

DM865 – Spring 2020  
Heuristics and Approximation Algorithms

# Construction Heuristics for Traveling Salesman Problem

Marco Chiarandini

Department of Mathematics & Computer Science  
University of Southern Denmark

# Outline

1. Combinatorial Optimization
2. Solution Approaches
3. TSP
4. Code Speed Up

# Outline

1. Combinatorial Optimization
2. Solution Approaches
3. TSP
4. Code Speed Up

# Aim of the Heuristic Part of the Course

To enable the student to solve **discrete optimization problems**  
that arise in practical applications

# Discrete and Combinatorial Optimization

- **Discrete optimization** emphasizes the difference to continuous optimization, solutions are described by **integer numbers** or **discrete structures**
- Combinatorial optimization is a subset of discrete optimization.
- Combinatorial optimization is the study of the ways **discrete structures** (eg, graphs) can be selected/arranged/combined: Finding an optimal object from a finite set of objects.
- Discrete/Combinatorial Optimization involves finding a way to efficiently allocate resources in mathematically formulated problems.

# Discrete Optimization Problems

## Discrete Optimization problems

They arise in many areas of

Computer Science, Artificial Intelligence, Operations Research....:

- allocating register memory
- planning, scheduling, timetabling
- Internet data packet routing
- protein structure prediction
- auction winner determination
- portfolio selection
- ...

# Discrete Optimization Problems

Simplified models are often used to formalize real life problems

- finding models of propositional formulae (SAT)
- finding variable assignment that satisfy constraints (CSP)
- partitioning graphs or digraphs
- partitioning, packing, covering sets
- finding shortest/cheapest round trips (TSP)
- coloring graphs (GCP)
- finding the order of arcs with minimal backward cost
- ...

## Example Problems

- They are chosen because conceptually concise, intended to illustrate the development, analysis and presentation of algorithms
- Although [real-world problems tend to have much more complex formulations](#), these problems capture their essence

# Elements of Combinatorial Problems

Combinatorial problems are characterized by an **input**, *i.e.*, a general description of **conditions** (or **constraints**) and **parameters**, and a **question** (or **task**, or **objective**) defining the properties of a **solution**.

They involve finding a **grouping**, **ordering**, or **assignment** of a **discrete**, **finite** set of objects that satisfies given conditions.

**Candidate solutions** are combinations of objects or **solution components** that need not satisfy all given conditions. They can be **partial solutions** or **complete solutions**.

**Feasible solutions** are candidate solutions that satisfy all given conditions.

**Optimal Solutions** are feasible solutions that maximize or minimize some criterion or objective function.

**Approximate solutions** are feasible candidate solutions that are not optimal but good in some sense.



# Traveling Salesman Problem

## Traveling Salesman Problem

Given: a weighted complete graph

Output: an Hamiltonian cycle of minimum total weight.

- <http://www.math.uwaterloo.ca/tsp/>
- “platform for the study of general methods that can be applied to a wide range of discrete optimization problems”
- arranging school bus routes to pick up the children in a school district.
- scheduling of service calls at cable firms
- delivery of meals to homebound persons
- scheduling of stacker cranes in warehouses
- scheduling of a machine to drill holes in a circuit board or other object
- routing of trucks for parcel post pickup

# General vs Instance

General problem vs problem instance:

General problem  $\mathcal{P}$ :

- Given *any* set of points  $X$  in a square, find a shortest Hamiltonian cycle
- *Solution*: Algorithm that finds shortest Hamiltonian cycle for any  $X$

Problem instantiation  $\pi$ :

- Given a *specific* set of points  $\pi$  in the square, find a shortest Hamiltonian cycle
- *Solution*: Shortest Hamiltonian cycle for  $\pi$

Problems can be formalized on sets of problem instances  $\Pi$  (*instance classes*)

# Traveling Salesman Problem

## Types of TSP instances:

- Complete vs incomplete graphs
- **Symmetric**: For all edges  $uv$  of the given graph  $G$ ,  $vu$  is also in  $G$ , and  $w_{uv} = w_{vu}$ .  
Otherwise: **asymmetric**.
- **Metric TSP**: symmetric + triangle inequality ( $w_{ij} \leq w_{ik} + w_{kj}$ )
  - **Euclidean**: Vertices = points in an Euclidean space,  
weight function = Euclidean distance metric.
  - **Geographic** (metric TSP): Vertices = points on a sphere,  
weight function = geographic (great circle) distance.

