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. Computational Complexity
4. Search Paradigms
5. 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

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.

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.

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

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.

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 vertices of G a colour from C such that two vertices connected by an edge are never assigned the same colour

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 vertices of G a colour from C such that two vertices connected by an edge are never assigned the same colour

Every decision problem has two variants:

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

Note: Search and decision variants are closely related; algorithms for one can be used for solving the other.

Every decision problem has two variants:

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

Note: Search and decision variants are closely related; algorithms for one can be used for solving the other.

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 problems and vice versa.

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 problems and vice versa.

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 problems and vice versa.

Variants of optimisation 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

Every optimisation 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 minimisation problems) or determine that no such solution exists.

Variants of optimisation 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

Every optimisation 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 minimisation problems) or determine that no such solution exists.

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.

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.

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.

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.

Note:

- Algorithms for *optimisation problems* can be used to solve *associated decision problems*
- Algorithms for *decision problems* can often be extended to related *optimisation problems*.
- *Caution:* This does not always solve the given problem most efficiently.

Note:

- Algorithms for *optimisation problems* can be used to solve *associated decision problems*
- Algorithms for *decision problems* can often be extended to related *optimisation problems*.
- *Caution:* This does not always solve the given problem most efficiently.

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 roundtrips in graphs (TSP)
  – prototypical optimisation problem

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.

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.

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.

Definition:

- *Formula in propositional logic*: well-formed string that may contain
  - propositional variables $x_1, x_2, \ldots, x_n$;
  - truth values $\top$ ('true'), $\perp$ ('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:

- *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:

- *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, $k(i) = k$).

Note: For every propositional formula, there is an equivalent formula in 3-CNF.

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, $k(i) = k$).

Note: For every propositional formula, there is an equivalent formula in 3-CNF.

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, $k(i) = k$).

Note: For every propositional formula, there is an equivalent formula in 3-CNF.

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*.

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*.

Example:

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

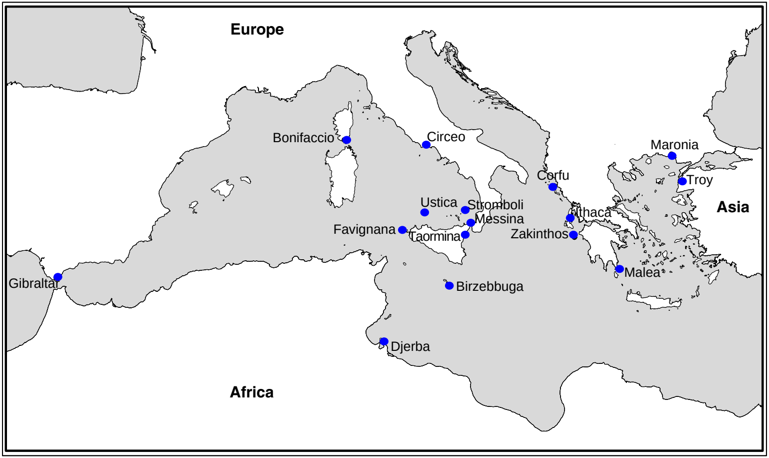
- F is in CNF.
- Is F satisfiable?

Example:

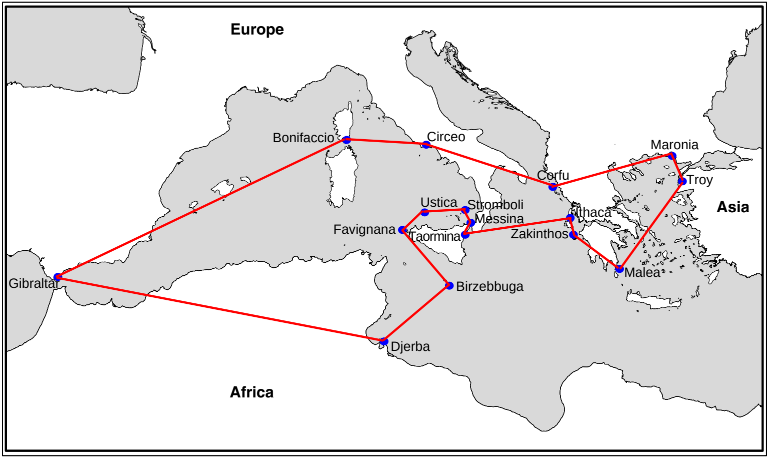
$$
\begin{aligned}
F := \quad & \wedge\ (\neg x_2 \vee x_1) \\
& \wedge\ (\neg x_1 \vee \neg x_2 \vee \neg x_3) \\
& \wedge\ (x_1 \vee x_2) \\
& \wedge\ (\neg x_4 \vee x_3) \\
& \wedge\ (\neg x_5 \vee x_3)
\end{aligned}
$$

- 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.

