HEURISTIC OPTIMIZATION

SLS Methods: An Overview

adapted from slides for SLS:FA, Chapter 2

Outline

1. Constructive Heuristics (Revisited)
2. Iterative Improvement (Revisited)
3. 'Simple' SLS Methods
4. Hybrid SLS Methods
5. Population-based SLS Methods

Constructive Heuristics (Revisited)

Constructive heuristics

- search space = partial candidate solutions
- search step = extension with one or more solution components

Constructive Heuristic (CH):

$s = \emptyset$
While s is not a complete solution do
  choose a solution component c
  $s := s + c$

Greedy construction heuristics

- rate the quality of solution components by a heuristic function
- choose at each step a best rated solution component
- possible tie-breaking often either randomly; rarely by a second heuristic function
- for some polynomially solvable problems "exact" greedy heuristics exist, e.g. Kruskal's algorithm for spanning trees
- static vs. adaptive greedy information in constructive heuristics
  - static: greedy values independent of partial solution
  - adaptive: greedy values depend on partial solution

Example: set covering problem

- **given**:
  - $A = \{a_1, \ldots, a_m\}$
  - family $F = \{A_1, \ldots, A_n\}$ of subsets $A_i \subseteq A$ that covers A
  - $w : F \mapsto \mathbb{R}^+$, weight function that assigns to each set of F a cost value
- **goal**: find C^* that covers all items of A with minimal total weight
  - *i.e.*, $C^* \in \mathit{argmin}_{C' \in \mathit{Covers}(A,F)} w(C')$
  - $w(C')$ of C' is defined as $\sum_{A' \in C'} w(A')$
- Example
  - $A = \{a, b, c, d, e, f, g\}$
  - $F = \{A_1 = \{a, b, d, g\}, A_2 = \{a, b, c\}, A_3 = \{e, f, g\}, A_4 = \{f, g\}, A_5 = \{d, e\}, A_6 = \{c, d\}\}$
  - $w(A_1) = 6,\ w(A_2) = 3,\ w(A_3) = 5,\ w(A_4) = 4,\ w(A_5) = 5,\ w(A_6) = 4$
  - Heuristics: see lecture

The SCP instance

	a	b	c	d	e	f	g	
A_1	★	★		★			★	6
A_2	★	★	★					3
A_3					★	★	★	5
A_4						★	★	4
A_5				★	★			5
A_6			★	★				4

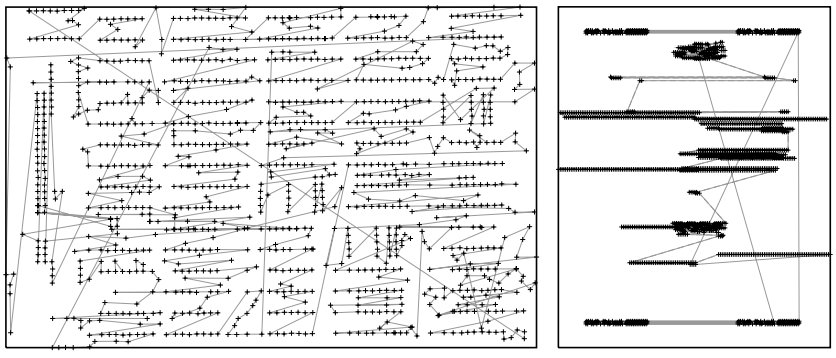
Constructive heuristics for TSP

- 'simple' SLS algorithms that quickly construct reasonably good tours
- are often used to provide an initial search position for more advanced SLS algorithms
- various types of constructive search algorithms exist
  - iteratively extend a connected partial tour
  - iteratively build tour fragments and patch them together into a complete tour
  - algorithms based on minimum spanning trees

Nearest neighbour (NN) construction heuristics:

- start with single vertex (chosen uniformly at random)
- in each step, follow minimal-weight edge to yet unvisited, next vertex
- complete Hamiltonian cycle by adding initial vertex to end of path
- results on length of NN tours
  - for TSP instances with *triangle inequality* NN tour is at most $1/2 \cdot (\lceil \log_2(n) \rceil + 1)$ worse than an optimal one

