DM811

Heuristics for Combinatorial Optimization

Lecture 8
Local Search (cntd.)

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Summary: Local Search Algorithms
(as in [Hoos, Stützle, 2005])

For given problem instance π:

1. search space S_π
2. neighborhood relation $\mathcal{N}_\pi \subseteq S_\pi \times S_\pi$
3. evaluation function $f_\pi : S \to \mathbf{R}$
4. set of memory states M_π
5. initialization function $\texttt{init} : \emptyset \to S_\pi \times M_\pi)$
6. step function $\texttt{step} : S_\pi \times M_\pi \to S_\pi \times M_\pi$
7. termination predicate $\texttt{terminate} : S_\pi \times M_\pi \to \{\top, \bot\}$

Outline

Outline

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)

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: $N(s) := \{s' \in S \mid \mathcal{N}(s, s')\}$

Notation: N when set, $\mathcal{N}$ when collection of sets or function

LS Algorithm Components

Neighborhood function

Also defined as: $\mathcal{N} : S \times S \rightarrow \{T, F\}$ 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 $|N(s)|$

Notation: N when set, $\mathcal{N}$ when collection of sets or function

LS Algorithm Components

Neighborhood function

Also defined as: $\mathcal{N} : S \times S \rightarrow \{T, F\}$ 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: $s' \in N(s) \rightarrow s \in N(s')$

Notation: N when set, $\mathcal{N}$ when collection of sets or function

LS Algorithm Components

Neighborhood function

Also defined as: $\mathcal{N} : S \times S \rightarrow \{T, F\}$ 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 $\rightsquigarrow$ undirected graph)

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

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

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)

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

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

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)

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 $s' \in N(s)$.

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 $s' \in N(s)$.

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.

LS Algorithm Components

Note:

- Local search implements a walk through the neighborhood graph

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.

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.*, $M_\pi := M_1 \times M_2 \times \ldots \times M_{l_\pi}$.

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 often Markov processes:
  behavior in any search state $\{s, m\}$ depends only
  on current position s
  higher order if (limited) memory m.

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 $< s_0, s_1, \dots, s_k >$ such that (s_{i-1}, s_i) is a *search step* for any $i \in \{1, \dots, k\}$ and the probability of initializing the search at s_0 is greater than zero, *i.e.*, $\texttt{init}(\{s_0, m\}) > 0$ for some memory state $m \in M$.

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
$\mathtt{step}(\{s, m\}, \{s', m'\}) > 0$ 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 than zero, *i.e.*, $\mathtt{init}(\{s_0, m\}) > 0$ 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

LS Algorithm Components

Evaluation (or cost) function:

- function $f_\pi : S_\pi \rightarrow \mathbf{R}$ 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.

LS Algorithm Components

Evaluation (or cost) function:

- function $f_\pi : S_\pi \rightarrow \mathbf{R}$ 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.

LS Algorithm Components

Evaluation (or cost) function:

- function $f_\pi : S_\pi \to \mathbf{R}$ 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.*, guided local search).

Outline

Iterative Improvement

Resume

- does not use memory
- `init`: uniform random choice from S or construction heuristic
- `step`: uniform random choice from improving neighbors

$$\Pr(s, s') = \begin{cases} 1/|I(s)| \text{ if } s' \in I(s) \\ 0 \text{ otherwise} \end{cases}$$

where $I(s) := \{s' \in S \mid \mathcal{N}(s, s') \text{ and } f(s') < f(s)\}$

Iterative Improvement

Resume

- does not use memory
- `init`: uniform random choice from S or construction heuristic
- `step`: uniform random choice from improving neighbors

$$\Pr(s, s') = \begin{cases} 1/|I(s)| \text{ if } s' \in I(s) \\ 0 \text{ otherwise} \end{cases}$$

where $I(s) := \{s' \in S \mid \mathcal{N}(s, s') \text{ and } f(s') < f(s)\}$

- terminates when no improving neighbor available

Iterative Improvement

Resume

- does not use memory
- `init`: uniform random choice from S or construction heuristic
- `step`: uniform random choice from improving neighbors

$$\Pr(s, s') = \begin{cases} 1/|I(s)| \text{ if } s' \in I(s) \\ 0 \text{ otherwise} \end{cases}$$

where $I(s) := \{s' \in S \mid \mathcal{N}(s, s') \text{ and } f(s') < f(s)\}$

- terminates when no improving neighbor available

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

Iterative Improvement (cntd)

Resume

Pivoting rule decides which neighbors go in $I(s)$

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

Iterative Improvement (cntd)

Resume

Pivoting rule decides which neighbors go in $I(s)$

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

  Note: Requires evaluation of all neighbors in each step!

Iterative Improvement (cntd)

Resume

Pivoting rule decides which neighbors go in $I(s)$

- Best Improvement (aka *gradient descent*, *steepest descent*, *greedy hill-climbing*): Choose maximally improving neighbors,
  i.e., $I(s) := \{s' \in N(s) \mid f(s') = g^*\}$,
  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 one encountered.

Iterative Improvement (cntd)

Resume

Pivoting rule decides which neighbors go in $I(s)$

- Best Improvement (aka *gradient descent*, *steepest descent*, *greedy hill-climbing*): Choose maximally improving neighbors,
  i.e., $I(s) := \{s' \in N(s) \mid f(s') = g^*\}$,
  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 one encountered.

  Note: Can be more efficient than Best Improvement but not in the worst case; order of evaluation can impact performance.

Examples

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)

Examples

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

Examples

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
- **memory:** not used, *i.e.*, $M := \{0\}$

Examples

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
- **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

Examples

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
- **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.)