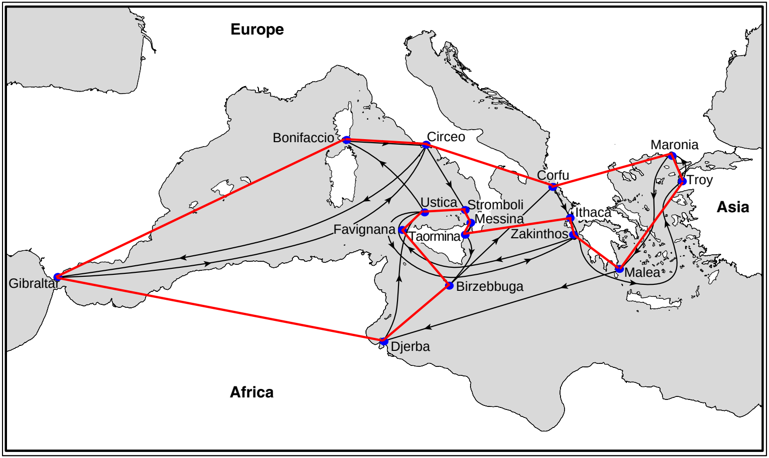
TSP: A simple example



TSP: A simple example (2)



TSP: A simple example (3)



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

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

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.

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.

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.

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.

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 ('$O(...)$' and '$\Theta(...)$' notations).

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 ('$O(...)$' and '$\Theta(...)$' notations).

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 ('$O(...)$' and '$\Theta(...)$' notations).

Important concepts (continued):

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

  Note: nondeterministic $\neq$ randomised; non-deterministic machines are idealised 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).

Important concepts (continued):

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

  Note: nondeterministic $\neq$ randomised; non-deterministic machines are idealised 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).

Important concepts (continued):

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

  Note: nondeterministic $\neq$ randomised; non-deterministic machines are idealised 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).

Important concepts (continued):

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

  Note: nondeterministic $\neq$ randomised; non-deterministic machines are idealised 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:

- SAT for general propositional formulae is $\mathcal{NP}$-complete.
- SAT for 3-CNF is $\mathcal{NP}$-complete.

- TSP is $\mathcal{NP}$-hard, the associated decision problem for optimal solution quality is $\mathcal{NP}$-complete.
- The same holds for Euclidean TSP instances.

- The Graph Colouring Problem is $\mathcal{NP}$-complete.
- Many scheduling and timetabling problems are $\mathcal{NP}$-hard.

Many combinatorial problems are hard:

- SAT for general propositional formulae is $\mathcal{NP}$-complete.
- SAT for 3-CNF is $\mathcal{NP}$-complete.

- TSP is $\mathcal{NP}$-hard, the associated decision problem for optimal solution quality is $\mathcal{NP}$-complete.
- The same holds for Euclidean TSP instances.

- The Graph Colouring Problem is $\mathcal{NP}$-complete.
- Many scheduling and timetabling problems are $\mathcal{NP}$-hard.

Many combinatorial problems are hard:

- SAT for general propositional formulae is $\mathcal{NP}$-complete.
- SAT for 3-CNF is $\mathcal{NP}$-complete.
- TSP is $\mathcal{NP}$-hard, the associated decision problem for optimal solution quality is $\mathcal{NP}$-complete.
- The same holds for Euclidean TSP instances.
- The Graph Colouring Problem is $\mathcal{NP}$-complete.
- Many scheduling and timetabling problems are $\mathcal{NP}$-hard.

Many combinatorial problems are hard:

- SAT for general propositional formulae is $\mathcal{NP}$-complete.
- SAT for 3-CNF is $\mathcal{NP}$-complete.

- TSP is $\mathcal{NP}$-hard, the associated decision problem for optimal solution quality is $\mathcal{NP}$-complete.
- The same holds for Euclidean TSP instances.

- The Graph Colouring Problem is $\mathcal{NP}$-complete.
- Many scheduling and timetabling problems are $\mathcal{NP}$-hard.

Many combinatorial problems are hard:

- SAT for general propositional formulae is $\mathcal{NP}$-complete.
- SAT for 3-CNF is $\mathcal{NP}$-complete.

- TSP is $\mathcal{NP}$-hard, the associated decision problem for optimal solution quality is $\mathcal{NP}$-complete.
- The same holds for Euclidean TSP instances.

- The Graph Colouring Problem is $\mathcal{NP}$-complete.
- Many scheduling and timetabling problems are $\mathcal{NP}$-hard.

But: Some combinatorial problems can be solved efficiently:

- Shortest Path Problem (Dijkstra's algorithm);
- 2-SAT (linear time algorithm);
- many special cases of TSP, *e.g.*, Euclidean instances where all vertices lie on a circle;
- sequence alignment problems (dynamic programming).

