Tabu Search

Key idea: Use aspects of search history (memory) to escape from local minima.

Simple Tabu Search:

- Usually extends underlying iterative (best) improvement procedures
- Associate *tabu status* with candidate solutions or solution components.
- Forbid steps to search positions that are tabu

Tabu Search (TS):

determine initial candidate solution s
While *termination criterion* is not satisfied:
  determine set N' of non-tabu neighbours of s
  choose a best improving candidate solution s' in N'
  update tabu attributes based on s'
  $s := s'$

Note:

- Non-tabu search positions in $N(s)$ are called *admissible neighbours of s*.
- After a search step, the current search position or the solution components just added/removed from it are declared *tabu* for a fixed number of subsequent search steps (*tabu tenure*).
- Often, an additional *aspiration criterion* is used: this specifies conditions under which tabu status may be overridden (*e.g.*, if considered step leads to improvement in incumbent solution).

Example: Tabu Search for SAT – GSAT/Tabu (1)

- **Search space:** set of all truth assignments for propositional variables in given CNF formula F.
- **Solution set:** models of F.
- Use 1-flip **neighbourhood relation**, *i.e.*, two truth assignments are neighbours iff they differ in the truth value assigned to one variable.
- **Memory:** Associate tabu status (Boolean value) with each variable in F.

Example: Tabu Search for SAT – GSAT/Tabu (2)

- **Initialisation:** random picking, *i.e.*, select uniformly at random from set of all truth assignments.
- **Search steps:**
  - variables are tabu iff they have been changed in the last tt steps;
  - neighbouring assignments are admissible iff they can be reached by changing the value of a non-tabu variable or have fewer unsatisfied clauses than the best assignment seen so far (*aspiration criterion*);
  - choose uniformly at random admissible assignment with minimal number of unsatisfied clauses.
- **Termination:** upon finding model of F or after given bound on number of search steps has been reached.

Note:

- *GSAT/Tabu* used to be state of the art for SAT solving.
- Crucial for efficient implementation:
  - keep time complexity of search steps minimal by using special data structures, incremental updating and caching mechanism for evaluation function values;
  - efficient determination of tabu status: store for each variable x the number of the search step when its value was last changed it_x; x is tabu iff $it - it_x < tt$, where it = current search step number.

Note: Performance of Tabu Search depends crucially on setting of tabu tenure tt:

- tt too low $\Rightarrow$ search stagnates due to inability to escape from local minima;
- tt too high $\Rightarrow$ search becomes ineffective due to overly restricted search path (admissible neighbourhoods too small)

Advanced TS methods:

- **Robust Tabu Search** [Taillard, 1991]:
  repeatedly choose tt from given interval;
  also: force specific steps that have not been made for a long time.
- **Reactive Tabu Search** [Battiti and Tecchiolli, 1994]:
  dynamically adjust tt during search;
  also: use escape mechanism to overcome stagnation.

Further improvements can be achieved by using *intermediate-term* or *long-term memory* to achieve additional *intensification* or *diversification*.

Examples:

- Occasionally backtrack to *elite candidate solutions*, *i.e.*, high-quality search positions encountered earlier in the search; when doing this, all associated tabu attributes are cleared.
- Freeze certain solution components and keep them fixed for long periods of the search.
- Occasionally force rarely used solution components to be introduced into current candidate solution.
- Extend evaluation function to capture frequency of use of candidate solutions or solution components.

Tabu search algorithms algorithms are state of the art for solving several combinatorial problems, including:

- SAT and MAX-SAT
- the Constraint Satisfaction Problem (CSP)
- several scheduling problems

Crucial factors in many applications:

- choice of neighbourhood relation
- efficient evaluation of candidate solutions (caching and incremental updating mechanisms)

Dynamic Local Search

- **Key Idea:** Modify the evaluation function whenever a local optimum is encountered in such a way that further improvement steps become possible.
- Associate *penalty weights* (*penalties*) with solution components; these determine impact of components on evaluation function value.
- Perform Iterative Improvement; when in local minimum, increase penalties of some solution components until improving steps become available.

Dynamic Local Search (DLS):

determine *initial candidate solution* s
initialise penalties
While *termination criterion* is not satisfied:
  compute *modified evaluation function* g' from g
    based on *penalties*
  perform *subsidiary local search* on s
    using *evaluation function* g'
  update penalties based on s

Dynamic Local Search (continued)

- **Modified evaluation function:**
  $g'(\pi, s) := g(\pi, s) + \sum_{i \in SC(\pi', s)} penalty(i)$, where

  $SC(\pi', s)$ = set of solution components
  of problem instance π' used in candidate solution s.

