DM811 – Fall 2009

Heuristics for Combinatorial Optimization

Lecture 8

Local Search

Marco Chiarandini

Deptartment of Mathematics & Computer Science
University of Southern Denmark

Outline

Definitions: Local Search Algorithm (1)

Given a (combinatorial) optimization problem Π and one of its instances π:

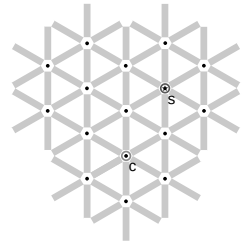
- search space $S(\pi)$
  specified by candidate solution representation:
  discrete structures: sequences, permutations, graphs, partitions
  (*e.g.*, for SAT: array, sequence of all truth assignments
  to propositional variables)

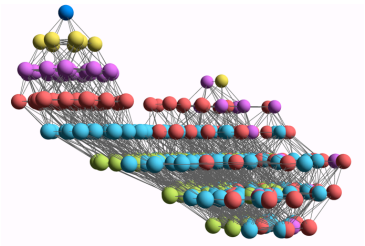
  Note: solution set $S'(\pi) \subseteq S(\pi)$
  (*e.g.*, for SAT: models of given formula)

- evaluation function $f(\pi) : S(\pi) \mapsto \mathbf{R}$
  (*e.g.*, for SAT: number of false clauses)

- neighborhood function, $\mathcal{N}(\pi) : S \mapsto 2^{S(\pi)}$
  (*e.g.*, for SAT: neighboring variable assignments differ
  in the truth value of exactly one variable)

Local search — global view





- vertices: candidate solutions (search positions)
- vertex labels: evaluation function
- edges: connect "neighboring" positions
- s: (optimal) solution
- c: current search position

Iterative Improvement

Iterative Improvement (II):
determine initial candidate solution s
while s has better neighbors **do**
  choose a neighbor s' of s such that $f(s') < f(s)$
  $s := s'$

- If more than one neighbor have better cost then need to choose one
  ➡ pivoting rule
- The procedure ends in a local optimum $\hat{s}$:
  Def.: Local optimum $\hat{s}$ wrt N if $f(\hat{s}) \leq f(s)\ \forall s \in N(\hat{s})$
- Issue: how to avoid getting trapped in bad local optima?
  - use more complex neighborhood functions
  - restart
  - allow non-improving moves

Definition: Local Search Algorithm (2)

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

LS-Decision(π)
input: problem instance $\pi \in \Pi$
output: solution $s \in S'(\pi)$ **or** $\emptyset$

$(s, m) := \texttt{init}(\pi)$

while not $\texttt{terminate}(\pi, s, m)$ **do**
  $(s, m) := \texttt{step}(\pi, s, m)$

if $s \in S'(\pi)$ **then**
  return s
else
  return $\emptyset$

LS-Minimization(π')
input: problem instance $\pi' \in \Pi'$
output: solution $s \in S'(\pi')$ **or** $\emptyset$

$(s, m) := \texttt{init}(\pi')$;
$\hat{s} := s$;
while not $\texttt{terminate}(\pi', s, m)$ **do**
  $(s, m) := \texttt{step}(\pi', s, m)$;
  if $f(\pi', s) < f(\pi', \hat{s})$ **then**
    $\hat{s} := s$;

if $\hat{s} \in S'(\pi')$ **then**
  return $\hat{s}$
else
  return $\emptyset$

Definition: Local Search Algorithm

For given problem instance π:

1. search space $S(\pi)$
2. neighborhood relation $\mathcal{N}(\pi) \subseteq S(\pi) \times S(\pi)$
3. evaluation function $f(\pi) : S \mapsto \mathbf{R}$
4. set of memory states $M(\pi)$
5. initialization function $\texttt{init} : \emptyset \mapsto \mathcal{P}(S(\pi) \times M(\pi))$
6. step function $\texttt{step} : S(\pi) \times M(\pi) \mapsto \mathcal{P}(S(\pi) \times M(\pi))$
7. termination predicate $\texttt{terminate} : S(\pi) \times M(\pi) \mapsto \mathcal{P}(\{\top, \bot\})$

Example: Uninformed random walk for SAT (1)

- **search space** S**:** set of all truth assignments to variables in given formula F
  (**solution set** S'**:** set of all models of F)
- **neighborhood relation** $\mathcal{N}$**:** *1-flip neighborhood*, *i.e.*, assignments are neighbors under $\mathcal{N}$ iff they differ in the truth value of exactly one variable
- **evaluation function** not used, or $f(s) = 0$ if model $f(s) = 1$ otherwise
- **memory:** not used, *i.e.*, $M := \{0\}$

Example: Uninformed random walk for SAT (2)

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

Definition: LS Algorithm Components

Search Space

Defined by the solution representation:

- permutations
  - linear (scheduling)
  - circular (TSP)
- arrays (assignment problems: GCP)
- sets or lists (partition problems: Knapsack)

Definition: LS Algorithm Components

Neighborhood function

Also defined as: $\mathcal{N} : S \times S \to \{T, F\}$ or $\mathcal{N} \subseteq S \times S$

- neighborhood (set) of candidate solution s: $N(s) := \{s' \in S \mid \mathcal{N}(s, s')\}$
- neighborhood size is $|N(s)|$
- neighborhood is symmetric if: $s' \in N(s) \Rightarrow s \in N(s')$
- neighborhood graph of (S, f, N, π) is a directed vertex-weighted graph: $G_{\mathcal{N}}(\pi) := (V, A)$ with $V = S(\pi)$ and $(uv) \in A \Leftrightarrow v \in N(u)$ (if symmetric neighborhood $\Rightarrow$ undirected graph)

A note on notation: N when set, $\mathcal{N}$ when collection of sets or function

A neighborhood function is also defined by means of an operator.

