STOCHASTIC LOCAL SEARCH
FOUNDATIONS AND APPLICATIONS

Introduction: Combinatorial Problems and Search

Holger H. Hoos & Thomas Stützle

Outline

1. Combinatorial Problems
2. Two Prototypical Combinatorial Problems
3. Search Paradigms
4. Stochastic Local Search

Combinatorial Problems

Combinatorial problems arise in many areas
of computer science and application domains:

- finding shortest/cheapest round trips (TSP)
- finding models of propositional formulae (SAT)
- planning, scheduling, time-tabling
- internet data packet routing
- protein structure prediction
- combinatorial auctions winner determination

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

Candidate solutions are combinations of *solution components* that may be encountered during a solutions attempt but need not satisfy all given conditions.

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

Example:

- *Given:* Set of points in the Euclidean plane
- *Objective:* Find the shortest round trip

Note:

- a round trip corresponds to a sequence of points
  (= assignment of points to sequence positions)
- *solution component:* trip segment consisting of two points that are visited one directly after the other
- *candidate solution:* round trip
- *solution*: round trip with minimal length

Problem *vs* problem instance:

- *Problem:* Given *any* set of points X, find a shortest round trip
- *Solution:* Algorithm that finds shortest round trips for any X

- *Problem instance:* Given *a specific* set of points P, find a shortest round trip
- *Solution:* Shortest round trip for P

Technically, problems can be formalised as sets of problem instances.

Decision problems:

solutions = candidate solutions that satisfy given *logical conditions*

Example: The Graph Colouring Problem

- *Given:* Graph G and set of colours C
- *Objective:* Assign to all vertices of G a colour from C
  such that two vertices connected by an edge
  are never assigned the same colour

Optimisation problems:

- can be seen as generalisations of decision problems
- *objective function f* measures *solution quality*
  (often defined on all candidate solutions)
- typical goal: find solution with optimal quality
  minimisation problem: optimal quality = *minimal* value of f
  maximisation problem: optimal quality = *maximal* value of f

Example:

Variant of the Graph Colouring Problem where the objective is to find a valid colour assignment that uses a minimal number of colours.

Note: Every minimisation problem can be formulated as a maximisation problem and vice versa.

Many optimisation problems have an objective function as well as logical conditions that solutions must satisfy.

A candidate solution is called *feasible* (or *valid*) iff it satisfies the given logical conditions.

Two Prototypical Combinatorial Problems

Studying conceptually simple problems facilitates development, analysis and presentation of algorithms

Two prominent, conceptually simple problems:

- Finding satisfying variable assignments of propositional formulae (SAT)
  – prototypical decision problem
- Finding shortest round trips in graphs (TSP)
  – prototypical optimisation problem

Concise definition of SAT:

- *Given:* Formula F in propositional logic.
- *Objective:* Decide whether F is satisfiable.

Note:

- In many cases, the restriction of SAT to CNF formulae is considered.
- The restriction of SAT to k-CNF formulae is called *k-SAT*.

Definition:

