LOCAL SEARCH METHODS
APPLICATIONS AND ENGINEERING

Lecture 1

Introduction

Marco Chiarandini

Outline

Course Presentation

- Communication tools
  - Web-site `http://www.imada.sdu.dk/~marco/Teaching/DMP86/`
  - Blackboard (currently only used for forums)
  - Mailing List
- Schedule (Current: weeks 5-20. Alternative weeks 5-17.)
- Evaluation
  - Project
    Selection, Formalization, Modeling, Solution Development, Analysis, Reporting
  - Oral exam (provisional date: 21st June)
- Text book:
  Hoos H. and Stützle T. (2004). Stochastic Local Search: Foundations and Applications. Morgan Kaufmann Publishers, San Francisco, CA, USA.
- Programming Languages: C vs C++ vs Java, and R
- Weekly Notes and Assignments

Content of the Course

- Heuristic Methods for Combinatorial Optimization
  - Greedy Heuristics and Neighborhood Structures
  - Metaheuristics
    (Probabilistic Iterative Improvement, GRASP, Variable Neighborhood Search, Tabu Search, Simulated Annealing, Iterated Local Search, Dynamic Local Search Dynasearch and Path Relinking Scatter Search and Memetic Algorithm Cross Entropy, Ant Colony Optimization)
- Applications
  Timetabling, Routing, Scheduling
- Empirical Methods
  - Descriptive Statistics (Exploratory Data Analysis)
  - Inference Statistics (Experimental Design, Statistical Tests)
    R, the free software environment for statistical computing and
- Further Notions (Multi-objective optimization, stochastic optimization)

Combinatorial Problems

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

- finding shortest/cheapest round trips (TSP)
- finding models of propositional formulae (SAT)
- register memory allocation (GCP)
- planning, scheduling, timetabling
- Internet data packet routing
- protein structure prediction
- combinatorial auctions winner determination
- portfolio selection
- ...

Combinatorial Problems (2)

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 (3)

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 (4)

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*

Example: Satisfiability problem

- *Given:* A formula in propositional logic, *e.g.*, $F := (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
- *Question:* is there an assignment of truth values to variables $x_1, x_2, \ldots, x_n$ that renders F true?

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

Example: MAX-SAT

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

Three Toy Problems

Toy problems: conceptually concise, exact description problems intended to illustrate the development, analysis and presentation of algorithms
Real-world problems: what people care about. Tend not to have a single agreed-upon description

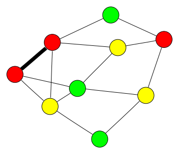
Three prominent, conceptually simple problems:

- Finding shortest round trips in graphs (TSP)
  unconstrained optimization problem — routing
- Finding an assignment of colors to the vertices of a graph such that any two vertices connected by an edge are not assigned the same color
  constrained optimization problem — assignment
- Finding the sequences of jobs to be processed by a single machine that minimize the total weighted tardiness (SMTWTP)
  unconstrained/constrained optimization problem – permutation

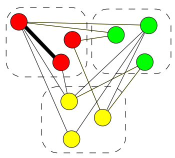
The Vertex Coloring Problem

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.



Assignment



Partition

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 Traveling Salesman Problem

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

Types of TSP instances:

- **Symmetric**: For all edges (v, v') of the given graph G, (v', v) is also in G, and $w((v, v')) = w((v', v))$.
  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.

The Single Machine Total Tardiness Problem

Given: a set of n jobs $\{J_1, \ldots, J_n\}$ to be processed on a single machine and for each job J_i a processing time p_i, a weight w_i and a due date d_i.

Task: Find a schedule that minimizes the total weighted tardiness $\sum_{i=1}^{n} w_i \cdot T_i$ where $T_i = \{C_i - d_i, 0\}$ (C_i completion time of job J_i)

Example:

Job	J_1	J_2	J_3	J_4	J_5	J_6
Processing Time	3	2	2	3	4	3
Due date	6	13	4	9	7	17
Weight	2	3	1	5	1	2

Sequence $\phi = J_3, J_1, J_5, J_4, J_1, J_6$

Job	J_3	J_1	J_5	J_4	J_1	J_6
C_i	2	5	9	12	14	17
T_i	0	0	2	3	1	0
$w_i \cdot T_i$	0	0	2	15	3	0

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).

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.

- The Graph Coloring Problem is $\mathcal{NP}$-complete
  but not on interval graphs

- Many scheduling and timetabling problems are $\mathcal{NP}$-hard

Application Scenarios

Practically solving hard combinatorial problems:

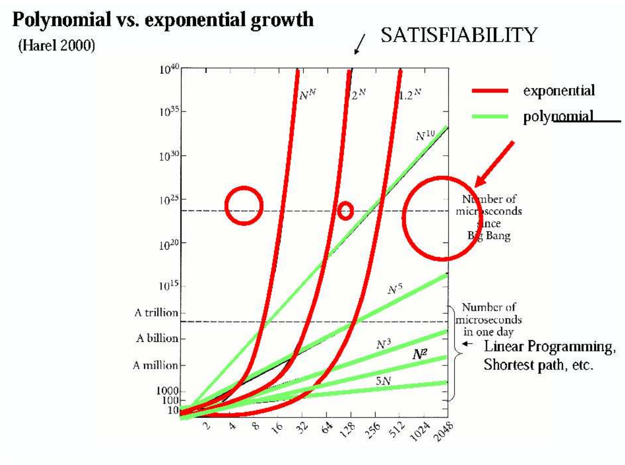
- Subclasses can often be solved efficiently
  (*e.g.*, 2-SAT);

- 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!