But: Some combinatorial problems can be solved efficiently:

- Shortest Path Problem (Dijkstra's algorithm);
- 2-SAT (linear time algorithm);
- many special cases of TSP, *e.g.*, Euclidean instances where all vertices lie on a circle;
- sequence alignment problems (dynamic programming).

But: Some combinatorial problems can be solved efficiently:

- Shortest Path Problem (Dijkstra's algorithm);
- 2-SAT (linear time algorithm);
- many special cases of TSP, *e.g.*, Euclidean instances where all vertices lie on a circle;
- sequence alignment problems (dynamic programming).

But: Some combinatorial problems can be solved efficiently:

- Shortest Path Problem (Dijkstra's algorithm);
- 2-SAT (linear time algorithm);
- many special cases of TSP, *e.g.*, Euclidean instances where all vertices lie on a circle;
- sequence alignment problems (dynamic programming).

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 optimisation);
- Approximation of optimal solutions:
  sometimes possible in polynomial time (*e.g.*, Euclidean TSP),
  but in many cases also intractable (*e.g.*, general TSP);
- Randomised computation is oftn practically
  (and possibly theoretically) more efficient;
- Asymptotic bounds *vs* true complexity:
  constants matter!

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 optimisation);
- Approximation of optimal solutions:
  sometimes possible in polynomial time (*e.g.*, Euclidean TSP),
  but in many cases also intractable (*e.g.*, general TSP);
- Randomised computation is oftn practically
  (and possibly theoretically) more efficient;
- Asymptotic bounds *vs* true complexity:
  constants matter!

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 optimisation);
- Approximation of optimal solutions: sometimes possible in polynomial time (*e.g.*, Euclidean TSP), but in many cases also intractable (*e.g.*, general TSP);
- Randomised computation is oftn practically (and possibly theoretically) more efficient;
- Asymptotic bounds *vs* true complexity: constants matter!

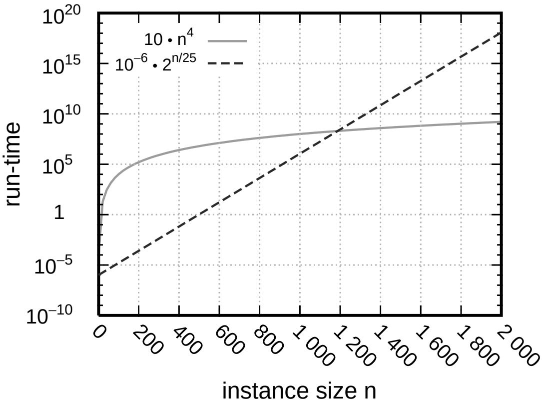
Practically solving hard combinatorial problems:

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

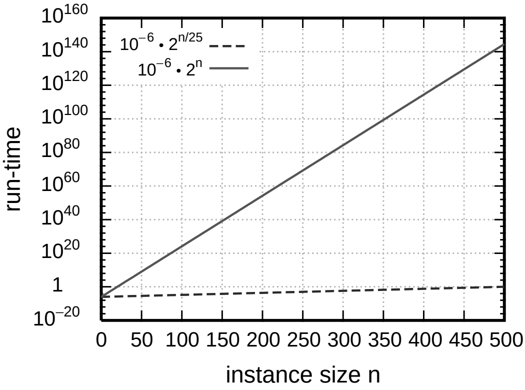
Practically solving hard combinatorial problems:

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

Example: Polynomial *vs* exponential growth



Example: Impact of constants



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

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

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

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

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 high-quality solutions, but is often and successfully used in combination with perturbative search methods.

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 high-quality solutions, but is often and successfully used in combination with perturbative search methods.

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 high-quality solutions, but 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

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

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

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*

Example: Uninformed random walk for SAT

```
procedure URW-for-SAT(F, maxSteps)
   input: propositional formula F, integer maxSteps
   output: model of F or ∅

   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 ∅
   end
end URW-for-SAT
```

Example: Uninformed random walk for SAT

```
procedure URW-for-SAT(F, maxSteps)
   input: propositional formula F, integer maxSteps
   output: model of F or ∅
   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 ∅
   end
end URW-for-SAT
```

Example: Uninformed random walk for SAT

```
procedure URW-for-SAT(F, maxSteps)
   input: propositional formula F, integer maxSteps
   output: model of F or ∅

   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 ∅
   end
end URW-for-SAT
```

Local search $\neq$ perturbative search:

- Construction heuristics can be seen as local search methods *e.g.*, the Nearest Neighbour Heuristic for TSP.

  Note: Many high-performance local search algorithms combine constructive and perturbative search.

- Perturbative search can provide the basis for systematic search methods.

Local search $\neq$ perturbative search:

- Construction heuristics can be seen as local search methods *e.g.*, the Nearest Neighbour Heuristic for TSP.

  Note: Many high-performance local search algorithms combine constructive and perturbative search.

- Perturbative search can provide the basis for systematic search methods.

