DM811
Heuristics for Combinatorial Optimization

Lecture 11
Efficient Local Search

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Efficiency vs Effectiveness

The performance of local search is determined by:

1. quality of local optima (effectiveness)
2. time to reach local optima (efficiency):
   A. time to move from one solution to the next
   B. number of solutions to reach local optima

Note:

- Local minima depend on evaluation function f and neighborhood function $\mathcal{N}$.
- Larger neighborhoods $\mathcal{N}$ induce
  - neighborhood graphs with smaller diameter;
  - fewer local minima.

Ideal case: exact neighborhood, *i.e.*, neighborhood function for which any local optimum is also guaranteed to be a global optimum.

- Typically, exact neighborhoods are too large to be searched effectively (exponential in size of problem instance).

Trade-off (to be assessed experimentally):

- Using larger neighborhoods
  can improve performance of LS algorithms.
- **But:** time required for determining improving search steps
  increases with neighborhood size.

Speedups Techniques for Efficient Neighborhood Search

1) Incremental updates

2) Neighborhood pruning

Speedups in Neighborhood Examination

1) Incremental updates (aka delta evaluations)

- **Key idea:** calculate effects of differences between
  current search position s and neighbors s' on
  evaluation function value.
- Evaluation function values often consist of
  independent contributions of solution components;
  hence, $f(s)$ can be efficiently calculated from $f(s')$ by differences
  between s and s' in terms of solution components.
- Typically crucial for the efficient implementation of
  II algorithms (and other LS techniques).

Do not do this:

```
tmp ← current
while ∃ unseen sol in N(current) do
    change current into sol
    evaluate current
    if current better than tmp then
        break;
    current ← tmp
```

Do this:

```
while ∃ unseen sol in N(current) do
    evaluate changes at current
    if improving then
        change current into sol
```


Example: Incremental updates for TSP

- solution components = edges of given graph G
- standard 2-exchange neighborhood, *i.e.*, neighboring
  round trips p, p' differ in two edges
- $w(p') := w(p)$ − edges in p but not in p'
  \+ edges in p' but not in p

Note: Constant time (4 arithmetic operations), compared to
linear time (n arithmetic operations for graph with n vertices)
for computing $w(p')$ from scratch.

2) Neighborhood Pruning

- **Idea:** Reduce size of neighborhoods by excluding neighbors that are likely (or guaranteed) not to yield improvements in f.
- **Note:** Crucial for large neighborhoods, but can be also very useful for small neighborhoods (*e.g.*, linear in instance size).

Example: Heuristic candidate lists for the TSP

- *Intuition:* High-quality solutions likely include short edges.
- Candidate list of vertex v: list of v's nearest neighbors (limited number), sorted according to increasing edge weights.
- Search steps (*e.g.*, 2-exchange moves) always involve edges to elements of candidate lists.
- Significant impact on performance of LS algorithms for the TSP.

Overview

Delta evaluations and neighborhood examinations in:

- Permutations
  - TSP
  - SMTWTP, Parallel Machine, Bin Packing
- Assignments
  - CSP, SAT, GCP, Bin Packing
- Sets
  - Set Covering, Max Independent Set, p-median

Local Search for the Traveling Salesman Problem

- k-exchange heuristics
  - 2-opt
  - 2.5-opt
  - Or-opt
  - 3-opt
- complex neighborhoods
  - Lin-Kernighan
  - Helsgaun's Lin-Kernighan
  - Dynasearch
  - ejection chains approach

Implementations exploit speed-up techniques

1. neighborhood pruning: fixed radius nearest neighborhood search
2. neighborhood lists: restrict exchanges to most interesting candidates
3. don't look bits: focus perturbative search to "interesting" part
4. sophisticated data structures

TSP data structures

Tour representation:

- `reverse`(a, b)
- `succ`
- `prec`
- `sequence(a,b,c)` – check whether b is within a and b

Possible choices:

- $|V| < 1.000$ array for π and π^{-1}
- $|V| < 1.000.000$ two level tree
- $|V| > 1.000.000$ splay tree

Moreover static data structure:

- priority lists
- k-d trees

Look at implementation of local search for TSP by T. Stützle:

File: http://www.imada.sdu.dk/~marco/DM811/Lab/ls.c

```
two_opt_b(tour); % best improvement, no speedup
two_opt_f(tour); % first improvement, no speedup
two_opt_best(tour); % first improvement including speed-ups (dlbs, fixed
     radius near neighbour searches, neughbourhood lists)
two_opt_first(tour); % best improvement including speed-ups (dlbs, fixed
     radius near neighbour searches, neughbourhood lists)
three_opt_first(tour); % first improvement
```


Table 17.1 Cases for k-opt moves.

k	No. of Cases
2	1
3	4
4	20
5	148
6	1,358
7	15,104
8	198,144
9	2,998,656
10	51,290,496

[Appelgate Bixby, Chvátal, Cook, 2006]

Table 17.2 Computer-generated source code for k-opt moves.

k	No. of Lines
6	120,228
7	1,259,863
8	17,919,296

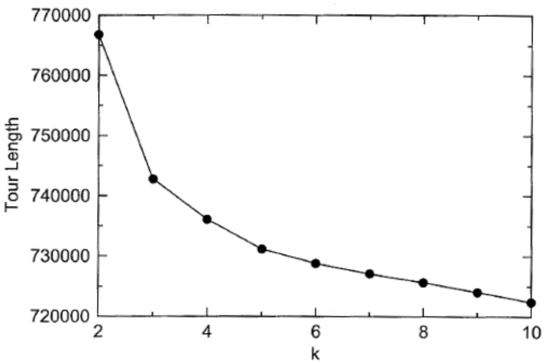

Figure 17.1 k-opt on a 10,000-city Euclidean TSP.

Single Machine Total Weighted Tardiness Problem

- Interchange: size $\binom{n}{2}$ and $O(|i-j|)$ evaluation each
  - first-improvement: π_j, π_k
    - $p_{\pi_j} \leq p_{\pi_k}$ for improvements, $w_j T_j + w_k T_k$ must decrease because jobs in $\pi_j, \ldots, \pi_k$ can only increase their tardiness.
    - $p_{\pi_j} \geq p_{\pi_k}$ possible use of auxiliary data structure to speed up the computation
  - best-improvement: π_j, π_k
    - $p_{\pi_j} \leq p_{\pi_k}$ for improvements, $w_j T_j + w_k T_k$ must decrease at least as the best interchange found so far because jobs in $\pi_j, \ldots, \pi_k$ can only increase their tardiness.
    - $p_{\pi_j} \geq p_{\pi_k}$ possible use of auxiliary data structure to speed up the computation
- Swap: size $n-1$ and $O(1)$ evaluation each
- Insert: size $(n-1)^2$ and $O(|i-j|)$ evaluation each
  But possible to speed up with systematic examination by means of swaps: an insert is equivalent to $|i-j|$ swaps hence overall examination takes $O(n^2)$

The Max Independent Set Problem

Max Independent Set (aka, stable set problem or vertex packing problem)

Given: an undirected graph $G(V, E)$ and a non-negative weight function ω on V ($\omega : V \to \mathbf{R}$)
Task: A largest weight independent set of vertices, i.e., a subset $V' \subseteq V$ such that no two vertices in V' are joined by an edge in E.

Related Problems:

Vertex Cover

Given: an undirected graph $G(V, E)$ and a non-negative weight function ω on V ($\omega : V \to \mathbf{R}$)
Task: A smallest weight vertex cover, i.e., a subset $V' \subseteq V$ such that each edge of G has at least one endpoint in V'.

Maximum Clique

Given: an undirected graph $G(V, E)$
Task: A maximum cardinality clique, i.e., a subset $V' \subseteq V$ such that every two vertices in V' are joined by an edge in E

The p-median Problem

- **Given:**
  a set U of locations for n users
  a set F of locations of m facilities
  a distance matrix $D = [d_{ij}] \in \mathbf{R}^{n \times m}$
- **Task:** Select p locations of F where to install facilities such that the sum of the distances of each user to its closest installed facility is minimized, *i.e.*,

$$\min_J \sum_{i \in U} \min_{j \in F} d_{ij} \qquad J \subseteq F \text{ and } |J| = p$$