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

Algorithms: instantiation of methods on a specific problem Π

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)

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

Perturbative search

- search space = complete candidate solutions
- search step = modification of one or more solution components
- iteratively generate and evaluate candidate solutions
  - decision problems: evaluation = test if solution
  - optimization problems: evaluation = check objective function value
- evaluating candidate solutions is typically computationally much cheaper than finding (optimal) solutions

Example

Construction Rule

Construction Heuristic (CH):

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

Perturbative Search

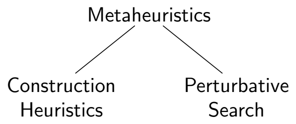
Iterative Improvement (II):

While s has better neighbors:
  choose a neighbor s' of s
    such that $g(s') < g(s)$
  $s := s'$

Local Search Methods

In order to obtain very high quality solutions
it is often successful to combine the two heuristic paradigms

Meta-heuristics
General guidance criteria over lower level heuristics.



Informed search based on *local* or incomplete knowledge as opposed to systematic search $\Rightarrow$ **Local Search Methods**

Stochastic Local Search (SLS) algorithms use *randomized choices* in generating and modifying candidate solutions. They are introduced whenever it is unknown which deterministic rules are profitable for all the instances of interest.

Example: Uninformed random walk for k-coloring

procedure *URW-for-k-col*(G, k, *maxSteps*)
  input: *graph G, integer k, integer maxSteps*
  output: *feasible coloring of G* **or** $\emptyset$
  choose assignment φ of k colors to all vertices in G
    uniformly at random;
  steps $:= 0$;
  while not((φ is a proper coloring) **and** (*steps* $<$ *maxSteps*)) **do**
    randomly select vertex v in G;
    change value of $\varphi(v)$;
    steps $:=$ *steps*$+1$;
  end
  if φ is feasible**then**
    return φ
  else
    return $\emptyset$
  end
end *URW-for-k-col*

Systematic search is often better suited when ...

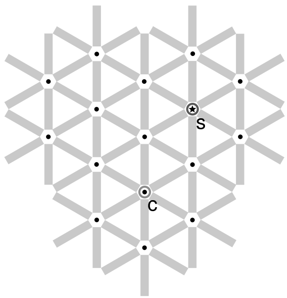
- proofs of insolubility or optimality are required;
- time constraints are not critical;
- problem-specific knowledge can be exploited.

Local search is often better suited when ...

- non linear constraints and non linear objective function;
- reasonably good solutions are required within a short time;
- problem-specific knowledge is rather limited.

Complementarity:
Local and systematic search can be fruitfully combined, *e.g.*, by using local search for finding solutions whose optimality is proved using systematic search.
Note: The largest instance proved optimal for the Traveling Salesman amounts to 24978 cities. See `http://www.tsp.gatech.edu/`.

Local search — global view



- vertices: candidate solutions (search positions)
- edges: connect neighboring positions
- s: (optimal) solution
- c: current search position

Definition: Local Search Algorithm (1)

For given problem instance π:

- **search space** $S(\pi)$
  (*e.g.*, for GCP: assignment of colors to each vertex)
- **solution set** $S'(\pi) \subseteq S(\pi)$
  (*e.g.*, for GCP: feasible assignment)
- **neighborhood relation** $N(\pi) \subseteq S(\pi) \times S(\pi)$
  (*e.g.*, for GCP: neighboring assignments differ in the color at one vertex)

Definition: Local Search Algorithm (2)

- **set of memory states** $M(\pi)$
  (may consist of a single state, for LS algorithms that do not use memory)
- **initialization function** *init* $: \emptyset \mapsto \mathcal{D}(S(\pi) \times M(\pi))$
  (specifies probability distribution over initial search positions and memory states)
- **step function** *step* $: S(\pi) \times M(\pi) \mapsto \mathcal{D}(S(\pi) \times M(\pi))$
  (maps each search position and memory state onto probability distribution over subsequent, neighboring search positions and memory states)
- **termination predicate** *terminate* $: S(\pi) \times M(\pi) \mapsto \mathcal{D}(\{\top, \bot\})$
  (determines the termination probability for each search position and memory state)

procedure *LS-Decision*(π)
  input: *problem instance* $\pi \in \Pi$
  output: *solution* $s \in S'(\pi)$ **or** $\emptyset$
  $(s, m) := init(\pi)$;

  while not $terminate(\pi, s, m)$ **do**
    $(s, m) := step(\pi, s, m)$;
  end

  if $s \in S'(\pi)$ **then**
    return s
  else
    return $\emptyset$
  end
end *LS-Decision*

procedure *LS-Minimization*(π')
  input: *problem instance* $\pi' \in \Pi'$
  output: *solution* $s \in S'(\pi')$ **or** $\emptyset$
  $(s, m) := init(\pi')$;
  $\hat{s} := s$;
  while not $terminate(\pi', s, m)$ **do**
    $(s, m) := step(\pi', s, m)$;
    if $f(\pi', s) < f(\pi', \hat{s})$ **then**
      $\hat{s} := s$;
    end
  end
  if $\hat{s} \in S'(\pi')$ **then**
    return $\hat{s}$
  else
    return $\emptyset$
  end
end *LS-Minimization*

Appendix

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 $F := (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
- *Objective:* Find an assignment of truth values to variables x_1, x_2 that renders F true, or decide that no such assignment exists.

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'), $\wedge$ ('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.

Definition:

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

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

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

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

- 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.

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 show 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) $\pi_1 \leq_P \pi_2 \Rightarrow \Pi_1$ is no more than a polynomial harder than Π_2.
- 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:**