DM811

Heuristics for Combinatorial Optimization

Lecture 6

Local Search

Marco Chiarandini

Department of Mathematics & Computer Science

University of Southern Denmark

Outline

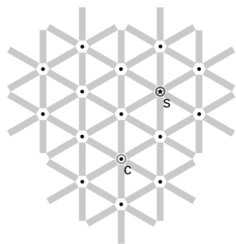
Local Search Algorithms

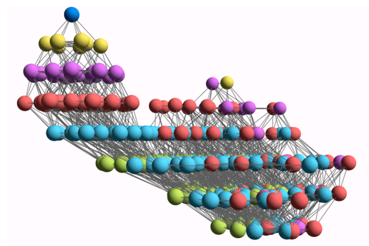
Given a (combinatorial) optimization problem Π and one of its instances π:

- 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 $\hat{s}$ 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

- set of memory states $M(\pi)$
  (may consist of a single state, for LS algorithms that do not use memory)
- initialization function $\texttt{init} : \emptyset \mapsto S(\pi)$
  (can be seen as a probability distribution $\Pr(S(\pi) \times M(\pi))$ over initial search positions and memory states)
- step function $\texttt{step} : S(\pi) \times M(\pi) \mapsto S(\pi) \times M(\pi)$
  (can be seen as a probability distribution $\Pr(S(\pi) \times M(\pi))$ over subsequent, neighboring search positions and memory states)
- termination predicate $\texttt{terminate} : S(\pi) \times M(\pi) \mapsto \{\top, \bot\}$
  (determines the termination state 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** $\emptyset$
$(s, m) := \texttt{init}(\pi')$;
$s_b := s$;
while not $\texttt{terminate}(\pi', s, m)$ **do**
  $(s, m) := \texttt{step}(\pi', s, m)$;
  if $f(\pi', s) < f(\pi', \hat{s})$ **then**
    $s_b := s$;
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)
- **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.

In Comet

Random Walk


```
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<<"change:␣queen["<<q<<"]␣:=␣"<<v<<"␣viol:␣"<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```


In Comet

Another Random Walk


```
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<<"change:␣queen["<<q<<"]␣:=␣"<<v<<"␣viol:␣"<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```


In Comet

Iterative Improvement


```
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<<"change:␣queen["<<q<<"]␣:=␣"<<v<<"␣viol:␣"<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

In Comet
Hill Climbing, Best Improvement (2/2)


```
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<<"change:␣queen["<<q<<"]␣:=␣"<<v<<"␣viol:␣"<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```


In Comet
First Improvement


```
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<<"change:␣queen["<<q<<"]␣:=␣"<<v<<"␣viol:␣"<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```


In Comet
Min Conflict Heuristic


```
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<<"change:␣queen["<<q<<"]␣:=␣"<<v<<"␣viol:␣"<<S.violations() <<
           endl;
    }
    it = it + 1;
  }
}
```


In Comet
Use of functions


```
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]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
   it = it + 1;
}
cout << queen << endl;
```

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

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 $\texttt{init} : \emptyset \mapsto S(\pi) \times M(\pi))$
6. step function $\texttt{step} : S(\pi) \times M(\pi) \mapsto S(\pi) \times M(\pi)$
7. termination predicate $\texttt{terminate} : S(\pi) \times M(\pi) \mapsto \{\top, \bot\}$