Local search $\neq$ perturbative search:

- Construction heuristics can be seen as local search methods *e.g.*, the Nearest Neighbour Heuristic for TSP.

  Note: Many high-performance local search algorithms combine constructive and perturbative search.

- Perturbative search can provide the basis for systematic search methods.

Tree search

- 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.

Tree search

- 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.

Example: NNH + Backtracking

- Construct complete candidate round trip using NNH.
- Backtrack to most recent choice point with unexplored alternatives.
- Complete tour using NNH (possibly creating new choice points).
- Recursively iterate backtracking and completion.

Efficiency of tree search can be substantially improved by pruning choices that cannot lead to (optimal) solutions.

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.

Efficiency of tree search can be substantially improved by pruning choices that cannot lead to (optimal) solutions.

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.

Efficiency of tree search can be substantially improved by pruning choices that cannot lead to (optimal) solutions.

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.

Variations on simple backtracking:

- Dynamical selection of solution components in construction or choice points in backtracking.
- Backtracking to other than most recent choice points (*back jumping*).
- Randomisation of construction method or selection of choice points in backtracking $\rightsquigarrow$ *randomised systematic search*.

Variations on simple backtracking:

- Dynamical selection of solution components in construction or choice points in backtracking.
- Backtracking to other than most recent choice points (*back jumping*).
- Randomisation of construction method or selection of choice points in backtracking $\rightsquigarrow$ *randomised systematic search*.

Variations on simple backtracking:

- Dynamical selection of solution components in construction or choice points in backtracking.
- Backtracking to other than most recent choice points (*back jumping*).
- Randomisation of construction method or selection of choice points in backtracking $\rightsquigarrow$ *randomised systematic search*.

Systematic *vs* Local Search:

- **Completeness:** Advantage of systematic search, but not always relevant,*e.g.*, when existence of solutions is guaranteed by construction or in real-time situations.
- **Any-time property:** Positive correlation between run-time and solution quality or probability; typically more readily achieved by local search.
- **Complementarity:** Local and systematic can be fruitfully combined, *e.g.*, by using local search for finding solutions whose optimality is proven using systematic search.

Systematic *vs* Local Search:

- **Completeness:** Advantage of systematic search, but not always relevant,*e.g.*, when existence of solutions is guaranteed by construction or in real-time situations.
- **Any-time property:** Positive correlation between run-time and solution quality or probability; typically more readily achieved by local search.
- **Complementarity:** Local and systematic can be fruitfully combined, *e.g.*, by using local search for finding solutions whose optimality is proven using systematic search.

Systematic *vs* Local Search:

- **Completeness:** Advantage of systematic search, but not always relevant,*e.g.*, when existence of solutions is guaranteed by construction or in real-time situations.
- **Any-time property:** Positive correlation between run-time and solution quality or probability; typically more readily achieved by local search.
- **Complementarity:** Local and systematic can be fruitfully combined, *e.g.*, by using local search for finding solutions whose optimality is proven using systematic search.

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.

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

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

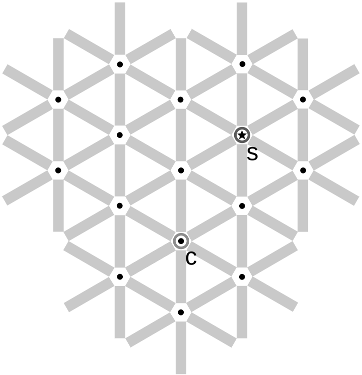
Many prominent local search algorithms use *randomised choices* in generating and modifying candidate solutions.

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:

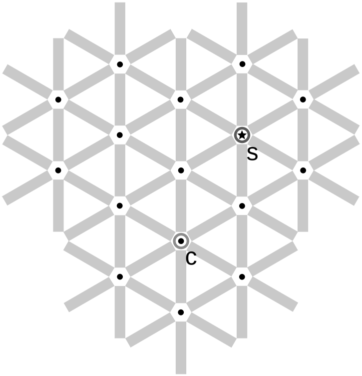
- Evolutionary Algorithms
- Simulated Annealing
- Lin-Kernighan Algorithm for TSP

Stochastic local search — global view



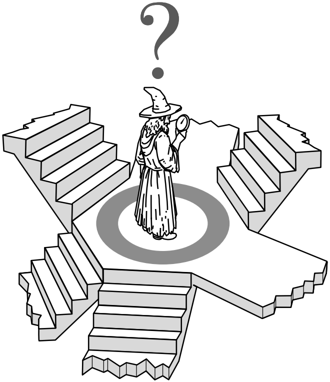
- vertices: candidate solutions (search positions)
- edges: connect neighbouring positions
- s: (optimal) solution
- c: current search position

Stochastic local search — 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** (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** (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 states 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)

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 states 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)

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 states 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)

