DM841
Discrete Optimization - Heuristics

Very Large-Scale Neighborhoods

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Course Overview

- ✔ Local Search: Components, Basic Algorithms
- ✔ Local Search: Neighborhoods and Search Landscape
- ✔ Metaheuristics: Construction based, LS based, Population based
- ✔ Working Environment and Implementation Frameworks
- ✔ Methods for the Analysis of Experimental Results
- ✔ Algorithm Configuration and Selection
- ✔ Efficient Local Search: neighborhood pruning, incremental updates and data structures
- ✖ Large Scale Neighborhoods

Examples: SAT, TSP, PB, GCP, Steiner Tree, Unrelated Parallel Machines, ...

Very Large Scale Neighborhoods

Small neighborhoods:

- might be short-sighted
- need many steps to traverse the search space

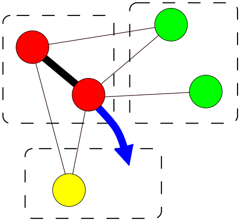
Large neighborhoods

- introduce large modifications to reach higher quality solutions
- allow to traverse the search space in few steps

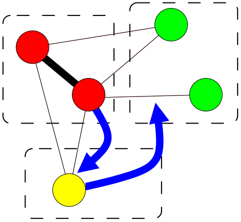
Key idea: use very large neighborhoods that can be searched efficiently (preferably in polynomial time) or are searched heuristically

Example (GCP)

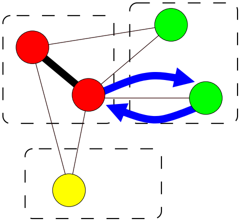
Neighborhood Structures: Very Large Scale Neighborhood



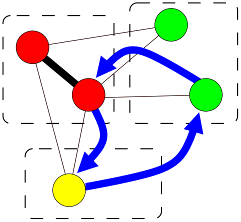
One Exchange



Path Exchange



Swap



Cyclic Exchange

Very large scale neighborhood search

1. define an exponentially large neighborhood
   (though, $O(n^3)$ might already be large)
2. define a polynomial time search algorithm to search the neighborhood
   (= solve the neighborhood search problem, NSP)
   - exactly (leads to a best improvement strategy)
   - heuristically (some improving moves might be missed)

Examples of VLSN Search

[Ahuja, Ergun, Orlin, Punnen, 2002]

- based on concatenation of simple moves
  - Variable Depth Search (Lin-Kernighan for TSP and Graph Partitioning)
  - Ejection Chains
- based on Dynamic Programming or Network Flows
  - Dynasearch (ex. SMTWTP)
  - Weighted Matching based neighborhoods (ex. TSP)
  - Cyclic exchange neighborhood (ex. VRP)
  - Shortest path
- based on polynomially solvable special cases of hard combinatorial optimization problems
  - Pyramidal tours
  - Halin Graphs

Outline

Variable Depth Search

- **Key idea:** *Complex steps* in large neighborhoods = variable-length sequences of *simple steps* in small neighborhood.
- Use various *feasibility restrictions* on selection of simple search steps to limit time complexity of constructing complex steps.
- Perform Iterative Improvement w.r.t. complex steps.

Variable Depth Search (VDS):

```
determine initial candidate solution s
while s is not locally optimal do
    t̂ := s
    repeat
        select best feasible neighbor t of t̂
        if f(t) < f(t̂) then
            t̂ := t
            s := t̂
    until construction of complex step has been completed;
```

VLSN for the Traveling Salesman Problem

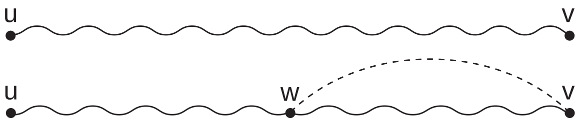
- k-exchange heuristics
  - 2-opt [Flood, 1956, Croes, 1958]
  - 2.5-opt or 2H-opt
  - Or-opt [Or, 1976]
  - 3-opt [Block, 1958]
  - k-opt [Lin 1965]

- complex neighborhoods
  - Lin-Kernighan [Lin and Kernighan, 1965]
  - Helsgaun's Lin-Kernighan
  - Dynasearch
  - Ejection chains approach

The Lin-Kernighan (LK) Algorithm for the TSP (1)

- Complex search steps correspond to sequences of 2-exchange steps and are constructed from sequences of *Hamiltonian paths*

- δ*-path:* Hamiltonian path p + 1 edge connecting one end of p to interior node of p

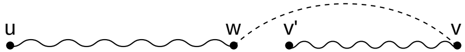


Basic LK exchange step:

- Start with Hamiltonian path $(u, \ldots, v)$:

- Obtain δ-path by adding an edge (v, w):

- Break cycle by removing edge (w, v'):



- *Note:* Hamiltonian path can be completed into Hamiltonian cycle by adding edge (v', u):

Construction of complex LK steps:

1. start with current candidate solution (Hamiltonian cycle) s;
   set $t^* := s$;
   set $p := s$
2. obtain δ-path p' by replacing one edge in p
3. consider Hamiltonian cycle t obtained from p by (uniquely) defined edge exchange
4. if $w(t) < w(t^*)$ then
      set $t^* := t$; $p := p'$; go to step 2
   else accept t^* as new current candidate solution s

Note: This can be interpreted as sequence of 1-exchange steps that alternate between δ-paths and Hamiltonian cycles.