An operator Δ is a collection of operator functions $\delta : S \to S$ such that

$$s' \in N(s) \iff \exists\, \delta \in \Delta, \delta(s) = s'$$

Definition

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

Examples:

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

Definition: LS Algorithm Components

Definition:

- Local minimum: search position without improving neighbors wrt given evaluation function f and neighborhood $\mathcal{N}$,
  i.e., position $s \in S$ such that $f(s) \leq f(s')$ for all $s' \in N(s)$.
- Strict local minimum: search position $s \in S$ such that
  $f(s) < f(s')$ for all $s' \in N(s)$.
- *Local maxima* and *strict local maxima*: defined analogously.

Definition: LS Algorithm Components

Note:

- Local search implements a walk through the neighborhood graph
- Procedural versions of `init`, `step` and `terminate` implement sampling from respective probability distributions.
- Memory state m can consist of multiple independent attributes, *i.e.*,
  $M(\pi) := M_1 \times M_2 \times \ldots \times M_{l(\pi)}$.
- Local search algorithms are Markov processes:
  behavior in any search state $\{s, m\}$ depends only
  on current position s and (limited) memory m.

Definition: LS Algorithm Components

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

- 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 initializing the search at s_0 is greater zero, *i.e.*, $\texttt{init}(\{s_0, m\}) > 0$ 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 LS algorithm.
  - random
  - based on evaluation function
  - based on memory

Uninformed Random Picking

- $\mathcal{N} := S \times S$
- does not use memory and evaluation function
- `init`, `step`: uniform random choice from S,
  i.e., for all $s, s' \in S$, $\texttt{init}(s) := \texttt{step}(\{s\}, \{s'\}) := 1/|S|$

Uninformed Random Walk

- does not use memory and evaluation function
- `init`: uniform random choice from S
- `step`: uniform random choice from current neighborhood,
  i.e., for all $s, s' \in S$, $\texttt{step}(\{s\}, \{s'\}) := \begin{cases} 1/|N(s)| & \text{if } s' \in N(s) \\ 0 & \text{otherwise} \end{cases}$

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

Definition: LS Algorithm Components

Evaluation (or cost) function:

- function $f(\pi) : S(\pi) \mapsto \mathbf{R}$ 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 neighbors of current search position to provide guidance to search process.

Evaluation *vs* objective functions:

- *Evaluation function*: part of LS algorithm.
- *Objective function*: integral part of optimization problem.
- Some LS methods use evaluation functions different from given objective function (*e.g.*, dynamic local search).

Iterative Improvement

- does not use memory
- `init`: uniform random choice from S
- `step`: uniform random choice from improving neighbors,
  i.e., $\texttt{step}(\{s\}, \{s'\}) := 1/|I(s)|$ if $s' \in I(s)$, and 0 otherwise,
  where $I(s) := \{s' \in S \mid \mathcal{N}(s, s') \text{and} f(s') < f(s)\}$
- terminates when no improving neighbor available
- different variants through modifications of step function

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

Pivoting rule

There might be more than one neighbor that have better cost.

Pivoting rule decides which to choose:

- Best Improvement (aka *gradient descent*, *steepest descent*, *greedy hill-climbing*): Choose maximally improving neighbor,
  i.e., randomly select from $I^*(s) := \{s' \in N(s) \mid f(s') = g^*\}$,
  where $g^* := \min\{f(s') \mid s' \in N(s)\}$.

  Note: Requires evaluation of all neighbors in each step.

- First Improvement: Evaluate neighbors in fixed order,
  choose first improving step encountered.

  Note: Can be much more efficient than Best Improvement; order of evaluation can have significant impact on performance.

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)
- **neighborhood relation** $\mathcal{N}$: 1-flip neighborhood
  (as in Uninformed Random Walk for SAT)
- **memory:** not used, *i.e.*, $M := \{0\}$
- **initialization:** uniform random choice from S, *i.e.*, $\texttt{init}(\emptyset, \{a'\}) := 1/|S|$ for all assignments a'
- **evaluation function:** $f(a) :=$ number of clauses in F that are *unsatisfied* under assignment a
  (*Note:* $f(a) = 0$ iff a is a model of F.)
- **step function**: uniform random choice from improving neighbors, *i.e.*,
  $\texttt{step}(a, a') := 1/\#I(a)$ if $s' \in I(a)$,
  and 0 otherwise, where $I(a) := \{a' \mid \mathcal{N}(a, a') \wedge f(a') < f(a)\}$
- **termination**: when no improving neighbor is available
  i.e., $\texttt{terminate}(a, \top) := 1$ if $I(a) = \emptyset$, and 0 otherwise.