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 states 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-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 *SLS-Decision*

procedure *SLS-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 *SLS-Decision*

```
procedure SLS-Decision(π)
   input: problem instance π ∈ Π
   output: solution s ∈ S′(π) or ∅
   (s, m) := init(π);

   while not terminate(π, s, m) do
      (s, m) := step(π, s, m);
   end

   if s ∈ S′(π) then
      return s
   else
      return ∅
   end
end SLS-Decision
```

```
procedure SLS-Decision(π)
   input: problem instance π ∈ Π
   output: solution s ∈ S′(π) or ∅
   (s, m) := init(π);

   while not terminate(π, s, m) do
       (s, m) := step(π, s, m);
   end



   if s ∈ S′(π) then
       return s
   else
       return ∅
   end
end SLS-Decision
```

```
procedure SLS-Minimisation(π')
   input: problem instance π' ∈ Π'
   output: solution s ∈ S'(π') or ∅
   (s, m) := init(π');
   ŝ := s;
   while not terminate(π', s, m) do
      (s, m) := step(π', s, m);
      if f(π', s) < f(π', ŝ) then
         ŝ := s;
      end
   end
   if ŝ ∈ S'(π') then
      return ŝ
   else
      return ∅
   end
end SLS-Minimisation
```

Note:

- Procedural versions of *init*, *step* and *terminate* implement sampling from respective probability distributions.

- Memory state m can consist of multiple independent attributes, *i.e.*, $M(\pi) := M_1 \times M_2 \times \ldots \times M_{l(\pi)}$.

- SLS algorithms realise *Markov processes*: behaviour in any *search state* (s, m) depends only on current position s and (limited) memory m.

Note:

- Procedural versions of *init*, *step* and *terminate* implement sampling from respective probability distributions.

- Memory state m can consist of multiple independent attributes, *i.e.*, $M(\pi) := M_1 \times M_2 \times \ldots \times M_{l(\pi)}$.

- SLS algorithms realise *Markov processes*: behaviour in any *search state* (s, m) depends only on current position s and (limited) memory m.

Note:

- Procedural versions of *init*, *step* and *terminate* implement sampling from respective probability distributions.

- Memory state m can consist of multiple independent attributes, *i.e.*, $M(\pi) := M_1 \times M_2 \times \ldots \times M_{l(\pi)}$.

- SLS algorithms realise *Markov processes*: behaviour in any *search state* (s, m) depends only on current position s and (limited) memory m.

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.*, $M := \{0\}$

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.*, $M := \{0\}$

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.*, $M := \{0\}$

Example: Uninformed random walk for SAT (continued)

- **initialisation:** uniform random choice from S, *i.e.*, $init()(a', m) := 1/\#S$ for all assignments a' and memory states m
- **step function:** uniform random choice from current neighbourhood, *i.e.*, $step(a, m)(a', m) := 1/\#N(a)$ for all assignments a and memory states m, where $N(a) := \{a' \in S \mid N(a, a')\}$ is the set of all neighbours of a.
- **termination:** when model is found, *i.e.*, $terminate(a, m) := 1$ if a is a model of F, and 0 otherwise.

Example: Uninformed random walk for SAT (continued)

- **initialisation:** uniform random choice from S, *i.e.*, $init()(a', m) := 1/\#S$ for all assignments a' and memory states m
- **step function:** uniform random choice from current neighbourhood, *i.e.*, $step(a, m)(a', m) := 1/\#N(a)$ for all assignments a and memory states m, where $N(a) := \{a' \in S \mid N(a, a')\}$ is the set of all neighbours of a.
- **termination:** when model is found, *i.e.*, $terminate(a, m) := 1$ if a is a model of F, and 0 otherwise.

Example: Uninformed random walk for SAT (continued)

- **initialisation:** uniform random choice from S, *i.e.*, $init()(a', m) := 1/\#S$ for all assignments a' and memory states m
- **step function:** uniform random choice from current neighbourhood, *i.e.*, $step(a, m)(a', m) := 1/\#N(a)$ for all assignments a and memory states m, where $N(a) := \{a' \in S \mid N(a, a')\}$ is the set of all neighbours of a.
- **termination:** when model is found, *i.e.*, $terminate(a, m) := 1$ if a is a model of F, and 0 otherwise.

Definition:

- *neighbourhood (set)* of candidate solution s:
  $N(s) := \{s' \in S \mid N(s, s')\}$
- *neighbourhood graph* of problem instance π:
  $G_N(\pi) := (S(\pi), N(\pi))$

Note: Diameter of G_N = worst-case lower bound for number of search steps required for reaching (optimal) solutions

Example:

SAT instance with n variables, 1-flip neighbourhood:
G_N = n-dimensional hypercube; diameter of $G_N = n$.

