of the queens. The next instruction

```
inc{int} queen[i in Size](m,Size) := distr.get();
```

is particularly interesting. It declares an array of incremental variables queen, each variable queen[c] representing the row where the queen in column c is located. These incremental variables take their values in range Size and are initialized with random positions. Incremental variables are central in the architecture. They are used in invariants and constraints, and changes to their values induce a propagation step that updates all invariants and constraints directly or indirectly affected by the changes. Observe that the above instruction is rather compact. In fact, it is just syntactic sugar for the code

```
inc{int} queen[] = new inc{int}[Size];
forall(i in Size) {
   queen[i] = new inc{int}(m,Size);
   queen[i] := distr.get();
}
```

which is closer to traditional object-oriented code. Observe the use of the assignment operator :=, which assigns a value of type T to an incremental variable of type inc{T}. By contrast, the operator = assigns references.

3.1 The Declarative Component

The next couple of instructions are fundamental and specify the declarative component of the program. They describe the problem constraints in a high-level declarative way. The instruction

```
ConstraintSystem S(m);
```

declares a constraint system S, while the instructions

```
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen,neg));
S.post(new AllDifferent(queen,pos));
```

add the problem constraints to S. The first constraint specifies that all queens must be placed on different rows and is simply the AllDifferent differentiable object presented earlier. The next two constraints specify that the queens must be placed on different diagonals. They use a more general form of the constraint. More precisely, a constraint AllDifferent(v,o) holds if v and o are arrays whose range is 1..n and the constraints

$$v[i] + o[i] \neq v[j] + o[j]$$

hold for all $1 \leq i < j \leq n$. Observe that these constraints specify the problem declaratively in a very natural way and capture a combinatorial substructure often used in combinatorial optimization problems.

The final instructions of the declarative component

```
inc{set{int}} conflictSet(m);
conflictSet <- argMax(q in Size) S.violations(queen[q]);
```

are particularly interesting as well. The first instruction declares an incremental variable conflictSet whose values are of type "set of integers". The second instruction imposes an invariant ensuring that the values of conflictSet always be the set of queens that violate the most constraints. Observe that S.violations(queen[q]) returns an incremental variable representing the number of violations in S. Operator argMax(i in S) E simply returns the set of values v in S which maximizes E. The last instruction

```
m.close()
```

closes the model, which enables COMET to construct a dependency graph to update the constraints and invariants under changes to the incremental variables.

Observe that the declarative component only specifies the properties of the solutions, as well as the data structures to maintain. It does not specify how to update the constraints, e.g., the violations of the variables, or how to update the conflict set. These are performed by the COMET runtime system, which uses optimal incremental algorithms in this case.

Local Search for CP, *some recent work*

- min-conflicts
  - Minton @ nasa
- WalkSat
  - reformulate CSP as SAT
- GENET
  - Tsang & Borrett
- Weak-commitment search
  - Yokoo AAAI-94
- Steve Prestwich
  - Cork
- Large Neighbourhood Search
  - Paul Shaw, ILOG
- Incorporating Local Search in CP (for VRP)
  - deBaker, Furnon, Kilby, Prosser, Shaw
- LOCALIZER
- COMET

Is local search used?
(aka “who cares”)

You bet!
(aka industry)

Early work on SA was Aart’s work on scheduling

BT use SA for Workforce management (claim $120M saving per year)

ILOG Dispatcher uses TS & GLS (TLS?)

COMET Dynadec

By eliminating symmetries we are producing a non-equivalent CSP which has fewer solutions. But the solutions eliminated can be derived from those which are still allowed, and for all practical purposes we have only eliminated solutions which are equivalent to solutions which still exist. On the other hand, implied constraints do not change the set of possible solutions at all.

11 When Systematic Search is Not Good Enough

Constraint satisfaction problems are difficult to solve; there is no known method which has reasonable complexity, so that as we try to solve larger and larger problems, sooner or later we shall meet a problem which cannot be solved, or proved not to have a solution, in a reasonable time. For instance, I have tried to solve instances of the template design problem where there are 50 different designs and 40 slots in each template, and sometimes no solution can be found even if the program is left running overnight. It is often possible to improve the performance of the algorithms by thinking of better variable and value ordering heuristics, new implied constraints, etc. (for instance, a program which previously found no solution overnight may now find a solution in a few seconds) but suppose we have done everything we can think of. What do we do then?

It is often tempting to assume that the problem we are trying to solve has no solution, especially if we have an optimisation problem; this means that the last solution we have (if we have one) is the optimal solution. However, in reality we may be a long way from the optimal solution, unless we have some independent evidence that suggests that the solution we already have may be 'good enough'.

