LOCAL SEARCH METHODS
APPLICATIONS AND ENGINEERING

Lecture 2

Basic LS Methods: Construction Heuristics and Perturbative Searches

Marco Chiarandini

Outline

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.*, $M(\pi) := M_1 \times M_2 \times \ldots \times M_{l(\pi)}$.
- LS algorithms realize *Markov processes*: behavior in any **search state** (s, m) depends only on current position s and (limited) memory m.

Example: Uninformed random walk for k-col

- **search space** S**:** set of all k-colorings of G
- **solution set** S'**:** set of all proper k-colorings of G
- **neighborhood relation** N**:** *1-exchange neighborhood*, *i.e.*, colorings are neighbors under N iff they differ in the color at one vertex
- **memory:** not used, *i.e.*, $M := \{0\}$
- **initialization:** uniform random choice from Γ, *i.e.*, $init(\varphi', m) := 1/|S|$ for all coloring φ' and memory states m
- **step function:** uniform random choice from current neighborhood, *i.e.*, $step(\varphi, m)(\varphi', m) := 1/|N(\varphi)|$ for all assignments φ and memory states m, where $N(a) := \{\varphi' \in S \mid N(\varphi, \varphi')\}$ is the set of all neighbors of φ.
- **termination:** when $terminate(\varphi, m) := 1$ if φ is a feasible coloring of G, and 0 otherwise.

Definition: LS Algorithm Components (continued)

Neighborhood relation (structure): $N : S \times S \rightarrow \{T, F\}$ or $N \subseteq S \times S$

- **neighborhood (set)** of candidate solution s: $N(s) := \{s' \in S \mid N(s, s')\}$
- **neighborhood 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:

k-col instance with n vertices, 1-exchange neighborhood:
$G_N = n$-dimensional hypercube; diameter of $G_N = n$.

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 k-col
(solution components = single vertex assignments)
- 2-exchange neighborhood for TSP
(solution components = edges in given graph)

Definition: LS Algorithm Components (continued)

Step function

- **Search step** (or **move**): pair of search positions 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 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.*, $init(s_0, m) > 0$ 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
  - **Evaluation function**
  - ...

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 neighborhood, *i.e.*, for all $s, s' \in S$, $step(s)(s') := 1/|N(s)|$ if $N(s, s')$, and 0 otherwise

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

Definition: LS Algorithm Components (continued)

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 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 (II)

- does not use memory
- $init$: uniform random choice from S
- $step$: uniform random choice from improving neighbors, *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 neighbor available (to be revisited later)
- different variants through modifications of step function (to be revisited later)

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

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.*, $init()(\varphi') := 1/|S|$ for all colorings φ'
- **evaluation function:** $g(\varphi) :=$ number of edges in G whose ending vertices are assigned the same color under assignment φ (*Note:* $g(\varphi) = 0$ iff φ is a proper coloring of G.)
- **step function**: uniform random choice from improving neighbors, *i.e.*, $step(\varphi)(\varphi') := 1/|I(\varphi)|$ if $s' \in I(\varphi)$, and 0 otherwise, where $I(\varphi) := \{\varphi' \mid N(\varphi, \varphi') \wedge g(\varphi') < g(\varphi)\}$
- **termination**: when no improving neighbor is available *i.e.*, $terminate(\varphi)(\top) := 1$ if $I(a) = \emptyset$, and 0 otherwise.

Incremental updates (aka delta evaluations)

- **Key idea:** calculate *effects of differences* between current search position s and neighbors 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 and s' in terms of solution components.
- Typically crucial for the efficient implementation of II algorithms (and other LS techniques).

Example: Incremental updates for TSP

- solution components = edges of given graph G
- standard 2-exchange neighborhood, *i.e.*, neighboring 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 arithmetic operations for graph with n vertices) for computing $w(p')$ from scratch.

Definition:

- **Local minimum:** search position without improving neighbors w.r.t. given evaluation function g and neighborhood 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.

Note:

- Local minima depend on g and neighborhood relation N.
- Larger neighborhoods $N(s)$ induce
  - neighborhood graphs with smaller diameter;
  - fewer local minima.

Ideal case: **exact neighborhood**, *i.e.*, neighborhood relation for which any local optimum is also guaranteed to be a global optimum.

- Typically, exact neighborhoods are too large to be searched effectively (exponential in size of problem instance).
- *But:* exceptions exist, *e.g.*, polynomially searchable neighborhood in Simplex Algorithm for linear programming.

Trade-off:

- Using larger neighborhoods can improve performance of II (and other LS methods).
- *But:* time required for determining improving search steps increases with neighborhood size.

More general trade-off:

Effectiveness *vs* Efficiency (= time complexity of search steps).

In II, various mechanisms (**pivoting rules**) can be used for choosing improving neighbor in each step:

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

  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: Random-order first improvement for the TSP

- **Given:** TSP instance G with vertices $v_1, v_2, \dots, v_n$.
- search space: Hamiltonian cycles in G; use standard 2-exchange neighborhood
- **Initialization:**

  search position := fixed canonical path $(v_1, v_2, \dots, v_n, v_1)$
  P := random permutation of {1,2, . . . , n}
- **Search steps:** determined using first improvement w.r.t. $g(p)$ = weight of path p, evaluating neighbors in order of P (does not change throughout search)
- **Termination:** when no improving search step possible (local minimum)

Speed-up Techniques: Neighborhood Pruning

- *Idea:* Reduce size of neighborhoods by excluding neighbors that are likely (or guaranteed) not to yield improvements in g.
- *Note:* Crucial for large neighborhoods, but can be also very useful for small neighborhoods (*e.g.*, linear in instance size).

