DM63
HEURISTICS FOR
COMBINATORIAL OPTIMIZATION PROBLEMS

Lecture 1
Combinatorial Optimization Problems

Marco Chiarandini

Outline
1. Course Introduction
2. Combinatorial Problems
3. Computational Complexity
4. Solution Methods
5. Construction Heuristics for the Traveling Salesman Problem
6. Software Development

Course Presentation
▶ Communication media
▶ Web-site http://www.imada.sdu.dk/~marco/DM63/
   (The Blackboard is redirected to this URL address)
▶ Course mailing list (please register!)
▶ Personal email

▶ Schedule: Monday 14.00, Wednesday 14.00
   (No lectures in week 41)
   Last lecture: Monday, November 5

▶ 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
  - Text Book
  - Articles and Chapters available from the website
  - …but take notes in class!
- Assignments
  - GCP Contest
    - Submit a program which accomplishes the required task
    - Details on the Weekly Notes
  - Class exercises
- Weekly notes and slides
- Students’ notes on lecture content

Course Presentation (3)

Why participating to the implementation Contest?
- to prepare the work for the project
  - algorithmic framework
  - experimental environment
- to address details that might be gone unsaid at the lecture
  (Computer Science is about implementation details!)
- to contribute to the part on experimental analysis
- for the challenge

Outline

1. Course Introduction
2. Combinatorial Problems
3. Computational Complexity
4. Solution Methods
5. Construction Heuristics for the Traveling Salesman Problem
6. Software Development

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)
▶ coloring graphs (GCP)
▶ finding variable assignment which satisfy constraints (CSP)
▶ partitioning graphs or digraphs
▶ partitioning, packing, covering sets
▶ finding the order of arcs with minimal backward 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)

The Traveling Salesman Problem

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

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

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.

---

Combinatorial Problems (5)

**General problem vs problem instance:**

**General problem** $\Pi$:

- 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 $I$

---

The Traveling Salesman Problem

**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 up to 85,900 cities) and at the 8th DIMACS challenge.

---

### The SAT Problem

**General SAT Problem (search variant):**
- **Given:** Formula $F$ in propositional logic
- **Objective:** Find an assignment of truth values to variables in $F$ that renders $F$ true, or decide that no such assignment exists.

**SAT: A simple example**
- **Given:** Formula $F := (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2)$
- **Objective:** Find an assignment of truth values to variables $x_1, x_2$ that renders $F$ true, or decide that no such assignment exists.

**MAX-SAT**
Which is the maximal number of clauses satisfiable in a propositional logic formula $F$?
Definition:

- **Formula in propositional logic**: well-formed string that may contain
  - propositional variables $x_1, x_2, \ldots, x_n$;
  - truth values $\top$ (‘true’), $\bot$ (‘false’);
  - operators $\neg$ (‘not’), $\land$ (‘and’), $\lor$ (‘or’);
  - parentheses (for operator nesting).

- **Model (or satisfying assignment)** of a formula $F$: Assignment of truth values to the variables in $F$ under which $F$ becomes true (under the usual interpretation of the logical operators).

- Formula $F$ is **satisfiable** iff there exists at least one model of $F$, unsatisfiable otherwise.
Definition:

- A formula is in conjunctive normal form (CNF) iff it is of the form

\[ \bigwedge_{i=1}^{m} \bigvee_{j=1}^{k(i)} l_{ij} = (l_{11} \lor \ldots \lor l_{1k(1)}) \ldots \lor (l_{m1} \lor \ldots \lor l_{mk(m)}) \]

where each literal \( l_{ij} \) is a propositional variable or its negation. The disjunctions \( (l_{i1} \lor \ldots \lor l_{ik(i)}) \) are called clauses.

- A formula is in \( k \)-CNF iff it is in CNF and all clauses contain exactly \( k \) literals (i.e., for all \( i \), \( k(i) = k \)).

- In many cases, the restriction of SAT to CNF formulae is considered.
- The restriction of SAT to \( k \)-CNF formulae is called \( k \)-SAT.
- For every propositional formula, there is an equivalent formula in 3-CNF.

Example:

\[ F := \land (\neg x_2 \lor x_1) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (x_1 \lor x_2) \land (\neg x_4 \lor x_3) \land (\neg x_5 \lor x_3) \]

- \( F \) is in CNF.
- Is \( F \) satisfiable?

Yes, e.g., \( x_1 := x_2 := \top, x_3 := x_4 := x_5 := \bot \) is a model of \( F \).

The Vertex Coloring Problem

Given: A graph \( G \) and a set of colors \( \Gamma \).

A proper coloring is an assignment of one color to each vertex of the graph such that adjacent vertices receive different colors.

Decision version (\( k \)-coloring)

Task: Find a proper coloring of \( G \) which uses at most \( k \) colors.

Optimization version (chromatic number)

Task: Find a proper coloring of \( G \) which uses the minimal number of colors.
Outline

1. Course Introduction
2. Combinatorial Problems
3. Computational Complexity
4. Solution Methods
5. Construction Heuristics for the Traveling Salesman Problem
6. Software Development

Computational Complexity

Fundamental question:
How hard is a given computational problem to solve?

Important concepts:

- **Time complexity of a problem II**: Computation time required for solving a given instance \( \pi \) of II using the most efficient algorithm for II.
- **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).

Important concepts (continued):

- **NP**: Class of problems that can be solved in polynomial time by a non-deterministic machine.