In other cases, we may simply be trying to find a solution that satisfies the constraints; then, if the algorithm fails to find a solution, and appears to be running for an indefinite amount of time, we have no answer to our problem at all. Sometimes in these circumstances, if we suspect that there really is no solution, we may be willing to relax some of the constraints so as to find a solution of some kind. Even if there is a solution, but the algorithm cannot find one, we may prefer to settle for a solution that satisfies most of the constraints rather than having no solution at all.

It is possible to express the problem of satisfying as many constraints as possible as a CSP; rather than posting every constraint, the constraints which we allow to be relaxed are simply defined. In Solver, each such constraint corresponds to a constrained Boolean expression. We can create another constrained (integer) expression representing the number of these constraints that are satisfied, and then maximise its value. Unfortunately, the resulting problem is likely to be even harder to solve than the original problem (although it will have a solution, whereas the original problem may have been infeasible). Unless a constraint definitely applies (i.e. has been posted) its effects cannot be propagated, so that the algorithm has to do correspondingly more search.

An alternative is to abandon the systematic search algorithms we have used so far, and use some kind of *local search* procedure, which always has a solution of sorts, and continually attempts to improve it.

An example is the *min-conflicts* heuristic (S. Minton, M.D. Johnston, A.B. Philips and P. Laird, 'Solving Large-Scale Constraint Satisfaction and Scheduling far, and use some kind of *local search* procedure, which always has a solution of sorts, and continually attempts to improve it.

An example is the *min-conflicts* heuristic (S. Minton, M.D. Johnston, A.B. Philips and P. Laird, 'Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method', Proceedings AAAI-90, pp. 17-254, 1990). This grew out of a neural network developed for scheduling the use of the Hubble Space Telescope. The neural network was extremely successful, and it was analysed to try to find out why; the min-conflicts heuristic is a simple algorithm which was developed from this analysis, and is reported to perform better than the neural network.

This heuristic can be built into a procedure which attempts to find a solution which maximises the number of satisfied constraints as follows:

- form an initial solution by assigning a value to each variable at random
- until a solution satisfying all the constraints is found, or a pre-set time limit is reached:
  - select a variable that is in conflict i.e. the value assigned to it violates one or more constraints involving one or more other variables
  - assign this variable a value that minimizes the number of conflicts (i.e. attempt to minimize the number of other variables that will need to be repaired)
  - break ties randomly

It is important to have a time-limit on an algorithm of this kind, because in abandoning systematic search, we have abandoned completeness; the algorithm has no means of telling if the problem has no solution. In common with other *local-search* algorithms (i.e. algorithms which repeatedly move from one solution to a 'neighbouring' solution), it may also get stuck in a *local optimum* where there is a neighbourhood of equally good solutions surrounded by worse solutions. The algorithm as defined above will then endlessly loop around this neighbourhood, but if this is not the best solution, the only way to improve is to temporarily choose a worse solution.

One way of avoiding this situation is to run the algorithm a number of times, starting with a different random initial solution each time.

Minton *et al.* claim that the min-conflicts heuristic works well in its original scheduling domain and on problems such as n-queens; they claim for instance that it can easily solve the one million queens problem (this is not all that impressive, apart from the memory problems associated with problems of this size, since the n-queens problem is not especially difficult, and gets relatively easier as n gets larger, because the constraints get looser).

One difficulty with methods of this kind is that they do not integrate easily with constraint propagation. Nevertheless, they may offer the only way of finding a solution of some kind to very difficult or very large problems, when the methods we have considered earlier fail. The latest version of Solver (5.0) does in fact offer local search as an option, as well as complete search.

12 Conclusions

Many different kinds of problem can be expressed as constraint satisfaction prob-

One difficulty with methods of this kind is that they do not integrate easily with constraint propagation. Nevertheless, they may offer the only way of finding a solution of some kind to very difficult or very large problems, when the methods we have considered earlier fail. The latest version of Solver (5.0) does in fact offer local search as an option, as well as complete search.

12 Conclusions

Many different kinds of problem can be expressed as constraint satisfaction problems, and constraint programming tools offer a relatively easy way to express such problems. We have seen a number of algorithms which will guarantee to find a solution (or even an optimal solution) if given enough time. We have also seen that solving a large problem successfully may require careful development of a solution strategy based on insights into how the available methods will go about solving our particular problem; so constraint programming is no panacea for difficult problems. Nevertheless, for the right kind of problem and with a good solution strategy, it can be the best available way to solve the problem by far.

THE END

That’s all for now folks