Iterative Improvement (2 OPT)

TSP-2opt-first(s)
input: *an initial candidate tour* $s \in S(\in)$
output: *a local optimum* $s \in S(\pi)$

$\Delta = 0$;
for $i = 1$ to $n - 2$ **do**
  if $i = 1$ **then** $n' = n - 1$ **else** $n' = n$
  for $j = i + 2$ to n' **do**
    $\Delta_{ij} = d(c_i, c_j) + d(c_{i+1}, c_{j+1}) - d(c_i, c_{i+1}) - d(c_j, c_{j+1})$
    if $\Delta_{ij} < 0$ **then**
      UpdateTour(s, i, j)

Iterative Improvement (2 OPT)

TSP-2opt-first(s)
input: *an initial candidate tour* $s \in S(\in)$
output: *a local optimum* $s \in S(\pi)$

$\Delta = 0$;
Improvement=TRUE;
while Improvement==TRUE **do**
  Improvement=FALSE;
  for $i = 1$ to $n - 2$ **do**
    if $i = 1$ **then** $n' = n - 1$ **else** $n' = n$
    for $j = i + 2$ to n' **do**
      $\Delta_{ij} = d(c_i, c_j) + d(c_{i+1}, c_{j+1}) - d(c_i, c_{i+1}) - d(c_j, c_{j+1})$
      if $\Delta_{ij} < 0$ **then**
        UpdateTour(s, i, j)
        Improvement=TRUE

Example: Iterative Improvement for k-col

- **search space** S: set of all k-colorings of G
- **solution set** S': set of all proper k-coloring of F
- **neighborhood relation** N: 1-exchange neighborhood
  (as in Uninformed Random Walk)
- **memory:** not used, *i.e.*, $M := \{0\}$
- **initialization:** uniform random choice from S, *i.e.*, $\texttt{init}()(\varphi') := 1/|S|$ for all colorings φ'
- **step function**:
  - **evaluation function:** $f(\varphi) :=$ number of edges in G whose ending vertices are assigned the same color under assignment φ
    (*Note:* $f(\varphi) = 0$ iff φ is a proper coloring of G.)
  - **move mechanism:** uniform random choice from improving neighbors, *i.e.*,
    $\texttt{step}(\varphi)(\varphi') := 1/|I(\varphi)|$ if $s' \in I(\varphi)$,
    and 0 otherwise, where $I(\varphi) := \{\varphi' \mid N(\varphi, \varphi') \wedge f(\varphi') < f(\varphi)\}$
- **termination**: when no improving neighbor is available
  i.e., $\texttt{terminate}(\varphi)(\top) := 1$ if $I(\varphi) = \emptyset$, and 0 otherwise.

Example: Random-order first improvement for the TSP

- **Given:** TSP instance G with vertices $v_1, v_2, \ldots, v_n$.
- search space: Hamiltonian cycles in G;
  use standard 2-exchange neighborhood
- **Initialization:**
  search position := fixed canonical path $< v_1, v_2, \ldots, v_n, v_1 >$
  P := random permutation of $\{1, 2, \ldots, n\}$
- **Search steps:** determined using first improvement w.r.t. $f(p)$ = weight of path p, evaluating neighbors in order of P (does not change throughout search)
- **Termination:** when no improving search step possible (local minimum)

Example: Random order first improvement for SAT

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

```
choose assignment φ of truth values to all variables in F
   uniformly at random;
steps := 0;
while ¬(φ satisfies F) and (steps < maxSteps) do
    select x uniformly at random from {x'|x' is a variable in F and
    changing value of x' in φ decreases the number of unsatisfied clauses}
    steps := steps+1;
if φ satisfies F then
    return φ
else
    return ∅
```


Escaping Local Optima

- Enlarge the neighborhood
- Restart: re-initialize search whenever a local optimum is encountered.
  (Often rather ineffective due to cost of initialization.)
- 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: None of these mechanisms is guaranteed to always escape effectively from local optima.

Diversification *vs* Intensification

- Goal-directed and randomized components of LS 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/Picking (URW/P): *diversification* strategy.

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

Computational Complexity of Local Search (1)

For a local search algorithm to be effective, search initialization 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 initialization
- 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

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

Outline

Learning goals of this section

- Review basic theoretical concepts
- Learn about techniques and goals of experimental search space analysis.
- Develop intuition on which features of local search are adequate to contrast a specific situation.

Definitions

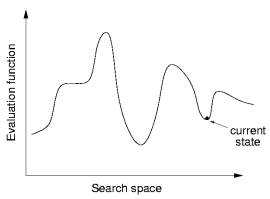
- Search space S
- Neighborhood function $\mathcal{N} : S \subseteq 2^S$
- Evaluation function $f(\pi) : S \mapsto \mathbf{R}$
- Problem instance π

Definition:

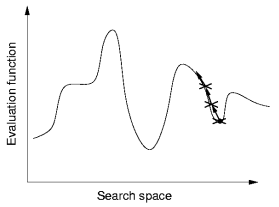
The **search landscape** L is the vertex-labeled neighborhood graph given by the triplet $\mathcal{L} = (S(\pi), N(\pi), f(\pi))$.

