DM841

Discrete Optimization

Part 2 – Lecture 3
Local Search
Overview

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:

1. search space $S(\pi)$
   - specified by the definition of (finite domain, integer) variables and their values handling implicit constraints
   - all together they determine the representation of candidate solutions
   - common solution representations are discrete structures such as: sequences, permutations, partitions, graphs
     (*e.g.*, for SAT: array, sequence of truth assignments
     to propositional variables)

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

Local Search Algorithms (cntd)

2. evaluation function $f_\pi : S(\pi) \to \mathbf{R}$
   - it handles the soft constraints and the objective function (*e.g.*, for SAT: number of false clauses)

3. neighborhood function, $\mathcal{N}_\pi : S \to 2^{S(\pi)}$
   - defines for each solution $s \in S(\pi)$ a set of solutions $N(s) \subseteq S(\pi)$ that are in some sense close to s.
     (*e.g.*, for SAT: neighboring variable assignments differ in the truth value of exactly one variable)

Local Search Algorithms (cntd)

Further components [according to [HS]]

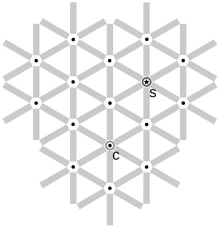
4. set of memory states $M(\pi)$
   (may consist of a single state, for LS algorithms that do not use memory)

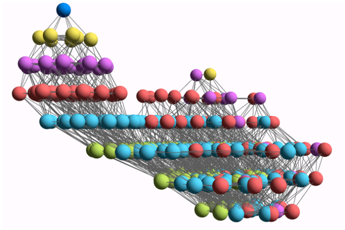
5. initialization function $\texttt{init} : \emptyset \to S(\pi)$
   (can be seen as a probability distribution $\Pr(S(\pi) \times M(\pi))$ over initial search positions and memory states)

6. 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)

7. termination predicate $\texttt{terminate} : S(\pi) \times M(\pi) \to \{\top, \bot\}$
   (determines the termination state for each search position and memory state)

Local search — global view





Neighborhood graph

- 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 (heuristic pivot 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

Example: Local Search for SAT

Example: Uninformed random walk for SAT (1)

- **solution representation and search space S:**
  array of boolean variables representing the truth assignments to variables in given formula F
  no implicit constraint
  (**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** handles clause and proposition constraints
  $f(s) = 0$ if model $f(s) = 1$ otherwise
- **memory:** not used, *i.e.*, $M := \emptyset$

Example: Uninformed random walk for SAT (2)

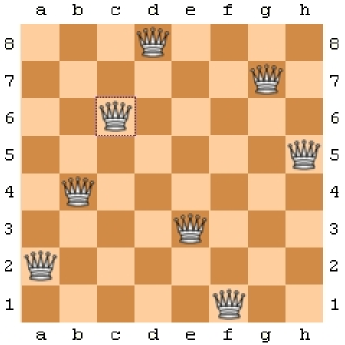
- **initialization:** uniform random choice from S, *i.e.*, $\mathtt{init}(,\{a', m\}) := 1/|S|$ for all assignments a' and memory states m
- **step function:** uniform random choice from current neighborhood, *i.e.*, $\mathtt{step}(\{a, m\}, \{a', m\}) := 1/|N(a)|$
  for all assignments a and memory 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.*, $\mathtt{terminate}(\{a, m\}) := \top$ if a is a model of F, and 0 otherwise.

N-Queens Problem

N-Queens problem

Input: A chessboard of size $N \times N$

Task: Find a placement of n queens on the board such that no two queens are on the same row, column, or diagonal.



Local Search Examples
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;
```

Local Search Examples
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;
```

Metaheuristics

- Variable Neighborhood Search and Large Scale Neighborhood Search
  diversified neighborhoods + incremental algorithmics ("diversified" $\equiv$ multiple, variable-size, and rich).
- Tabu Search: Online learning of moves
  Discard undoing moves,
  Discard inefficient moves
  Improve efficient moves selection
- Simulated annealing
  Allow degrading solutions
- "Restart" + parallel search
  Avoid local optima
  Improve search space coverage

Summary: Local Search Algorithms

For given problem instance π:

1. search space S_π, solution representation: variables + implicit constraints
2. evaluation function $f_\pi : S \to \mathbf{R}$, soft constraints + objective
3. neighborhood relation $\mathcal{N}_\pi \subseteq S_\pi \times S_\pi$
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 $\texttt{terminate} : S_\pi \times M_\pi \to \{\top, \bot\}$

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

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 $\texttt{terminate}(\pi', \texttt{s}, \texttt{m})$ **do**
  $(s, m) := \texttt{step}(\pi', \texttt{s}, \texttt{m})$;
  if $f(\pi', s) < f(\pi', \hat{s})$ **then**
    $s_b := s$;

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

However, the algorithm on the left has little guidance, hence most often decision problems are transformed in optimization problems by, eg, couting number of violations.

Outline

1. Local Search Algorithms

2. Basic Algorithms

Iterative Improvement

- 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)

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.*, $\texttt{terminate}(a) := \top$ 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 $\emptyset$

Local Search Algorithms
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;
```

Local Search Algorithms
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;
```

Local Search Algorithms

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;
```

Local Search Algorithms

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;
```