STOCHASTIC LOCAL SEARCH
FOUNDATIONS AND APPLICATIONS

SLS Methods: An Overview

Holger H. Hoos & Thomas Stützle

Outline

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

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.

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.

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.

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.

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 neighbhourhood size.

More general trade-off:

Effectiveness *vs* time complexity of search steps.

Trade-off:

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

More general trade-off:

Effectiveness *vs* time complexity of search steps.

Trade-off:

- Using larger neighbourhoods can improve performance of II (and other SLS methods).
- *But:* time required for determining improving search steps increases with neighbhourhood 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).

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).

Example: Candidate lists for the TSP

- *Intuition:* High-quality solutions likely include short edges.
- *Candidate list* of vertex v: list of v's nearest neighbours (limited number), sorted according to increasing edge weights.
- Search steps (*e.g.*, 2-exchange moves) always involve edges to elements of candidate lists.
- Significant impact on performance of SLS algorithms for the TSP.

Example: Candidate lists for the TSP

- *Intuition:* High-quality solutions likely include short edges.
- *Candidate list* of vertex v: list of v's nearest neighbours (limited number), sorted according to increasing edge weights.
- Search steps (*e.g.*, 2-exchange moves) always involve edges to elements of candidate lists.
- Significant impact on performance of SLS algorithms for the TSP.

Example: Candidate lists for the TSP

- *Intuition:* High-quality solutions likely include short edges.
- *Candidate list* of vertex v: list of v's nearest neighbours (limited number), sorted according to increasing edge weights.
- Search steps (*e.g.*, 2-exchange moves) always involve edges to elements of candidate lists.
- Significant impact on performance of SLS algorithms for the TSP.

Example: Candidate lists for the TSP

- *Intuition:* High-quality solutions likely include short edges.
- *Candidate list* of vertex v: list of v's nearest neighbours (limited number), sorted according to increasing edge weights.
- Search steps (*e.g.*, 2-exchange moves) always involve edges to elements of candidate lists.
- Significant impact on performance of SLS algorithms for the TSP.

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.

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.

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.

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.

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.

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 (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 $\{1,2, \ldots, n\}$

- **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 (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 (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 $\{1,2, \ldots, n\}$

- **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 (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.

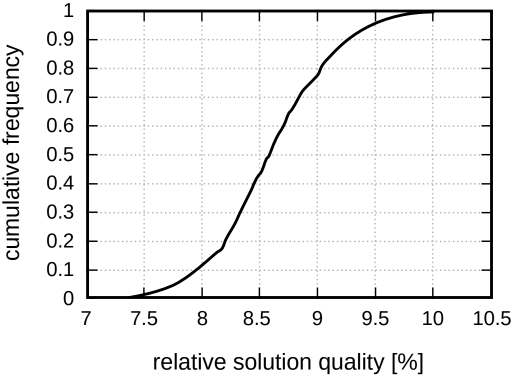
Example: Random-order first improvement for the TSP (2)

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.



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

- *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

- *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

- *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

- *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 $i > k$

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 $i > k$

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 $i > k$

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
```

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].

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].

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].

Variable Depth Search

- **Key idea:** *Complex steps* in large neighbourhoods = variable-length sequences of *simple steps* in small neighbourhood.
- 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

- **Key idea:** *Complex steps* in large neighbourhoods = variable-length sequences of *simple steps* in small neighbourhood.
- 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

- **Key idea:** *Complex steps* in large neighbourhoods = variable-length sequences of *simple steps* in small neighbourhood.
- 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
- $s := \hat{t}$

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
  $s := \hat{t}$

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 2-exchange steps and are constructed from sequences of *Hamiltonian paths* (= paths that visit every node in given graph exactly once).

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

Example: 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* (= paths that visit every node in given graph exactly once).
- δ*-path:* Hamiltonian path p + 1 edge connecting one end of p to interior node of p ('lasso' structure):

Example: 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* (= paths that visit every node in given graph exactly once).
- δ*-path:* Hamiltonian path p + 1 edge connecting one end of p to interior node of p ('lasso' structure):

Example: 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* (= paths that visit every node in given graph exactly once).

- *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 $(u, \ldots, v)$:

Basic LK exchange step:

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

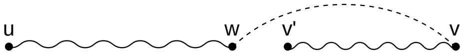
- Obtain δ-path by adding an edge (v, w):

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'):

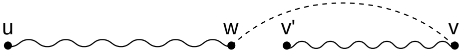


Basic LK exchange step:

- 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
5. 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.

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
5. 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.

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
5. 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.

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
5. 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.

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
5. 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.

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
5. 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.

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 neighbhourhood

- (For further details, see book and/or module on TSP.)

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 neighbhourhood

- (For further details, see book and/or module on TSP.)

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 neighbhourhood

- (For further details, see book and/or module on TSP.)

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 (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 (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 (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]

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]

'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

'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

'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

'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

'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

'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

'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
  $s := s'$

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
  $s := s'$

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
- $s := s'$

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
⌊ $s := s'$

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.

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.

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.

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.

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 ∅

   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' | 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 ∅
   end
end GUWSAT
```