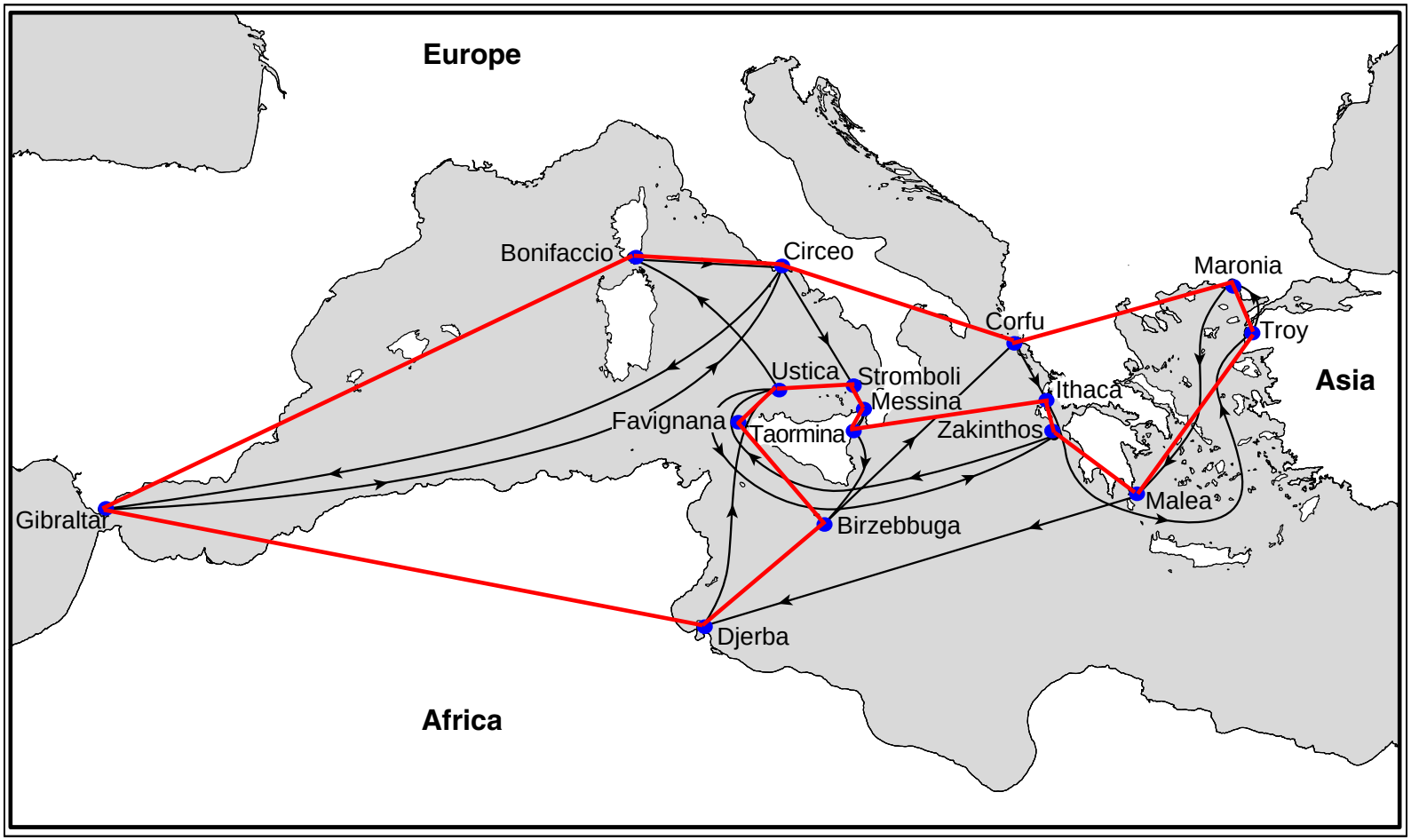
- A formula is in *conjunctive normal form (CNF)* iff it is of the form

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

  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, $k(i) = k$).

TSP: A simple example



Definition:

- *Hamiltonian cycle* in graph $G := (V, E)$:
  cyclic path that visits every vertex of G exactly once
  (except start/end point).

- *Weight* of path $p := (u_1, \ldots, u_k)$ in edge-weighted graph $G := (V, E, w)$: total weight of all edges on p, *i.e.*:

$$w(p) := \sum_{i=1}^{k-1} w((u_i, u_{i+1}))$$

The Travelling Salesman Problem (TSP)

- *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 a Euclidean space,
  weight function = Euclidean distance metric.
- *Geographic*: Vertices = points on a sphere,
  weight function = geographic (great circle) distance.

