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 $f_\pi : S \rightarrow \mathbf{R}$
4. set of memory states M_π
5. initialization function $\mathtt{init} : \emptyset \rightarrow S_\pi \times M_\pi)$
6. step function $\mathtt{step} : S_\pi \times M_\pi \rightarrow S_\pi \times M_\pi$
7. termination predicate $\mathtt{terminate} : S_\pi \times M_\pi \rightarrow \{\top, \bot\}$

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: graph partitioning, max indep. set)

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: $s' \in N(s) \to s \in N(s')$
- neighborhood graph of (S, N, π) is a directed 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)
- 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 $s' \in N(s)$.
- 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

Evaluation (or cost) function:

- function $f_\pi : S_\pi \to \mathbf{Q}$ that maps candidate solutions of a given problem instance π onto rational numbers (most often integer), such that global optima correspond to solutions of π;
- used for assessing or ranking 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).

Constraint-based local search

From [B4]

What is a violation?
Constraint specific:

- decomposition-based violations
  number of violated constraints, eg: alldiff
- variable-based violations
  min number of variables that must be changed to satisfy c.
- value-based violations
  for constraints on number of occurences of values
- arithmetic violations
- combinations of these

Constraint-based local search

From [B4]

Arithmetic constraints

- $l \leq r \rightsquigarrow \mathrm{viol} = \max(l - r, 0)$
- $l = r \rightsquigarrow \mathrm{viol} = |l - r|$
- $l \neq r \rightsquigarrow \mathrm{viol} = 1$ if $l = r$, 0 otherwise

Combinatorial constraints

- $\texttt{alldiff}(x_1, \ldots, x_n)$:
  Let a be an assignment with values $V = \{a(x_1), \ldots, (x_n)\}$ and $c_v = \#_a(v, x)$ be the number of variables with the same value.
  Possible definitions for violations are:
  - $\mathrm{viol} = \sum_{v \in V} I(\max(c_v - 1, 0) > 0)$ value-based
  - $\mathrm{viol} = \max_{v \in V} \max(c_v - 1, 0)$ value-based
  - $\mathrm{viol} = \sum_{v \in V} \max(c_v - 1, 0)$ value-based
  - $\#$ variables with same value, variable-based, here leads to same definitions as previous three

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.
- Local search algorithms can be described as Markov processes:
  behavior in any search state $\{s, m\}$ depends only
  on current position s
  higher order MP 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 $< s_0, s_1, \ldots, s_k >$ 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.*, $\texttt{init}(\{s_0, m\}) > 0$ 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

Outline

1. Local Search Revisited
    Components

2. Examples

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 $g^* := \min\{f(s') \mid s' \in N(s)\}$.

  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)
- **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** $\emptyset$

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*$+1$;
if φ satisfies F **then**
    return φ
else
    return $\emptyset$

In Comet
Iterative Improvement

queensLS00.co

```
import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  select(q in Size, v in Size : S.getAssignDelta(queen[q],v) < 0) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"]:="<<v<<" viol: "<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

In Comet
Best Improvement

queensLS0.co

```
import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  selectMin(q in Size,v in Size)(S.getAssignDelta(queen[q],v)) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"] := "<<v<<" viol: "<<S.violations() <<
          endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

In Comet

First Improvement

queensLS2.co

```
import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  selectFirst(q in Size, v in Size: S.getAssignDelta(queen[q],v) < 0) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"] := "<<v<<" viol: "<<S.violations() <<
          endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

In Comet
Min Conflict Heuristic

queensLS0b.co

```
import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  select(q in Size : S.violations(queen[q])>0) {
    selectMin(v in Size)(S.getAssignDelta(queen[q],v)) {
      queen[q] := v;
      cout<<"chng @ "<<it<<": queen["<<q<<"] := "<<v<<" viol: "<<S.violations() <<
            endl;
    }
    it = it + 1;
  }
}
cout << queen << endl;
```

In Comet
General procedure

queensLS-generic.co

```
function void conflictSearch (Constraint<LS> c, int itLimit) {
    int it = 0;
    var{int}[] x = c.getVariables();
    range Size = x.getRange();
    while (!c.isTrue() && it < itLimit) {
        selectMax(i in Size)(c.violations(x[i]))
            selectMin(v in x[i].getDomain())(c.getAssignDelta(x[i],v))
                x[i] := v;
        it = it + 1;
    }
}

import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

conflictSearch(S,50*n);
cout << queen << endl;
```

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)

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;
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)
```

is it really?