Two examples of nearest neighbour tours for TSPLIB instances left: `pcb1173`; right: `fl1577`:



- for metric and TSPLIB instances, nearest neighbour tours are typically 20–35% above optimal
- typically, NN tours are locally close to optimal but contain few long edges

Insertion heuristics:

- *insertion heuristics* iteratively extend a partial tour p by inserting a heuristically chosen vertex such that the path length increases minimally
- various heuristics for the choice of the next vertex to insert
  - nearest insertion
  - cheapest insertion
  - farthest insertion
  - random insertion
- nearest and cheapest insertion guarantee approximation ratio of two for TSP instances with *triangle inequality*
- in practice, farthest and random insertion perform better; typically, 13 to 15% above optimal for metric and TSPLIB instances

Greedy, Quick-Borůvka and Savings heuristic:

- *greedy heuristic*
  - first sort edges in graph according to increasing weight
  - scan list and add feasible edges to partial solution
  - complete Hamiltonian cycle by adding initial vertex to end of path
  - greedy tours are at most $(1 + \log n)/2$ longer than optimal for TSP instances with triangle inequality
- *Quick-Borůvka*
  - inspired by minimum spanning tree algorithm of Borůvka, 1926
  - first, sort vertices in arbitrary order
  - for each vertex in this order insert a feasible minimum weight edge
  - two such scans are done to generate a tour

- *savings heuristic*
  - based on savings heuristic for the vehicle routing problem
  - choose a base vertex u_b and $n-1$ cyclic paths (u_b, u_i, u_b)
  - at each step, remove an edge incident to u_b in two path p_1 and p_2 and create a new cyclic path p_{12}
  - edges removed are chosen as to maximise cost reduction
  - savings tours are at most $(1 + \log n)/2$ longer than optimal for TSP instances with triangle inequality

- *empirical results*
  - savings produces better tours than greedy or Quick-Borůvka
  - on RUE instances approx. 12% above optimal (savings), 14% (greedy) and 16% (Quick-Borůvka)
  - computation times are modest ranging from 22 seconds (Quick-Borůvka) to around 100 seconds (Greedy, Savings) for 1 million RUE instances on 500MHz Alpha CPU (see Johnson and McGeoch, 2002)

Construction heuristics based on minimum spanning trees:

- *minimum spanning tree heuristic*
  - compute a minimum spanning tree (MST) t
  - double each edge in t obtaining a graph G'
  - compute an Eulerian tour p in G'
  - convert p into a Hamiltonian cycle by short-cutting subpaths of p
  - for TSP instances with *triangle inequality* the result is at most twice as long as the optimal tour

- *Christofides heuristic*
  - similar to algorithm above but computes a minimum weight perfect matching of the odd–degree vertices of the MST
  - this converts MST into an Eulerian graph, *i.e.*, a graph with an Eulerian tour
  - for TSP instances with *triangle inequality* the result is at most 1.5 times as long as the optimal tour
  - very good performance w.r.t. solution quality if heuristics are used for converting Eulerian tour into a Hamiltonian cycle

Iterative Improvement (Revisited)

Iterative Improvement (II):

determine initial candidate solution s
While s is not a local optimum:
  choose a neighbour s' of s such that $g(s') < g(s)$
  $s := s'$

In II, various mechanisms (*pivoting rules*) can be used for choosing improving neighbour in each step:

- *Best Improvement* (aka *gradient descent*, *greedy hill-climbing*): Choose maximally improving neighbour, *i.e.*, randomly select from $I^*(s) := \{s' \in N(s) \mid g(s') = g^*\}$, where $g^* := \min\{g(s') \mid s' \in N(s)\}$.

  Note: Requires evaluation of all neighbours in each step.

- *First Improvement:* Evaluate neighbours in fixed order, choose first improving step encountered.