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 ∅
   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' | 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 ∅
   end
end GUWSAT
```

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*

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*

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 ∅
   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' | 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 ∅
   end
end GUWSAT
```

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.

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.

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.

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.

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.

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.

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 (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 (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:

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

  (*Metropolis condition*), where *temperature* parameter T controls likelihood of accepting worsening steps.

- **Termination:** upon exceeding given bound on run-time.

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.

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.

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:

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

  (*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 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

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 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

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 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

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 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

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 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

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 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*

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*

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*

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.

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.

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.

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.

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.

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 $< 2\%$.

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 $< 2\%$.

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 $\exp\left[(f(s) - f(s'))/T\right]$);
- *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 $< 2\%$.

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.

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.

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.

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.

'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, 1998].

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)!

‘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, 1998].

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)!

‘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, 1998].

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)!

Tabu Search

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

Simple Tabu Search:

- Associate *tabu attributes* with candidate solutions or solution components.
- Forbid steps to search positions recently visited by underlying iterative best improvement procedure based on tabu attributes.

Tabu Search

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

Simple Tabu Search:

- Associate *tabu attributes* with candidate solutions or solution components.
- Forbid steps to search positions recently visited by underlying iterative best improvement procedure based on tabu attributes.

Tabu Search

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

Simple Tabu Search:

- Associate *tabu attributes* with candidate solutions or solution components.
- Forbid steps to search positions recently visited by underlying iterative best improvement procedure based on tabu attributes.

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'
- $s := s'$

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'
- $s := s'$

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'
- $s := s'$

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).

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).

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 (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 (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 (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.

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.

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.

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.

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:

- *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:

- *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 ⇒ search stagnates due to inability to escape from local minima;
- tt too high ⇒ 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.

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

- *tt* too low ⇒ search stagnates due to inability to escape from local minima;
- *tt* too high ⇒ 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.

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

- *tt* too low ⇒ search stagnates due to inability to escape from local minima;
- *tt* too high ⇒ 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.

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

- *tt* too low ⇒ search stagnates due to inability to escape from local minima;
- *tt* too high ⇒ 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.

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

- *tt* too low ⇒ search stagnates due to inability to escape from local minima;
- *tt* too high ⇒ 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.

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.

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.

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.

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 many combinatorial problems, including:

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

Crucial factors in many applications:

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

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

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

Crucial factors in many applications:

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

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

- SAT and MAX-SAT
- the Constraint Satisfaction Problem (CSP)
- many 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

- **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

- **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 (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 (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*.

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*.

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*.

Dynamic Local Search (continued)

- **Modified evaluation function:**

  $$g'(\pi, s) := g(\pi, s) + \sum_{i \in SC(\pi', s)} \mathit{penalty}(i),$$

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

- **Penalty initialisation:** For all i: $\mathit{penalty}(i) := 0$.

- **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**:

[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'.

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**:

[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'.

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**:

[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'.

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**:

[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.

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.

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.

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.

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.

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.

Earlier, closely 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]

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

- SAT [Hutter *et al.*, 2002]
- MAX-SAT [Tompkins and Hoos, 2003]
- MAX-CLIQUE [Pullan *et al.*, to appear]

Earlier, closely 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]

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

- SAT [Hutter *et al.*, 2002]
- MAX-SAT [Tompkins and Hoos, 2003]
- MAX-CLIQUE [Pullan *et al.*, to appear]

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

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

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

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

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 $s := r$

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 $s := r$

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.