- **Penalty initialisation:** For all i: $penalty(i) := 0$.

- **Penalty update** in local minimum s: Typically involves *penalty increase* of some or all solution components of s; often also occasional *penalty decrease* or *penalty smoothing*.

- **Subsidiary local search:** Often *Iterative Improvement*.

Potential problem:

Solution components required for (optimal) solution may also be present in many local minima.

Possible solutions:

- **A:** Occasional decreases/smoothing of penalties.
- **B:** Only increase penalties of solution components that are least likely to occur in (optimal) solutions.

Implementation of **B** (Guided local search):

[Voudouris and Tsang, 1995] Only increase penalties of solution components i with maximal utility:

$$util(s', i) := \frac{f_i(\pi, s')}{1 + penalty(i)}$$

where $f_i(\pi, s') =$ solution quality contribution of i in s'.

Example: Guided Local Search (GLS) for the TSP

[Voudouris and Tsang 1995; 1999]

- **Given:** TSP instance G
- Search space: Hamiltonian cycles in G with n vertices; use standard 2-exchange neighbourhood; solution components = edges of G; $f(G, p) := w(p)$; $f_e(G, p) := w(e)$;
- **Penalty initialisation:** Set all edge penalties to zero.
- **Subsidiary local search:** Iterative First Improvement.
- **Penalty update:** Increment penalties for all edges with maximal utility by

$$\lambda := 0.3 \cdot \frac{w(s_{2\text{-}opt})}{n}$$

where $s_{2\text{-}opt}$ = 2-optimal tour.

Related methods:

- Breakout Method [Morris, 1993]
- GENET [Davenport *et al.*, 1994]
- Clause weighting methods for SAT
  [Selman and Kautz, 1993; Cha and Iwama, 1996; Frank, 1997]
- several long-term memory schemes of tabu search

Dynamic local search algorithms are state of the art
for several problems, including:

- SAT, MAX-SAT
- MAX-CLIQUE [Pullan *et al.*, 2006]

Hybrid SLS Methods

Combination of 'simple' SLS methods often yields
substantial performance improvements.

Simple examples:

- Commonly used restart mechanisms can be seen
  as hybridisations with Uninformed Random Picking
- Iterative Improvement + Uninformed Random Walk
  = Randomised Iterative Improvement

Iterated Local Search

Key Idea: Use two types of SLS steps:

- *subsidiary local search* steps for reaching local optima as efficiently as possible (intensification)
- *perturbation steps* for effectively escaping from local optima (diversification).

Also: Use *acceptance criterion* to control diversification *vs* intensification behaviour.

Iterated Local Search (ILS):

determine initial candidate solution s
perform *subsidiary local search* on s
While termination criterion is not satisfied:
  $r := s$
  perform *perturbation* on s
  perform *subsidiary local search* on s
  based on *acceptance criterion*, keep s or revert to $s := r$

Note:

- *Subsidiary local search* results in a local minimum.
- ILS trajectories can be seen as walks in the space of local minima of the given evaluation function.
- *Perturbation phase* and *acceptance criterion* may use aspects of *search history* (*i.e.*, limited memory).
- In a high-performance ILS algorithm, *subsidiary local search*, *perturbation mechanism* and *acceptance criterion* need to complement each other well.

In what follows: A closer look at ILS

ILS — algorithmic outline

procedure *Iterated Local Search*
  $s_0 \leftarrow$ GenerateInitialSolution
  $s^* \leftarrow$ LocalSearch(s_0)
  repeat
    $s' \leftarrow$ Perturbation(s^*, history)
    $s^{*\prime} \leftarrow$ LocalSearch(s')
    $s^* \leftarrow$ AcceptanceCriterion($s^*, s^{*\prime}$, history)
  until *termination condition met*
end

basic version of ILS

- initial solution: random or construction heuristic
- subsidiary local search: often readily available
- perturbation: random moves in higher order neighborhoods
- acceptance criterion: force cost to decrease

such a version of ILS ..