  Note: Can be much more efficient than Best Improvement; order of evaluation can have significant impact on performance.


```
procedure iterative best-improvement
    while improvement
        improvement ← false
        for i ← 1 to n do
            for j ← 1 to n do
                CheckMove(i, j);
                if move is new best improvement then
                    (k, l) ← MemorizeMove(i, j);
                    improvement ← true
            endfor
        end
        ApplyBestMove(k, l);
    until (improvement = false)
end iterative best-improvement
```

```
procedure iterative first-improvement
    while improvement
        improvement ← false
        for i ← 1 to n do
            for j ← 1 to n do
                CheckMove(i, j);
                if move improves then
                    ApplyMove(i, j);
                    improvement ← true
            endfor
        end
    until (improvement = false)
end iterative first-improvement
```


Example: Random-order first improvement for the TSP (1)

- **given:** TSP instance G with vertices $v_1, v_2, \ldots, v_n$.
- search space: Hamiltonian cycles in G;
  use standard 2-exchange neighbourhood
- **initialisation:**

  search position := fixed canonical path $(v_1, v_2, \ldots, v_n, v_1)$
  P := random permutation of {1,2, . . . , n}

- **search steps:** determined using first improvement w.r.t. $g(p)$ = weight of path p, evaluating neighbours in order of P (does not change throughout search)
- **termination:** when no improving search step possible (local minimum)

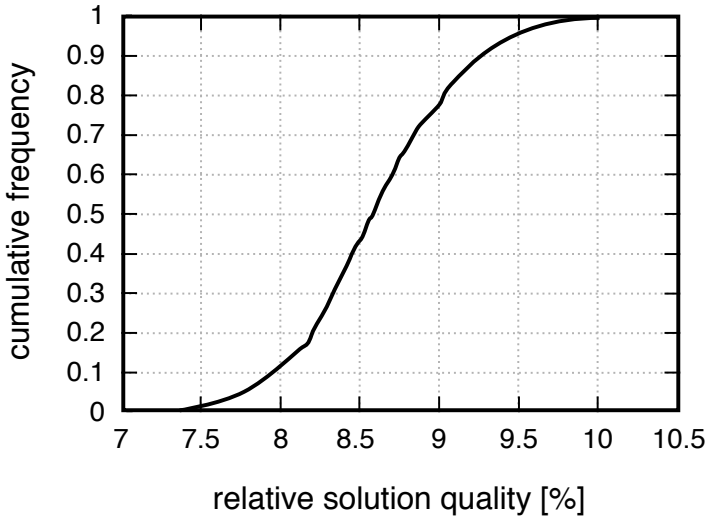
Example: Random-order first improvement for the TSP (2)

Empirical performance evaluation:

- perform 1000 runs of algorithm on benchmark instance pcb3038
- record *relative solution quality* (= percentage deviation from known optimum) of final tour obtained in each run
- plot *cumulative distribution function* of relative solution quality over all runs.

example: Random-order first improvement for the TSP (3)

result: substantial variability in solution quality between runs.



Iterative Improvement (Revisited)

Iterative Improvement (II):

determine initial candidate solution s
While s is not a local optimum:
  choose a neighbour s' of s such that $g(s') < g(s)$
  $s := s'$

Main Problem:
stagnation in local optima of evaluation function g.

Note:

- local minima depend on g and neighbourhood relation, N.
- larger neighbourhoods $N(s)$ induce
  - neighbhourhood graphs with smaller diameter;
  - fewer local minima.

ideal case: *exact neighbourhood*, *i.e.*, neighbourhood relation for which any local optimum is also guaranteed to be a global optimum.

- typically, exact neighbourhoods are too large to be searched effectively (exponential in size of problem instance).
- *but:* exceptions exist, *e.g.*, polynomially searchable neighbourhood in Simplex Algorithm for linear programming.

Trade-off:

- using larger neighbourhoods can improve performance of II (and other SLS methods).
- *but:* time required for determining improving search steps increases with neighbourhood size.

more general trade-off:

effectiveness *vs* time complexity of search steps.

neighbourhood Pruning:

- *idea:* reduce size of neighbourhoods by exluding neighbours that are likely (or guaranteed) not to yield improvements in g.
- *note:* crucial for large neighbourhoods, but can be also very useful for small neighbourhoods (*e.g.*, linear in instance size).

next: example of speed-up techniques for TSP

Observation:

- for any improving 2–exchange move from s to neighbour s', at least one vertex incident to an edge e in s that is replaced by a different edge e' with $w(e') < w(e)$

