Variable Neighbourhood Descent

- *recall:* Local minima are relative to neighbourhood relation.
- **key idea:** To escape from local minimum of given neighbourhood relation, switch to different neighbhourhood relation.
- use k neighbourhood relations $N_1, \ldots, N_k$, (typically) ordered according to increasing neighbourhood size.
- always use smallest neighbourhood that facilitates improving steps.
- upon termination, candidate solution is locally optimal w.r.t. all neighbourhoods

Variable Neighbourhood Descent (VND):

determine initial candidate solution s
$i := 1$
Repeat:
  choose a most improving neighbour s' of s in N_i
  If $g(s') < g(s)$:
    $s := s'$
    $i := 1$
  Else:
    $i := i + 1$
Until $i > k$

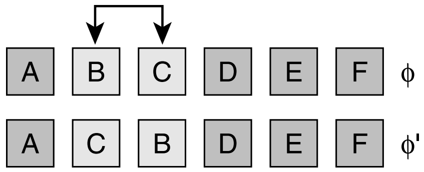
piped VND

- different iterative improvement algorithms $II_1 \dots II_k$ available
- **key idea:** build a chain of iterative improvement algorithms
- different orders of algorithms often reasonable, typically same as would be done in standard VND
- substantial performance improvements possible without modifying code of existing iterative improvement algorithms

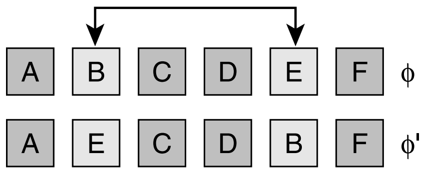
piped VND for single-machine total weighted tardiness problem (SMTWTP)

- **given**:
  - single machine, continuously available
  - n jobs, for each job j is given its processing time p_j, its due date d_j and its importance w_j
- lateness $L_j = C_j - d_j$, $\quad C_j$: completion time of job j
- tardiness $T_j = \max\{L_j, 0\}$
- **goal**:
  - minimise the sum of the weighted tardinesses of all jobs
- SMTWTP $\mathcal{NP}$-hard.
- candidate solutions are permutations of job indices

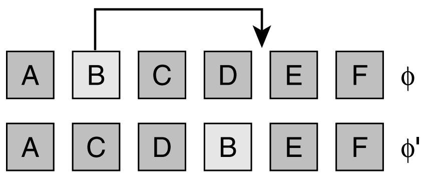
Neighbourhoods for SMTWTP



transpose neighbourhood



exchange neighbourhood



insert neighbourhood

SMTWTP example:

computational results for three different starting solutions

Δ_{avg}: deviation from best-known solutions, averaged over 125 instances
t_{avg}: average computation time on a Pentium II 266MHz

initial	exchange		insert		exchange+insert		insert+exchange	
solution	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}
EDD	0.62	0.140	1.19	0.64	*0.24*	0.20	0.47	0.67
MDD	0.65	0.078	1.31	0.77	*0.40*	0.14	0.44	0.79
AU	0.92	0.040	0.56	0.26	0.59	0.10	*0.21*	0.27

Note:

- VND often performs substantially better than simple II or II in large neighbourhoods [Hansen and Mladenović, 1999]
- many variants exist that switch between neighbhourhoods in different ways.
- more general framework for SLS algorithms that switch between multiple neighbourhoods: *Variable Neighbourhood Search (VNS)* [Mladenović and Hansen, 1997].

Very large scale neighborhood search (VLSN)

- VLSN algorithms are iterative improvement algorithms that make use of very large neighborhoods, often exponentially-sized ones
- very large scale neighborhoods require efficient neighborhood search algorithms, which is facilitated through special-purpose neighborhood structures
- two main classes
  - explore heuristically very large scale neighborhoods
    example: *variable depth search*
  - define special neighborhood structures that allow for efficient search (often in polynomial time)
    example: *Dynasearch*

Variable Depth Search

- **Key idea:** *Complex steps* in large neighbourhoods = variable-length sequences of *simple steps* in small neighbourhood.
- the number of solution components that is exchanged in the complex step is variable and changes from one complex step to another.
- 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.