- often leads to very good performance
- only requires few lines of additional code to existing local search algorithm
- state-of-the-art results with further optimizations

basic ILS algorithm for TSP

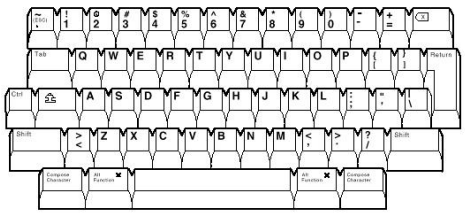
- GenerateInitialSolution: greedy heuristic
- LocalSearch: 2-opt, 3-opt, LK, (whatever available)
- Perturbation: double-bridge move (a specific 4-opt move)
- AcceptanceCriterion: accept $s^{*\prime}$ only if $f(s^{*\prime}) \leq f(s^*)$

basic ILS algorithm for SMTWTP

- GenerateInitialSolution: random initial solution or by EDD heuristic
- LocalSearch: piped VND using local searches based on interchange and insert neighborhoods
- Perturbation: random k-opt move, $k > 2$
- AcceptanceCriterion: accept $s^{*\prime}$ only if $f(s^{*\prime}) \leq f(s^*)$

Quadratic Assignment Problem (QAP)

- given: matrix of inter–location distances; d_{ij}: distance from location i to location j
- given: matrix of flows between objects; f_{rs}: flow from object r to object s



- *objective*: find an assignment (represented as a permutation) of the n objects to the n locations that minimizes

$$\min_{\pi \in \Pi(n)} \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} f_{\pi(i)\pi(j)}$$

  $\pi(i)$ gives object at location i
- interest: among most difficult combinatorial optimization problems for exact methods

basic ILS algorithm for QAP

- GenerateInitialSolution: random initial solution
- LocalSearch: iterative improvement in 2-exchange neighborhood
- Perturbation: random k-opt move, $k > 2$
- AcceptanceCriterion: accept $s^{*\prime}$ only if $f(s^{*\prime}) \leq f(s^*)$

basic ILS algorithm for SAT

- GenerateInitialSolution: random initial solution
- LocalSearch: short tabu search runs based on 1-flip neighborhood
- Perturbation: random k-flip move, $k >> 2$
- AcceptanceCriterion: accept $s^{*\prime}$ only if $f(s^{*\prime}) \leq f(s^*)$

ILS is a modular approach

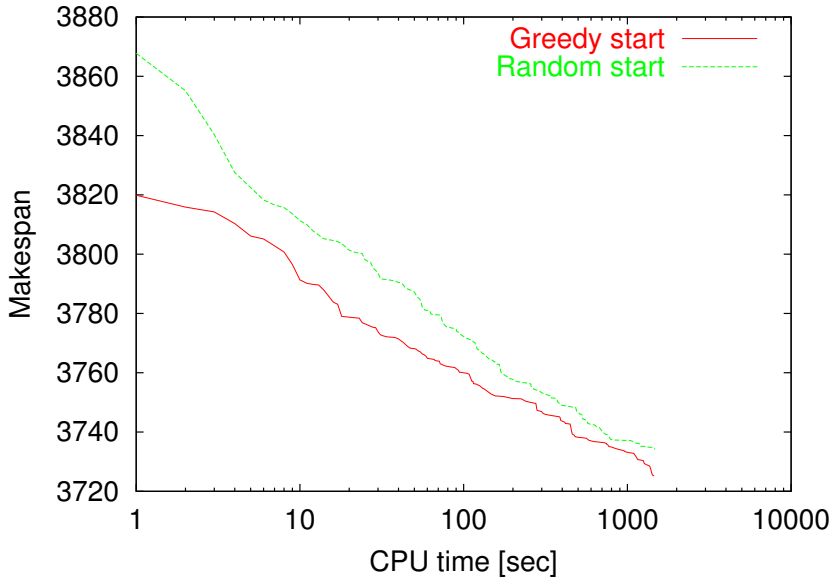
Performance improvement by optimization of modules

- consider different implementation possibilities for modules
- fine-tune modules step-by-step
- optimize single modules without considering interactions among modules
  $\rightsquigarrow$ *local optimization of ILS*

ILS — initial solution

- determines starting point s_0^* of walk in $\mathcal{S}^*$
- random vs. greedy initial solution
- greedy initial solutions appear to be recomendable
- for long runs dependence on s_0^* should be very low