Speed-up 1: Fixed Radius Search

- for a vertex u_i perform two searches, considering each of its two tour neighours as u_j
- search for a vertex u_k around u_i that is closer than $w((u_i, u_j))$
- for each such vertex examine effect of 2–exchange move and perform first improving move found
- results in large reduction of computation time
- technique is extendable to 3–exchange

Speed-up 2: Candidate lists

- lists of neighbouring vertices sorted according to edge weight
- supports fixed radius near neighbour searches

Construction of candidate lists:

- full candidate lists require $O(n^2)$ memory and $O(n^2 \log n)$ time to construct
- therefore: often *bounded-length* candidate lists
- typical bound: 10 to 40
- *quadrant-nearest neighbour lists* helpful on clustered instances

Observation:

- if no improving k–exchange move was found for vertex v_i, it is unlikely that an improving step will be found in future search steps, unless an edge incident to v_i has changed

Speed-up 3: don't look bits

- associate to each vertex v_i a don't look bit
  - *0*: start search at v_i
  - *1*: don't start search at v_i
- initially, all don't look bits are set to zero
- if search centred at vertex v_i for improving move fails, set don't look bit to one (turn on)
- for all vertices incident to changed edges in a move the don't look bits are set to zero (turned off)
- leads to significant reductions in computation time
- can be integrated into complex SLS methods (ILS, MAs)

- don't look bits can be generalised for applications to other combinatorial problems

```
procedure iterative improvement
    while improvement
        improvement ← false
        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, improvement ← true
            endfor
            if improve_flag = false then dlb[i] ← 1;
        end
    until (improvement = false)
end iterative improvement
```

Example:

computational results for different variants of 2-opt and 3-opt

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

	2-opt-**std**		2-opt-**fr** + **cl**		2-opt-**fr** + **cl** + **dlb**		3-opt-**fr** + **cl**	
Instance	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}
rat783	13.0	93.2	3.9	3.9	8.0	3.3	3.7	34.6
pcb1173	14.5	250.2	8.5	10.8	9.3	7.1	4.6	66.5
d1291	16.8	315.6	10.1	13.0	11.1	7.4	4.9	76.4
fl1577	13.6	528.2	7.9	21.1	9.0	11.1	22.4	93.4
pr2392	15.0	1 421.2	8.8	47.9	10.1	24.9	4.5	188.7
pcb3038	14.7	3 862.4	8.2	73.0	9.4	40.2	4.4	277.7
fnl4461	12.9	19 175.0	6.9	162.2	8.0	87.4	3.7	811.6
pla7397	13.6	80 682.0	7.1	406.7	8.6	194.8	6.0	2 260.6
rl11849	16.2	360 386.0	8.0	1 544.1	9.9	606.6	4.6	8 628.6
usa13509	—		7.4	1 560.1	9.0	787.6	4.4	7 807.5

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 $i > k$

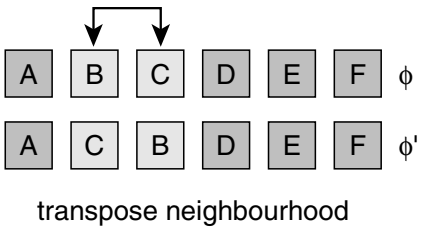
piped VND

- different iterative improvement algorithms $II_1 \ldots II_k$ 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

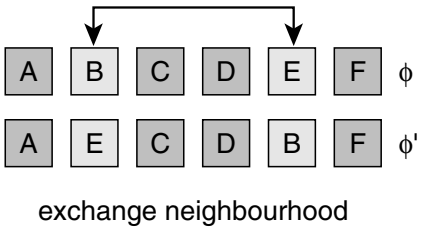
piped VND for single-machine total weighted tardiness problem (SMTWTP)

- **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

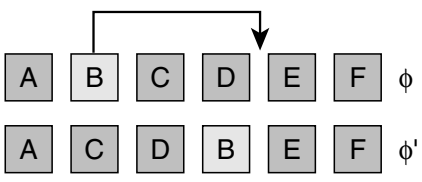
Neighbourhoods for SMTWTP



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]
- several 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*, *cyclic exchange neighbourhoods*