Search Paradigms

Solving combinatorial problems through search:

- iteratively generate and evaluate candidate solutions
- decision problems: evaluation = test if solution
- optimisation problems: evaluation = check objective function value
- evaluating candidate solutions is typically computationally much cheaper than finding (optimal) solutions

Perturbative search

- search space = complete candidate solutions
- search step = modification of one or more solution components

Example: SAT

- search space = complete variable assignments
- search step = modification of truth values for one or more variables

Constructive search (aka construction heuristics)

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

Example: Nearest Neighbour Heuristic (NNH) for TSP

- start with single vertex (chosen uniformly at random)
- in each step, follow minimal-weight edge to yet unvisited, next vertex
- complete Hamiltonian cycle by adding initial vertex to end of path

Note: NNH typically does not find very high quality solutions, but it is often and successfully used in combination with perturbative search methods.

Systematic search:

- traverse search space for given problem instance in a systematic manner
- *complete*: guaranteed to eventually find (optimal) solution, or to determine that no solution exists

Local Search:

- start at some position in search space
- iteratively move from position to neighbouring position
- typically *incomplete*: not guaranteed to eventually find (optimal) solutions, cannot determine insolubility with certainty

Example: Uninformed random walk for SAT

procedure *URW-for-SAT*(F, *maxSteps*)
  input: *propositional formula* F, *integer maxSteps*
  output: *model of* F **or** $\emptyset$

  choose assignment a of truth values to all variables in F
    uniformly at random;
  steps := 0;
  while not((a satisfies F) **and** (*steps* $<$ *maxSteps*)) **do**
    randomly select variable x in F;
    change value of x in a;
    steps := *steps*+1;
  end
  if a satisfies F **then**
    return a
  else
    return $\emptyset$
  end
end *URW-for-SAT*

Systematic search is often better suited when ...

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

Local search is often better suited when ...

- reasonably good solutions are required within a short time;
- parallel processing is used;
- problem-specific knowledge is rather limited.

Stochastic Local Search

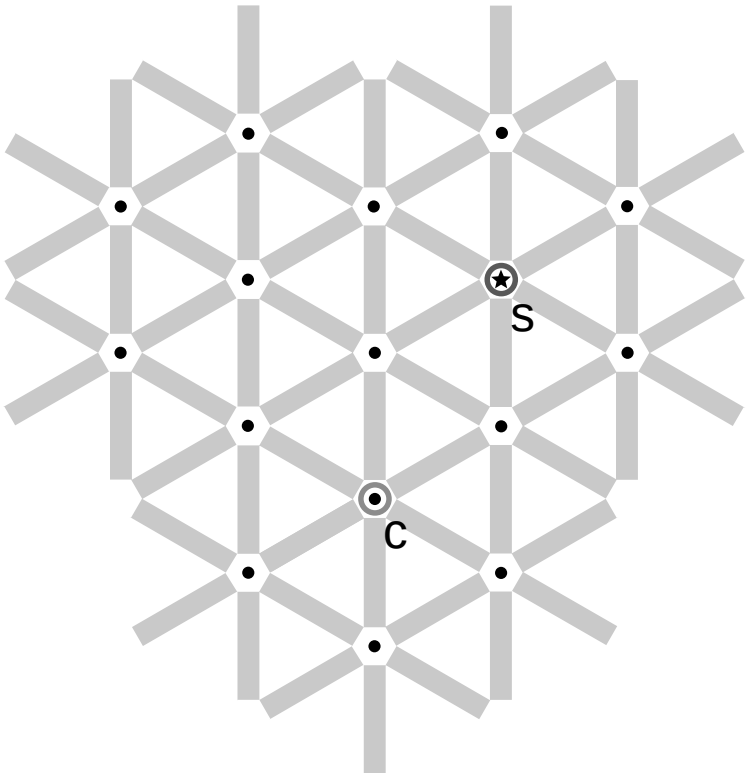
Many prominent local search algorithms use *randomised choices* in generating and modifying candidate solutions.

These *stochastic local search (SLS) algorithms* are one of the most successful and widely used approaches for solving hard combinatorial problems.