ILS for FSP, initial solution

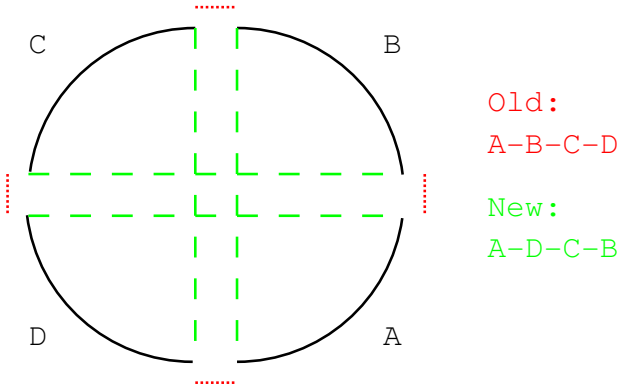


ILS — perturbation

- important: *strength* of perturbation
  - *too strong*: close to random restart
  - *too weak*: LocalSearch may undo perturbation easily
- random perturbations are simplest but not necessarily best
- perturbation should be complementary to LocalSearch

double-bridge move for TSP

- small perturbation good also for very large-size TSP instances
- complementary to most implementations of LK local search
- low cost increase



sometimes large perturbations needed

- example: basic ILS for QAP

  given is average deviation from best-known solutions for different sizes of the perturbation (from 3 to n); averages over 10 trials; 60 seconds on a 500MHz Pentium III.

instance	3	$n/12$	$n/6$	$n/4$	$n/3$	$n/2$	$3n/4$	n
kra30a	2.51	2.51	2.04	1.06	0.83	0.42	*0.0*	0.77
sko64	0.65	1.04	0.50	0.37	*0.29*	*0.29*	0.82	0.93
tai60a	2.31	2.24	1.91	*1.71*	1.86	2.94	3.13	3.18
tai60b	2.44	0.97	0.67	0.96	0.82	0.50	*0.14*	0.43

Adaptive perturbations

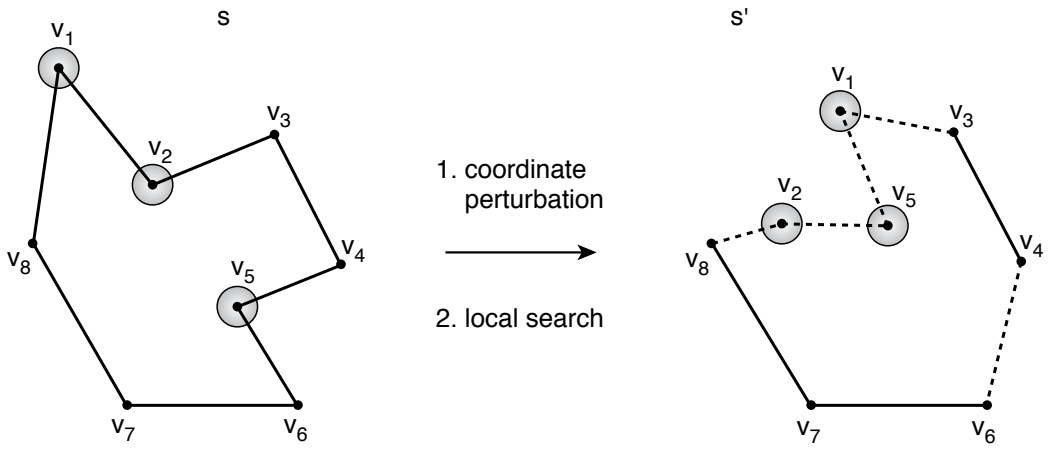
- single perturbation size not necessarily optimal
- perturbation size may vary at run-time;
  done in *basic Variable Neighborhood Search*
- perturbation size may be adapted at run-time;
  leads to *reactive search*

Complex perturbation schemes

- optimizations of subproblems *[Lourenço, 1995]*
- input data modifications
  - modify data definition of instance
  - on modified instance run LocalSearch using input s^*, output is perturbed solution s'

example of a complex perturbation

[Codenotti et al., 1993, 1996]



ILS — speed

- on many problems, small perturbations are sufficient
- LocalSearch in such a case will execute very fast; very few improvement steps
- sometimes access to LocalSearch in combination with Perturbation increases strongly speed (e.g. don't look bits)
- example: TSP

ILS — speed, example

- compare No. local searches of 3-opt in fixed computation time
- $\#_{\text{LS}_{\text{RR}}}$: No. local searches with random restart
- $\#_{\text{LS}_{\text{1-DB}}}$: No. local searches with one double bridge move as Perturbation
- time limit: 120 sec on a Pentium II 266 MHz PC

instance	$\#_{\text{LS}_{\text{RR}}}$	$\#_{\text{LS}_{\text{1-DB}}}$	$\#_{\text{LS}_{\text{1-DB}}}/\#_{\text{LS}_{\text{RR}}}$
kroA100	17507	56186	3.21
d198	7715	36849	4.78
lin318	4271	25540	5.98
pcb442	4394	40509	9.22
rat783	1340	21937	16.38
pr1002	910	17894	19.67
d1291	835	23842	28.56
fl1577	742	22438	30.24
pr2392	216	15324	70.94
pcb3038	121	13323	110.1
fl3795	134	14478	108.0
rl5915	34	8820	259.4