Alternatively, these features can become part of the general problem description and exploited in the development of the solution algorithm

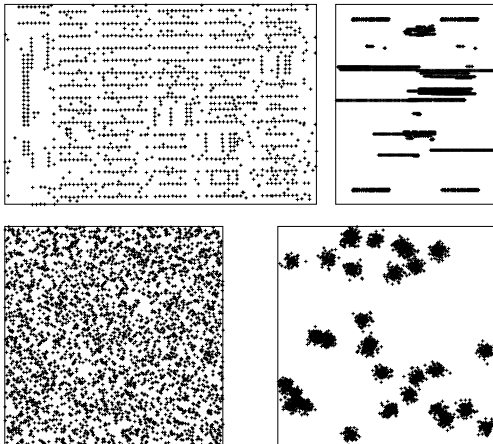
# TSP: Benchmark Instances

## Instance classes

- Real-life applications (geographic, VLSI)
- Random Euclidean
- Random Clustered Euclidean
- Random Distance

Available at the TSPLIB (more than 100 instances upto 85.900 cities)  
and at the 8th DIMACS challenge

# TSP: Instance Examples



# Complete Algorithms and Lower Bounds

## Reference Results

- Branch & cut algorithms (Concorde: <http://www.math.uwaterloo.ca/tsp/concorde>)
  - cutting planes + branching
  - use LP-relaxation for lower bounding schemes
  - effective heuristics for upper bounds

Solution times with Concorde		
Instance	Computing nodes	CPU time (secs)
att532	7	109.52
rat783	1	37.88
pcb1173	19	468.27
fl1577	7	6705.04
d2105	169	11179253.91
pr2392	1	116.86
rl5934	205	588936.85
usa13509	9539	ca. 4 years
d15112	164569	ca. 22 years
s24978	167263	84.8 CPU years

- Lower bounds: (within less than one percent of optimum for random Euclidean, up to two percent for TSPLIB instances)

# Outline

1. Combinatorial Optimization
2. Solution Approaches
3. TSP
4. Code Speed Up

# Enumeration

Good way to start approaching a problem:

- Make a small example and a drawing of the problem
- Represent a solution: decision variables, data structures.  
For the TSP: permutation as array of all different values corresponds to cycle notation, alternative notation: Cauchy's notation or (node images)
- Enumerate all possible solutions and determine the optimal solution
- For TSP: solution representation is a permutation of vertices, construct all possible permutations by, for example, tree search.  
Consider which parts of the tree can be spared.
  - Rotating permutations: keep starting node fixed
  - Symmetric permutations

Overall complexity  $O((n-1)!/2)$



# Held Karp Algorithm: Dynamic Programming

- Consider the problem as a multistage decision problem.
- fix the origin at some city, say 0 (wlog).
- Suppose that at a certain stage of an optimal tour starting at 0 one has reached a city  $i$  and there remain  $k$  cities  $j_1, j_2, \dots, j_k$  to be visited before returning to 0.
- **Principle of Optimality** for the tour being optimal, the path from  $i$  through  $j_1, j_2, \dots, j_k$  in some order and then to 0 must be of minimum length (if not the entire tour could not be optimal, since its total length could be reduced by choosing a shorter path from  $i$  through  $j_1, j_2, \dots, j_k$  to 0).
- $f(i; \{j_1, j_2, \dots, j_k\})$  length of a path of minimal length from  $i$  to 0 which passes exactly once through each of the remaining  $k$  unvisited cities  $j_1, j_2, \dots, j_k$
- $f(0; \{j_1, j_2, \dots, j_n\})$  is the solution to the problem

- Recursive relation

$$f(i; \{j_1, j_2, \dots, j_k\}) = \min_{1 \leq m \leq k} \{d_{ij_m} + f(j_m; \{j_1, j_2, \dots, j_{m-1}, j_{m+1}, \dots, j_k\})\}$$

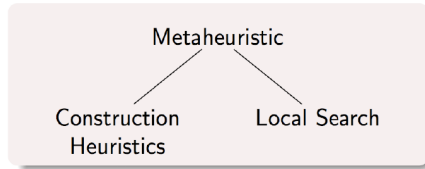
- $f(i; \{j\}) = d_{ij} + d_{j0}$
- $f(i; \{j_1, j_2\}) = \min_{j_1, j_2} \{d_{ij_1} + f(j_1; \{j_2\}), d_{ij_2} + f(j_2; \{j_1\})\}$
- $n2^n$  values  $f(i; j_1, j_2, \dots, j_k)$  to calculate  
each value costs up to  $n$  operations if previous values available  
Overall time complexity:  $O(n^2 2^n)$ ; memory usage  $O(n 2^n)$ .
- This was a backward implementation. See wikipedia for a forward implementation and a numerical example

