DM811

Heuristics for Combinatorial Optimization

Lecture 7

Local Search

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

Local Search Algorithms

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

- search space S_π
  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)

Local Search Algorithms

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

- search space S_π
  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 $f_\pi : S_\pi \to \mathbf{R}$
  (*e.g.*, for SAT: number of false clauses)

Local Search Algorithms

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

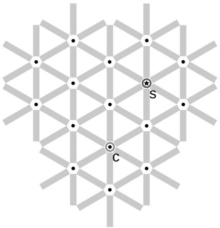
- search space S_π
  specified by candidate solution representation:
  discrete structures: sequences, permutations, graphs, partitions
  (*e.g.*, for SAT: array, sequence of all truth assignments
  to propositional variables)

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

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

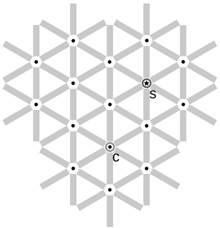
- neighborhood function, $\mathcal{N}_\pi : S \to 2^{S_\pi}$
  (*e.g.*, for SAT: neighboring variable assignments differ
  in the truth value of exactly one variable)

Local search — global view



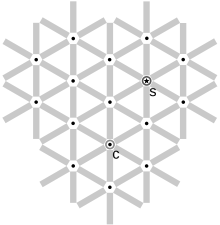
- vertices: candidate solutions (search positions)
- vertex labels: evaluation function
- edges: connect "neighboring" positions
- s: (optimal) solution
- c: current search position

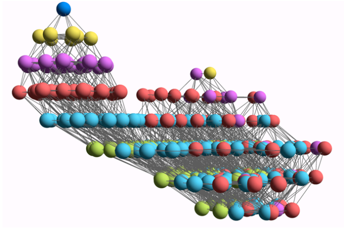
Local search — global view



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

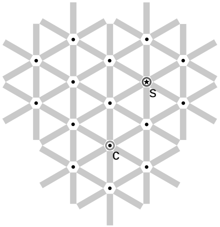
Local search — global view

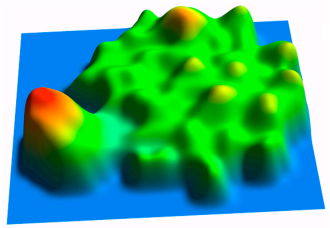




- vertices: candidate solutions (search positions)
- vertex labels: evaluation function
- edges: connect "neighboring" positions
- s: (optimal) solution
- c: current search position

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

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 $\hat{s}$ w.r.t. N if $f(\hat{s}) \leq f(s)\ \forall s \in N(\hat{s})$

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 $\hat{s}$ w.r.t. 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

Local Search Algorithm

Further components [according to B5]

- set of memory states M_π
  (may consist of a single state, for LS algorithms that do not use memory)

Local Search Algorithm

Further components [according to B5]

- set of memory states M_π
  (may consist of a single state, for LS algorithms that do not use memory)

- initialization function $\texttt{init} : \emptyset \to S_\pi$
  (can be seen as a probability distribution $\Pr(S_\pi \times M_\pi)$ over initial search positions and memory states)

Local Search Algorithm

Further components [according to B5]

- set of memory states M_π
  (may consist of a single state, for LS algorithms that do not use memory)
- initialization function $\texttt{init} : \emptyset \to S_\pi$
  (can be seen as a probability distribution $\Pr(S_\pi \times M_\pi)$ over initial search positions and memory states)
- step function $\texttt{step} : S_\pi \times M_\pi \to S_\pi \times M_\pi$
  (can be seen as a probability distribution $\Pr(S_\pi \times M_\pi)$ over subsequent, neighboring search positions and memory states)

Local Search Algorithm

Further components [according to B5]

- set of memory states M_π
  (may consist of a single state, for LS algorithms that do not use memory)
- initialization function $\texttt{init} : \emptyset \to S_\pi$
  (can be seen as a probability distribution $\Pr(S_\pi \times M_\pi)$ over initial search positions and memory states)
- step function $\texttt{step} : S_\pi \times M_\pi \to S_\pi \times M_\pi$
  (can be seen as a probability distribution $\Pr(S_\pi \times M_\pi)$ over subsequent, neighboring search positions and memory states)
- termination predicate $\texttt{terminate} : S_\pi \times M_\pi \to \{\top, \bot\}$
  (determines the termination state for each search position and memory state)

Decision vs Minimization

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

Decision vs Minimization

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

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

Decision vs Minimization

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

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

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

Decision vs Minimization

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 $\emptyset$

Decision vs Minimization

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

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

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

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** $\emptyset$

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

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

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)

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

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

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.

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.

In Comet
Random Walk

queensLS0a.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) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"]:="<<v<<" viol: "<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

In Comet
Another Random Walk

queensLS1.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, v in Size) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"]:="<<v<<" viol: "<<S.violations()<<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

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
Use of functions

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;
```

Constraint-based local search

From [B4]

What is a violation?
Constraint specific:

- variable-based violations
- value-based violations
- decomposition-based violations
- 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
  - here variable-based, eg: # variables with same value, lead to same definitions as previous three

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 $\texttt{init} : \emptyset \rightarrow S_\pi \times M_\pi)$
6. step function $\texttt{step} : S_\pi \times M_\pi \rightarrow S_\pi \times M_\pi$
7. termination predicate $\texttt{terminate} : S_\pi \times M_\pi \rightarrow \{\top, \bot\}$