ILS — acceptance criterion

- AcceptanceCriterion has strong influence on nature and effectiveness of walk in $\mathcal{S}^*$
- controls balance between intensification and diversification
- simplest case: Markovian acceptance criteria
- extreme intensification:
  `Better`$(s^*, s^{*\prime}, \mathit{history})$: accept $s^{*\prime}$ only if $f(s^{*\prime}) < f(s^*)$
- extreme diversification:
  `RW`$(s^*, s^{*\prime}, \mathit{history})$: accept $s^{*\prime}$ always
- many intermediate choices possible

example: influence of acceptance criterion on TSP

- small perturbations are known to be enough
- high quality solutions are known to cluster;
  "big valley structure"
  $\rightsquigarrow$ good strategy incorporates intensification

- compare average dev. from optimum (Δ_{avg}) over 25 trials
- Δ_{avg}(RR): random restart
- Δ_{avg}(RW): random walk as AcceptanceCriterion
- Δ_{avg}(Better): first descent in $\mathcal{S}^*$ as AcceptanceCriterion
- time limit: 120 sec on a Pentium II 266 MHz PC

instance	Δ_{avg}(RR)	Δ_{avg}(RW)	Δ_{avg}(Better)
kroA100	0.0	0.0	0.0
d198	0.003	0.0	0.0
lin318	0.66	0.30	0.12
pcb442	0.83	0.42	0.11
rat783	2.46	1.37	0.12
pr1002	2.72	1.55	0.14
pcb1173	3.12	1.63	0.40
d1291	2.21	0.59	0.28
fl1577	10.3	1.20	0.33
pr2392	4.38	2.29	0.54
pcb3038	4.21	2.62	0.47
fl3795	38.8	1.87	0.58
rl5915	6.90	2.13	0.66

exploitation of search history

- many of the bells and whistles of other strategies (diversification, intensification, tabu, adaptive perturbations and acceptance criteria, etc...) are applicable

simplest usage of search history

- extremely simple use of history:
  Restart($s^*, s^{*\prime}$, *history*): Restart search if for a number of iterations no improved solution is found

ILS — QAP, example results

instance	accept	3	$n/12$	$n/6$	$n/4$	$n/3$	$n/2$	$3n/4$	n
kra30a	Better	2.51	2.51	2.04	1.06	0.83	0.42	*0.0*	0.77
kra30a	RW	*0.0*	*0.0*	*0.0*	*0.0*	*0.0*	0.02	0.47	0.77
kra30a	Restart	*0.0*	*0.0*	*0.0*	*0.0*	*0.0*	*0.0*	*0.0*	0.77
sko64	Better	0.65	1.04	0.50	0.37	*0.29*	*0.29*	0.82	0.93
sko64	RW	*0.11*	0.14	0.17	0.24	0.44	0.62	0.88	0.93
sko64	Restart	0.37	0.31	*0.14*	*0.14*	0.15	0.41	0.79	0.93
tai60a	Better	2.31	2.24	1.91	*1.71*	1.86	2.94	3.13	3.18
tai60a	RW	*1.36*	1.44	2.08	2.63	2.81	3.02	3.14	3.18
tai60a	Restart	1.83	1.74	*1.45*	1.73	2.29	3.01	3.10	3.18
tai60b	Better	2.44	0.97	0.67	0.96	0.82	0.50	*0.14*	0.43
tai60b	RW	0.79	0.80	0.52	0.21	*0.08*	0.14	0.28	0.43
tai60b	Restart	0.08	0.08	*0.005*	0.02	0.03	0.07	0.17	0.43

ILS — local search

- in the simplest case, use LocalSearch as black box
- any improvement method can be used as LocalSearch
- best performance with optimization of this choice
- often it is necessary to have direct access to LocalSearch (e.g. when using don't look bits)

complex local search algorithms

- variable depth local search, ejection chains
- dynasearch
- variable neighborhood descent
- any other local search can be used within ILS, including *short* runs of
  - tabu search
  - simulated annealing
  - dynamic local search