Definition:

- *neighbourhood (set)* of candidate solution s:
  $N(s) := \{s' \in S \mid N(s, s')\}$
- *neighbourhood graph* of problem instance π:
  $G_N(\pi) := (S(\pi), N(\pi))$

Note: Diameter of G_N = worst-case lower bound for number of search steps required for reaching (optimal) solutions

Example:

SAT instance with n variables, 1-flip neighbourhood:
G_N = n-dimensional hypercube; diameter of $G_N = n$.

Definition:

- *neighbourhood (set)* of candidate solution s:
  $N(s) := \{s' \in S \mid N(s, s')\}$
- *neighbourhood graph* of problem instance π:
  $G_N(\pi) := (S(\pi), N(\pi))$

Note: Diameter of G_N = worst-case lower bound for number of search steps required for reaching (optimal) solutions

Example:

SAT instance with n variables, 1-flip neighbourhood:
G_N = n-dimensional hypercube; diameter of $G_N = n$.

Definition:

- *neighbourhood (set)* of candidate solution s:
  $N(s) := \{s' \in S \mid N(s, s')\}$
- *neighbourhood graph* of problem instance π:
  $G_N(\pi) := (S(\pi), N(\pi))$

Note: Diameter of G_N = worst-case lower bound for number of search steps required for reaching (optimal) solutions

Example:

SAT instance with n variables, 1-flip neighbourhood:
G_N = n-dimensional hypercube; diameter of $G_N = n$.

Definition:

k-exchange neighbourhood: candidate solutions s, s' are neighbours iff s differs from s' in at most k solution components

Examples:

- 1-flip neighbourhood for SAT
  (solution components = single variable assignments)
- 2-exchange neighbourhood for TSP
  (solution components = edges in given graph)

Definition:

k-exchange neighbourhood: candidate solutions s, s' are neighbours iff s differs from s' in at most k solution components

Examples:

- 1-flip neighbourhood for SAT
  (solution components = single variable assignments)
- 2-exchange neighbourhood for TSP
  (solution components = edges in given graph)

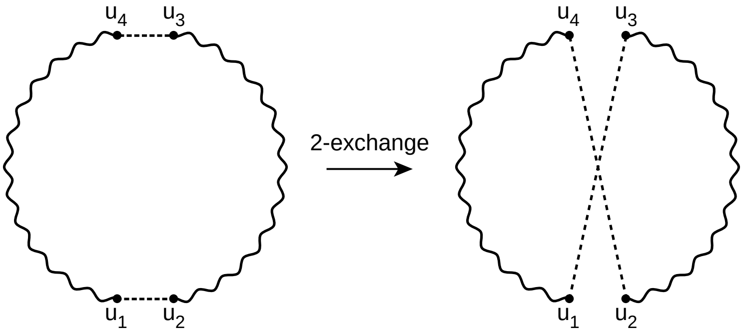
Definition:

k-exchange neighbourhood: candidate solutions s, s' are neighbours iff s differs from s' in at most k solution components

Examples:

- 1-flip neighbourhood for SAT
  (solution components $=$ single variable assignments)
- 2-exchange neighbourhood for TSP
  (solution components $=$ edges in given graph)

Search steps in the 2-exchange neighbourhood for the TSP



Definition:

- *Search step* (or *move*): pair of search states s, s' for which s can be reached from s in one step, *i.e.*, $N(s, s')$ and $step(s, m)(s', m') > 0$ for some memory states $m, m' \in M$.
- *Search trajectory*: finite sequence of search states $(s_0, s_1, \ldots, s_k)$ 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 zero, *i.e.*, $init(s_0, m) > 0$ for some memory state $m \in M$.
- *Search strategy*: specified by *init* and *step* function; to some extent independent of problem instance and other components of SLS algorithm.

Definition:

- *Search step* (or *move*): pair of search states s, s' for which s can be reached from s in one step, *i.e.*, $N(s, s')$ and $step(s, m)(s', m') > 0$ for some memory states $m, m' \in M$.
- *Search trajectory*: finite sequence of search states $(s_0, s_1, \ldots, s_k)$ 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 zero, *i.e.*, $init(s_0, m) > 0$ for some memory state $m \in M$.
- *Search strategy*: specified by *init* and *step* function; to some extent independent of problem instance and other components of SLS algorithm.

Definition:

- *Search step* (or *move*): pair of search states s, s' for which s can be reached from s in one step, *i.e.*, $N(s, s')$ and $step(s, m)(s', m') > 0$ for some memory states $m, m' \in M$.

- *Search trajectory*: finite sequence of search states $(s_0, s_1, \ldots, s_k)$ 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 zero, *i.e.*, $init(s_0, m) > 0$ for some memory state $m \in M$.