Example: Heuristic candidate lists for the TSP

- *Intuition:* High-quality solutions likely include short edges.
- **Candidate list** of vertex v: list of v's nearest neighbors (limited number), sorted according to increasing edge weights.
- Search steps (*e.g.*, 2-exchange moves) always involve edges to elements of candidate lists.
- Significant impact on performance of LS algorithms for the TSP.

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

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 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.

Simple Mechanisms for Escaping from Local Optima

- *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: Neither 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.

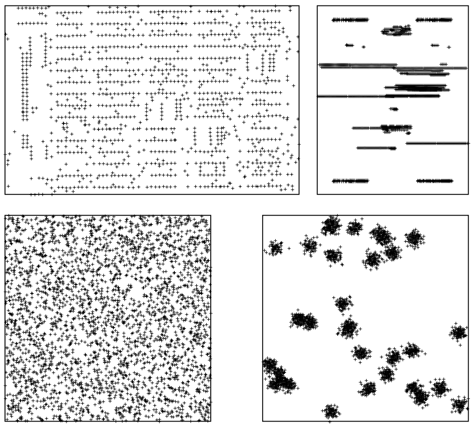
TSP: Benchmark Instances

Instance classes

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

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

TSP: Benchmark Instances, Examples



Complete Algorithms and Lower Bounds

- Branch & cut algorithms
  - use LP-relaxation for lower bounding schemes
  - effective heuristics for upper bounds
  - branch if cuts cannot be found easily

Solution times with Concorde

Instance	No. nodes	CPU time (secs)
att532	7	109.52
rat783	1	37.88
pcb1173	19	468.27
fl1577	7	6705.04
d2105	169	11179253.91
pr2392	1	116.86
rl5934	205	588936.85
usa13509	9539	ca. 4 years
d15112	164569	ca. 22 years

- Lower bounds: (within less than one percent of optimum for random Euclidean, up to two percent for TSPLIB instances)

Construction Heuristics

In general *construction heuristics* are closely related to search tree techniques

- a single path from root to leaf
- possible extension: beam search/pilot method
  - maintains a set B of bw (beam width) partial candidate solutions
  - at each level extend fw (filter width) candidate solutions and rank
  - complete candidate solutions obtained by B are maintained in B_f

Construction heuristics specific for TSP

- nearest neighborhood heuristics
- insertion heuristics
- greedy heuristics
- savings heuristics
- Christofides heuristics

Software Development: Extreme Programming

Planning

Release planning creates the schedule // Make frequent small releases // The project is divided into iterations // A stand-up meeting starts each day

Designing

Simplicity Choose a system metaphor // No functionality is added early // Refactor: eliminate unused functionality and redundancy

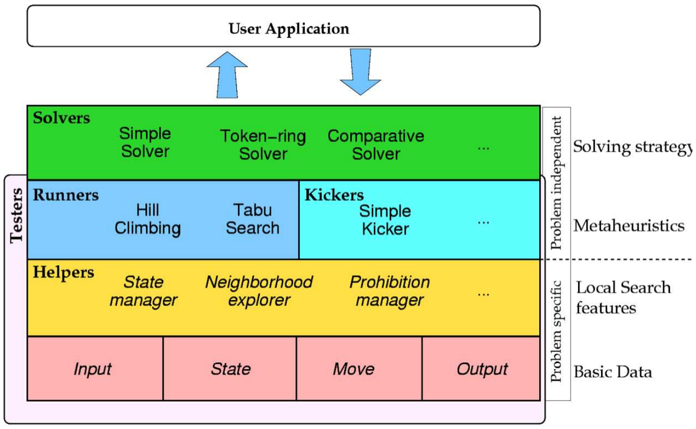
Coding

The customer is always available // Code must be written to agreed standards // Code the unit test first // All production code is pair programmed // Only one pair integrates code at a time // Use collective code ownership // Leave optimization till last // No overtime

Testing

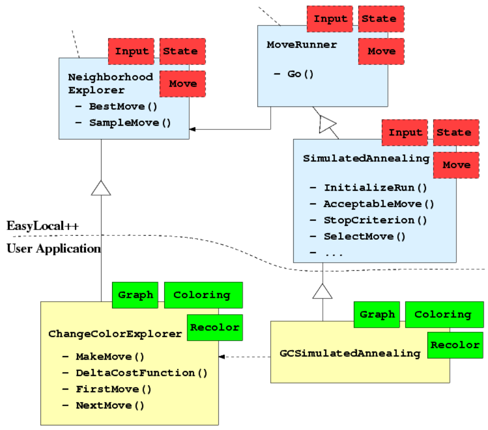
All code must have unit tests // All code must pass all unit tests before it can be released // When a bug is found tests are created

Software Framework for LS Methods



From EasyLocal++ by Schaerf and Di Gaspero (2003).

Software Framework for LS Methods



From EasyLocal++ by Schaerf and Di Gaspero (2003).