Topics in Local Search

Thomas Stützle & Irina Dumitrescu

{tom,irina}@intellektik.informatik.tu-darmstadt.de

http://www.intellektik.informatik.tu-darmstadt.de/~tom,~irina

Darmstadt University of Technology
Department of Computer Science
Intellectics Group
Outline

- local search
- efficiency
  - cost function evaluation
  - neighborhood pruning
  - speed-up techniques / data structures
- large neighborhoods
  - variable depth search
  - very large neighborhood search
- concluding remarks

Thomas Stützle, Irina Dumitrescu, Topics in Local Search — MN Summerschool, Tenerife, 2003 – p.2
Example problem — TSP

- given: fully connected, weighted Graph
  \( G = (V, E, d) \)
- goal: find shortest Hamiltonian cycle
- hardness: \( \mathcal{NP} \)-hard
- interest: standard benchmark problem for algorithmic ideas
Local search

Ingredients

- (candidate) solution representation, search space definition $S$
  TSP: set of all possible permutations of the city indices

- solution set $S' \subseteq S$
  TSP: set of all shortest Hamiltonian cycles (tours)

- cost function $f : S \leftrightarrow \mathbb{R}_0^+$
  TSP: sum of the weights of the edges in a tour

- neighborhood relation $\mathcal{N} \subseteq S \times S$
  TSP: e.g. $k$-exchange neighborhood;
  two tours differ in (at most) $k$ edges

- an examination scheme of the neighborhood
  how to search neighborhood and to choose a new solution
Local search

Main issues

- neighborhood definition
  - problem specific
  - essential influence on efficiency and effectiveness of local search
  - tradeoff: size and solution quality vs. time to search
- neighborhood examination mechanism
  - in which order to search neighborhood
  - which neighboring solution becomes new one (pivoting rule)
- remark: neighborhoods are typically defined through moves that are applicable to solutions
in this presentation we only talk about issues when implementing iterative improvement algorithms

procedure Iterative improvement

while \{s' \mid s' \in \mathcal{N}(s), f(s') < f(s)\} \neq \emptyset do

\quad s \leftarrow s'

end
Advantages of local search

Why can local search be good (trivial reasons)?

- cost of generating neighboring solutions
  - typically, for generating a neighboring solution the computational complexity is much lower than generating a new solution from scratch
  - for evaluating a neighboring solution, it often does not need to generate it explicitly at all

- cost of evaluating neighboring solutions
  - typically $\Delta$-evaluation can be done in a computational cost that is much less than computing solution cost from scratch
Disadvantages of local search?

- iterative improvement may take exponential time in the worst case
  
  *but usually this occurs only rarely and for few problems*

- exponential increase of the number of local minima with instance size

- short-sightedness of local search

- general: problem of local optimality

Experience has shown that the disadvantages are for many problems by far outweighted by the advantages
Pivoting rules

gives a rule which of the neighboring solutions is accepted
  - best improvement
  - first improvement
  - (worst improvement)  please, don’t use this one

"checkout-time"

it is problem dependent, which pivoting rule results in better quality solutions or gives place to faster local search algorithms

pivoting rules can have significant influence on the performance of local search algorithms
**Example where first is faster than best**

- TSP, 2-opt, averages over 10 local searches  
  (CPU: UltraSparc 300MHz)

<table>
<thead>
<tr>
<th></th>
<th>best imp</th>
<th>first imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>No.moves</td>
<td>Δ_avg</td>
</tr>
<tr>
<td>d198</td>
<td>0.93</td>
<td>220 4.8%</td>
</tr>
<tr>
<td>lin318</td>
<td>4.35</td>
<td>380 7.9%</td>
</tr>
<tr>
<td>pcb442</td>
<td>11.60</td>
<td>500 10.9%</td>
</tr>
<tr>
<td>rat783</td>
<td>72.06</td>
<td>750 10.0%</td>
</tr>
<tr>
<td>pcb1173</td>
<td>—</td>
<td>— — —</td>
</tr>
<tr>
<td>fl1577</td>
<td>—</td>
<td>— — —</td>
</tr>
<tr>
<td>pr2392</td>
<td>—</td>
<td>— — —</td>
</tr>
</tbody>
</table>

Thomas Stützle, Irina Dumitrescu, Topics in Local Search — MN Summerschool, Tenerife, 2003 – p.10
Neighborhood examination

- order of searching the neighborhood
  - deterministic order
  - random order

- where to continue the local search after an exchange
  - continue from where you are
  - restart from where you started scanning the neighborhood