- *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,
  i.e., for all $s, s' \in S$, $init(s) := step(s)(s') := 1/\#S$

Uninformed Random Walk

- does not use memory
- *init*: uniform random choice from S
- *step*: uniform random choice from current neighbourhood,
  i.e., for all $s, s' \in S$, $step(s)(s') := 1/\#N(s)$ if $N(s, s')$,
  and 0 otherwise

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

Uninformed Random Picking

- $N := S \times S$
- does not use memory
- *init*, *step*: uniform random choice from S,
  i.e., for all $s, s' \in S$, $init(s) := step(s)(s') := 1/\#S$

Uninformed Random Walk

- does not use memory
- *init*: uniform random choice from S
- *step*: uniform random choice from current neighbourhood,
  i.e., for all $s, s' \in S$, $step(s)(s') := 1/\#N(s)$ if $N(s, s')$,
  and 0 otherwise

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

Uninformed Random Picking

- $N := S \times S$
- does not use memory
- *init*, *step*: uniform random choice from S,
  i.e., for all $s, s' \in S$, $init(s) := step(s)(s') := 1/\#S$

Uninformed Random Walk

- does not use memory
- *init*: uniform random choice from S
- *step*: uniform random choice from current neighbourhood,
  i.e., for all $s, s' \in S$, $step(s)(s') := 1/\#N(s)$ if $N(s, s')$,
  and 0 otherwise

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 (*e.g.*, dynamic local search).

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 (*e.g.*, dynamic local search).

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 (*e.g.*, dynamic local search).

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 (*e.g.*, dynamic local search).

Iterative Improvement (II)

- does not use memory
- *init*: uniform random choice from S
- *step*: uniform random choice from improving neighbours, *i.e.*, $step(s)(s') := 1/\#I(s)$ if $s' \in I(s)$, and 0 otherwise, where $I(s) := \{s' \in S \mid N(s, s') \wedge g(s') < g(s)\}$
- terminates when no improving neighbour available (to be revisited later)

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

Iterative Improvement (II)

- does not use memory
- *init*: uniform random choice from S
- *step*: uniform random choice from improving neighbours, *i.e.*, $step(s)(s') := 1/\#I(s)$ if $s' \in I(s)$, and 0 otherwise, where $I(s) := \{s' \in S \mid N(s, s') \wedge g(s') < g(s)\}$
- terminates when no improving neighbour available (to be revisited later)

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

Iterative Improvement (II)

- does not use memory
- *init*: uniform random choice from S
- *step*: uniform random choice from improving neighbours, *i.e.*, $step(s)(s') := 1/\#I(s)$ if $s' \in I(s)$, and 0 otherwise, where $I(s) := \{s' \in S \mid N(s, s') \wedge g(s') < g(s)\}$
- terminates when no improving neighbour available (to be revisited later)

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

Iterative Improvement (II)

- does not use memory
- *init*: uniform random choice from S
- *step*: uniform random choice from improving neighbours, *i.e.*, $step(s)(s') := 1/\#I(s)$ if $s' \in I(s)$, and 0 otherwise, where $I(s) := \{s' \in S \mid N(s, s') \wedge g(s') < g(s)\}$
- terminates when no improving neighbour available (to be revisited later)

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.*, $init()(a') := 1/\#S$ for all assignments a'

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.*, $init()(a') := 1/\#S$ 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.*, $terminate(a) = \top$ if $I(a) = \emptyset$, and $\bot$ otherwise.

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.*, $terminate(a) = \top$ if $I(a) = \emptyset$, and $\bot$ otherwise.

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.*, $terminate(a) = \top$ if $I(a) = \emptyset$, and $\bot$ otherwise.

Incremental updates (aka delta evaluations)

- **Key idea:** calculate *effects of differences* between current search position s and neighbours s' on evaluation function value.
- Evaluation function values often consist of *independent contributions of solution components*; hence, $g(s)$ can be efficiently calculated from $g(s')$ by differences between s, s' in terms of solution components.
- Typically crucial for the efficient implementation of II algorithms (and other SLS techniques).

Incremental updates (aka delta evaluations)

- **Key idea:** calculate *effects of differences* between current search position s and neighbours s' on evaluation function value.
- Evaluation function values often consist of *independent contributions of solution components*; hence, $g(s)$ can be efficiently calculated from $g(s')$ by differences between s, s' in terms of solution components.
- Typically crucial for the efficient implementation of II algorithms (and other SLS techniques).

Incremental updates (aka delta evaluations)

- **Key idea:** calculate *effects of differences* between current search position s and neighbours s' on evaluation function value.
- Evaluation function values often consist of *independent contributions of solution components*; hence, $g(s)$ can be efficiently calculated from $g(s')$ by differences between s, s' in terms of solution components.
- Typically crucial for the efficient implementation of II algorithms (and other SLS techniques).

Example: Incremental updates for TSP

- solution components = edges of given graph G
- standard 2-exchange neighbhourhood, *i.e.*, neighbouring round trips p, p' differ in two edges
- $w(p') := w(p)$ − edges in p but not in p'
  \+ edges in p' but not in p

Note: Constant time (4 arithmetic operations), compared to linear time (n arithmethic operations for graph with n vertices) for computing $w(p')$ from scratch.