Variable Depth Search

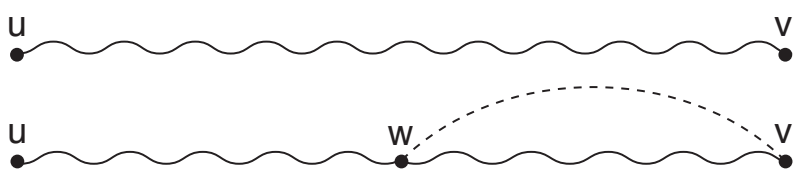
- **Key idea:** *Complex steps* in large neighbourhoods = variable-length sequences of *simple steps* in small neighbourhood.
- the number of solution components that is exchanged in the complex step is variable and changes from one complex step to another.
- Use various *feasibility restrictions* on 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
  if $g(\hat{t}) < g(s)$ then $s := \hat{t}$

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):

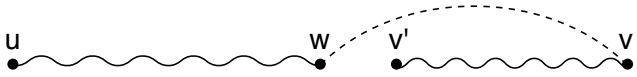


Basic LK exchange step:

- 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 $X = \{x_1, \dots x_i\}$ and $Y = \{y_1, \dots y_i'\}$ 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 $w(p_{i*}) < w(p)$

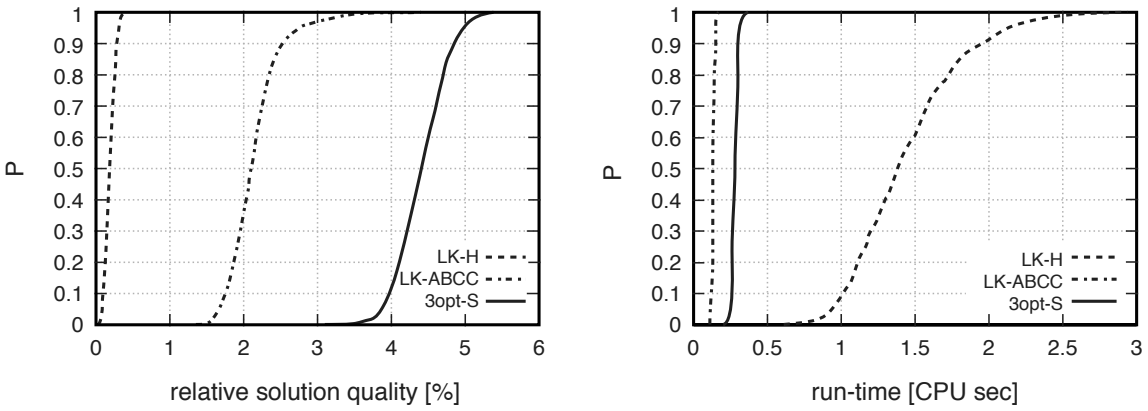
Search guidance in LK

- 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*	
*I*nstance	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:

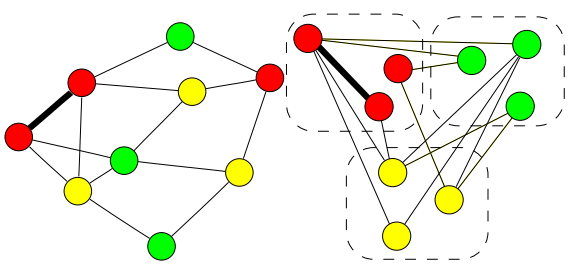
  Therefore: Overall effect of complex search step = sum of effects of constituting simple steps; complex search steps maintain feasibility of candidate solutions.

