DM811
HEURISTICS AND LOCAL SEARCH ALGORITHMS
FOR COMBINATORIAL OPTIMZATION

Lecture 1

Introduction, Terminology, Combinatorial Probelms

Marco Chiarandini

Outline

Outline

Course Presentation

- Communication media
  - Blackboard (alternatively, `http://www.imada.sdu.dk/~marco/DM811/`)
  - Lecture diary (internal to Blackboard)
  - Personal email
- Schedule:
  - Monday 10-12
  - Thursday 8-10

  Last lecture: Monday, 9th October, 2008
- Course content
- Evaluation: final project (internal examiner)
  - Individual work on a commonly posed problem,
  - Implementation of heuristic algorithms and experimentation,
  - Final report

Course Material

- Books:
  - B1 *Constraint-Based Local Search*, P. Van Hentenryck and L. Michel. The MIT Press (2005).
  - B2 *Stochastic Local Search: Foundations and Applications*, H. Hoos and T. StÃijtzle, 2005, Morgan Kaufmann
  - B3 *Handbook of Approximation Algorithms and Metaheuristics*. T.F. Gonzalez, Chapman & Hall/CRC Computer and Information Science, 2007.
  - B4 *Search methodologies: introductory tutorials in optimization and decision support techniques* E.K. Burke, G. Kendall, 2005, Springer, New York
  - B5 *Introduction to algorithms* T.H. Cormen and C.E. Leiserson and R.L. Rivest, MIT press (2001).
- Photocopies (from the Blackboard)
- Lecture slides
- Assignments
- ...but take notes in class!

Active Learning and Laboratory

Practical experience is important to learn to use heuristics
Implementation details play an important role.

- Laboratory sessions
  - Home preparation
  - Implementation of heuristics for a certain problem
  - Experimental analysis of performance
  - Groups in competition
  - (worthwhile in preparation of the project)
- Problem solving in class

Outline

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
- ...

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

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)

Traveling Salesman Problem

- **Given:** edge-weighted, undirected graph G
- **Task:** 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

Hamiltonian cycle problem

- **Given:** undirected graph G
- **Question:** Does it contain a Hamiltonian cycle?

solutions = candidate solutions that satisfy given *logical conditions*

Two variants:

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

Optimization problems

Traveling Salesman Problem

- **Given:** edge-weighted, undirected graph G
- **Task:** Find a minimal-weight Hamiltonian cycle in G.

- 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 constraints (= logical conditions) that solutions must satisfy.
- A candidate solution is called feasible (or valid) iff it satisfies the given constraints.
- Approximate solutions are feasible candidate solutions that are not optimal. (to be refined later).
- **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 Π:

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

Problem instantiation $\pi = \Pi(\mathrm{I})$:

- 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 $\mathcal{I}$

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.

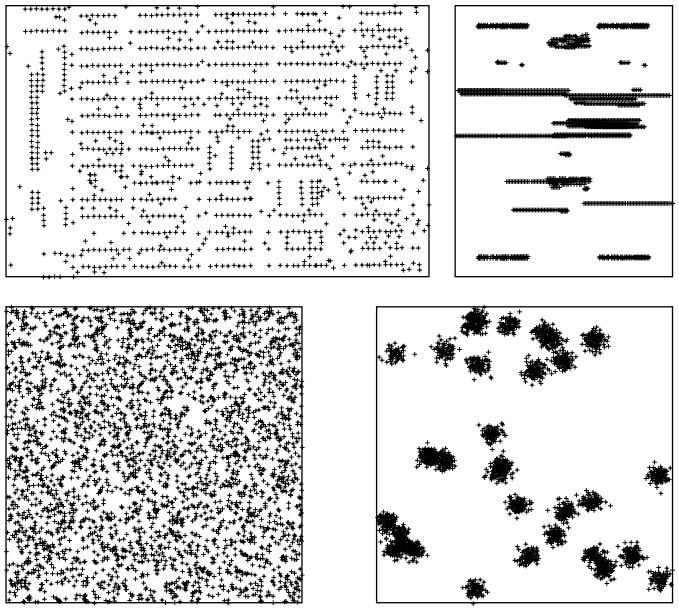
TSP: Benchmark Instances

Instance classes

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

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

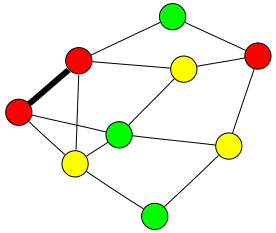
TSP: Benchmark Instances, Examples



The Vertex Coloring Problem

Given: A graph G and a set of colors Γ.

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.

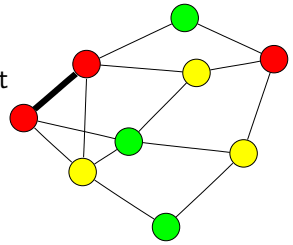
Optimization version (chromatic number)

Task: Find a proper coloring of G which uses the minimal number of colors.

The Vertex Coloring Problem

Given: A graph G and a set of colors Γ.

