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
- allows 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

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 (TSP, GP)
  - 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

⇒ Idea: turn a special case into a neighborhood
VLSN allows to use the literature on polynomial time algorithms

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 $s$

\[
\hat{t} := s
\]

while $s$ is not locally optimal do

repeat

- select best feasible neighbor $t$
  - if $g(t) < g(\hat{t})$ then $\hat{t} := t$
  - $s := \hat{t}$

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
  - 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
- $\delta$-path: Hamiltonian path $p + 1$ edge connecting one end of $p$ to interior node of $p$
Basic LK exchange step:

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

\[
\begin{array}{c}
\text{u} \\
\text{---------------} \\
\text{v}
\end{array}
\]

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

\[
\begin{array}{c}
\text{u} \\
\text{---------------} \\
\text{w} \\
\text{---------------} \\
\text{v}
\end{array}
\]

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

\[
\begin{array}{c}
\text{u} \\
\text{---------------} \\
\text{w} \\
\text{---------------} \\
\text{v'} \\
\text{---------------} \\
\text{v}
\end{array}
\]

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

\[
\begin{array}{c}
\text{u} \\
\text{---------------} \\
\text{w} \\
\text{---------------} \\
\text{v'} \\
\text{---------------} \\
\text{v}
\end{array}
\]

Construction of complex LK steps:

1. start with current candidate solution (Hamiltonian cycle) \( s \);
   set \( t^* := s \);
   set \( p := s \)
2. obtain \( \delta \)-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 \( \delta \)-paths and Hamiltonian cycles.

Additional 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)

Elements for an efficient neighborhood search

- fast delta evaluations
- neighborhood pruning: fixed radius nearest neighborhood search
- neighborhood lists: restrict exchanges to most interesting candidates
- don’t look bits: focus perturbative search to “interesting” part
- sophisticated data structures for fast updates

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

Static data structures:
- priority lists
- k-d trees

Tour representation. Operations needed:
- reverse \((a, b)\)
- \(\text{succ}(a)\)
- \(\text{prec}(a)\)
- sequence \((a, b, c)\) – check whether \(b\) is within \(a\) and \(b\)

Possible choices (dynamic data structure):
- \(|V| < 1.000\) arries \(\pi\) and \(\pi^{-1}\)
- \(|V| < 1.000.000\) two level tree
- \(|V| > 1.000.000\) splay tree

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 \(c_4\).

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 w.r.t. effect on evaluation function and feasibility of candidate solutions.

Example: Independent 2-exchange steps for the TSP:

\[
\begin{array}{cccccccc}
  u_1 & u_i & u_{i+1} & \cdots & u_j & u_{j+1} & u_k & u_{k+1} & \cdots & u_n & u_{n+1} \\
\end{array}
\]

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.

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 (non-)bipartite improvement graph

Example (TSP)
Neighborhood: Eject \(k\) nodes and reinsert them optimally
Cyclic Exchange Neighborhoods

- Possible for problems where solution can be represented as form of partitioning
- Definition of a partitioning problem:
  Given: a set \( W \) of \( n \) elements, a collection \( T = \{ T_1, T_2, \ldots, T_k \} \) 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 \( c : T \rightarrow \mathbb{R} \):
  Task: Find another partition \( T' \) of \( W \) by means of single exchanges between the sets such that
  \[
  \min \sum_{i=1}^{k} c(T_i)
  \]
- 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)

Exponential size but can be searched efficiently

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

if $(e(p), s(p)) \in D'$ and $c(p) + c(e(p), s(p)) < c^*$ then

$q^* = \text{the cycle obtained by closing } p \text{ and } c^* = c(q^*)$

while $P \neq \emptyset$ do

Let $\hat{P}$ be the set of untreated paths $P = \emptyset$

while $\exists p \in \hat{P}$ untreated do

Select some untreated path $p \in \hat{P}$ and mark it as treated

for all $(e(p), j) \in D'$ s.t. $\psi (v_j(p) = 0$ and $c(p) + c(e(p), j) < 0$ do

Add the extended path $\{s(p), \ldots, e(p), j\}$ to $P$ as untreated

if $(j, s(p)) \in D'$ and $c(p) + c(e(p), j) + c(j, s(p)) < c^*$ then

$q^* = \text{the cycle obtained closing the path } \{s(p), \ldots, e(p), j\}$

$c^* = c(q^*)$

for all $p' \in P$ 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 $c^*$

---

**Example (GCP)**

Cyclic exchanges

- negative cost cycles can be detected rather easily thanks to Lin-Kernighan Lemma

If a sequence of edge costs has negative sum, then there is a cyclic permutation of these edges such that every partial sum is negative.

Path exchanges

- dynamic programming algorithm requires modification to also check for path exchanges (easy)
- require a correction term due to the definition of the improvement graph
- unfortunately, the above lemma is not anymore applicable if we require to find all path exchanges.

---

**Iterative Improvement**

Very Large Scale Neighborhood, effectiveness

<table>
<thead>
<tr>
<th>Num. vertices</th>
<th>Num. distinct colorings</th>
<th>Path and cyclic exchanges</th>
<th>One exchange</th>
<th>Path and cyclic exchanges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>7 (2)</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>63 (6)</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>756 (21)</td>
<td></td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>14113 (112)</td>
<td></td>
<td>83</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>421555 (853)</td>
<td></td>
<td>532</td>
<td>260</td>
</tr>
<tr>
<td>8</td>
<td>22965511 (11117)</td>
<td></td>
<td>348</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>2461096985 (261080)</td>
<td></td>
<td>134</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>134</td>
<td>54</td>
</tr>
</tbody>
</table>
Variable Neighborhood Search (VNS)

Variable Neighborhood Search is an SLS method that is based on the systematic change of the neighborhood during the search.