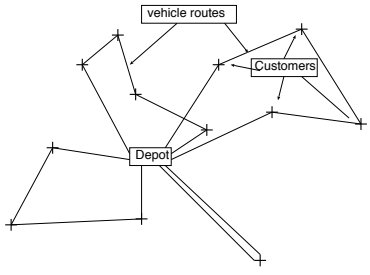
Dynasearch (2)

- **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]

Cyclic exchange neighbourhoods

- In many problems, elements of a set S need to be partitioned into disjunct subsets (that is, partitions) $S_i, i = 1, \ldots, m$
- independent costs for each partition S_i : $g(S_i)$
- total evaluation function value: $\sum_{i=1}^{m} g(S_i)$
- examples
  - graph coloring
  - vehicle routing



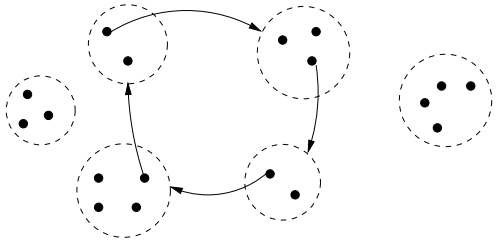


Simple neighbourhoods

- move of a single element into another subset
- exchange of two elements of two different subsets
- often, general exchanges of more than two elements too time consuming

Cyclic exchange neighbourhoods

- cyclic exchange of one element each across different subsets
- neighbourhood size in $O(n^m)$



Approach

- generate a directed *improvement graph*
  - vertices: one for each element of S
  - edges: for each pair $(k, l) \in S$ such that $k \in S_i,\ l \in s_j,\ i \neq j$
  - edge (k, l) indicates that element k is moved from S_i to S_j and l is removed from S_j
  - edge weight corresponds to evaluation function difference in subset S_j, that is, $g((k, l)) = g(S_j \cup \{k\} \setminus \{l\}) - g(S_j)$
- determine a cycle in this graph with negative total weight and such that all vertices belong to different subsets
- such a cycle corresponds to a cyclic exchange that improves the solution
- finding a best such a cycle is itself NP-hard, but efficient heuristics exist
- high-performing method for various problems

Summary VLNS

- very large neighborhoods are specially defined so that they can be searched efficiently and effectively in a heuristic or an exact way
- for several problems crucial to obtain state-of-the-art results
- neighborhoods and efficient neighborhood searches are rather problem specific
- sometimes high implementation effort necessary
- a variety of other techniques exist (large neighborhood search, ejection chains, special purpose neighborhoods etc.)

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.

Simulated Annealing

Key idea: Vary temperature parameter, *i.e.*, probability of accepting worsening moves, in Probabilistic Iterative Improvement according to *annealing schedule* (aka *cooling schedule*).

Inspired by a simulation of the physical annealing process:

- candidate solutions $\cong$ states of physical system
- evaluation function $\cong$ thermodynamic energy
- globally optimal solutions $\cong$ ground states
- parameter $T \cong$ physical temperature

Note: In physical process (*e.g.*, annealing of metals), perfect ground states are achieved by very slow lowering of temperature.

Simulated Annealing (SA):

determine initial candidate solution s
set initial temperature T according to *annealing schedule*
While termination condition is not satisfied:
  probabilistically choose a neighbour s' of s
    using *proposal mechanism*
  If s' satisfies probabilistic *acceptance criterion* (depending on T):
    $s := s'$
  update T according to *annealing schedule*

Note:

- 2-stage step function based on
  - proposal mechanism (often uniform random choice from $N(s)$)
  - acceptance criterion (often *Metropolis condition*)
- Annealing schedule (function mapping run-time t onto temperature $T(t)$):
  - initial temperature T_0
    (may depend on properties of given problem instance)
  - temperature update scheme
    (*e.g.*, geometric cooling: $T := \alpha \cdot T$)
  - number of search steps to be performed at each temperature
    (often multiple of neighbourhood size)
- Termination predicate: often based on *acceptance ratio*, *i.e.*, ratio of proposed *vs* accepted steps.