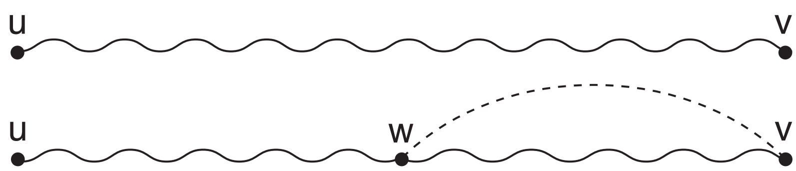
Variable Depth Search (VDS):

determine initial candidate solution s
$\hat{t} := s$
While s is not locally optimal:
  Repeat:
    select best feasible neighbour t
    If $g(t) < g(\hat{t})$: $\hat{t} := t$
  Until construction of complex step has been completed
  $s := \hat{t}$

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

- Complex search steps correspond to sequences of 1-exchange steps and are constructed from sequences of *Hamiltonian paths*

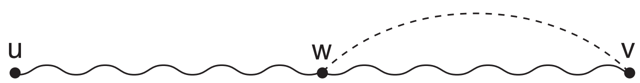
- *δ-path:* Hamiltonian path p + 1 edge connecting one end of p to interior node of p ('lasso' structure):



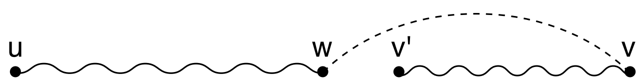
Basic LK exchange step:

- 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 $t^* := t$; $p := p'$
5. if termination criteria of LK step construction not met, go to step 2
6. accept t^* as new current candidate solution s if $w(t^*) < w(s)$

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

Additional mechanisms used by LK algorithm:

- *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 neighbourhood

Lin-Kernighan (LK) Algorithm for the TSP

- k–exchange neighbours with $k > 3$ can reach better solution quality, but require significantly increased computation times
- LK constructs complex search steps by iteratively concatenating 2-exchange steps
- in each complex step, a set of edges $X = \{x_1, \ldots x_r\}$ is deleted from a current tour p and replaced by a set of edges $Y = \{y_1, \ldots y_r\}$ to form a new tour p'
- the number of edges that are exchanged in the complex step is variable and changes from one complex step to another
- termination of the construction process is guaranteed through a gain criterion and additional conditions on the simple moves

Construction of complex step

- the two sets X and Y are constructed iteratively
- edges x_i and y_i as well as y_i and x_{i+1} need to share an endpoint; this results in *sequential moves*
- at any point during the construction process, there needs to be an alternative edge y_i' such that complex step defined by $X = \{x_1, \ldots x_i\}$ and $Y = \{y_1, \ldots y_i'\}$ yields a valid tour

Gain criterion

- at each step compute length of tour defined through $X = \{x_1, \dots x_i\}$ and $Y = \{y_1, \dots y_i'\}$
- also compute gain $g_i := \sum_{j=1}^{i}(w(y_j) - w(x_j))$ for $X = \{x_1, \dots x_i\}$ and $Y = \{y_1, \dots y_i\}$
- terminate construction if $w(p) - g_i < w(p_{i*})$, where p is current tour and p_{i*} best tour found during construction
- p_{i*} becomes new tour if $w(p_{i*}) < w(p)$