Ideal visualization of metaheuristic principles

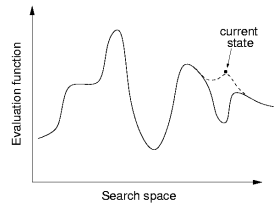
- Simplified landscape representation



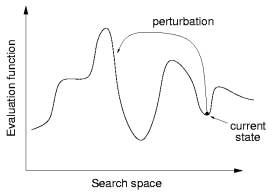
- Tabu Search



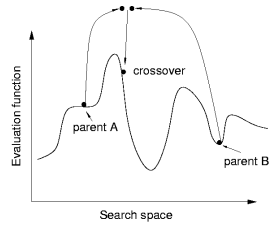
- Guided Local Search



- Iterated Local Search



- Evolutionary Alg.



Fundamental Search Space Properties

The behavior and performance of an LS algorithm on a given problem instance crucially depends on properties of the respective search space.

Simple properties of search space S:

- search space size $|S|$
- reachability: solution j is reachable from solution i if neighborhood graph has a path from i to j.
  - strongly connected neighborhood graph
  - weakly optimally connected neighborhood graph
- distance between solutions
- neighborhood size (ie, degree of vertices in neigh. graph)
- neighborhood examination
- relation between different neighborhood functions

Solution Representations and Neighborhoods

Three different types of solution representations:

- Permutation
  - linear permutation: Single Machine Total Weighted Tardiness Problem
  - circular permutation: Traveling Salesman Problem
- Assignment: Graph Coloring Problem, SAT, CSP
- Set, Partition: Knapsack, Max Independent Set

A neighborhood function $\mathcal{N} : S \to S \times S$ is also defined through an operator. An operator Δ is a collection of operator functions $\delta : S \to S$ such that

$$s' \in N(s) \iff \exists \delta \in \Delta \mid \delta(s) = s'$$

Permutations

$\Pi(n)$ indicates the set all permutations of the numbers $\{1, 2, \ldots, n\}$

$(1, 2 \ldots, n)$ is the identity permutation ι.

If $\pi \in \Pi(n)$ and $1 \leq i \leq n$ then:

- π_i is the element at position i
- $pos_\pi(i)$ is the position of element i

Alternatively, a permutation is a bijective function $\pi(i) = \pi_i$

The permutation product $\pi \cdot \pi'$ is the composition $(\pi \cdot \pi')_i = \pi'(\pi(i))$

For each π there exists a permutation such that $\pi^{-1} \cdot \pi = \iota$

$$\Delta_N \subset \Pi$$

Neighborhood Operators for Linear Permutations

Swap operator

$$\Delta_S = \{\delta_S^i | 1 \leq i \leq n\}$$

$$\delta_S^i(\pi_1 \ldots \pi_i \pi_{i+1} \ldots \pi_n) = (\pi_1 \ldots \pi_{i+1} \pi_i \ldots \pi_n)$$

Interchange operator

$$\Delta_X = \{\delta_X^{ij} | 1 \leq i < j \leq n\}$$

$$\delta_X^{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \pi_{i+1} \ldots \pi_{j-1} \pi_i \pi_{j+1} \ldots \pi_n)$$

($\equiv$ set of all transpositions)

Insert operator

$$\Delta_I = \{\delta_I^{ij} | 1 \leq i \leq n, 1 \leq j \leq n, j \neq i\}$$

$$\delta_I^{ij}(\pi) = \begin{cases} (\pi_1 \ldots \pi_{i-1} \pi_{i+1} \ldots \pi_j \pi_i \pi_{j+1} \ldots \pi_n) & i < j \\ (\pi_1 \ldots \pi_j \pi_i \pi_{j+1} \ldots \pi_{i-1} \pi_{i+1} \ldots \pi_n) & i > j \end{cases}$$

Neighborhood Operators for Circular Permutations

Reversal (2-edge-exchange)

$$\Delta_R = \{\delta_R^{ij} | 1 \leq i < j \leq n\}$$

$$\delta_R^{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \ldots \pi_i \pi_{j+1} \ldots \pi_n)$$

Block moves (3-edge-exchange)

$$\Delta_B = \{\delta_B^{ijk} | 1 \leq i < j < k \leq n\}$$

$$\delta_B^{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \ldots \pi_k \pi_i \ldots \pi_{j-1} \pi_{k+1} \ldots \pi_n)$$

Short block move (Or-edge-exchange)

$$\Delta_{SB} = \{\delta_{SB}^{ij} | 1 \leq i < j \leq n\}$$

$$\delta_{SB}^{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \pi_{j+1} \pi_{j+2} \pi_i \ldots \pi_{j-1} \pi_{j+3} \ldots \pi_n)$$

Neighborhood Operators for Assignments

An assignment can be represented as a mapping
$\sigma : \{X_1 \ldots X_n\} \to \{v : v \in D, |D| = k\}$:

$$\sigma = \{X_i = v_i, X_j = v_j, \ldots\}$$

One-exchange operator

$$\Delta_{1E} = \{\delta_{1E}^{il} | 1 \leq i \leq n, 1 \leq l \leq k\}$$

$$\delta_{1E}^{il}(\sigma) = \{\sigma : \sigma'(X_i) = v_l \text{ and } \sigma'(X_j) = \sigma(X_j) \ \forall j \neq i\}$$

Two-exchange operator

$$\Delta_{2E} = \{\delta_{2E}^{ij} | 1 \leq i < j \leq n\}$$

$$\delta_{2E}^{ij}\{\sigma : \sigma'(X_i) = \sigma(X_j), \ \sigma'(X_j) = \sigma(X_i) \text{ and } \sigma'(X_l) = \sigma(X_l) \ \forall l \neq i, j\}$$