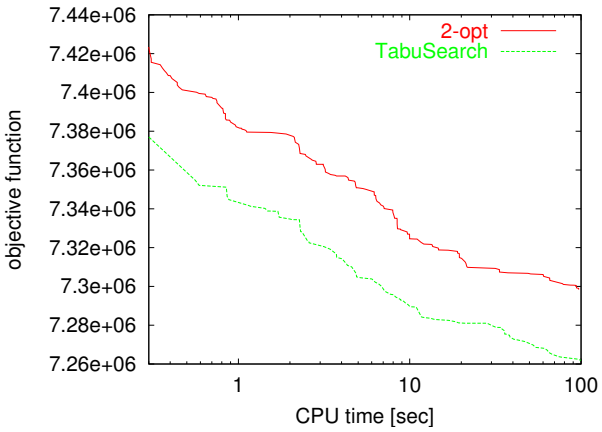
effectiveness of local search?

- *often*: the more effective the local search the better performs ILS
  - example TSP: 2-opt vs. 3-opt *vs*. Lin-Kernighan
- *sometimes*: preferable to have fast but less effective local search

the tradeoff between effectiveness and efficiency of the local search procedure is an important point to be adressed when optimizing an ILS algorithm

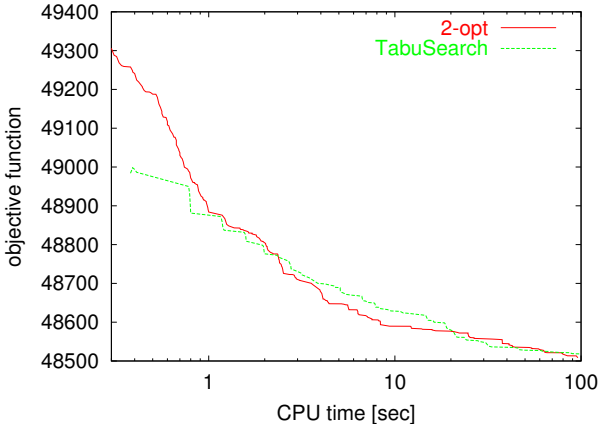
tabu search vs. 2-opt, tai60a

- short tabu search runs ($6n$ iterations) vs. 2-opt



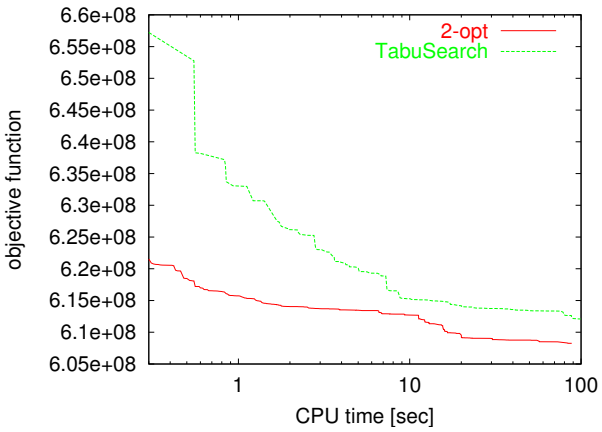
tabu search vs. 2-opt, sko64

- short tabu search runs ($6n$ iterations) vs. 2-opt



tabu search vs. 2-opt, tai60b

- short tabu search runs ($6n$ iterations) vs. 2-opt



optimize performance of ILS algorithms

- important to reach peak performance
- optimization goal has to be given (optimize average solution quality, etc.)
- robustness is an important issue
- *start:* basic ILS

ad-hoc optimization

- optimize single components, e.g. in the order GenerateInitialSolution, LocalSearch, Perturbation, AcceptanceCriterion
- iterate through this process

closer look

- optimal configuration of one component depends on other components
- complex interactions among components exist
- directly address these dependencies to perform a *global optimization of ILS performance*

main dependencies

- perturbation should not be easily undone by LocalSearch; if LocalSearch has obvious short-comings, a good perturbation should compensate for them.
- combination Perturbation — AcceptanceCriterion determines the relative balance of intensification and diversification; large perturbations are only useful if they can be accepted

but see also later chapter on automatic algorithm configuration!

ILS is ..

- based on simple principles
- easy to understand
- basic versions are easy to implement
- flexible, allowing for many additional optimizations if needed
- highly effective in many applications

Remark

- we have seen ILS in more detail than other methods
- main reason: one more in depth illustration of important trade-offs in the design of (meta-)heuristic algorithms
- note that for many other methods similar tradeoffs / issues exist and need to be tackled when trying to develop high-performing algorithms.