→ best variant needs to be determined in an experimental way
compute cost of a \(2\text{-opt}\) move in constant time (\(\mathcal{O}(1)\)) as

\[
\Delta_{ij} = d({i, j}) + d({s(i), s(j)}) - d({i, s(i)}) - d({j, s(j)})
\]

\(i, s(i)\): city \(i\) and its successor in the tour
\(j, s(j)\): city \(j\) and its successor in the tour

cost of evaluation function evaluation from scratch: linear time (\(\mathcal{O}(n)\))
Example: QAP

- **given:** \( n \) objects and \( n \) locations with
  - \( a_{ij} \): flow from object \( i \) to object \( j \)
  - \( d_{rs} \): distance between location \( r \) and location \( s \)

- **goal:** find an assignment (i.e. a permutation) of the \( n \) objects to the \( n \) locations that minimizes

\[
\min_{\pi \in \Pi(n)} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} d_{\pi(i)\pi(j)}
\]

\( \pi(i) \) gives location of object \( i \)

- **interest:** it is among the “hardest” combinatorial optimization problems; several applications
Example: QAP

basic neighborhood for the QAP

- **LocalSearch**: 2-opt

![Diagram of a 2-opt operation on a QAP example]

- computation of the cost function from scratch in $\mathcal{O}(n^2)$
- for 2-opt we can again use $\Delta$-evaluation
  - a neighboring solution can be evaluated in $\mathcal{O}(n)$
- one full neighborhood scan can be completed in $\mathcal{O}(n^3)$
Exchange of objects $r, s$

for asymmetric instances

$$\Delta(\pi, r, s) = a_{rr} \cdot (d_{\pi_s\pi_r} - d_{\pi_r\pi_r}) + a_{rs} \cdot (d_{\pi_s\pi_r} - d_{\pi_r\pi_s}) +$$

$$a_{sr} \cdot (d_{\pi_r\pi_s} - d_{\pi_s\pi_r}) + a_{ss} \cdot (d_{\pi_r\pi_r} - d_{\pi_s\pi_s}) +$$

$$\sum_{k=1, k \neq r, s}^{n} (a_{kr} \cdot (d_{\pi_k\pi_s} - d_{\pi_k\pi_r}) + a_{ks} \cdot (d_{\pi_k\pi_r} - d_{\pi_k\pi_s}) +$$

$$a_{rk} \cdot (d_{\pi_s\pi_k} - d_{\pi_r\pi_k}) + a_{sk} \cdot (d_{\pi_r\pi_k} - d_{\pi_s\pi_k}))$$

for symmetric instances

$$\text{(1) } \Delta(\pi, r, s) = 2 \cdot \sum_{k=1, k \neq r, s}^{n} (a_{sk} - a_{rk}) \cdot (d_{\pi(r)\pi(k)} - d_{\pi(s)\pi(k)})$$
**Fast update**

- \( \pi' \) results from \( \pi \) by exchanging objects \( r, s \)
- computation of \( \Delta(\pi', u, v) \), with \( \{u, v\} \cap \{r, s\} = \emptyset \)

\[
\Delta(\pi', u, v) = \Delta(\pi, u, v) + (a_{ru} - a_{rv} + a_{sv} - a_{su}) \cdot \\
(d_{\pi s \pi u} - d_{\pi s \pi v} + d_{\pi r \pi v} - d_{\pi r \pi u}) + (d_{\pi u \pi s} - d_{\pi v \pi s} + d_{\pi v \pi r} - d_{\pi u \pi r}) \cdot \\
(a_{ur} - a_{vr} + a_{vs} - a_{us})
\]
**Fast update**

- fast update can be used within best improvement local search (ie. also tabu search)
- requires: additional memorization of the $\Delta(\pi, i, j)$ value for all pairs $r, s$ in a table
- first local search iteration in $O(n^3)$ for initializing the table
- in the subsequent iterations exchanges can be computed in $O(1)$
- exception: objects that were moved in previous iteration
**Example — 2-opt for QAP**

Local search variants of 2-opt, average results over 100 restarts; times measured on a Pentium III 500MHz