# Heuristics

Get inspired by approach to problem solving in human mind

[A. Newell and H.A. Simon. "Computer science as empirical inquiry: symbols and search." Communications of the ACM, ACM, 1976, 19(3)]

- effective rules without theoretical support
- trial and error



Applications:

- Optimization
- But also in Psychology, Economics, Management [Tversky, A.; Kahneman, D. (1974). "Judgment under uncertainty: Heuristics and biases". Science 185]

Basis on empirical evidence rather than mathematical logic. Getting things done in the given time. 19

# Outline

1. Combinatorial Optimization
2. Solution Approaches
3. TSP
4. Code Speed Up

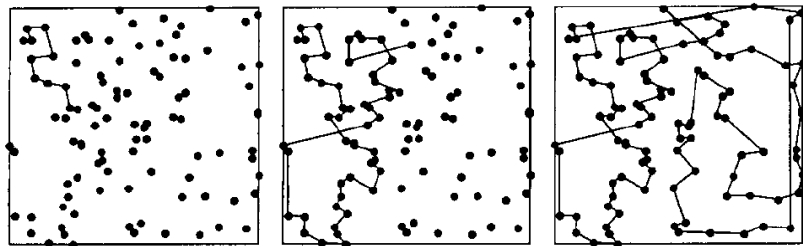
# Construction Heuristics for TSP

Construction heuristics specific for TSP

- Heuristics that Grow Fragments
  - Nearest neighborhood heuristics
  - Double-Ended Nearest Neighbor heuristic
  - Multiple Fragment heuristic (aka, greedy heuristic)
- Heuristics that Grow Tours
  - Nearest Addition
  - Farthest Addition
  - Random Addition
  - Clarke-Wright savings heuristic
  - Nearest Insertion
  - Farthest Insertion
  - Random Insertion
- Heuristics based on Trees
  - Minimum spanning tree heuristic
  - Christofides' heuristics
  - Fast recursive partitioning heuristic

# Nearest Neighbor Heuristic

[Bentley, 1992]



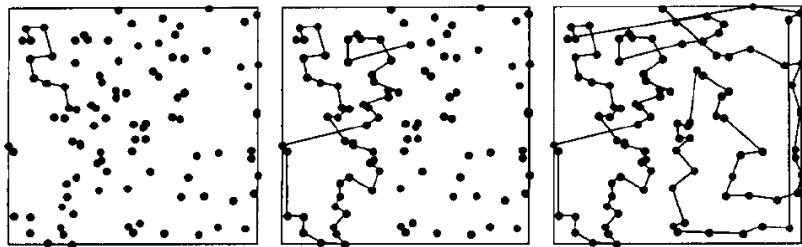
**Figure 1.** The Nearest Neighbor heuristic.

NN (Flood, 1956)

- 1 Randomly select a starting node
- 2 Add to the last node the closest node until no more nodes are available
- 3 Connect the last node with the first node

Running time  $O(N^2)$

## Nearest Neighbor Heuristic



**Figure 1.** The Nearest Neighbor heuristic.

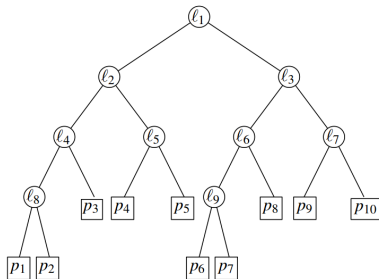
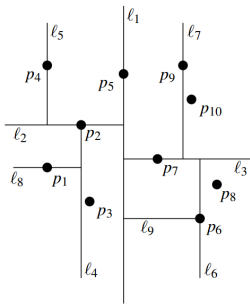
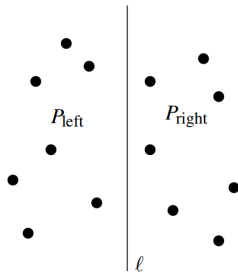
- In geometric instances:  $NN < \frac{(\lceil \log N \rceil + 1)}{2} \cdot OPT$
- Double-Ended NN