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.

What are we after?
Finding an algorithm a general method for solving general instances of the graph coloring problem.

Problem Solving as by George Pólya

George Pólya's 1945 book *How to Solve It*:

1. Understand the problem.
2. Make a plan.
3. Carry out the plan.
4. Look back on your work. How could it be better?

`http://en.wikipedia.org/wiki/How_to_Solve_It`

Pólya's First Principle: Understand the Problem

- Do you understand all the words used in stating the problem?
- What are you asked to find or show?
- Is there enough information to enable you to find a solution?
- Can you restate the problem in your own words?
- Can you think of a picture or a diagram that might help you understand the problem?

Pólya's Second Principle: Devise a plan

There are many reasonable ways to solve problems.
The skill at choosing an appropriate strategy is best learned by solving many problems.
You will find choosing a strategy increasingly easy. A partial list of strategies is included:

- Guess and check
- Make an orderly list
- Eliminate possibilities
- Use symmetry
- Consider special cases
- Use direct reasoning

Also suggested:

- Look for a pattern
- Draw a picture
- Solve a simpler problem
- Use a model
- Work backward

Pólya's third Principle: Carry out the plan

"Needed is care and patience, given that you have the necessary skills. Persist with the plan that you have chosen. If it continues not to work discard it and choose another. Don't be misled, this is how mathematics is done, even by professionals."

Pólya's fourth Principle: Review/Extend

"Much can be gained by taking the time to reflect and look back at what you have done, what worked and what didn't. Doing this will enable you to predict what strategy to use to solve future problems."

"If you can't solve a problem, then there is an easier problem you can solve: find it"

Heuristic	Informal Description	Formal analogue
Analogy	Can you find a problem analogous to your problem and solve that?	Map
Generalization	Can you find a problem more general than your problem?	Generalization
Induction	Can you solve your problem by deriving a generalization from some examples?	Induction
Variation of the Problem	Can you vary or change your problem to create a new problem (or set of problems) whose solution(s) will help you solve your original problem?	Search
Auxiliary Problem	Can you find a subproblem or side problem whose solution will help you solve your problem?	Subgoal
Here is a problem related to yours and solved before	Can you find a problem related to yours that has already been solved and use that to solve your problem?	Pattern recognition Pattern matching Reduction
Specialization	Can you find a problem more specialized?	Specialization
Decomposing and Recombining	Can you decompose the problem and "recombine its elements in some new manner"?	Divide and conquer
Working backward	Can you start with the goal and work backwards to something you already know?	Backward chaining
Draw a Figure	Can you draw a picture of the problem?	Diagrammatic Reasoning [1]
Auxiliary Elements	Can you add some new element to your problem to get closer to a solution?	Extension

SAT Problem

Definitions:

- 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'), $\wedge$ ('and'), $\vee$ ('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.

SAT Problem (decision problem, search variant):

- **Given:** Formula F in propositional logic
- **Task:** 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 $F := (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
- **Task:** 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 (optimization problem)

Which is the maximal number of clauses satisfiable in a propositional logic formula F?

Definitions:

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

$$\bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} l_{ij} = (l_{11} \vee \ldots \vee l_{1k_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots \vee l_{mk_m})$$

  where each literal l_{ij} is a propositional variable or its negation. The disjunctions $c_i = (l_{i1} \vee \ldots \vee l_{ik_i})$ 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:

$$\begin{aligned} F := \quad & \wedge (\neg x_2 \vee x_1) \\ & \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \\ & \wedge (x_1 \vee x_2) \\ & \wedge (\neg x_4 \vee x_3) \\ & \wedge (\neg x_5 \vee x_3) \end{aligned}$$

- 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.

Exercise

- Design algorithms for the 3-SAT problem.
- Show how the GCP problem can be encoded in a SAT problem.

Outline

Basic Concepts to Design and Analyze Algorithms

- Notation and terminology
- Machine models
- Pseudo-code
- Analysis of algorithms
- Computational complexity

Computational Complexity

Questions:

1. How good is the algorithm designed?
2. How hard, computationally, is a given a problem to solve using the most efficient algorithm for that problem?

1. Asymptotic notation, running time bounds
   Approximation theory
2. Complexity theory

Running time and asymptotic notation

$n \in \mathbf{N}$ instance size

max time	worst case	$T(n) = \max\{T(\pi) : \pi \in \Pi_n\}$
average time	average case	$T(n) = \frac{1}{\lvert\Pi_n\rvert}\{\sum_\pi T(\pi) : \pi \in \Pi_n\}$
min time	best case	$T(n) = \min\{T(\pi) : \pi \in \Pi_n\}$

Growth rate or asymptotic analysis

$f(n)$ and $g(n)$ same growth rate if $c \leq \frac{f(n)}{g(n)} \leq d$ for n large
$f(n)$ grows faster than $g(n)$ if $f(n) \geq c \cdot g(n)$ for all c and n large

big O	$O(f) = \{g(n) : \exists c > 0, \forall n > n_0 : g(n) \leq c \cdot f(n)\}$
big omega	$\Omega(f) = \{g(n) : \exists c > 0, \forall n > n_0 : g(n) \geq c \cdot f(n)\}$
theta	$\Theta(f) = O(f) \cap \Omega(f)$
little O	$o(f) = g$ grows strictly more slowly

Machine model

For asymptotic analysis we use RAM machine

- single processor unit
- all memory access take same amount of time

It is an abstraction from machine architecture: it ignores caches, memories hierarchies, parallel processing (SIMD, multi-threading), etc.