Neighborhood Operators for Partitions or Sets

An assignment can be represented as a partition of objects selected and not selected $s : \{X\} \to \{C, \overline{C}\}$
(it can also be represented by a bit string)

One-addition operator

$$\Delta_{1E} = \{\delta^{v}_{1E} | v \in \overline{C}\}$$

$$\delta^{v}_{1E}(s) = \left\{s : C' = C \cup v \text{ and } \overline{C}' = \overline{C} \setminus v\right\}$$

One-deletion operator

$$\Delta_{1E} = \{\delta^{v}_{1E} | v \in C\}$$

$$\delta^{v}_{1E}(s) = \left\{s : C' = C \setminus v \text{ and } \overline{C}' = \overline{C} \cup v\right\}$$

Swap operator

$$\Delta_{1E} = \{\delta^{v}_{1E} | v \in C, u \in \overline{C}\}$$

$$\delta^{v}_{1E}(s) = \left\{s : C' = C \cup u \setminus v \text{ and } \overline{C}' = \overline{C} \cup v \setminus u\right\}$$

Distances

Set of paths in $G_{\mathcal{N}}$ with $s, s' \in S$:

$$\Phi(s, s') = \{(s_1, \ldots, s_h) | s_1 = s, s_h = s' \, \forall i : 1 \leq i \leq h-1, \langle s_i, s_{i+1} \rangle \in E_N\}$$

If $\phi = (s_1, \ldots, s_h) \in \Phi(s, s')$ let $|\phi| = h$ be the length of the path; then the distance between any two solutions s, s' is the length of shortest path between s and s' in $G_{\mathcal{N}}$:

$$d_{\mathcal{N}}(s, s') = \min_{\phi \in \Phi(s,s')} |\Phi|$$

$\mathtt{diam}(G_{\mathcal{N}}) = \max\{d_{\mathcal{N}}(s, s') \mid s, s' \in S\}$ (= maximal distance between any two candidate solutions)
(= worst-case lower bound for number of search steps required for reaching (optimal) solutions.

Note: with permutations it is easy to see that:

$$d_{\mathcal{N}}(\pi, \pi') = d_{\mathcal{N}}(\pi^{-1} \cdot \pi', \iota)$$

Distances for Linear Permutation Representations

- Swap neighborhood operator
  computable in $O(n^2)$ by the precedence based distance metric:
  $d_S(\pi, \pi') = \#\{\langle i, j \rangle | 1 \leq i < j \leq n, pos_{\pi'}(\pi_j) < pos_{\pi'}(\pi_i)\}$.
  $\mathtt{diam}(G_{\mathcal{N}}) = n(n-1)/2$

- Interchange neighborhood operator
  Computable in $O(n) + O(n)$ since
  $d_X(\pi, \pi') = d_X(\pi^{-1} \cdot \pi', \iota) = n - c(\pi^{-1} \cdot \pi')$
  where $c(\pi)$ is the number of disjoint cycles that decompose a permutation.
  $\mathtt{diam}(G_{\mathcal{N}_X}) = n - 1$

- Insert neighborhood operator
  Computable in $O(n) + O(n \log(n))$ since
  $d_I(\pi, \pi') = d_I(\pi^{-1} \cdot \pi', \iota) = n - |lis(\pi^{-1} \cdot \pi')|$ where $lis(\pi)$ denotes the length of the longest increasing subsequence.
  $\mathtt{diam}(G_{\mathcal{N}_I}) = n - 1$

Distances for Circular Permutation Representations

- Reversal neighborhood operator
  sorting by reversal is known to be NP-hard
  surrogate in TSP: bond distance

- Block moves neighborhood operator
  unknown whether it is NP-hard but there does not exist a proved polynomial-time algorithm

Distances for Assignment Representations

- Hamming Distance

- An assignment can be seen as a partition of n in k mutually exclusive non-empty subsets

  One-exchange neighborhood operator
  The *partition-distance* $d_{1E}(\mathcal{P}, \mathcal{P}')$ between two partitions $\mathcal{P}$ and $\mathcal{P}'$ is the minimum number of elements that must be moved between subsets in $\mathcal{P}$ so that the resulting partition equals $\mathcal{P}'$.

  The partition-distance can be computed in polynomial time by solving an assignment problem. Given the assignment matrix M where in each cell (i, j) it is $|S_i \cap S'_j|$ with $S_i \in \mathcal{P}$ and $S'_j \in \mathcal{P}'$ and defined $A(\mathcal{P}, \mathcal{P}')$ the assignment of maximal sum then it is $d_{1E}(\mathcal{P}, \mathcal{P}') = n - A(\mathcal{P}, \mathcal{P}')$

Example: Search space size and diameter for the TSP

- Search space size $= (n-1)!/2$

- Insert neighborhood
  size $= (n-3)n$
  diameter $= n-2$

- 2-exchange neighborhood
  size $= \binom{n}{2} = n \cdot (n-1)/2$
  diameter in $[n/2, n-2]$

- 3-exchange neighborhood
  size $= \binom{n}{3} = n \cdot (n-1) \cdot (n-2)/6$
  diameter in $[n/3, n-1]$

Example: Search space size and diameter for SAT

SAT instance with n variables, 1-flip neighborhood:
$G_{\mathcal{N}} = n$-dimensional hypercube; diameter of $G_{\mathcal{N}} = n$.

Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two different neighborhood functions for the same instance (S, f, π) of a combinatorial optimization problem.

If for all solutions $s \in S$ we have $N_1(s) \subseteq N_2(s')$ then we say that $\mathcal{N}_2$ dominates $\mathcal{N}_1$

Example:

In TSP, 1-insert is dominated by 3-exchange.
(1-insert corresponds to 3-exchange and there are 3-exchnages that are not 1-insert)

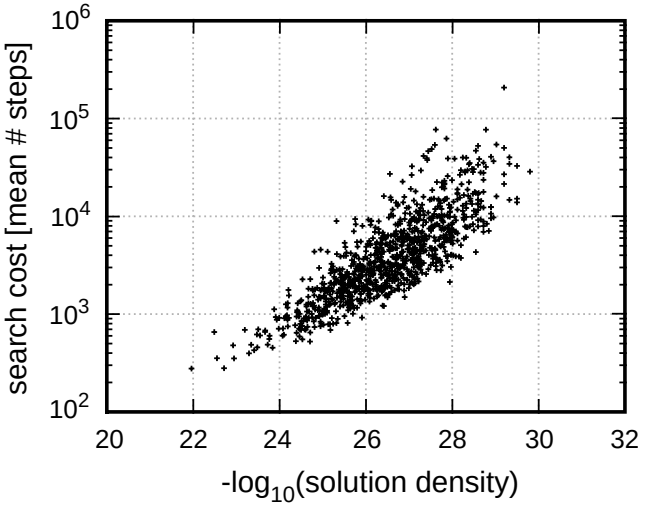
Other Search Space Properties

- number of (optimal) solutions $|S'|$, solution density $|S'|/|S|$
- distribution of solutions within the neighborhood graph

Solution densities and distributions can generally be determined by:

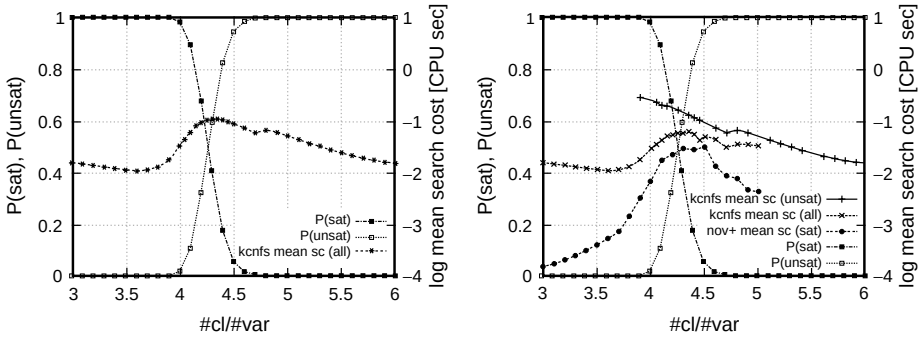
- exhaustive enumeration;
- sampling methods;
- counting algorithms (often variants of complete algorithms).

Example: Correlation between solution density and search cost for GWSAT over set of hard Random-3-SAT instances:

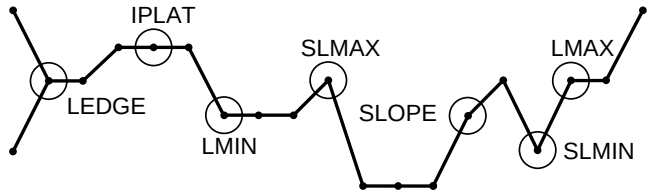


Phase Transition for 3-SAT

Random instances $\Rightarrow$ m clauses of n uniformly chosen variables



Classification of search positions



position type	$>$	$=$	$<$
SLMIN (strict local min)	+	−	−
LMIN (local min)	+	+	−
IPLAT (interior plateau)	−	+	−
SLOPE	+	−	+
LEDGE	+	+	+
LMAX (local max)	−	+	+
SLMAX (strict local max)	−	−	+

"+" = present, "−" absent; table entries refer to neighbors with larger (">") , equal ("="), and smaller ("<") evaluation function values

Example: Complete distribution of position types for hard Random-3-SAT instances

instance	*avg sc*	SLMIN	LMIN	IPLAT
uf20-91/easy	13.05	0%	0.11%	0%
uf20-91/medium	83.25	$< 0.01\%$	0.13%	0%
uf20-91/hard	563.94	$< 0.01\%$	0.16%	0%

instance	SLOPE	LEDGE	LMAX	SLMAX
uf20-91/easy	0.59%	99.27%	0.04%	$< 0.01\%$
uf20-91/medium	0.31%	99.40%	0.06%	$< 0.01\%$
uf20-91/hard	0.56%	99.23%	0.05%	$< 0.01\%$

(based on exhaustive enumeration of search space;
sc refers to search cost for GWSAT)

Example: Sampled distribution of position types for hard Random-3-SAT instances

instance	*avg sc*	SLMIN	LMIN	IPLAT
uf50-218/medium	615.25	0%	47.29%	0%
uf100-430/medium	3 410.45	0%	43.89%	0%
uf150-645/medium	10 231.89	0%	41.95%	0%

instance	SLOPE	LEDGE	LMAX	SLMAX
uf50-218/medium	$< 0.01\%$	52.71%	0%	0%
uf100-430/medium	0%	56.11%	0%	0%
uf150-645/medium	0%	58.05%	0%	0%

(based on sampling along GWSAT trajectories;
sc refers to search cost for GWSAT)

Local Minima

Note: Local minima impede local search progress.