Example: Simulated Annealing for the TSP

Extension of previous PII algorithm for the TSP, with

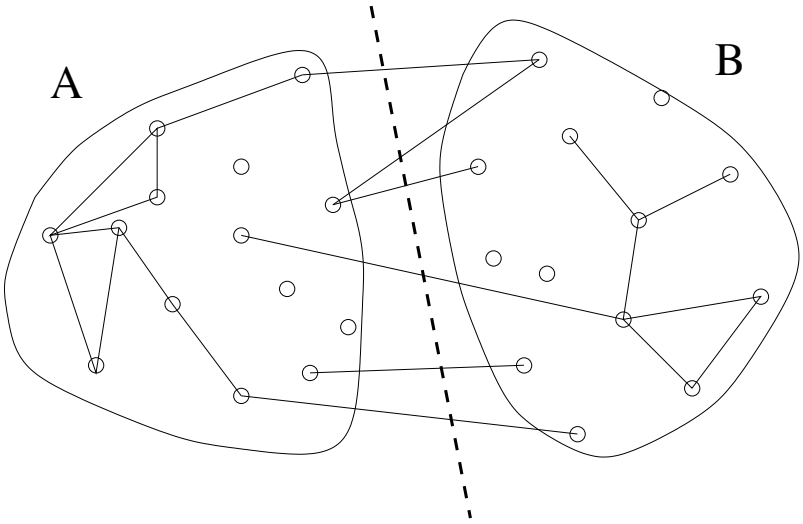
- *proposal mechanism:* uniform random choice from 2-exchange neighbourhood;
- *acceptance criterion:* Metropolis condition (always accept improving steps, accept worsening steps with probability $\exp[(f(s) - f(s'))/T]$);
- *annealing schedule*: geometric cooling $T := 0.95 \cdot T$ with $n \cdot (n-1)$ steps at each temperature ($n =$ number of vertices in given graph), T_0 chosen such that 97% of proposed steps are accepted;
- *termination:* when for five successive temperature values no improvement in solution quality and acceptance ratio $< 2\%$.

Improvements:

- neighbourhood pruning (*e.g.*, candidate lists for TSP)
- greedy initialisation (*e.g.*, by using NNH for the TSP)
- *low temperature starts* (to prevent good initial candidate solutions from being too easily destroyed by worsening steps)
- *look-up tables* for acceptance probabilities:
  instead of computing exponential function $\exp(\Delta/T)$ for each step with $\Delta := f(s) - f(s')$ (expensive!), use precomputed table for range of argument values Δ/T.

Example: Simulated Annealing for the graph bipartitioning

- for a given graph $G := (V, E)$, find a partition of the nodes in two sets V_1 and V_2 such that $|V_1| = |V_2|$, $V_1 \cup V_2 = V$, and that the number of edges with vertices in each of the two sets is minimal



SA example: graph bipartitioning Johnson et al. 1989

- tests were run on random graphs ($G_{n,p}$) and random geometric graphs $U_{n,d}$
- modified cost function (α: imbalance factor)

$$f(V_1, V_2) = |\{(u, v) \in E \mid u \in V_1 \wedge v \in V_2\}| + \alpha(|V_1| - |V_2|)^2$$

$\rightsquigarrow$ allows infeasible solutions but punishes the amount of infeasibility

- *side advantage*: allows to use 1–exchange neighborhoods of size $\mathcal{O}(n)$ instead of the typical neighborhood that exchanges two nodes at a time and is of size $\mathcal{O}(n^2)$

SA example: graph bipartitioning Johnson et al. 1989

- initial solution is chosen randomly
- standard geometric cooling schedule
- experimental comparison to Kernighan–Lin heuristic
  - Simulated Annealing gave better performance on $G_{n,p}$ graphs
  - just the opposite is true for $U_{n,d}$ graphs
- several further improvements were proposed and tested

general remark: Although relatively old, Johnson et al.'s experimental investigations on SA are still worth a detailed reading!

'Convergence' result for SA:

Under certain conditions (extremely slow cooling),
any sufficiently long trajectory of SA is guaranteed to end in
an optimal solution [Geman and Geman, 1984; Hajek, 1988].

Note:

- Practical relevance for combinatorial problem solving
  is very limited (impractical nature of necessary conditions)
- In combinatorial problem solving, *ending* in optimal solution
  is typically unimportant, but *finding* optimal solution
  during the search is (even if it is encountered only once)!

- SA is historically one of the first SLS methods (metaheuristics)
- raised significant interest due to simplicity, good results, and theoretical properties
- rather simple to implement
- on standard benchmark problems (e.g. TSP, SAT) typically outperformed by more advanced methods (see following ones)
- nevertheless, for some (messy) problems sometimes surprisingly effective