Central observations

- a local minimum w.r.t. one neighborhood function is not necessarily locally minimal w.r.t. another neighborhood function
- a global optimum is locally optimal w.r.t. all neighborhood functions

Principle: change the neighborhood during the search

Several adaptations of this central principle

- (Basic) Variable Neighborhood Descent (VND)
- Variable Neighborhood Search (VNS)
- Reduced Variable Neighborhood Search (RVNS)
- Variable Neighborhood Decomposition Search (VNDS)
- Skewed Variable Neighborhood Search (SVNS)

Notation

- $N_k, k = 1, 2, \ldots, k_m$ is a set of neighborhood functions
- $N_k(s)$ is the set of solutions in the $k$-th neighborhood of $s$

How to generate the various neighborhood functions?

- for many problems different neighborhood functions (local searches) exist / are in use
- change parameters of existing local search algorithms
- use k-exchange neighborhoods; these can be naturally extended
- many neighborhood functions are associated with distance measures; in this case increase the distance

Basic Variable Neighborhood Descent (BVND)

Procedure VND

input: $N_k, k = 1, 2, \ldots, k_{\text{max}}$, and an initial solution $s$

output: a local optimum $s$ for $N_k, k = 1, 2, \ldots, k_{\text{max}}$

$k \leftarrow 1$

repeat

$s' \leftarrow \text{FindBestNeighbor}(s, N_k)$

if $g(s') < g(s)$ then

$s \leftarrow s'$

$k \leftarrow 1$

else

$k \leftarrow k + 1$

until $k = k_{\text{max}}$
Variable Neighborhood Descent (VND)

Procedure VND
input : $N_k$, $k = 1, 2, \ldots, k_{\text{max}}$, and an initial solution $s$
output: a local optimum $s$ for $N_k$, $k = 1, 2, \ldots, k_{\text{max}}$

$k \leftarrow 1$
repeat
$s' \leftarrow \text{IterativeImprovement}(s, N_k)$
if $g(s') < g(s)$ then
    $s \leftarrow s'$
else
    $k \leftarrow k + 1$
until $k = k_{\text{max}}$

Final solution is locally optimal w.r.t. all neighborhoods
First improvement may be applied instead of best improvement
Typically, order neighborhoods from smallest to largest
If iterative improvement algorithms II$_k$, $k = 1, \ldots, k_{\text{max}}$ are available as black-box procedures:
- order black-boxes
- apply them in the given order
- possibly iterate starting from the first one
- order chosen by: solution quality and speed

Example
VND for single-machine total weighted tardiness problem
- Candidate solutions are permutations of job indexes
- Two neighborhoods: swap and insert
- Influence of different starting heuristics also considered

<table>
<thead>
<tr>
<th>initial solution</th>
<th>swap</th>
<th>insert</th>
<th>swap+insert</th>
<th>insert+swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta_{\text{avg}}$</td>
<td>$\text{tavg}$</td>
<td>$\Delta_{\text{avg}}$</td>
<td>$\text{tavg}$</td>
<td>$\Delta_{\text{avg}}$</td>
</tr>
<tr>
<td>EDD</td>
<td>0.62</td>
<td>0.140</td>
<td>1.19</td>
<td>0.64</td>
</tr>
<tr>
<td>MDD</td>
<td>0.65</td>
<td>0.078</td>
<td>1.31</td>
<td>0.77</td>
</tr>
</tbody>
</table>

$\Delta_{\text{avg}}$ deviation from best-known solutions, averaged over 100 instances

Basic Variable Neighborhood Search (VNS)

Procedure BVNS
input : $N_k$, $k = 1, 2, \ldots, k_{\text{max}}$, and an initial solution $s$
output: a local optimum $s$ for $N_k$, $k = 1, 2, \ldots, k_{\text{max}}$

$k \leftarrow 1$
repeat
$s' \leftarrow \text{RandomPicking}(s, N_k)$
$s'' \leftarrow \text{IterativeImprovement}(s', N_k)$
if $g(s'') < g(s)$ then
    $s \leftarrow s''$
    $k \leftarrow 1$
else
    $k \leftarrow k + 1$
until $k = k_{\text{max}}$
until Termination Condition
To decide:
▶ which neighborhoods
▶ how many
▶ which order
▶ which change strategy
▶ Extended version: parameters \( k_{\text{min}} \) and \( k_{\text{step}} \); set \( k \leftarrow k_{\text{min}} \) and increase by \( k_{\text{step}} \) if no better solution is found (achieves diversification)

---

**Extensions (1)**

**Reduced Variable Neighborhood Search (RVNS)**
▶ same as VNS except that no IterativeImprovement procedure is applied
▶ only explores different neighborhoods randomly
▶ can be faster than standard local search algorithms for reaching good quality solutions

---

**Variable Neighborhood Decomposition Search (VNDS)**
▶ same as in VNS but in IterativeImprovement all solution components are kept fixed except \( k \) randomly chosen
▶ IterativeImprovement is applied on the \( k \) unfixed components

---

**Extensions (2)**

**Skewed Variable Neighborhood Search (SVNS)**
▶ Derived from VNS
▶ Accept \( s \leftarrow s'' \) when \( s'' \) is worse
  ▶ according to some probability
  ▶ skewed VNS: accept if
  \[
  g(s'') - \alpha \cdot d(s, s'') < g(s)
  \]
  \( d(s, s'') \) measure the distance between solutions
  (underlying idea: avoiding degeneration to multi-start)

---

**Extensions (3)**