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.

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.

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.

Subsidiary local search:

- More effective subsidiary local search procedures lead to better ILS performance.

  Example: 2-opt *vs* 3-opt *vs* LK for TSP.

- Often, subsidiary local search = iterative improvement, but more sophisticated SLS methods can be used. (*e.g.*, Tabu Search).

Subsidiary local search:

- More effective subsidiary local search procedures lead to better ILS performance.

  Example: 2-opt *vs* 3-opt *vs* LK for TSP.

- Often, subsidiary local search = iterative improvement, but more sophisticated SLS methods can be used. (*e.g.*, Tabu Search).

Perturbation mechanism:

- Needs to be chosen such that its effect *cannot* be easily undone by subsequent local search phase.
  (Often achieved by search steps larger neighbourhood.)

  Example: local search = 3-opt, perturbation = 4-exchange steps in ILS for TSP.

- A perturbation phase may consist of one or more perturbation steps.

Perturbation mechanism:

- Needs to be chosen such that its effect *cannot* be easily undone by subsequent local search phase.
  (Often achieved by search steps larger neighbourhood.)

  Example: local search = 3-opt, perturbation = 4-exchange steps in ILS for TSP.

- A perturbation phase may consist of one or more perturbation steps.

Perturbation mechanism (continued):

- Weak perturbation ⇒ short subsequent local search phase; *but:* risk of revisiting current local minimum.

- Strong perturbation ⇒ more effective escape from local minima; *but:* may have similar drawbacks as random restart.

- Advanced ILS algorithms may change nature and/or strength of perturbation adaptively during search.

Perturbation mechanism (continued):

- Weak perturbation ⇒ short subsequent local search phase; *but:* risk of revisiting current local minimum.
- Strong perturbation ⇒ more effective escape from local minima; *but:* may have similar drawbacks as random restart.
- Advanced ILS algorithms may change nature and/or strength of perturbation adaptively during search.

Perturbation mechanism (continued):

- Weak perturbation ⇒ short subsequent local search phase; *but:* risk of revisiting current local minimum.
- Strong perturbation ⇒ more effective escape from local minima; *but:* may have similar drawbacks as random restart.
- Advanced ILS algorithms may change nature and/or strength of perturbation adaptively during search.

Acceptance criteria:

- Always accept the *better* of the two candidate solutions

  ⇒ ILS performs Iterative Improvement in the space of local optima reached by subsidiary local search.

- Always accept the *more recent* of the two candidate solutions

  ⇒ ILS performs random walk in the space of local optima reached by subsidiary local search.

- Intermediate behaviour: select between the two candidate solutions based on the *Metropolis criterion* (*e.g.*, used in *Large Step Markov Chains* [Martin *et al.*, 1991].

- Advanced acceptance criteria take into account search history, *e.g.*, by occasionally reverting to *incumbent solution*.

Acceptance criteria:

- Always accept the *better* of the two candidate solutions

  ⇒ ILS performs Iterative Improvement in the space of local optima reached by subsidiary local search.

- Always accept the *more recent* of the two candidate solutions

  ⇒ ILS performs random walk in the space of local optima reached by subsidiary local search.

- Intermediate behaviour: select between the two candidate solutions based on the *Metropolis criterion* (*e.g.*, used in *Large Step Markov Chains* [Martin *et al.*, 1991].

- Advanced acceptance criteria take into account search history, *e.g.*, by occasionally reverting to *incumbent solution*.

Acceptance criteria:

- Always accept the *better* of the two candidate solutions

  ⇒ ILS performs Iterative Improvement in the space of local optima reached by subsidiary local search.

- Always accept the *more recent* of the two candidate solutions

  ⇒ ILS performs random walk in the space of local optima reached by subsidiary local search.

- Intermediate behaviour: select between the two candidate solutions based on the *Metropolis criterion* (*e.g.*, used in *Large Step Markov Chains* [Martin *et al.*, 1991].

- Advanced acceptance criteria take into account search history, *e.g.*, by occasionally reverting to *incumbent solution*.

Acceptance criteria:

- Always accept the *better* of the two candidate solutions

  ⇒ ILS performs Iterative Improvement in the space of local optima reached by subsidiary local search.