# Nearest Neighbor Heuristic

```
Build(PtSet)
Perm[1]:=StartPt
DeletePt(Perm[1])
for i:=2 to N do
  | Perm[i]:=NN(Perm[i-1])
  | DeletePt(Perm[i])
```

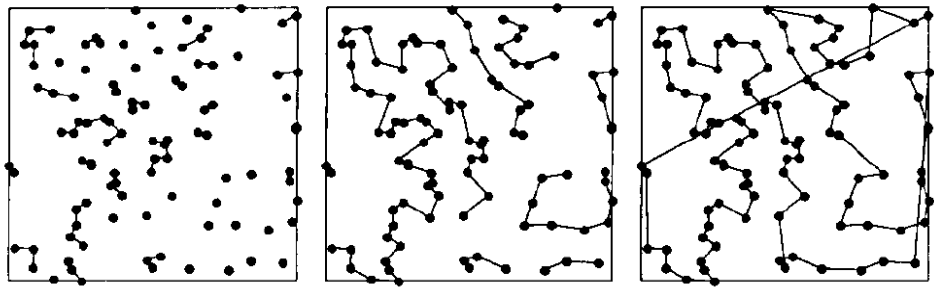


# Data Structure: *kd-tree*



## Data Structure: *kd-tree*

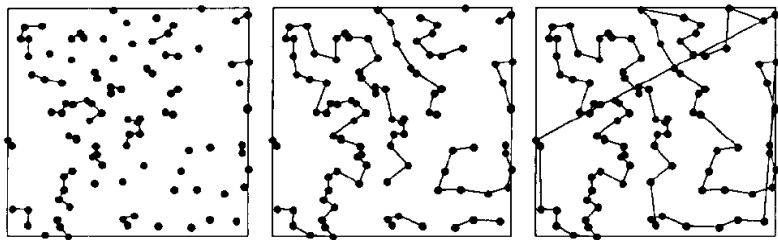
- Construction in  $O(n \log n)$  time and  $O(n)$  space
- Range search: reports the leaves from a split node.
- Delete(PointNum) amortized constant time
- NearestNeighbor(PointNum) bottom-up search  
visit nodes + compute distances  
 $A + BN^C$ ,  $A > 0, B < 0, -1 < C < 0$  (expected constant time) if no deletions happened and data uniform
- FixedRadiusNearestNeighbor(PointNum, Radius, function)
- BallSearch(PointNum, function) ball centered at point
- SetRadius(PointNum, float Radius)
- SphereOfInfluence(PointNum, float Radius) ball centered at point with given radius



**Figure 5.** The Multiple Fragment heuristic.

## Greedy Heuristic for TSP

[Bentley, 1992]



**Figure 5.** The Multiple Fragment heuristic.

- Add the cheapest edge provided it does not create a cycle.
- $O(\sqrt{N})$  approximation

# Greedy Heuristic: Implementation Details

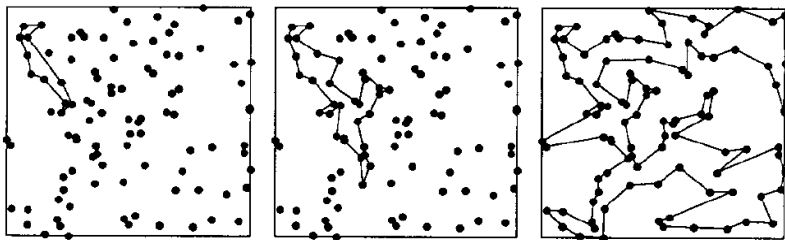
- Array Degree num. of tour edges
- K-*d* tree for nearest neighbor searching (only eligible nodes)
- Array NNLink containing index to nearest neighbor of *i* not in the fragment of *i*
- Priority queue (heap) with nearest neighbor links
- Array Tail link to the other tail of current fragments.

