DM63
HEURISTICS FOR
COMBINATORIAL OPTIMIZATION PROBLEMS

Lecture 1

Introduction
and
Heuristics for the Travelling Salesman Problem

Marco Chiarandini

Outline

Course Presentation

- Communication media
  - Web-site `http://www.imada.sdu.dk/~marco/DM63/` (The Blackboard is redirected to this URL address)
  - Course mailing list (compulsory registration from the web site)
  - Personal email
- Schedule: Monday 8.15, Thursday 10.15 (Weeks 36-45. No lectures in week 37.) Last lecture: Monday, November 6
- Course content
- Evaluation: final project
  - Implementation of metaheuristics and experimentation.
  - Individual work on a commonly posed problem,
  - The final report will be evaluated by an external examiner.

Course Presentation (2)

- Literature
  - Notes available from the Book Store
  - Articles and Books referenced from the website
  - ...but take notes in class!
- Course resources (programming languages: C, C++, Java)
- Assignments
  - TSP competition
    - Submit a program which implements the required task
    - Details on the Weekly Notes
  - Class exercises
- Weekly notes and slides

Combinatorial Problems

Combinatorial problems arise in many areas of Computer Science, Artificial Intelligence and Operations Research:

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

Combinatorial Problems (2)

Simplified models are often used to formalize real life problems

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

Combinatorial Problems (3)

Combinatorial problems are characterized by an input, *i.e.*, a general description of conditions 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.

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

Combinatorial Problems (4)

Example:

- *Input:* a set of points in the Euclidean plane
- *Task:* find the shortest Hamiltonian cycle

Note:

- *solution component:* segment consisting of two points that are visited one directly after the other
- *candidate solution:* one of the $(n-1)!$ possible sequences of points to visit one directly after the other.
- *solution:* Hamiltonian cycle of minimal length

Combinatorial Problems (5)

General problem *vs* problem instance:

General problem Π:

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

Problem instantiation $\pi = \Pi(I)$:

- Given *a specific* set of points I, find a shortest Hamiltonian cycle
- *Solution:* Shortest Hamiltonian cycle for I

Problems can be formalized on sets of problem instances $\mathcal{I}$

Decision problems

solutions = candidate solutions that satisfy given *logical conditions*

Two variants:

- Search variant: Find a solution for given problem instance (or determine that no solution exists)
- Existence variant: Determine whether solutions for given problem instance exists

Optimization problems

- objective function f measures solution quality (often defined on all candidate solutions)
- find solution with optimal quality, *i.e.*, minimize/maximize f

Variants of optimization problems:

- Search variant: Find a solution with optimal objective function value for given problem instance
- Evaluation variant: Determine optimal objective function value for given problem instance

Remarks

- Every optimization problem has *associated decision problems*: Given a problem instance and a fixed solution quality bound b, find a solution with objective function value $\leq b$ (for minimization problems) or determine that no such solution exists.
- Many optimization problems have an objective function as well as logical conditions, constraints that solutions must satisfy.
- A candidate solution is called feasible (or valid) iff it satisfies the given logical conditions.
- *Note:* Logical conditions can always be captured by an objective function such that feasible candidate solutions correspond to solutions of an associated decision problem with a specific bound.

Computational Complexity

Fundamental question:

How hard is a given computational problems to solve?

Important concepts:

- Time complexity of a problem Π: Computation time required for solving a given instance π of Π using the most efficient algorithm for Π.
- Worst-case time complexity: Time complexity in the worst case over all problem instances of a given size, typically measured as a function of instance size, neglecting constants and lower-order terms ($O(...)$ upper, $\Theta(...)$ tight, $\Omega(...)$ lower).

Computation Complexity

Equivalent Notions
Consider Decision Problems

- A problem Π is in $\mathcal{P}$ if $\exists$ algorithm A that finds a solution in polynomial time.
- in $\mathcal{NP}$ if $\exists$ verification algorithm $A(s,k)$ that verifies a binary certificate (whether it is a solution to the problem) in polynomial time.
- Polynomial time reduction formally shows that one problem Π_1 is at least as hard as another Π_2, to within a polynomial factor. (there exists a polynomial time transformation) $\Pi_2 \leq_P \Pi_1 \Rightarrow \Pi_2$ is no more than a polynomial harder than Π_1.
- Π_1 is in $\mathcal{NP}$-complete if
  1. $\Pi_1 \in \mathcal{NP}$
  2. $\forall \Pi_2 \in \mathcal{NP}\ \Pi_2 \leq_P \Pi_1$
- If Π_1 satisfies property 2, but not necessarily property 1, we say that it is $\mathcal{NP}$-hard:

Important concepts (continued):

- $\mathcal{NP}$: Class of problems that can be solved in polynomial time by a non-deterministic machine.