<table>
<thead>
<tr>
<th></th>
<th>best Imp.</th>
<th>first Imp</th>
<th>first Imp+dlbs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>secs</td>
<td>moves</td>
<td>Δ&lt;sub&gt;avg&lt;/sub&gt;</td>
</tr>
<tr>
<td>tai50b</td>
<td>0.06</td>
<td>56</td>
<td>6.9%</td>
</tr>
<tr>
<td>tai60b</td>
<td>0.12</td>
<td>72</td>
<td>8.3%</td>
</tr>
<tr>
<td>tai80b</td>
<td>0.33</td>
<td>100</td>
<td>6.3%</td>
</tr>
<tr>
<td>sko72</td>
<td>0.21</td>
<td>77</td>
<td>2.6%</td>
</tr>
<tr>
<td>sko80</td>
<td>0.31</td>
<td>89</td>
<td>2.3%</td>
</tr>
<tr>
<td>sko90</td>
<td>0.46</td>
<td>105</td>
<td>2.2%</td>
</tr>
</tbody>
</table>
**don’t look bits**

- a technique that allows to focus the local search around the part where potentially there can happen something
- allows to reduce the checkout time
- only applicable with first-improvement pivoting rule
- proceeds as
  - associate to each "component" a don’t look bit
  - if don’t look bit is zero, the component can be used in an outer loop of an improvement search
  - if no improving move is found for the component: set its don’t look bit to one
  - if a component is involved in a move: set don’t look bit to zero
  - if for the "outer-loop component" no improving move is found: set its don’t look bit to one
procedure iterative improvement

for $i = 1$ to $n$ do
    if $dlb[i] = 1$ then continue

    improve_flag ← false

    for $j = 1$ to $n$ do
        CheckMove($i, j$)
        if move improves then
            ApplyMove($i, j$); $dlb[i] ← 0, dlb[j] ← 0$
            improve_flag ← true
        endfor
    endfor

    if improve_flag = false then $dlb[i] ← 1$
end

end iterative improvement
often: significant speed-up at only low loss of solution quality

integration possibilities between perturbation and local search in ILS

reset don’t look bits to zero only of "moved" solution components in a perturbation

same possibility is available for memetic algorithms after applying recombination or mutation

some SLS methods do not allow for an easy integration of don’t look bits
Perturbation — Speed, ILS for TSP

- compare No. local searches (here, 3-opt) in fixed computation time
- \#LS_{RR}: No. local searches with random restart
- \#LS_{1-DB}: No. local searches with one double bridge move as Perturbation
- \#LS_{1-DB}/\#LS_{RR}: factor between \#LS_{1-DB} and \#LS_{RR}
- time limit: 120 sec on a Pentium II 266 MHz PC

<table>
<thead>
<tr>
<th>instance</th>
<th>#LS_{RR}</th>
<th>#LS_{1-DB}</th>
<th>#LS_{1-DB}/#LS_{RR}</th>
</tr>
</thead>
<tbody>
<tr>
<td>kroA100</td>
<td>17507</td>
<td>56186</td>
<td>3.21</td>
</tr>
<tr>
<td>d198</td>
<td>7715</td>
<td>36849</td>
<td>4.78</td>
</tr>
<tr>
<td>lin318</td>
<td>4271</td>
<td>25540</td>
<td>5.98</td>
</tr>
<tr>
<td>pcb442</td>
<td>4394</td>
<td>40509</td>
<td>9.22</td>
</tr>
<tr>
<td>rat783</td>
<td>1340</td>
<td>21937</td>
<td>16.38</td>
</tr>
<tr>
<td>pr1002</td>
<td>910</td>
<td>17894</td>
<td>19.67</td>
</tr>
<tr>
<td>d1291</td>
<td>835</td>
<td>23842</td>
<td>28.56</td>
</tr>
<tr>
<td>f11577</td>
<td>742</td>
<td>22438</td>
<td>30.24</td>
</tr>
<tr>
<td>pr2392</td>
<td>216</td>
<td>15324</td>
<td>70.94</td>
</tr>
<tr>
<td>pcb3038</td>
<td>121</td>
<td>13323</td>
<td>110.1</td>
</tr>
<tr>
<td>f13795</td>
<td>134</td>
<td>14478</td>
<td>108.0</td>
</tr>
<tr>
<td>rl5915</td>
<td>34</td>
<td>8820</td>
<td>259.4</td>
</tr>
</tbody>
</table>
Neighborhood pruning

Example: TSP, 2-opt local search

- **important property**: for any improving $2$-$\text{opt}$ move, there is at least one node that is incident to an edge $e$ that is replaced by a different edge $e'$ with lower weight.