# Important Elements

- Exploit the locality inherent in the problem to solve it (NN search, Fixed-radius search, ball search)
- Search time modelled by a function  $A + BN^C$
- Number of searches
- Priority queue of links to nearest neighbors

## Addition Heuristics

[Bentley, 1992]



**Figure 8.** The Nearest Addition heuristic.

NA

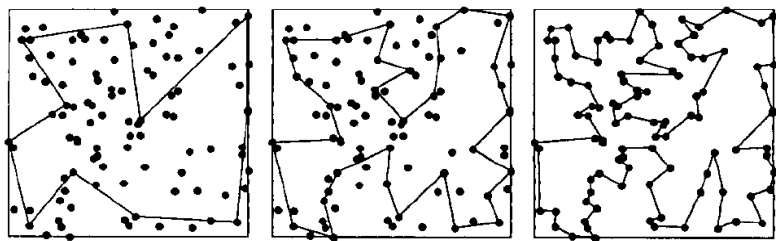
- ① Select a node and its closest node and build a tour of two nodes
- ② Insert in the tour the closest node  $v$  until no more node are available

Tour maintained as a double lined list

Running time  $O(N^3)$

## Addition Heuristics

[Bentley, 1992]



**Figure 11.** The Farthest Addition heuristic.

FA

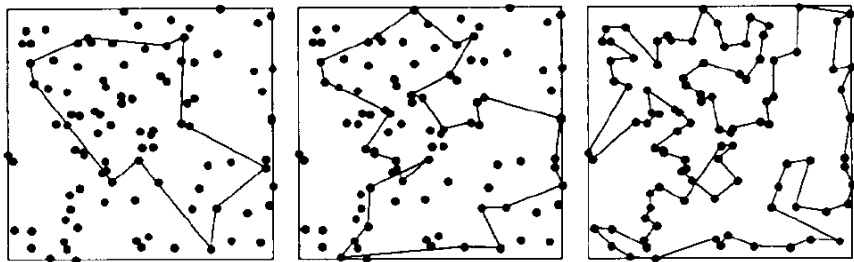
- ① Select a node and its farthest and build a tour of two nodes
- ② Insert in the tour the farthest node  $v$  until no more node are available

FA is more effective than NA because the first few farthest points sketch a broad outline of the tour that is refined after.

Running time  $O(N^3)$



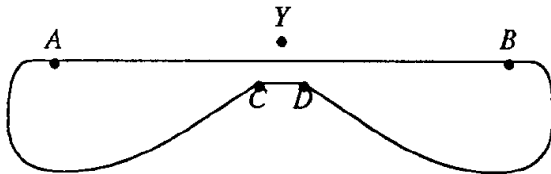
## Addition Heuristics



**Figure 14.** The Random Addition heuristic.

# Insertion Heuristics

Motivation:



- $A$  and  $B$  are far from  $Y$  relative to the distance from  $Y$ 's nearest neighbor
- $Y$  is near to  $A$  relative to the length of the edge  $AB$ .

**Nearest Neighbor-ball** at a point  $Y$  with scale  $S$  is a ball centered at  $Y$  with radius  $S$  times the distance from  $Y$  to its nearest neighbor among the points in the tour (eg,  $D(Y, C)$ ).

**Sphere of influence** at tour vertex  $A$  with scale  $S$  is a ball centered at  $A$  with radius  $S$  times the length of the longer edge adjacent to  $A$  (eg,  $D(A, B)$ ).

## Theorem

$Y$  not yet in tour

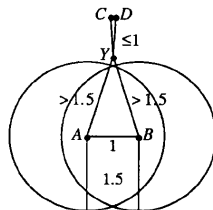
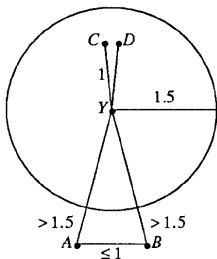
$C$  nearest neighbor of  $Y$

$D$  neighbor of  $C$  in tour that minimize  $C(Y, CD)$

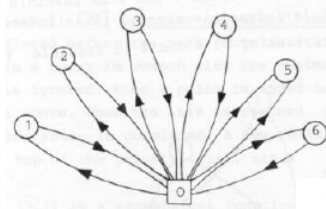
There exists an edge  $AB$  such that  $C(Y, AB) < C(Y, CD)$  only if one of the following is true:

- $D(A, B) \leq D(Y, C)$  and  $A$  or  $B$  is in  $Y$ 's nearest-neighbor-ball with scale 1.5
- $D(A, B) \geq D(Y, C)$  and  $Y$  is in  $A$  or  $B$ 's sphere of influence with scale 1.5

Proof:  $C(Y, CD) \leq 2D(Y, C)$



# Saving Heuristic



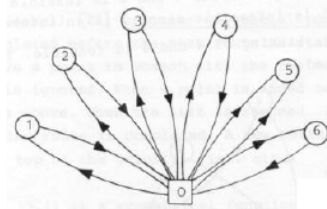
## Clarke-Wright Saving Heuristic (1964)