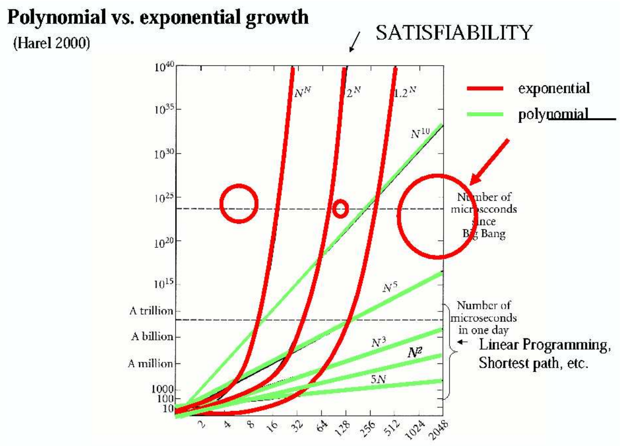
  Note: non-deterministic $\neq$ randomized; non-deterministic machines are idealized models of computation that have the ability to make perfect guesses.
- $\mathcal{NP}$-complete: Among the most difficult problems in $\mathcal{NP}$; believed to have at least exponential time-complexity for any realistic machine or programming model.
- $\mathcal{NP}$-hard: At least as difficult as the most difficult problems in $\mathcal{NP}$, but possibly not in $\mathcal{NP}$ (*i.e.*, may have even worse complexity than $\mathcal{NP}$-complete problems).

Application Scenarios

Practically solving hard combinatorial problems:

- Subclasses can often be solved efficiently (*e.g.*, 2-SAT);
- Average-case *vs* worst-case complexity (*e.g.* Simplex Algorithm for linear optimization);
- Approximation of optimal solutions: sometimes possible in polynomial time (*e.g.*, Euclidean TSP), but in many cases also intractable (*e.g.*, general TSP);
- Randomized computation is often practically (and possibly theoretically) more efficient;
- Asymptotic bounds *vs* true complexity: constants matter!

An online compendium on the computational complexity of optimization problems:
`http://www.nada.kth.se/~viggo/problemlist/compendium.html`



Methods and Algorithms

A Method is a general framework for the development of a solution algorithm. It is not problem-specific.

An Algorithmic model (or simply algorithm) is the instantiation of a method on a specific problem Π.
The level of instantiation may vary:

- minimally instantiated (few details, algorithm template)
- lowly instantiated (which data structure to use)
- highly instantiated (programming tricks that give speedups)
- maximally instantiated (details specific of a programming language and computer architecture)

A Program is the implementation of an algorithm.

Solution Methods

- Exact methods:
  systematic enumeration
  complete: guaranteed to eventually find (optimal) solution, or to determine that no solution exists
  - Search algorithms
  - Dynamic programming
  - Constraint programming
  - Integer programming
- Approximate methods:
  incomplete: not guaranteed to find (optimal) solution, and unable to prove that no solution exists
  - Integer programming relaxations
  - Randomized backtracking
  - Heuristic algorithms
- Approximation methods
  worst-case solution guarantee
  `http://www.nada.kth.se/~viggo/problemlist/compendium.html`

Complete Search Paradigms

Tree search

- uninformed search: breadth first, depth first
- informed search: greedy best-first search, A* search, branch & bound
  Example: branch & bound / A* search for TSP
  - Compute lower bound on length of completion of given partial round trip.
  - Terminate search on branch if length of current partial round trip + lower bound on length of completion exceeds length of shortest complete round trip found so far.
- Combination of constructive search and backtracking, *i.e.*, revisiting of choice points after construction of complete candidate solutions.
- Performs *systematic search* over constructions.
- *Complete*, but visiting all candidate solutions becomes rapidly infeasible with growing size of problem instances.

Incomplete Search Paradigms

Heuristic: a commonsense rule (or set of rules) intended to increase the probability of solving some problem

Construction rules (aka construction heuristics)

They are closely related to search tree techniques but correspond to a single path from root to leaf

- search space = partial candidate solutions
- search step = extension with one or more solution components

Construction Heuristic (CH):

$s := \emptyset$
While s is not a complete solution:
  choose a solution component c
  add the solution component to s

Incomplete Search Paradigms

An important class of Construction Heuristics are greedy algorithms.