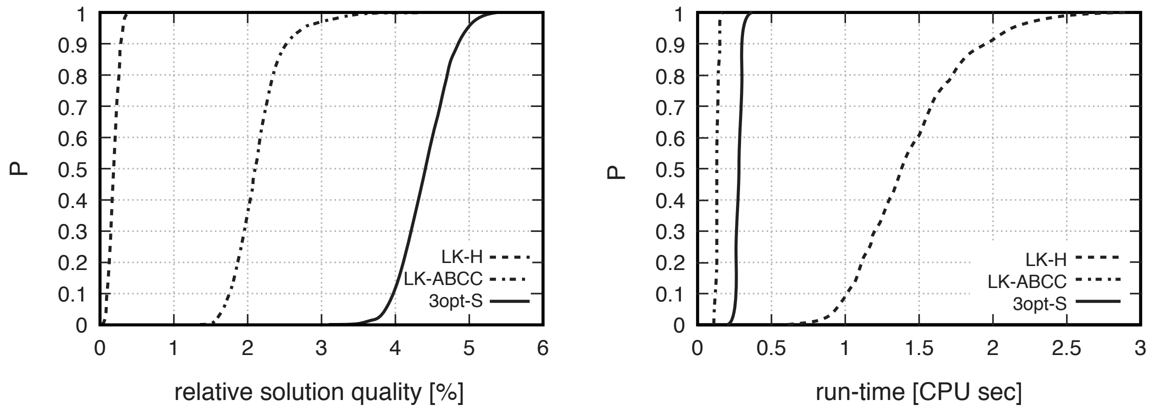
Search guidance in LK

- search for improving move starts with selecting a vertex u_1
- the sets X and Y are required to be disjoint and, hence, *bounds the depth of moves to n*
- 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 for x_1, x_2, y_1, y_2)
  - consider alternative edges in order of increasing weight $w(y_i)$
  - at last backtrack level consider alternative starting nodes u_1
  - backtracking ensures that final tours are at least 2–opt and 3–opt
- some few additional cases receive special treatment
- important are techniques for pruning the search

Variants of LK

- details of LK implementations can vary in many details
  - depth of backtracking
  - width of backtracking
  - rules for guiding the search
  - bounds on length of complex LK steps
  - type and length of candidate lists
  - search initialisation
- essential for good performance on large TSP instances are fine-tuned data structures
- wide range of performance trade-offs of available implementations (Helsgaun's LK, Neto's LK, LK implementation in concorde)
- noteworthy advancement through Helsgaun's LK

Solution Quality distributions for LK-H, LK-ABCC, and 3-opt on TSPLIB instance pcb3038:



Example:

Computational results for LK-ABCC, LK-H, and 3-opt

averages across 1 000 trials; times in ms on Athlon 1.2 GHz CPU, 1 GB RAM

	LK-ABCC		LK-H		3-opt-fr + cl	
Instance	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}
`rat783`	1.85	21.0	0.04	61.8	3.7	34.6
`pcb1173`	2.25	45.3	0.24	238.3	4.6	66.5
`d1291`	5.11	63.0	0.62	444.4	4.9	76.4
`fl1577`	9.95	114.1	5.30	1 513.6	22.4	93.4
`pr2392`	2.39	84.9	0.19	1 080.7	4.5	188.7
`pcb3038`	2.14	134.3	0.19	1 437.9	4.4	277.7
`fnl4461`	1.74	239.3	0.09	1 442.2	3.7	811.6
`pla7397`	4.05	625.8	0.40	8 468.6	6.0	2 260.6
`rl11849`	6.00	1 072.3	0.38	9 681.9	4.6	8 628.6
`usa13509`	3.23	1 299.5	0.19	13 041.9	4.4	7 807.5

Note:

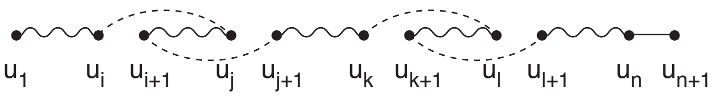
Variable depth search algorithms have been very successful for other problems, including:

- the Graph Partitioning Problem [Kernigan and Lin, 1970];
- the Unconstrained Binary Quadratic Programming Problem [Merz and Freisleben, 2002];
- the Generalised Assignment Problem [Yagiura *et al.*, 1999].

Dynasearch (1)

- Iterative improvement method based on building complex search steps from combinations of simple search steps.
- Simple search steps constituting any given complex step are required to be *mutually independent*, *i.e.*, 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:

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

Dynasearch (2)

- **Key idea:** Efficiently find optimal combination of mutually independent simple search steps using *Dynamic Programming*.
- Successful applications to various combinatorial optimisation problems, including:
  - the TSP and the Linear Ordering Problem [Congram, 2000]
  - the Single Machine Total Weighted Tardiness Problem (scheduling) [Congram *et al.*, 2002]

The methods we have seen so far are iterative ***improvement*** *methods, that is, they get stuck in local optima.*

Simple mechanisms for escaping from local optima:

- *Restart:* re-initialise search whenever a local optimum is encountered.
- *Non-improving steps*: in local optima, allow selection of candidate solutions with equal or worse evaluation function value, *e.g.*, using minimally worsening steps.

Note: Neither of these mechanisms is guaranteed to always escape effectively from local optima.