Mechanisms used by LK algorithm:

- *Pruning exact rule:* If a sequence of numbers has a positive sum, there is a cyclic permutation of these numbers such that every partial sum is positive.
  ➨ need to consider only gains whose partial sum remains positive

- *Tabu restriction:* Any edge that has been added cannot be removed and any edge that has been removed cannot be added in the same LK step.

  Note: This limits the number of simple steps in a complex LK step.

- *Limited form of backtracking* ensures that local minimum found by the algorithm is optimal w.r.t. standard 3-exchange neighborhood

- (For further details, see original article)

[LKH Helsgaun's implementation http://www.akira.ruc.dk/~keld/research/LKH/ (99 pages report)]

Graph Partitioning

Graph Partitioning

Given: $G = (V, E)$, weighted function $\omega : V \to \mathrm{R}$, a positive number p: $0 < w_i \leq p, \forall i$ and a connectivity matrix $C = [c_{ij}] \in \mathrm{R}^{|V| \times |V|}$.

Task: A k-partition of G, $V_1, V_2, \ldots, V_k$: $\bigcup_{i=1}^{n} V_i = G$ such that:

- it is admissible, ie, $|V_i| \leq p$ for all i and
- it has minimum cost, ie, the sum of c_{ij}, i, j that belong to different subsets is mimimal

- Focus on 2-way partitions in sets A and B of equal size $(n+n)$
- For each $a \in A$: external cost, $E_a = \sum_{b \in B} c_{ab}$; internal cost: $I_a = \sum_{c \in A} c_{ac}$
  For each $z \in S$: $D_z = E_z - I_z$
- Interchange of a and b has gain: $g_{ab} = D_a + D_b - 2c_{ab}$
- Select a_1^* and b_1^* with maximum gain.
- Recalculate the D values for elements of $A \setminus \{a_1\}$ and $B \setminus \{b_1\}$:
  $D'_y = D_y + 2c_{ya_1} - 2c_{yb_1}$ for $y \in A \setminus \{a_1\}$
  $D'_x = D_x + 2c_{xb_1} - 2c_{xa_1}$ for $x \in B \setminus \{b_1\}$
- Repeat the last two steps conditional to the previous until all vertices are swapped.
- Since $\sum g_i = 0$ then there is an intereation k that maximizes the partial sum $\sum_{i=1}^{k} g_i = G$
- If $G > 0$ then make the change, if $G = 0$ then we are in a local optimum.

Outline

Ejection Chains

- Attempt to use large neighborhoods without examining them exhaustively
- Sequences of successive steps each influenced by the precedent and determined by myopic choices
- Limited in length
- Local optimality in the large neighborhood is not guaranteed.

Example (on TSP):
successive 2-exchanges where each exchange involves one edge of the previous exchange

Example (on GCP):
successive 1-exchanges: a vertex v_1 changes color from $\varphi(v_1) = c_1$ to c_2, in turn forcing some vertex v_2 with color $\varphi(v_2) = c_2$ to change to another color c_3 (which may be different or equal to c_1) and again forcing a vertex v_3 with color $\varphi(v_3) = c_3$ to change to color c_4.

Outline

Single Machine Total Weighted Tardiness

Given: a set of n jobs $\{J_1, \ldots, J_n\}$ to be processed on a single machine and for each job J_i a processing time p_i, a weight w_i and a due date d_i.

Task: Find a schedule that minimizes
the total weighted tardiness $\sum_{i=1}^{n} w_i \cdot T_i$
where $T_i = \max\{C_i - d_i, 0\}$ (C_i completion time of job J_i)

Example:

Job	J_1	J_2	J_3	J_4	J_5	J_6
Processing Time	3	2	2	3	4	3
Due date	6	13	4	9	7	17
Weight	2	3	1	5	1	2

Sequence $\phi = J_3, J_1, J_5, J_4, J_1, J_6$

Job	J_3	J_1	J_5	J_4	J_2	J_6
C_i	2	5	9	12	14	17
T_i	0	0	2	3	1	0
$w_i \cdot T_i$	0	0	2	15	3	0

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 interchange is equivalent to $|i-j|$ swaps hence overall examination takes $O(n^2)$

Dynasearch

- Iterative improvement method based on building complex search steps from combinations of mutually independent search steps
- Mutually independent search steps do not interfere with each other wrt effect on evaluation function and feasibility of candidate solutions.