- Always accept the *more recent* of the two candidate solutions

  ⇒ ILS performs random walk in the space of local optima reached by subsidiary local search.

- Intermediate behaviour: select between the two candidate solutions based on the *Metropolis criterion* (*e.g.*, used in *Large Step Markov Chains* [Martin *et al.*, 1991].

- Advanced acceptance criteria take into account search history, *e.g.*, by occasionally reverting to *incumbent solution*.

Example: Iterated Local Search for the TSP (1)

- **Given:** TSP instance G.
- Search space: Hamiltonian cycles in G;
  use 4-exchange neighbourhood.
- **Subsidiary local search:**
  Lin-Kernighan variable depth search algorithm

Example: Iterated Local Search for the TSP (1)

- **Given:** TSP instance G.
- Search space: Hamiltonian cycles in G;
  use 4-exchange neighbourhood.
- **Subsidiary local search:**
  Lin-Kernighan variable depth search algorithm

Example: Iterated Local Search for the TSP (2)

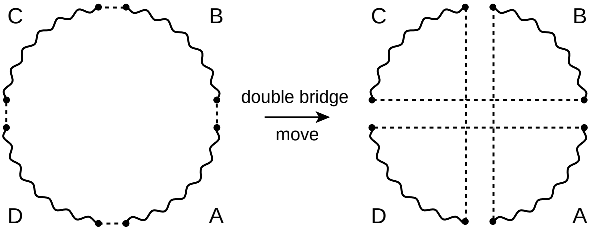
- **Perturbation mechanism:**
  'double-bridge move' = particular 4-exchange step:

Note:

- Cannot be directly reversed by a sequence of 2-exchange steps as performed by "usual" LK implementations.
- Empirically shown to be effective independent of instance size.

Example: Iterated Local Search for the TSP (2)

- **Perturbation mechanism:**
  'double-bridge move' = particular 4-exchange step:

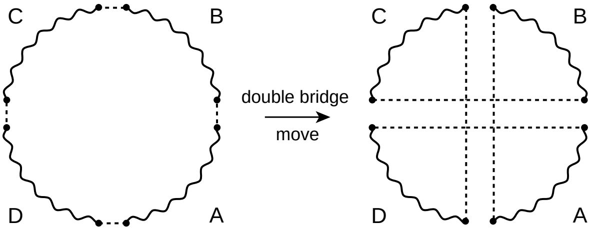


Note:

- Cannot be directly reversed by a sequence of 2-exchange steps as performed by "usual" LK implementations.
- Empirically shown to be effective independent of instance size.

Example: Iterated Local Search for the TSP (2)

- **Perturbation mechanism:**
  'double-bridge move' = particular 4-exchange step:



Note:

- Cannot be directly reversed by a sequence of 2-exchange steps as performed by "usual" LK implementations.
- Empirically shown to be effective independent of instance size.

Example: Iterated Local Search for the TSP (3)

- **Acceptance criterion:** Always return the better of the two given candidate round trips.

Note:

- This ILS algorithm for the TSP is known as *Iterated Lin-Kernighan (ILK) Algorithm*.
- Although ILK is structurally rather simple, an efficient implementation was shown to achieve excellent performance [Johnson and McGeoch, 1997].

Example: Iterated Local Search for the TSP (3)

- **Acceptance criterion:** Always return the better of the two given candidate round trips.

Note:

- This ILS algorithm for the TSP is known as *Iterated Lin-Kernighan (ILK) Algorithm*.
- Although ILK is structurally rather simple, an efficient implementation was shown to achieve excellent performance [Johnson and McGeoch, 1997].

Example: Iterated Local Search for the TSP (3)

- **Acceptance criterion:** Always return the better of the two given candidate round trips.

Note:

- This ILS algorithm for the TSP is known as *Iterated Lin-Kernighan (ILK) Algorithm*.
- Although ILK is structurally rather simple, an efficient implementation was shown to achieve excellent performance [Johnson and McGeoch, 1997].

Iterated local search algorithms . . .

- are typically rather easy to implement (given existing implementation of subsidiary simple SLS algorithms);
- achieve state-of-the-art performance on many combinatorial problems, including the TSP.