Examples

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
- **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 $a' \in I(a)$, and 0 otherwise, where $I(a) := \{a' \mid \mathcal{N}(a, a') \wedge f(a') < f(a)\}$

Examples

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
- **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 $a' \in I(a)$, 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.

Examples

Random order first improvement for SAT

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

Examples

Random order first improvement for SAT

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

choose assignment φ of truth values to all variables in F
uniformly at random;
steps := 0;

Examples

Random order first improvement for SAT

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

choose assignment φ of truth values to all variables in F
uniformly at random;
steps := 0;
while $\neg$(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*$+1$;

Examples

Random order first improvement for SAT

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

choose assignment φ of truth values to all variables in F
    uniformly at random;
steps := 0;
while $\neg(\varphi$ 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 $\emptyset$

Examples

Iterative Improvement for TSP

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**
      $\texttt{UpdateTour}(s, i, j)$

is it really?

Examples

Iterative Improvement for TSP

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

```
Δ = 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
            Δ_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 Δ_ij < 0 then
                UpdateTour(s, i, j)
                Improvement=TRUE
```

Examples

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;
- **neighborhood relation** N**:** standard 2-exchange neighborhood

Examples

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;
- **neighborhood relation N:** standard 2-exchange neighborhood

- **Initialization:**

  search position := fixed canonical tour $< v_1, v_2, \ldots, v_n, v_1 >$
  P := random permutation of $\{1, 2, \ldots, n\}$

Examples

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;
- **neighborhood relation** N**:** standard 2-exchange neighborhood
- **Initialization:**

  search position := fixed canonical tour $< v_1, v_2, \ldots, v_n, v_1 >$
  P := random permutation of $\{1, 2, \ldots, n\}$

- **Search steps:** determined using first improvement w.r.t. $f(s)$ = cost of tour s, evaluating neighbors in order of P (does not change throughout search)

Examples

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;
- **neighborhood relation** N**:** standard 2-exchange neighborhood
- **Initialization:**

  search position := fixed canonical tour $< v_1, v_2, \ldots, v_n, v_1 >$
  P := random permutation of $\{1, 2, \ldots, n\}$

- **Search steps:** determined using first improvement w.r.t. $f(s)$ = cost of tour s, evaluating neighbors in order of P (does not change throughout search)
- **Termination:** when no improving search step possible (local minimum)

The Max Independent Set Problem

Also called "stable set problem" or "vertex packing problem".
Given: an undirected graph $G(V, E)$ and a non-negative weight function ω on V ($\omega : V \to \mathbf{R}$)

Task: A largest weight independent set of vertices, i.e., a subset $V' \subseteq V$ such that no two vertices in V' are joined by an edge in E.

The Max Independent Set Problem

Also called "stable set problem" or "vertex packing problem".
Given: an undirected graph $G(V, E)$ and a non-negative weight function ω on V ($\omega : V \to \mathbf{R}$)

Task: A largest weight independent set of vertices, i.e., a subset $V' \subseteq V$ such that no two vertices in V' are joined by an edge in E.

Related Problems:

Vertex Cover

Given: an undirected graph $G(V, E)$ and a non-negative weight function ω on V ($\omega : V \to \mathbf{R}$)
Task: A smallest weight vertex cover, i.e., a subset $V' \subseteq V$ such that each edge of G has at least one endpoint in V'.

Maximum Clique

Given: an undirected graph $G(V, E)$
Task: A maximum cardinality clique, i.e., a subset $V' \subseteq V$ such that every two vertices in V' are joined by an edge in E

Single Machine Total Weighted Tardiness

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

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

Example:

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

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

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

Graph Partitioning

Input: A graph $G = (V, E)$, weights $w(v) \in Z^+$ for each $v \in V$ and $l(e) \in Z^+$ for each $e \in E$.
Task: Find a partition of V into disjoint sets $V_1, V_2, \ldots, V_m$ such that $\sum_{v \in V_i} w(v) \leq K$ for $1 \leq i \leq m$ and such that if $E' \subseteq E$ is the set of edges that have their two endpoints in two different sets V_i, then $\sum_{e \in E'} l(e)$ is minimal.

Consider the specific case of graph bipartitioning, that is, the case $|V| = 2n$ and $K = n$ and $w(v) = 1, \forall v \in V$.

Outline

Example: Scheduling in Parallel Machines

Total Weighted Completion Time on Unrelated Parallel Machines Problem

Input: A set of jobs J to be processed on a set of parallel machines M. Each job $j \in J$ has a weight w_j and processing time p_{ij} that depends on the machine $i \in M$ on which it is processed.

Task: Find a schedule of the jobs on the machines such that the sum of weighted completion time of the jobs is minimal.

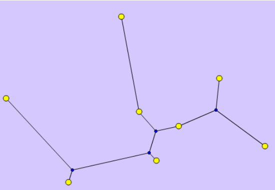
Example: Steiner Tree

Steiner Tree Problem

Input: A graph $G = (V, E)$, a weight function $\omega : E \mapsto \mathbf{N}$, and a subset $U \subseteq V$.

Task: Find a Steiner tree, that is, a subtree $T = (V_T, E_T)$ of G that includes all the vertices of U and such that the sum of the weights of the edges in the subtree is minimal.

Vertices in U are the special vertices and
vertices in $S = V \setminus U$ are Steiner vertices.



Examples, Resume

- Permutations
  - TSP
  - SMWTP
- Assignments
  - SAT
  - Coloring
  - Parallel machines
- Sets
  - Max Weighted Independent Set
  - Steiner tree