Example: Independent 2-exchange steps for the TSP:

Therefore: Overall effect of complex search step = sum of effects of constituting simple steps; complex search steps maintain feasibility of candidate solutions.

- **Key idea:** Efficiently find optimal combination of mutually independent simple search steps using *Dynamic Programming*.

Dynasearch for SMTWTP

- two interchanges δ_{jk} and δ_{lm} are independent
  if $\max\{j,k\} < \min\{l,m\}$ or $\min\{j,k\} > \max\{l,m\}$;
- the dynasearch neighborhood is obtained by a series of independent interchanges;
- it has size $2^{n-1} - 1$ (the number of subsets of $n-1$ positions, the nth position involvement being decided consequently and minus 1 because of the case of no change);
- but a best move can be found in $O(n^3)$ searched by dynamic programming;
- it yields in average better results than the interchange neighborhood alone.

Table 1 Data for the Problem Instance

Job j	1	2	3	4	5	6
Processing time p_j	3	1	1	5	1	5
Weight w_j	3	5	1	1	4	4
Due date d_j	1	5	3	1	3	1

Table 2 Swaps Made by Best-Improve Descent

Iteration	Current Sequence	Total Weighted Tardiness
	1 2 3 4 5 6	109
1	1 2 3 5 4 6	90
2	1 2 3 5 6 4	75
3	5 2 3 1 6 4	70

Table 3 Dynasearch Swaps

Iteration	Current Sequence	Total Weighted Tardiness
	1 2 3 4 5 6	109
1	1 3 2 5 4 6	89
2	1 5 2 3 6 4	68
3	5 1 2 3 6 4	67

- state (k, π)
- π_k is the partial sequence at state (k, π) that has min $\sum_j w_j T_j$
- π_k is obtained from state (i, π)

$$\begin{cases} \text{appending job } \pi(k) \text{ after } \pi(i) & i = k-1 \\ \text{appending job } \pi(k) \text{ and interchanging } \pi(i+1) \text{ and } \pi(k) & 0 \le i < k-1 \end{cases}$$

- $F(\pi_0) = 0; \qquad F(\pi_1) = w_{\pi(1)} \left(p_{\pi(1)} - d_{\pi(1)}\right)^+;$

$$F(\pi_k) = \min \begin{cases} F(\pi_{k-1}) + w_{\pi(k)} \left(C_{\pi(k)} - d_{\pi(k)}\right)^+, \\ \min\limits_{1 \le i < k-1} \{F(\pi_i) + w_{\pi(k)} \left(C_{\pi(i)} + p_{\pi(k)} - d_{\pi(k)}\right)^+ + \\ \quad + \sum_{j=i+2}^{k-1} w_{\pi(j)} \left(C_{\pi(j)} + p_{\pi(k)} - p_{\pi(i+1)} - d_{\pi(j)}\right)^+ + \\ \quad + w_{\pi(i+1)} \left(C_{\pi(k)} - d_{\pi(i+1)}\right)^+ \} \end{cases}$$

- The best choice is computed by recursion in $O(n^3)$ and the optimal series of interchanges for $F(\pi_n)$ is found by backtrack.

- Local search with dynasearch neighborhood starts from an initial sequence, generated by ATC, and at each iteration applies the best dynasearch move, until no improvement is possible (that is, $F(\pi_n^t) = F(\pi_n^{(t-1)})$, for iteration t).

- Speedups:
  - pruning with considerations on $p_{\pi(k)}$ and $p_{\pi(i+1)}$
  - maintainig a string of late, no late jobs
  - h_t largest index s.t. $\pi^{(t-1)}(k) = \pi^{(t-2)}(k)$ for $k = 1, \ldots, h_t$ then $F(\pi_k^{(t-1)}) = F(\pi_k^{(t-2)})$ for $k = 1, \ldots, h_t$ and at iter t no need to consider $i < h_t$.

Dynasearch, refinements:

- [Grosso et al. 2004] add insertion moves to interchanges.
- [Ergun and Orlin 2006] show that dynasearch neighborhood can be searched in $O(n^2)$.