Simple properties of local minima:

- *number of local minima:* $|\mathrm{lmin}|$, *local minima density* $|\mathrm{lmin}|/|S|$
- *localization of local minima:* distribution of local minima within the neighborhood graph

Problem: Determining these measures typically requires exhaustive enumeration of search space.

$\Rightarrow$ Approximation based on sampling or estimation from other measures (such as autocorrelation measures, see below).

Example: Distribution of local minima for the TSP

Goal: Empirical analysis of distribution of local minima for Euclidean TSP instances.

Experimental approach:

- Sample sets of local optima of three TSPLIB instances using multiple independent runs of two TSP algorithms (3-opt, ILS).
- Measure pairwise distances between local minima (using *bond distance* = number of edges in which two given tours differ).
- Sample set of purportedly globally optimal tours using multiple independent runs of high-performance TSP algorithm.
- Measure minimal pairwise distances between local minima and respective closest optimal tour (using bond distance).

Empirical results:

Instance	avg sq [%]	avg d_{lmin}	avg d_{opt}
	Results for 3-opt		
rat783	3.45	197.8	185.9
pr1002	3.58	242.0	208.6
pcb1173	4.81	274.6	246.0
	Results for ILS algorithm		
rat783	0.92	142.2	123.1
pr1002	0.85	177.2	143.2
pcb1173	1.05	177.4	151.8

(based on local minima collected from 1 000/200 runs of 3-opt/ILS)
avg sq [%]: average solution quality expressed in percentage deviation from optimal solution

Interpretation:

- Average distance between local minima is small compared to maximal possible bond distance, n.

  $\Rightarrow$ *Local minima are concentrated in a relatively small region of the search space.*
- Average distance between local minima is slightly larger than distance to closest global optimum.

  $\Rightarrow$ *Optimal solutions are located centrally in region of high local minima density.*
- Higher-quality local minima found by ILS tend to be closer to each other and the closest global optima compared to those determined by 3-opt.

  $\Rightarrow$ *Higher-quality local minima tend to be concentrated in smaller regions of the search space.*

Note: These results are fairly typical for many types of TSP instances and instances of other combinatorial problems.
In many cases, local optima tend to be clustered; this is reflected in multi-modal distributions of pairwise distances between local minima.

Fitness-Distance Correlation (FDC)

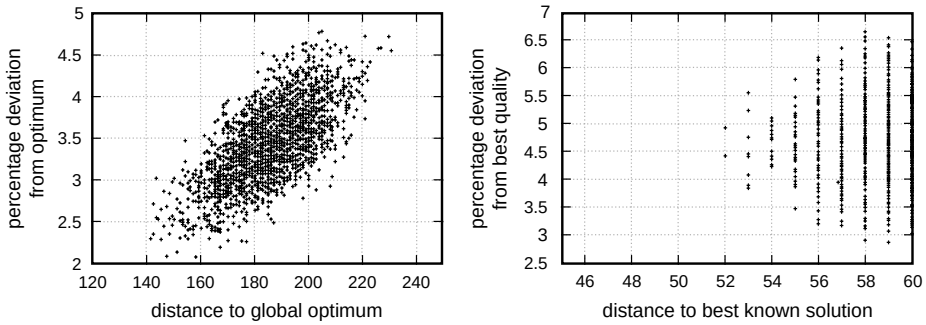
Idea: Analyze correlation between solution quality (fitness) g of candidate solutions and distance d to (closest) optimal solution.

Measure for FDC: *empirical correlation coefficient* r_{fdc}.

Fitness-distance plots, *i.e.*, scatter plots of the (g_i, d_i) pairs underlying an estimate of r_{fdc}, are often useful to graphically illustrate fitness distance correlations.

- The FDC coefficient, r_{fdc} depends on the given neighborhood relation.
- r_{fdc} is calculated based on a sample of m candidate solutions (typically: set of local optima found over multiple runs of an iterative improvement algorithm).

Example: FDC plot for TSPLIB instance rat783, based on 2500 local optima obtained from a 3-opt algorithm



High FDC (r_{fdc} close to one):

- 'Big valley' structure of landscape provides guidance for local search;
- search initialization: high-quality candidate solutions provide good starting points;
- search diversification: (weak) perturbation is better than restart;
- typical, *e.g.*, for TSP.

Low FDC (r_{fdc} close to zero):

- global structure of landscape does not provide guidance for local search;
- typical for very hard combinatorial problems, such as certain types of QAP (Quadratic Assignment Problem) instances.

Applications of fitness-distance analysis:

- algorithm design: use of strong intensification (including initialization) and relatively weak diversification mechanisms;
- comparison of effectiveness of neighborhood relations;
- analysis of problem and problem instance difficulty.

Limitations and short-comings:

- *a posteriori* method, requires set of (optimal) solutions, **but:** results often generalize to larger instance classes;
- optimal solutions are often not known, using best known solutions can lead to erroneous results;
- can give misleading results when used as the sole basis for assessing problem or instance difficulty.

Ruggedness

Idea: Rugged search landscapes, *i.e.*, landscapes with high variability in evaluation function value between neighboring search positions, are hard to search.