Example: Incremental updates for TSP

- solution components = edges of given graph G
- standard 2-exchange neighbhourhood, *i.e.*, neighbouring round trips p, p' differ in two edges
- $w(p') := w(p) -$ edges in p but not in p'
  $+$ edges in p' but not in p

Note: Constant time (4 arithmetic operations), compared to linear time (n arithmethic operations for graph with n vertices) for computing $w(p')$ from scratch.

Example: Incremental updates for TSP

- solution components = edges of given graph G
- standard 2-exchange neighbhourhood, *i.e.*, neighbouring round trips p, p' differ in two edges
- $w(p') := w(p)$ $-$ edges in p but not in p'
  $+$ edges in p' but not in p

Note: Constant time (4 arithmetic operations), compared to linear time (n arithmethic operations for graph with n vertices) for computing $w(p')$ from scratch.

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.

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.

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.

Computational complexity of local search

For a local search algorithm to be effective, search initialisation and individual search steps should be efficiently computable.

Complexity class $\mathcal{PLS}$: class of problems for which a local search algorithm exists with polynomial time complexity for:

- search initialisation
- any single search step, including computation of any evaluation function value

For any problem in $\mathcal{PLS}$ ...

- local optimality can be verified in polynomial time
- improving search steps can be computed in polynomial time
- *but:* finding local optima may require super-polynomial time

Note: All time-complexities are stated for deterministic machines.

Computational complexity of local search

For a local search algorithm to be effective, search initialisation and individual search steps should be efficiently computable.

Complexity class $\mathcal{PLS}$: class of problems for which a local search algorithm exists with polynomial time complexity for:

- search initialisation
- any single search step, including computation of any evaluation function value

For any problem in $\mathcal{PLS}$ ...

- local optimality can be verified in polynomial time
- improving search steps can be computed in polynomial time
- *but:* finding local optima may require super-polynomial time

Note: All time-complexities are stated for deterministic machines.

Computational complexity of local search

For a local search algorithm to be effective, search initialisation and individual search steps should be efficiently computable.

Complexity class $\mathcal{PLS}$: class of problems for which a local search algorithm exists with polynomial time complexity for:

- search initialisation
- any single search step, including computation of any evaluation function value

For any problem in $\mathcal{PLS}$ ...

- local optimality can be verified in polynomial time
- improving search steps can be computed in polynomial time
- *but:* finding local optima may require super-polynomial time

Note: All time-complexities are stated for deterministic machines.

Computational complexity of local search

For a local search algorithm to be effective, search initialisation and individual search steps should be efficiently computable.

Complexity class $\mathcal{PLS}$: class of problems for which a local search algorithm exists with polynomial time complexity for:

- search initialisation
- any single search step, including computation of any evaluation function value

For any problem in $\mathcal{PLS}$ ...

- local optimality can be verified in polynomial time
- improving search steps can be computed in polynomial time
- *but:* finding local optima may require super-polynomial time

Note: All time-complexities are stated for deterministic machines.

Computational complexity of local search (2)

$\mathcal{PLS}$-complete: Among the most difficult problems in $\mathcal{PLS}$;
if for any of these problems local optima can be found
in polynomial time, the same would hold for all problems in $\mathcal{PLS}$.

Some complexity results:

- TSP with k-exchange neighbourhood with $k > 3$ is $\mathcal{PLS}$-complete.
- TSP with 2- or 3-exchange neighbourhood is in $\mathcal{PLS}$, but $\mathcal{PLS}$-completeness is unknown.

Computational complexity of local search (2)

$\mathcal{PLS}$-complete: Among the most difficult problems in $\mathcal{PLS}$;
if for any of these problems local optima can be found
in polynomial time, the same would hold for all problems in $\mathcal{PLS}$.

Some complexity results:

- TSP with k-exchange neighbourhood with $k > 3$ is $\mathcal{PLS}$-complete.
- TSP with 2- or 3-exchange neighbourhood is in $\mathcal{PLS}$, but $\mathcal{PLS}$-completeness is unknown.

Computational complexity of local search (2)

$\mathcal{PLS}$-complete: Among the most difficult problems in $\mathcal{PLS}$;
if for any of these problems local optima can be found
in polynomial time, the same would hold for all problems in $\mathcal{PLS}$.

Some complexity results:

- TSP with k-exchange neighbourhood with $k > 3$ is $\mathcal{PLS}$-complete.
- TSP with 2- or 3-exchange neighbourhood is in $\mathcal{PLS}$, but $\mathcal{PLS}$-completeness is unknown.

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.

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.

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 evalution 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.

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 evalution 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.

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 evalution 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.

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 evalution 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.

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 evalution 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.