Equivalent Notions
Consider Decision Problems

- A problem II 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 \( \Pi_1 \) is at least as hard as another \( \Pi_2 \), to within a polynomial factor. (there exists a polynomial time transformation) \( \Pi_2 \leq \mathcal{P} \Pi_1 \Rightarrow \Pi_2 \) is no more than a polynomial harder than \( \Pi_1 \).
- \( \Pi_1 \) is in \( \mathcal{NP} \)-complete if
  1. \( \Pi_1 \in \mathcal{NP} \)
  2. \( \forall \Pi_2 \in \mathcal{NP} \Pi_2 \leq \mathcal{P} \Pi_1 \)
- If \( \Pi_1 \) satisfies property 2, but not necessarily property 1, we say that it is \( \mathcal{NP} \)-hard:

- **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.
- **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).
Important concepts (continued):

- **NP**: Class of problems that can be solved in polynomial time by a non-deterministic machine. 
  
  *Note*: non-deterministic ≠ randomized; non-deterministic machines are idealized models of computation that have the ability to make perfect guesses.

- **NP-complete**: Among the most difficult problems in *NP*; believed to have at least exponential time-complexity for any realistic machine or programming model.

- **NP-hard**: At least as difficult as the most difficult problems in *NP*, but possibly not in *NP* (i.e., may have even worse complexity than *NP*-complete problems).

**Many combinatorial problems are hard but some problems can be solved efficiently**

- Longest path problem is *NP*-hard but not shortest path problem
Many combinatorial problems are hard but some problems can be solved efficiently

- Longest path problem is $\mathcal{NP}$-hard but not shortest path problem
- SAT for 3-CNF is $\mathcal{NP}$-complete but not 2-CNF (linear time algorithm)
- TSP is $\mathcal{NP}$-hard, the associated decision problem (for any solution quality) is $\mathcal{NP}$-complete but not the Euler tour problem
- TSP on Euclidean instances is $\mathcal{NP}$-hard but not where all vertices lie on a circle.

Application Scenarios

Practically solving hard combinatorial problems:

- Average-case vs worst-case complexity (e.g. Simplex Algorithm for linear optimization);
Practically solving hard combinatorial problems:

- 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
Outline

1. Course Introduction
2. Combinatorial Problems
3. Computational Complexity
4. Solution Methods
5. Construction Heuristics for the Traveling Salesman Problem
6. Software Development

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 $\Pi$.

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.

Incomplete Search Paradigms

Heuristic: a common-sense 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:
  \[ \text{choose a solution component } c \]
  \[ \text{add the solution component to } s \]
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.)

Outline

1. Course Introduction
2. Combinatorial Problems
3. Computational Complexity
4. Solution Methods
5. Construction Heuristics for the Traveling Salesman Problem
6. Software Development

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

<table>
<thead>
<tr>
<th>Instance</th>
<th>No. nodes</th>
<th>CPU time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>att532</td>
<td>7</td>
<td>109.52</td>
</tr>
<tr>
<td>rat783</td>
<td>1</td>
<td>37.88</td>
</tr>
<tr>
<td>pcb1173</td>
<td>19</td>
<td>468.27</td>
</tr>
<tr>
<td>fi1577</td>
<td>7</td>
<td>6705.04</td>
</tr>
<tr>
<td>d2105</td>
<td>169</td>
<td>11179253.91</td>
</tr>
<tr>
<td>pr2392</td>
<td>1</td>
<td>116.86</td>
</tr>
<tr>
<td>rl5934</td>
<td>205</td>
<td>588936.85</td>
</tr>
<tr>
<td>usa13509</td>
<td>9539</td>
<td>ca. 4 years</td>
</tr>
<tr>
<td>d15112</td>
<td>164569</td>
<td>ca. 22 years</td>
</tr>
<tr>
<td>s24978</td>
<td>167263</td>
<td>84.8 CPU years</td>
</tr>
</tbody>
</table>

- 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
  - Clarke-Wright savings heuristic

- **Heuristics based on Trees**
  - Minimum span tree heuristic
  - Christofides’ heuristics
  - Fast recursive partitioning heuristic

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

<table>
<thead>
<tr>
<th>Instance</th>
<th>No. nodes</th>
<th>CPU time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>att532</td>
<td>7</td>
<td>109.52</td>
</tr>
<tr>
<td>rat783</td>
<td>1</td>
<td>37.88</td>
</tr>
<tr>
<td>pcb1173</td>
<td>19</td>
<td>468.27</td>
</tr>
<tr>
<td>fi1577</td>
<td>7</td>
<td>6705.04</td>
</tr>
<tr>
<td>d2105</td>
<td>169</td>
<td>11179253.91</td>
</tr>
<tr>
<td>pr2392</td>
<td>1</td>
<td>116.86</td>
</tr>
<tr>
<td>rl5934</td>
<td>205</td>
<td>588936.85</td>
</tr>
<tr>
<td>usa13509</td>
<td>9539</td>
<td>ca. 4 years</td>
</tr>
<tr>
<td>d15112</td>
<td>164569</td>
<td>ca. 22 years</td>
</tr>
<tr>
<td>s24978</td>
<td>167263</td>
<td>84.8 CPU years</td>
</tr>
</tbody>
</table>

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

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

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

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

**Figure 11.** The Farthest Addition heuristic.
Construction Heuristics for TSP

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

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

**Figure 19.** Christofides' heuristic.

**Outline**

1. Course Introduction
2. Combinatorial Problems
3. Computational Complexity
4. Solution Methods
5. Construction Heuristics for the Traveling Salesman Problem
6. Software Development
**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

---

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