Example: Smooth *vs* rugged search landscape

Note: Landscape ruggedness is closely related to local minima density: rugged landscapes tend to have many local minima.

The ruggedness of a landscape L can be measured by means of the empirical autocorrelation function $r(i)$:

$$r(i) := \frac{1/(m-i) \cdot \sum_{k=1}^{m-i} (g_k - \bar{g}) \cdot (g_{k+i} - \bar{g})}{1/m \cdot \sum_{k=1}^{m} (g_k - \bar{g})^2}$$

where $g_1, \ldots g_m$ are evaluation function values sampled along an uninformed random walk in L.

Note: $r(i)$ depends on the given neighborhood relation.

- Empirical autocorrelation analysis is computationally cheap compared to, *e.g.*, fitness-distance analysis.
- (Bounds on) AC can be theoretically derived in many cases, *e.g.*, the TSP with the 2-exchange neighborhood.
- There are other measures of ruggedness, such as *empirical autocorrelation coefficient* and *(empirical) correlation length*.

High AC (close to one):

- "smooth" landscape;
- evaluation function values for neighboring candidate solutions are close on average;
- low local minima density;
- problem typically relatively easy for local search.

Low AC (close to zero):

- very rugged landscape;
- evaluation function values for neighboring candidate solutions are almost uncorrelated;
- high local minima density;
- problem typically relatively hard for local search.

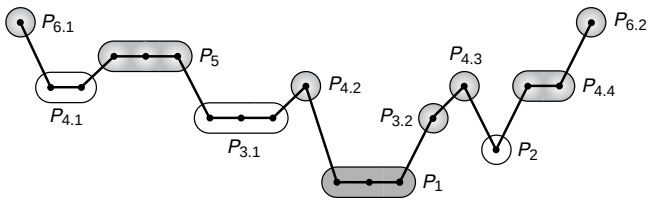
Note:

- Measures of ruggedness, such as AC, are often insufficient for distinguishing between the hardness of individual problem instances;
- but they can be useful for
  - analyzing differences between neighborhood relations for a given problem,
  - studying the impact of parameter settings of a given SLS algorithm on its behavior,
  - classifying the difficulty of combinatorial problems.

Plateaux

Plateaux, *i.e.*, 'flat' regions in the search landscape

Intuition: Plateaux can impede search progress due to lack of guidance by the evaluation function.



Definitions

- **Region:** connected set of search positions.
- **Border of region** R**:** set of search positions with at least one direct neighbor outside of R (**border positions**).
- **Plateau region:** region in which all positions have the same level, *i.e.*, evaluation function value, l.
- **Plateau:** maximally extended plateau region, *i.e.*, plateau region in which no border position has any direct neighbors at the plateau level l.
- **Solution plateau:** Plateau that consists entirely of solutions of the given problem instance.
- **Exit of plateau region** R**:** direct neighbor s of a border position of R with lower level than plateau level l.
- **Open / closed plateau:** plateau with / without exits.

Measures of plateau structure:

- *plateau diameter* = diameter of corresponding subgraph of $G_{\mathcal{N}}$
- *plateau width* = maximal distance of any plateau position to the respective closest border position
- *number of exits*, *exit density*
- *distribution of exits within a plateau*, *exit distance distribution* (in particular: avg./max. distance to closest exit)

Some plateau structure results for SAT:

- Plateaux typically don't have an interior, *i.e.*, almost every position is on the border.
- The diameter of plateaux, particularly at higher levels, is comparable to the diameter of search space. (In particular: plateaux tend to span large parts of the search space, but are quite well connected internally.)
- For open plateaux, exits tend to be clustered, but the average exit distance is typically relatively small.

Barriers and Basins

Observation:

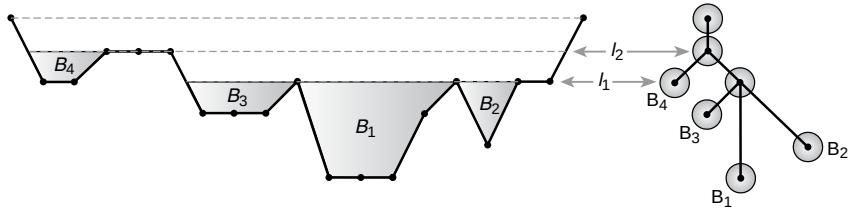
The *difficulty of escaping* from closed plateaux or strict local minima is related to the *height of the barrier*, *i.e.*, the difference in evaluation function, that needs to be overcome in order to reach better search positions:

Higher barriers are typically more difficult to overcome (this holds, *e.g.*, for Probabilistic Iterative Improvement or Simulated Annealing).

Definitions:

- Positions s, s' are *mutually accessible at level* l iff there is a path connecting s' and s in the neighborhood graph that visits only positions t with $g(t) \leq l$.
- The *barrier level between positions* s, s', $bl(s, s')$ is the lowest level l at which s' and s' are mutually accessible;
  the difference between the level of s and $bl(s, s')$ is called the *barrier height between* s *and* s'.
- **Basins**, *i.e.*, maximal (connected) regions of search positions below a given level, form an important basis for characterizing search space structure.

Example: Basins in a simple search landscape and corresponding basin tree



Note: The basin tree only represents basins just below the critical levels at which neighboring basins are joined (by a *saddle*).