- **fixed radius nearest neighbour search**
  - consider both tour neighbours of a node $v_i$, say $v_j$
  - search around $v_i$ for nodes $v_k$ for which holds $d(v_i, v_k) < d(v_i, v_j)$
  - for each such city $v_k$ delete unique edge to make feasible $2$-$\text{opt}$ move and test for improvement
  - if fixed radius near neighbor searches for all nodes are unsuccessful, the tour is $2$-$\text{opt}$
Neighborhood pruning

- support fixed radius search by appropriate data structures
- nearest neighbor lists for each city
  - for each node $v_i$, $nl(v_i, r)$ gives the $r$–nearest neighbour of $v_i$
  - several possibilities available of how to construct candidate sets

neighborhood pruning applicable similarly to many geometric problems; for other problems often more complicated than for TSP or not possible at all
**Example results: TSP**

Timings for 1000 local searches with 2-opt and 3-opt variants from random initial solutions on a Pentium III 500 MHz CPU. **std**: no speed-up techniques; **fr+cl**: fixed radius and unbounded candidate lists, **dlb**: don’t look bits.

<table>
<thead>
<tr>
<th>instance</th>
<th>2-opt-std</th>
<th>2-opt-fr+cl</th>
<th>2-opt-fr+cl+dlb</th>
<th>3-opt-fr+cl+dlb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Delta_{avg}$</td>
<td>$t_{avg}$</td>
<td>$\Delta_{avg}$</td>
<td>$t_{avg}$</td>
</tr>
<tr>
<td>kroA100</td>
<td>8.9</td>
<td>1.6</td>
<td>6.4</td>
<td>0.5</td>
</tr>
<tr>
<td>d198</td>
<td>5.7</td>
<td>6.4</td>
<td>4.2</td>
<td>1.2</td>
</tr>
<tr>
<td>lin318</td>
<td>10.6</td>
<td>22.1</td>
<td>7.5</td>
<td>2.1</td>
</tr>
<tr>
<td>pcb442</td>
<td>12.7</td>
<td>55.7</td>
<td>7.1</td>
<td>2.9</td>
</tr>
<tr>
<td>rat783</td>
<td>13.0</td>
<td>239.7</td>
<td>7.5</td>
<td>7.5</td>
</tr>
<tr>
<td>pr1002</td>
<td>12.8</td>
<td>419.5</td>
<td>8.4</td>
<td>13.2</td>
</tr>
<tr>
<td>pcb1173</td>
<td>14.5</td>
<td>603.1</td>
<td>8.5</td>
<td>16.7</td>
</tr>
<tr>
<td>d1291</td>
<td>16.8</td>
<td>770.3</td>
<td>10.1</td>
<td>16.9</td>
</tr>
<tr>
<td>fl1577</td>
<td>13.6</td>
<td>1251.1</td>
<td>7.9</td>
<td>25.8</td>
</tr>
<tr>
<td>pr2392</td>
<td>15.0</td>
<td>2962.8</td>
<td>8.8</td>
<td>65.5</td>
</tr>
</tbody>
</table>
Efficient computation of move values

Example: SAT, GSAT local search architecture

- 1-opt neighbourhood
- one of the first local search algorithms for SAT
- best improvement pivoting rule
Local Search for SAT

procedure local search for SAT
    input CNF formula $\Phi$, maxTries, maxSteps
    output model for $\Phi$ or “no solution found"
    for $i := 1$ to maxTries do
        $s := \text{initAssign}(\Phi)$;
        for $j := 1$ to maxSteps do
            if $s$ satisfies $\Phi$ then return $s$;
            else
                $x := \text{chooseVariable}(\Phi, s)$;
                $s := s$ with truth value flipped for $x$;
            end if
        end for
    end for
    return “no solution found”; 
end local search for SAT
Implementation issues

for each iteration one needs to compute scores of variables

simple approach

- recompute scores after each iteration from scratch
- requires effort in $O(m \cdot CL(n))$

$m$: number of clauses, $CL(n)$: bound on maximum clause length

efficient computation of flip effects required

idea

- use dynamic update of the scores
- use appropriate data structures to allow for the dynamic update
Implementation issues

- central observation
  - only the score of variables $x'$ is affected by flipping a variable $x$ that occur in a same clause as variable $x$
  - for an update of the score only clauses are interesting in which the flipped variable $x$ occurs

\[
C_{dep}(\Phi, x) = \{ c \text{ is a clause of } \Phi \mid x \text{ appears in clause } c \}
\]