There are many SLS approaches that are closely related to ILS, including:

- Large Step Markov Chains [Martin *et al.*, 1991]
- Chained Local Search [Martin and Otto, 1996]
- Variants of Variable Neighbourhood Search (VNS) [Hansen and Mladenovìc, 2002]

Iterated local search algorithms . . .

- are typically rather easy to implement (given existing implementation of subsidiary simple SLS algorithms);
- achieve state-of-the-art performance on many combinatorial problems, including the TSP.

There are many SLS approaches that are closely related to ILS, including:

- Large Step Markov Chains [Martin *et al.*, 1991]
- Chained Local Search [Martin and Otto, 1996]
- Variants of Variable Neighbourhood Search (VNS) [Hansen and Mladenovìc, 2002]

Greedy Randomised Adaptive Search Procedures

Key Idea: Combine randomised constructive search with subsequent perturbative local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by perturbative local search.
- Perturbative local search methods typically often require substantially fewer steps to reach high-quality solutions when initialised using greedy constructive search rather than random picking.
- By iterating cycles of constructive + perturbative search, further performance improvements can be achieved.

Greedy Randomised Adaptive Search Procedures

Key Idea: Combine randomised constructive search with subsequent perturbative local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by perturbative local search.
- Perturbative local search methods typically often require substantially fewer steps to reach high-quality solutions when initialised using greedy constructive search rather than random picking.
- By iterating cycles of constructive + perturbative search, further performance improvements can be achieved.

Greedy Randomised Adaptive Search Procedures

Key Idea: Combine randomised constructive search with subsequent perturbative local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by perturbative local search.
- Perturbative local search methods typically often require substantially fewer steps to reach high-quality solutions when initialised using greedy constructive search rather than random picking.
- By iterating cycles of constructive + perturbative search, further performance improvements can be achieved.

Greedy Randomised Adaptive Search Procedures

Key Idea: Combine randomised constructive search with subsequent perturbative local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by perturbative local search.
- Perturbative local search methods typically often require substantially fewer steps to reach high-quality solutions when initialised using greedy constructive search rather than random picking.
- By iterating cycles of constructive + perturbative search, further performance improvements can be achieved.

Greedy Randomised "Adaptive" Search Procedure (GRASP):

While *termination criterion* is not satisfied:

  generate candidate solution s using
    subsidiary greedy randomised constructive search

  perform *subsidiary local search* on s

Note:

Randomisation in *constructive search* ensures that a large number of good starting points for *subsidiary local search* is obtained.

Greedy Randomised "Adaptive" Search Procedure (GRASP):

While *termination criterion* is not satisfied:

- generate candidate solution *s* using
  subsidiary greedy randomised constructive search
- perform *subsidiary local search* on *s*

Note:

Randomisation in *constructive search* ensures that a large number of good starting points for *subsidiary local search* is obtained.

Greedy Randomised "Adaptive" Search Procedure (GRASP):

While *termination criterion* is not satisfied:

- generate candidate solution s using
  subsidiary greedy randomised constructive search
- perform *subsidiary local search* on s

Note:

Randomisation in *constructive search* ensures that a large number of good starting points for *subsidiary local search* is obtained.

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a *restricted candidate list (RCL)*.
- RCLs are constructed in each step usig a *heuristic function* h.
- RCLs based on *cardinality restriction* comprise the k best-ranked solution components. (k is a parameter of the algorithm.)
- RCLs based on *value restriction* comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$, where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a *restricted candidate list (RCL)*.
- RCLs are constructed in each step usig a *heuristic function* h.
- RCLs based on *cardinality restriction* comprise the k best-ranked solution components. (k is a parameter of the algorithm.)
- RCLs based on *value restriction* comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$, where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a *restricted candidate list (RCL)*.
- RCLs are constructed in each step usig a *heuristic function* h.
- RCLs based on *cardinality restriction* comprise the k best-ranked solution components. (k is a parameter of the algorithm.)
- RCLs based on *value restriction* comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$, where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a *restricted candidate list (RCL)*.
- RCLs are constructed in each step usig a *heuristic function* h.
- RCLs based on *cardinality restriction* comprise the k best-ranked solution components. (k is a parameter of the algorithm.)
- RCLs based on *value restriction* comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$, where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)

Note:

- Constructive search in GRASP is 'adaptive':
Heuristic value of solution component to be added to given partial candidate solution r may depend on solution components present in r.
- Variants of GRASP without perturbative local search phase (aka *semi-greedy heuristics*) typically do not reach the performance of GRASP with perturbative local search.