Diversification *vs* Intensification

- Goal-directed and randomised components of SLS strategy need to be balanced carefully.
- *Intensification*: aims to greedily increase solution quality or probability, *e.g.*, by exploiting the evaluation function.
- *Diversification*: aim to prevent search stagnation by preventing search process from getting trapped in confined regions.

Examples:

- Iterative Improvement (II): *intensification* strategy.
- Uninformed Random Walk (URW): *diversification* strategy.

Balanced combination of intensification and diversification mechanisms forms the basis for advanced SLS methods.

‘Simple’ SLS Methods

Goal:

Effectively escape from local minima of given evaluation function.

General approach:

For fixed neighbourhood, use step function that permits *worsening search steps*.

Specific methods:

- Randomised Iterative Improvement
- Probabilistic Iterative Improvement
- Simulated Annealing
- Tabu Search
- Dynamic Local Search

Randomised Iterative Improvement

Key idea: In each search step, with a fixed probability perform an uninformed random walk step instead of an iterative improvement step.

Randomised Iterative Improvement (RII):

determine initial candidate solution s
While termination condition is not satisfied:
  With probability *wp*:
    choose a neighbour s' of s uniformly at random
  Otherwise:
    choose a neighbour s' of s such that $g(s') < g(s)$ or,
     if no such s' exists, choose s' such that $g(s')$ is minimal
  $s := s'$

Note:

- No need to terminate search when local minimum is encountered

  Instead: Bound number of search steps or CPU time from beginning of search or after last improvement.

- Probabilistic mechanism permits arbitrary long sequences of random walk steps

  Therefore: When run sufficiently long, RII is guaranteed to find (optimal) solution to any problem instance with arbitrarily high probability.

- A variant of RII has successfully been applied to SAT (GWSAT algorithm), but generally, RII is often outperformed by more complex SLS methods.

Example: Randomised Iterative Best Improvement for SAT

procedure *GUWSAT*(F, wp, $maxSteps$)
  input: *propositional formula* F, *probability* wp, *integer* $maxSteps$
  output: *model of* F **or** $\emptyset$
  choose assignment a of truth values to all variables in F
    uniformly at random;
  $steps$:= 0;
  while not(a satisfies F) **and** ($steps < maxSteps$) **do**
    with probability wp **do**
      select x uniformly at random from set of all variables in F;
    otherwise
      select x uniformly at random from $\{x' \mid x'$ is a variable in F and
        changing value of x' in a max. decreases number of unsat. clauses$\}$;
    change value of x in a;
    $steps$:= $steps$+1;
  end
  if a satisfies F **then return** a
  else return $\emptyset$
  end
end *GUWSAT*

Note:

- A variant of GUWSAT, GWSAT [Selman et al., 1994], was at some point state-of-the-art for SAT
- Generally, RII is often outperformed by more complex SLS methods
- Very easy to implement
- Very few parameters

Probabilistic Iterative Improvement

Key idea: Accept worsening steps with probability that depends on respective deterioration in evaluation function value: bigger deterioration $\cong$ smaller probability

Realisation:

- Function $p(g, s)$: determines probability distribution over neighbours of s based on their values under evaluation function g.
- Let $step(s)(s') := p(g, s)(s')$.

Note:

- Behaviour of PII crucially depends on choice of p.
- II and RII are special cases of PII.

Example: Metropolis PII for the TSP (1)

- **Search space:** set of all Hamiltonian cycles in given graph G.
- **Solution set:** same as search space (*i.e.*, all candidate solutions are considered feasible).
- **Neighbourhood relation:** reflexive variant of 2-exchange neighbourhood relation (includes s in $N(s)$, *i.e.*, allows for steps that do not change search position).

Example: Metropolis PII for the TSP (2)

- **Initialisation:** pick Hamiltonian cycle uniformly at random.
- **Step function:** implemented as 2-stage process:
  1. select neighbour $s' \in N(s)$ uniformly at random;
  2. accept as new search position with probability:

  $$p(T, s, s') := \begin{cases} 1 & \text{if } f(s') \leq f(s) \\ \exp(\frac{f(s)-f(s')}{T}) & \text{otherwise} \end{cases}$$

  (*Metropolis condition*), where *temperature* parameter T controls likelihood of accepting worsening steps.
- **Termination:** upon exceeding given bound on run-time.