Performance:

- exact solution via branch and bound feasible up to 40 jobs [Potts and Wassenhove, Oper. Res., 1985]
- exact solution via time-indexed integer programming formulation used to lower bound in branch and bound solves instances of 100 jobs in 4-9 hours [Pan and Shi, Math. Progm., 2007]
- dynasearch: results reported for 100 jobs within a 0.005% gap from optimum in less than 3 seconds [Grosso et al., Oper. Res. Lett., 2004]

Outline

Weighted Matching Neighborhoods

- **Key idea** use basic polynomial time algorithms, example: weighted matching in bipartied graphs, shortest path, minimum spanning tree.
- Neighborhood defined by finding a minimum cost matching on a (bipartite) improvement graph

Example (TSP)
Neighborhood: Eject k nodes and reinsert them optimally

Outline

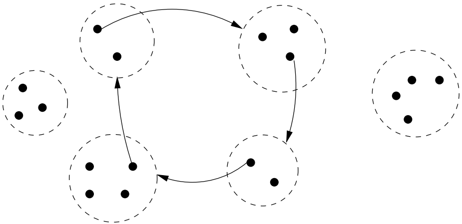
Cyclic Exchange Neighborhoods

- Possible for problems where solution can be represented as form of partitioning
- Definition of a neighborhood search problem in a partitioning problem ($\min_{T_i \in \mathcal{T}} \sum_{i=1}^{k} c(T_i)$):

  Given: a set W of n elements, a collection $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of subsets of W, such that $W = T_1 \cup \ldots \cup T_k$ and $T_i \cap T_j = \emptyset$, and a cost function $c : \mathcal{T} \to \mathbb{R}$:
  Task: Find a partition $\mathcal{T}'$ of W by means of single exchanges between the sets such that

  $$c(\mathcal{T}') < c(\mathcal{T})$$

- Cyclic exchange:

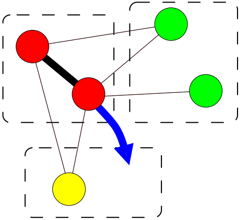


Neighborhood search

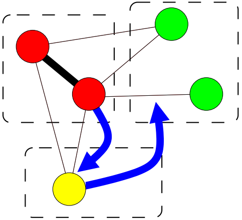
- Define an improvement graph
- Solve the relative
  - Subset Disjoint *Negative* Cost Cycle Problem
  - Subset Disjoint *Minimum* Cost Cycle Problem

Example (GCP)

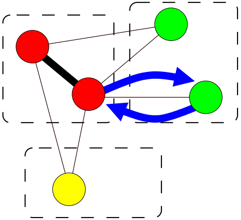
Neighborhood Structures: Very Large Scale Neighborhood



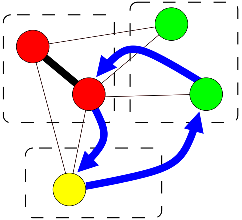
One Exchange



Path Exchange



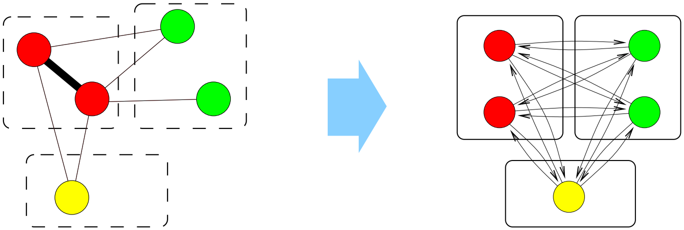
Swap



Cyclic Exchange

Example (GCP)
Examination of the Very Large Scale Neighborhood

Exponential size but can be searched efficiently



Improvement Graph

A Subset Disjoint Negative Cost Cycle Problem in the Improvement Graph can be solved by dynamic programming in $\mathcal{O}(|V|^2 2^k |D'|)$.
Yet, heuristic rules can be adopted to reduce the complexity to $\mathcal{O}(|V'|^2)$

```
Procedure SDNCC(G'(V', D'))
Let 𝒫 all negative cost paths of length 1, Mark all paths in 𝒫 as untreated
Initialize the best cycle q* = () and c* = 0
for all p ∈ 𝒫 do
    if (e(p), s(p)) ∈ D' and c(p) + c(e(p), s(p)) < c* then
        q* = the cycle obtained by closing p and c* = c(q*)
while 𝒫 ≠ ∅ do
    Let 𝒫̂ = 𝒫 be the set of untreated paths
    𝒫 = ∅
    while ∃ p ∈ 𝒫̂ untreated do
        Select some untreated path p ∈ 𝒫̂ and mark it as treated
        for all (e(p), j) ∈ D' s.t. w_{φ(v_j)}(p) = 0 and c(p) + c(e(p), j) < 0 do
            Add the extended path (s(p), ..., e(p), j) to 𝒫 as untreated
            if (j, s(p)) ∈ D' and c(p) + c(e(p), j) + c(j, s(p)) < c* then
                q* = the cycle obtained closing the path (s(p), ..., e(p), j)
                c* = c(q*)
    for all p' ∈ 𝒫 subject to w(p') = w(p), s(p') = s(p), e(p') = e(p) do
        Remove from 𝒫 the path of higher cost between p and p'
return a minimal negative cost cycle q* of cost c*
```