Note:

- Constructive search in GRASP is 'adaptive':
  Heuristic value of solution component to be added to given partial candidate solution r may depend on solution components present in r.
- Variants of GRASP without perturbative local search phase (aka *semi-greedy heuristics*) typically do not reach the performance of GRASP with perturbative local search.

Example: GRASP for SAT [Resende and Feo, 1996]

- **Given:** CNF formula F over variables $x_1, \ldots, x_n$
- **Subsidiary constructive search:**
  - start from empty variable assignment
  - in each step, add one atomic assignment (*i.e.*, assignment of a truth value to a currently unassigned variable)
  - heuristic function $h(i, v) :=$ number of clauses that become satisfied as a consequence of assigning $x_i := v$
  - RCLs based on cardinality restriction (contain fixed number k of atomic assignments with largest heuristic values)
- **Subsidiary local search:**
  - iterative best improvement using 1-flip neighbourhood
  - terminates when model has been found or given number of steps has been exceeded

Example: GRASP for SAT [Resende and Feo, 1996]

- **Given:** CNF formula F over variables $x_1, \ldots, x_n$
- **Subsidiary constructive search:**
  - start from empty variable assignment
  - in each step, add one atomic assignment (*i.e.*, assignment of a truth value to a currently unassigned variable)
  - heuristic function $h(i, v) :=$ number of clauses that become satisfied as a consequence of assigning $x_i := v$
  - RCLs based on cardinality restriction (contain fixed number k of atomic assignments with largest heuristic values)
- **Subsidiary local search:**
  - iterative best improvement using 1-flip neighbourhood
  - terminates when model has been found or given number of steps has been exceeded

Example: GRASP for SAT [Resende and Feo, 1996]

- **Given:** CNF formula F over variables $x_1, \ldots, x_n$
- **Subsidiary constructive search:**
  - start from empty variable assignment
  - in each step, add one atomic assignment (*i.e.*, assignment of a truth value to a currently unassigned variable)
  - heuristic function $h(i, v) :=$ number of clauses that become satisfied as a consequence of assigning $x_i := v$
  - RCLs based on cardinality restriction (contain fixed number k of atomic assignments with largest heuristic values)
- **Subsidiary local search:**
  - iterative best improvement using 1-flip neighbourhood
  - terminates when model has been found or given number of steps has been exceeded

Example: GRASP for SAT [Resende and Feo, 1996]

- **Given:** CNF formula F over variables $x_1, \ldots, x_n$
- **Subsidiary constructive search:**
  - start from empty variable assignment
  - in each step, add one atomic assignment (*i.e.*, assignment of a truth value to a currently unassigned variable)
  - heuristic function $h(i, v) :=$ number of clauses that become satisfied as a consequence of assigning $x_i := v$
  - RCLs based on cardinality restriction (contain fixed number k of atomic assignments with largest heuristic values)
- **Subsidiary local search:**
  - iterative best improvement using 1-flip neighbourhood
  - terminates when model has been found or given number of steps has been exceeded

GRASP has been applied to many combinatorial problems, including:

- SAT, MAX-SAT
- the Quadratic Assignment Problem
- various scheduling problems

Extensions and improvements of GRASP:

- reactive GRASP (*e.g.*, dynamic adaptation of α during search)
- combinations of GRASP with Tabu Search and other SLS methods

GRASP has been applied to many combinatorial problems, including:

- SAT, MAX-SAT
- the Quadratic Assignment Problem
- various scheduling problems

Extensions and improvements of GRASP:

- reactive GRASP (*e.g.*, dynamic adaptation of α during search)
- combinations of GRASP with Tabu Search and other SLS methods

GRASP has been applied to many combinatorial problems, including:

- SAT, MAX-SAT
- the Quadratic Assignment Problem
- various scheduling problems

Extensions and improvements of GRASP:

- reactive GRASP (*e.g.*, dynamic adaptation of α during search)
- combinations of GRASP with Tabu Search and other SLS methods

Adaptive Iterated Construction Search

Key Idea: Alternate construction and perturbative local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate *weights* with possible decisions made during constructive search.
- Initialise all weights to some small value τ_0 at beginning of search process.
- After every cycle (= constructive + perturbative local search phase), update weights based on solution quality and solution components of current candidate solution.