- data structure
  - each variables has a list of clauses where it occurs
  - for each variable store truth value, score; for each clause store it satisfaction status
Score update

- the score of variable $x$ changes its sign
- go through all the clauses $c \in C_{dep}(\Phi, x)$ and for all variables in each clause do
  - if the clause has become unsatisfied by flipping $x$, then increase the score of the other variables by one
  - if the clause was unsatisfied and has become satisfied, then decrease score of all other variables by one
  - if flipping $x$ makes two variables instead of one satisfying the clause, then search the other one and increase its score by one

Analogous update schemes are essential for many problems like graph coloring, time tabling, set covering, etc.
Large neighborhoods

- several local search algorithms search large, typically exponentially sized neighborhoods
- exploration of the neighborhoods through appropriate techniques typically possible in polynomial time either through
  - insight into neighborhood structure and searching it with exact algorithms
  - a heuristically guided search in the neighborhood
- advantages: typically much better solution quality reachable
Variable depth search algorithms

- complex moves are build as being a concatenation of a number of simple moves
- the number of simple moves composing a complex one is variable and determined based on gain criteria
- the simple moves need not be independent of each other
- termination is guaranteed through additional conditions on the simple moves

example

- Lin-Kernighan algorithm for TSPs
Lin-Kernighan for TSPs

- at each complex search step a set of edges $X = \{x_1, \ldots, x_r\}$ is deleted and another set $Y = \{y_1, \ldots, y_r\}$ is added to a tour
- the number of edges $r$ is determined dynamically
- the two sets $X$ and $Y$ are constructed iteratively, element by element
- edges $x_i$ and $y_i$ as well as $y_i$ and $x_{i+1}$ must share an endpoint (sequential moves)
- at any point in the search there needs to be an edge $y'_i$ such that the complex step defined by $X = \{x_1, \ldots, x_i\}$ and $Y = \{y_1, \ldots, y'_i\}$ is a feasible tour
- illustration: $\delta$-path
Lin-Kernighan for TSPs

(a)

(b)

(c)

(d)
\( \delta\text{-path} \)

\( \delta\text{-path}: \) (spanning tree plus one edge)
convert a $\delta$-path into a tour
new $\delta$-path from previous $\delta$-path
Limitations on moves

- length restrictions
  - edges that are included in set $Y$ (added edges) may not be deleted anymore
  - edges that are included in set $X$ (deleted edges) may not be added again

$\Rightarrow$ bounds the depth of the search to a maximum of $n$ moves

- cost restrictions
  - stop the construction process of the complex move if the resulting $\delta$-path has higher weight than the shortest tour found in the process
at each step, try to include a least costly possible edge $y_i$

if no improved complex move is found

- apply backtracking on the first and second level of the construction steps (choices of $x_1, y_1, x_2, y_2$)
- consider alternative choices in order of increasing weight of candidates up to a maximum number of candidates
- at the last level, consider different starting nodes for search
- backtracking assures final tour to be 2-opt and 3-opt
- important are techniques for pruning the search
Lin-Kernighan algorithm is best performing local search for TSP

many variants of the algorithm are available (see also recent DIMACS challenge)

an efficient implementation requires sophisticated data structures (several articles available on this subject)

implementation is quite time consuming, but

at least three very good implementations are publically available (concorde, Helsgaun, Neto)

variable depth search algorithms are now available for many problems and for many they show excellent performance
Ejection chains

- similar approach as in variable depth search algorithms
- differences concern mainly that ejection chains allow for more flexibility in move generation
- complex moves are composed of a sequence of dependent, simple moves
- “Ejection moves”: moves that allow to do a transition to a different solution by ejecting some solution components
- “trial moves”: moves that try to restore feasible solutions
Conclusions (1)

- implementation aspects
  - efficient evaluation of cost functions (Δ–evaluation etc.)
  - seemingly minor implementation details can have significant influence on search performance (pivoting rules, order in which neighborhoods are scanned etc.)
  - exploitation of problem specific properties can improve strongly the speed of local search
  - appropriate data structures are essential for efficient implementations of local search algorithms

But: they do not make up for a poor choice of a neighborhood
Conclusions (2)

- complex (large-scale) neighborhoods
  - allow to obtain better solution quality in a single local search step than simple exchange neighborhoods
  - often explore exponentially sized neighborhoods but that are explored typically in polynomial time
  - often relatively complex to implement them efficiently
  - often require deep knowledge about the problem for their development
  - but for several problems they are the (by far) best performing local search algorithms