- 1 Start with an initial allocation of one vehicle to each customer (0 is the depot for VRP or any chosen city for TSP)

### Sequential:

2. consider in turn route  $(0, i, \dots, j, 0)$  determine  $s_{ki}$  and  $s_{jl}$
3. merge with  $(k, 0)$  or  $(0, l)$

# Saving Heuristic



## Clarke-Wright Saving Heuristic (1964)

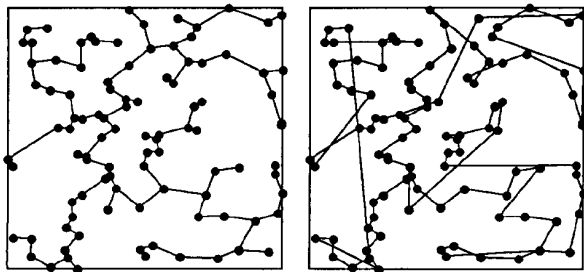
- 1 Start with an initial allocation of one vehicle to each customer (0 is the depot for VRP or any chosen city for TSP)

Parallel:

2. Calculate saving  $s_{ij} = c_{0i} + c_{0j} - c_{ij}$  and order the saving in non-increasing order
3. scan  $s_{ij}$ 
  - i)  $i$  and  $j$  are not in the same tour
  - ii) neither  $i$  and  $j$  are interior to an existing route
  - iii) vehicle and time capacity are not exceeded

# Minimum Spanning Tree Heuristics

[Bentley, 1992]



**Figure 18.** The Minimum Spanning Tree heuristic.

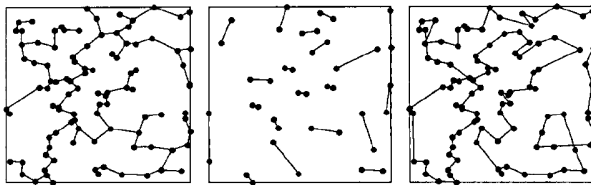
- ① Find a minimum spanning tree  $O(N^2)$
- ② Append the nodes in the tour in a depth-first, inorder traversal

Running time  $O(N^2)$

$$A = MST/OPT \leq 2$$

# Christofides' Heuristic

[Bentley, 1992]



**Figure 19.** Christofides' heuristic.

- 1 Find the minimum spanning tree  $T$ .  $O(N^2)$
- 2 Find nodes in  $T$  with odd degree and find the cheapest perfect matching  $M$  in the complete graph consisting of these nodes only. Let  $G$  be the multigraph of all nodes and edges in  $T$  and  $M$ .  $O(N^3)$
- 3 Find an Eulerian walk (each node appears at least once and each edge exactly once) on  $G$  and an embedded tour.  $O(N)$

Running time  $O(N^3)$

$A = CH/OPT \leq 3/2$  tight and best known

# Outline

1. Combinatorial Optimization
2. Solution Approaches
3. TSP
4. Code Speed Up



# Where do speedups come from?

Where can maximum speedup be achieved?  
How much speedup should you expect?

# Code Tuning

- Caution: proceed carefully! Let the optimizing compiler do its work!
  - optimizing flags
  - just-in-time-compilation: it converts code at runtime prior to executing it natively, for example bytecode into native machine code. (module numba [https://www.ibm.com/developerworks/community/blogs/jfp/entry/Fast\\_Computation\\_of\\_AUC\\_ROC\\_score?lang=en](https://www.ibm.com/developerworks/community/blogs/jfp/entry/Fast_Computation_of_AUC_ROC_score?lang=en))
- Caching, memoization (`@functools.lru_cache(None)`)
- Profiling (module `cProfile`)

- Expression Rules: Recode for smaller instruction counts.
- Loop and procedure rules: Recode to avoid loop or procedure call overhead.
- Hidden costs of high-level languages
- String comparisons: proportional to length of the string, not constant
- Object construction / de-allocation: very expensive
- Matrix access: row-major order  $\neq$  column-major order
- Exploit algebraic identities
- Avoid unnecessary computations inside the loops

# Where Speedups Come From?

McGeoch reports conventional wisdom, based on studies in the literature.

- Concurrency is tricky: bad -7x to good 500x
- Classic algorithms: to 1trillion and beyond
- Data-aware: up to 100x
- Memory-aware: up to 20x
- Algorithm tricks: up to 200x
- Code tuning: up to 10x
- Change platforms: up to 10x