Total execution of a program = total number of instruction executed

We are not interested in constant and lower order terms

Pseudo-code

We express algorithms in natural language and mathematical notation, and in pseudo-code, which is an abstraction from programming languages C, C++, Java, etc.

(In implementation you can choose your favorite language)

Programs must be correct.
Certifying algorithm: computes a certificate for a post condition (without increasing asymptotic running time)

Good Algorithms

We say that an algorithm A is

$$\text{Efficient} = \text{good} = \text{polynomial time} = \text{polytime}$$
$$\text{iff}$$
$$\text{there exists } p(n) \text{ such that } T(A) = O(p(n))$$

There are problems for which no polytime algorithm is known. This course is about those problems.

Complexity theory classifies problems

Computational 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).

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

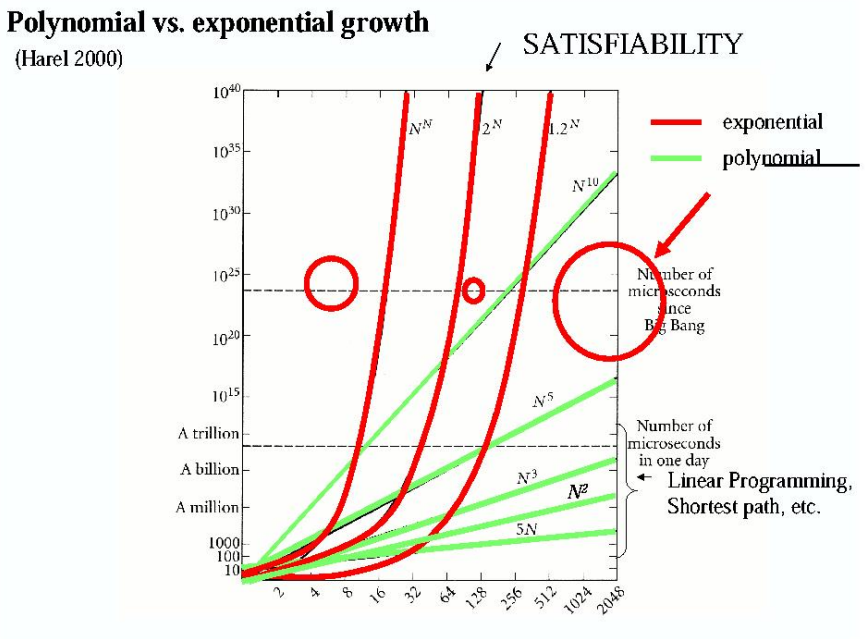
- Longest path problem is $\mathcal{NP}$-hard
  but not shortest path problem
- SAT for 3-CNF is $\mathcal{NP}$-complete
  but not 2-CNF (linear time algorithm)
- TSP is $\mathcal{NP}$-hard, the associated decision problem (for any solution quality) is $\mathcal{NP}$-complete
  but not the Euler tour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

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

Application Scenarios

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!



Analytical Analysis

Definition: Approximation Algorithms

An algorithm $\mathcal{A}$ is said to be a δ-approximation algorithm if it runs in *polynomial* time and for every problem instance π with optimal solution value $\mathrm{OPT}(\pi)$

$$\text{minimization:} \quad \frac{\mathcal{A}(\pi)}{\mathrm{OPT}(\pi)} \leq \delta \quad \delta \geq 1$$

$$\text{maximization:} \quad \frac{\mathcal{A}(\pi)}{\mathrm{OPT}(\pi)} \geq \delta \quad \delta \leq 1$$

(δ is called *worst case bound*, *worst case performance*, *approximation factor*, *approximation ratio*, *performance bound*, *performance ratio*, *error ratio*)

Definition: Polynomial approximation scheme

A family of approximation algorithms for a problem Π, $\{\mathcal{A}_\epsilon\}_\epsilon$, is called a polynomial approximation scheme (PAS), if algorithm $\mathcal{A}_\epsilon$ is a $(1+\epsilon)$-approximation algorithm and its running time is polynomial in the size of the input for a fixed ϵ

Definition: Fully polynomial approximation scheme

A family of approximation algorithms for a problem Π, $\{\mathcal{A}_\epsilon\}_\epsilon$, is called a fully polynomial approximation scheme (FPAS), if algorithm $\mathcal{A}_\epsilon$ is a $(1+\epsilon)$-approximation algorithm and its running time is polynomial in the size of the input and $1/\epsilon$