Some well-known SLS methods and algorithms:

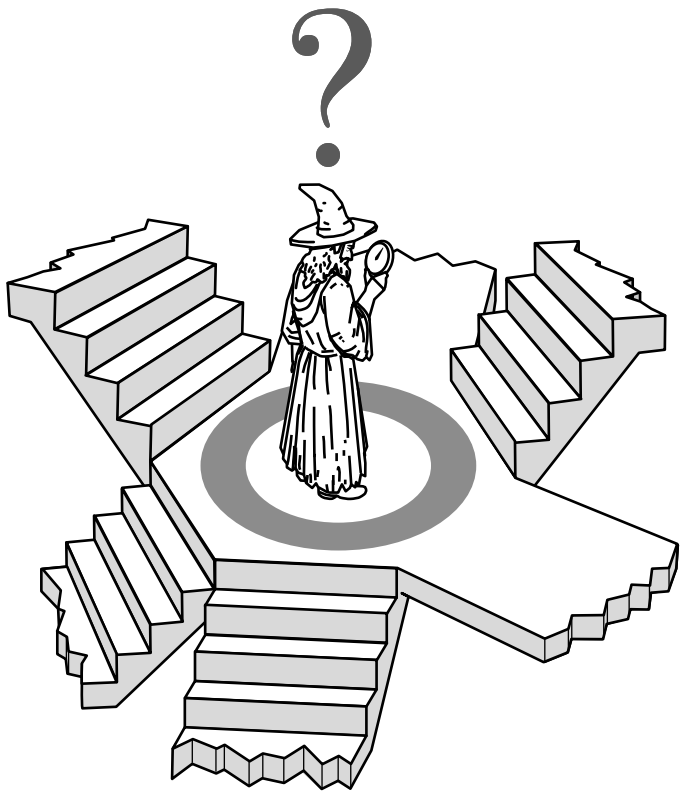
- Evolutionary Algorithms
- Simulated Annealing
- Lin-Kernighan Algorithm for TSP

Stochastic local search — global view



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

Stochastic local search — local view



Next search position is selected from

local neighbourhood
based on local information, *e.g.*, heuristic values.

Definition: **Stochastic Local Search Algorithm** (1)

For given problem instance π:

- *search space* $S(\pi)$
  (*e.g.*, for SAT: set of all complete truth assignments to propositional variables)
- *solution set* $S'(\pi) \subseteq S(\pi)$
  (*e.g.*, for SAT: models of given formula)
- *neighbourhood relation* $N(\pi) \subseteq S(\pi) \times S(\pi)$
  (*e.g.*, for SAT: neighbouring variable assignments differ in the truth value of exactly one variable)

Definition: **Stochastic Local Search Algorithm** (2)

- *set of memory states* $M(\pi)$
  (may consist of a single state, for SLS algorithms that do not use memory)
- *initialisation 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, neighbouring 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 *SLS-Minimisation*(π')
  input: *problem instance* $\pi' \in \Pi'$
  output: *solution* $s \in S'(\pi')$ **or** $\emptyset$
  $(s, m) := \mathit{init}(\pi')$;
  $\hat{s} := s$;
  while not $\mathit{terminate}(\pi', s, m)$ **do**
    $(s, m) := \mathit{step}(\pi', s, m)$;
    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 *SLS-Minimisation*

Example: Uninformed random walk for SAT

- **search space S:** set of all truth assignments to variables in given formula F
- **solution set S':** set of all models of F
- **neighbourhood relation N:** *1-flip neighbourhood*, *i.e.*, assignments are neighbours under N iff they differ in the truth value of exactly one variable
- **memory:** not used, *i.e.*, $M := \{0\}$
- **initialisation:** uniform random choice from S
- **step function:** uniform random choice from current neighbourhood
- **termination:** when model is found

Definition:

- *Search step* (or *move*): pair of search positions s, s' for which s' can be reached from s in one step,
- *Search trajectory*: finite sequence of search positions $(s_0, s_1, \ldots, s_k)$ such that (s_{i-1}, s_i) is a *search step* for any $i \in \{1, \ldots, k\}$ and the probability of initialising the search at s_0 is greater than zero,
- *Search strategy*: specified by *init* and *step* function; to some extent independent of problem instance and other components of SLS algorithm.

