DM811

Heuristics for Combinatorial Optimization

Lecture 7
Local Search

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Course Overview

- ✔ Combinatorial Optimization, Methods and Models
- ✔ CH and LS: overview
- ✔ Working Environment and Solver Systems
- ~ Methods for the Analysis of Experimental Results
- ✔ Construction Heuristics
- Local Search: Components, Basic Algorithms
- Local Search: Neighborhoods and Search Landscape
- Efficient Local Search: Incremental Updates and Neighborhood Pruning
- Stochastic Local Search & Metaheuristics
- Configuration Tools: F-race
- Very Large Scale Neighborhoods

Examples: GCP, CSP, TSP, SAT, MaxIndSet, SMTWP, Steiner Tree, p-median, set covering

DM811

Heuristics for Combinatorial Optimization

Lecture 7
Local Search

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Construction Heuristics

- sequential heuristics
  1. choose a variable (vertex)
     a) static order: random (ROS), largest degree first, smallest degree last
     b) dynamic order: saturation degree (DSATUR) [Brélaz, 1979]
  2. choose a value (color): greedy heuristic

```
Procedure ROS
RandomPermutation π(Vertices);
forall i in 1, ..., n do
    v := π(i);
    select min{c : c not in saturated[v]};
    col[v] := c;
    add c in saturated[w] for all w adjacent v;
```

$\mathcal{O}(nk + m) \rightsquigarrow \mathcal{O}(n^2)$

```
Procedure DSATUR
select vertex v uncolored with max degree;
while uncolored vertices do
    select min{c : c not in saturated[v]};
    col[v] := c;
    add c in saturated[w] for all w adjacent v;
    select uncolored v with max size of
        saturated[v];
```

$\mathcal{O}(n(n + k) + m) \rightsquigarrow \mathcal{O}(n^2)$

- partitioning heuristics
  - recursive largest first (RLF) [Leighton, 1979]
    iteratively extract stable sets

Alternative form of pseudo-code

```
Procedure ROS
RandomPermutation π(Vertices);
forall i in 1, ..., n do
    v := π(i);
    selectMin {c : c not in saturated[v]} do
        col[v] := c;
        forall w in Vertices: adj[v,w] do
            saturated[w].insert(c);
```

```
Procedure DSATUR
RandomPermutation π(Vertices);
forall i in 1, ..., n do
    v := π(i);
    selectMin {c : c not in saturated[v]} do
        col[v] := c;
        forall w in Vertices: adj[v,w] do
            saturated[w].insert(c);
```

RLF [Leighton, 1979]

Procedure Recursive Largest First(G)
In $G = (V, E)$: input graph;
Out k : upper bound on $\chi(G)$;
Out c : a coloring $c : V \mapsto K$ of G;

$k \leftarrow 0$ **while** $|V| > 0$ **do**
    $k \leftarrow k + 1$ /* Use an additional color */
    FindStableSet(V, E, k) /* $G = (V, E)$ is reduced */
return k

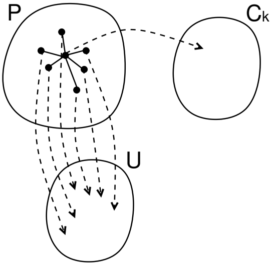
RLF

Key idea: extract stable sets trying to maximize edges removed.

Procedure FindStableSet(G, k)
In $G = (V, E)$: input graph
In k : color for current stable set
Var P : set of potential vertices for stable set
Var U : set of vertices that cannot go in current stable set

$P \leftarrow V$; $U \leftarrow \emptyset$;
forall $v \in P$ **do** $d_U(v) \leftarrow 0$; /* degree induced by U */
while P not empty **do**
    select v in P with max d_U;
    move v from P to C_k; $V \leftarrow V \setminus \{v\}$
    forall $w \in \delta_P(v)$ **do** /* neighbors of v in P */
        move w from P to U; $E \leftarrow E \setminus \{v, w\}$
        forall $u \in \delta_P(w)$ **do**
            $d_U(u) \leftarrow d_U(u) + 1$

$$\mathcal{O}(m + n\Delta^2) \rightsquigarrow \mathcal{O}(n^3)$$



Examples


```
import cotls;
include "loadDIMACS";
// int nv;
// int me;
// float alpha;
// bool adj[nv,nv];
range Vertices = 1..nv;
range Colors = 1..nv;
int nbc = Colors.getUp();

Solver<LS> m();

var{int} col[Vertices](m,Colors) := 1;
ConstraintSystem<LS> S(m);

forall (i in Vertices, j in Vertices: j>i && adj[i,j])
S.post(col[i] != col[j]);
S.close();

m.close();

// CONSTRUCTION HEURISTIC
set{int} dom[v in Vertices] = setof(c in Colors) true;
RandomPermutation perm(Vertices);
forall (i in 1..nv) {
  int v = perm.get();
  selectMin(c in dom[v])(c) {
    col[v] := c;
    forall(w in Vertices: adj[v,w])
      dom[w].delete(c);
  }
}
nbc = max(v in Vertices) col[v];
Colors = 1..nbc;
cout<<"Construction heuristic, done: "<<nbc<<" colors"<< endl;
```

code1.java/png code3.cpp

Outline

Local Search Algorithms

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

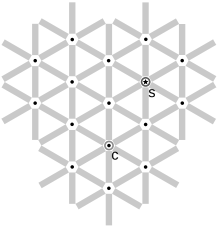
- search space $S(\pi)$
  specified by candidate solution representation:
  discrete structures: sequences, permutations, graphs, partitions
  (*e.g.*, for SAT: array, sequence of all truth assignments
  to propositional variables)

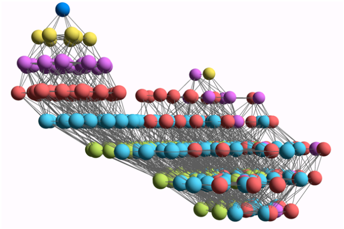
  Note: solution set $S'(\pi) \subseteq S(\pi)$
  (*e.g.*, for SAT: models of given formula)

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

- neighborhood function, $\mathcal{N}_\pi : S \rightarrow 2^{S(\pi)}$
  (*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

Further components [according to B5]

- set of memory states $M(\pi)$
  (may consist of a single state, for LS algorithms that do not use memory)
- initialization function $\texttt{init} : \emptyset \to S(\pi)$
  (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** $\emptyset$

```
(s, m) := init(π)

while not terminate(π, s, m) do
    (s, m) := step(π, s, m)

if s ∈ S'(π) then
    return s
else
    return ∅
```

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

```
(s, m) := init(π');
s_b := s;
while not terminate(π', s, m) do
    (s, m) := step(π', s, m);
    if f(π', s) < f(π', ŝ) then
        s_b := s;

if s_b ∈ S'(π') then
    return s_b
else
    return ∅
```

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

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

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;
```

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 $f(\pi) : S \to \mathbf{R}$
4. set of memory states $M(\pi)$
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 $\texttt{terminate} : S(\pi) \times M(\pi) \to \{\top, \bot\}$

References

Brélaz D. (1979). New methods to color the vertices of a graph. ***Communications of the ACM***, **22(4), pp. 251–256.**

Leighton F.T. (1979). A graph coloring algorithm for large scheduling problems. ***Journal of Research of the National Bureau of Standards***, **84(6), pp. 489–506.**