- Strategy: always make the choice which is the best at the moment.
- They are not generally guaranteed to find globally optimal solutions (but sometimes they do: Minimum Spanning Tree, Single Source Shortest Path, etc.)

The Traveling Salesman Problem

- *Given:* Directed, edge-weighted graph G (complete and with triangle inequality).
- *Objective:* Find a minimal-weight Hamiltonian cycle in G.

Types of TSP instances:

- Symmetric: For all edges uv of the given graph G, vu is also in G, and $w(uv) = w(vu)$.
  Otherwise: asymmetric.
- Euclidean: Vertices = points in an Euclidean space, weight function = Euclidean distance metric.
- Geographic: Vertices = points on a sphere, weight function = geographic (great circle) distance.

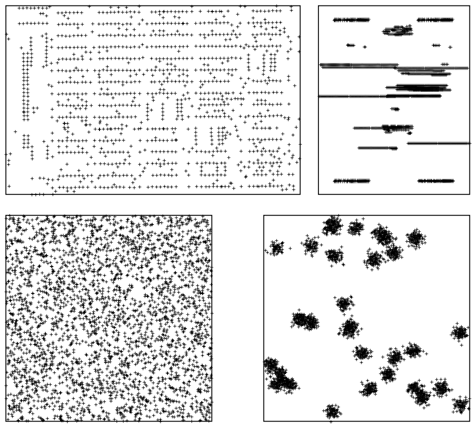
TSP: Benchmark Instances

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: Benchmark Instances, Examples



Complete Algorithms and Lower Bounds

- Branch & cut algorithms (Concorde: http://www.tsp.gatech.edu/)
  - cutting planes + branching
  - use LP-relaxation for lower bounding schemes
  - effective heuristics for upper bounds

Solution times with Concorde

Instance	No. 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)

Construction Heuristics

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
  - Nearest Insertion
  - Farthest Insertion
  - Random Insertion
  - Clarke-Wright savings heuristic
- Heuristics based on Trees
  - Minimum span tree heuristic
  - Christofides' heuristics
  - Fast recursive partitioning heuristic

Software Development: Extreme Programming & Scrum

Planning

Release planning creates the schedule // Make frequent small releases // The project is divided into iterations

Designing

Simplicity // No functionality is added early // Refactor: eliminate unused functionality and redundancy

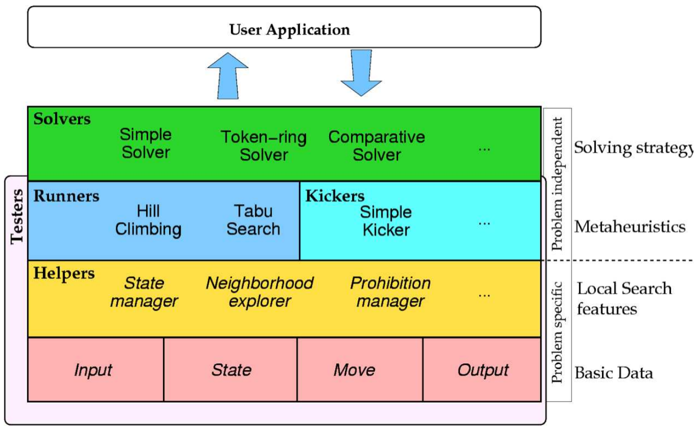
Coding

Code must be written to agreed standards // Code the unit test first // All production code is pair programmed // Leave optimization till last // No overtime

Testing

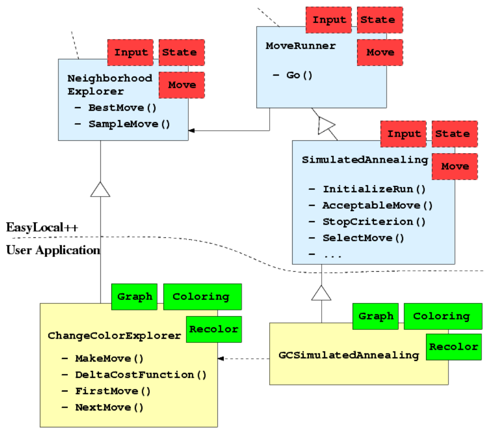
All code must have unit tests // All code must pass all unit tests before it can be released // When a bug is found tests are created

Software Framework for LS Methods



From EasyLocal++ by Schaerf and Di Gaspero (2003).

Software Framework for LS Methods



From EasyLocal++ by Schaerf and Di Gaspero (2003).