Uninformed Random Picking

- $N := S \times S$
- does not use memory
- *init*, *step*: uniform random choice from S,

Uninformed Random Walk

- does not use memory
- *init*: uniform random choice from S
- *step*: uniform random choice from current neighbourhood,

Note: These uninformed SLS strategies are quite ineffective,
but play a role in combination with more directed search strategies.

Evaluation function:

- function $g(\pi) : S(\pi) \mapsto \mathbb{R}$ that maps candidate solutions of a given problem instance π onto real numbers, such that global optima correspond to solutions of π;
- used for ranking or assessing neighbhours of current search position to provide guidance to search process.

Evaluation *vs* objective functions:

- *Evaluation function*: part of SLS algorithm.
- *Objective function*: integral part of optimisation problem.
- Some SLS methods use evaluation functions different from given objective function

Iterative Improvement (II)

- does not use memory
- *init*: uniform random choice from S
- *step*: uniform random choice from improving neighbours,
- terminates when no improving neighbour available
  (to be revisited later)
- different variants through modifications of step function

Note: II is also known as *iterative descent* or *hill-climbing*.

Example: Iterative Improvement for SAT

- **search space S:** set of all truth assignments to variables in given formula F
- **solution set S':** set of all models of F
- **neighbourhood relation N:** 1-flip neighbourhood (as in Uninformed Random Walk for SAT)
- **memory:** not used, *i.e.*, $M := \{0\}$
- **initialisation:** uniform random choice from S, *i.e.*, $\mathit{init}()(a') := 1/\#S$ for all assignments a'

Example: Iterative Improvement for SAT (continued)

- **evaluation function:** $g(a) :=$ number of clauses in F that are *unsatisfied* under assignment a
  (*Note:* $g(a) = 0$ iff a is a model of F.)
- **step function**: uniform random choice from improving neighbours, *i.e.*, $step(a)(a') := 1/\#I(a)$ if $s' \in I(a)$, and 0 otherwise, where $I(a) := \{a' \mid N(a, a') \wedge g(a') < g(a)\}$
- **termination**: when no improving neighbour is available
  i.e., $terminate(a)(\top) := 1$ if $I(a) = \emptyset$, and 0 otherwise.

Definition:

- *Local minimum:* search position without improving neighbours w.r.t. given evaluation function g and neighbourhood N, *i.e.*, position $s \in S$ such that $g(s) \leq g(s')$ for all $s' \in N(s)$.
- *Strict local minimum:* search position $s \in S$ such that $g(s) < g(s')$ for all $s' \in N(s)$.
- *Local maxima* and *strict local maxima*: defined analogously.

Simple mechanisms for escaping from local optima:

- *Restart:* re-initialise search whenever a local optimum is encountered.
  (Often rather ineffective due to cost of initialisation.)
- *Non-improving steps*: in local optima, allow selection of candidate solutions with equal or worse evaluation function value, *e.g.*, using minimally worsening steps.
  (Can lead to long walks in *plateaus*, *i.e.*, regions of search positions with identical evaluation function.)

Note: Neither of these mechanisms is guaranteed to always escape effectively from local optima.

Diversification *vs* Intensification

- Goal-directed and randomised components of SLS strategy need to be balanced carefully.
- *Intensification*: aims to greedily increase solution quality or probability, *e.g.*, by exploiting the evaluation function.
- *Diversification*: aim to prevent search stagnation by preventing search process from getting trapped in confined regions.

Examples:

- Iterative Improvement (II): *intensification* strategy.
- Uninformed Random Walk (URW): *diversification* strategy.

Balanced combination of intensification and diversification mechanisms forms the basis for advanced SLS methods.