Adaptive Iterated Construction Search

Key Idea: Alternate construction and perturbative local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate *weights* with possible decisions made during constructive search.
- Initialise all weights to some small value τ_0 at beginning of search process.
- After every cycle (= constructive + perturbative local search phase), update weights based on solution quality and solution components of current candidate solution.

Adaptive Iterated Construction Search

Key Idea: Alternate construction and perturbative local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate *weights* with possible decisions made during constructive search.
- Initialise all weights to some small value τ_0 at beginning of search process.
- After every cycle (= constructive + perturbative local search phase), update weights based on solution quality and solution components of current candidate solution.

Adaptive Iterated Construction Search

Key Idea: Alternate construction and perturbative local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate *weights* with possible decisions made during constructive search.
- Initialise all weights to some small value τ_0 at beginning of search process.
- After every cycle (= constructive + perturbative local search phase), update weights based on solution quality and solution components of current candidate solution.

Adaptive Iterated Construction Search (AICS):

initialise weights

While *termination criterion* is not satisfied:

- generate candidate solution s using
  subsidiary randomised constructive search
- perform *subsidiary local search* on s
- *adapt weights* based on s

Adaptive Iterated Construction Search (AICS):

initialise weights

While *termination criterion* is not satisfied:

- generate candidate solution *s* using
  subsidiary randomised constructive search
- perform *subsidiary local search* on *s*
- *adapt weights* based on *s*

Adaptive Iterated Construction Search (AICS):

initialise weights

While *termination criterion* is not satisfied:

- generate candidate solution s using *subsidiary randomised constructive search*
- perform *subsidiary local search* on s
- *adapt weights* based on s

Adaptive Iterated Construction Search (AICS):

initialise weights

While *termination criterion* is not satisfied:

- generate candidate solution s using *subsidiary randomised constructive search*
- perform *subsidiary local search* on s
- *adapt weights* based on s

Subsidiary constructive search:

- The solution component to be added in each step of *constructive search* is based on *weights* and heuristic function h.
- h can be standard heuristic function as, *e.g.*, used by greedy construction heuristics, GRASP or tree search.
- It is often useful to design solution component selection in constructive search such that any solution component may be chosen (at least with some small probability) irrespective of its weight and heuristic value.

Subsidiary constructive search:

- The solution component to be added in each step of *constructive search* is based on *weights* and heuristic function h.
- h can be standard heuristic function as, *e.g.*, used by greedy construction heuristics, GRASP or tree search.
- It is often useful to design solution component selection in constructive search such that any solution component may be chosen (at least with some small probability) irrespective of its weight and heuristic value.

Subsidiary constructive search:

- The solution component to be added in each step of *constructive search* is based on *weights* and heuristic function h.
- h can be standard heuristic function as, *e.g.*, used by greedy construction heuristics, GRASP or tree search.
- It is often useful to design solution component selection in constructive search such that any solution component may be chosen (at least with some small probability) irrespective of its weight and heuristic value.

Subsidiary perturbative local search:

- As in GRASP, perturbative local search phase is typically important for achieving good performance.
- Can be based on Iterative Improvement or more advanced SLS method (the latter often results in better performance).
- Tradeoff between computation time used in construction phase *vs* local search phase (typically optimised empirically, depends on problem domain).

Subsidiary perturbative local search:

- As in GRASP, perturbative local search phase is typically important for achieving good performance.
- Can be based on Iterative Improvement or more advanced SLS method (the latter often results in better performance).
- Tradeoff between computation time used in construction phase *vs* local search phase (typically optimised empirically, depends on problem domain).

Subsidiary perturbative local search:

- As in GRASP, perturbative local search phase is typically important for achieving good performance.
- Can be based on Iterative Improvement or more advanced SLS method (the latter often results in better performance).
- Tradeoff between computation time used in construction phase *vs* local search phase (typically optimised empirically, depends on problem domain).

Weight updating mechanism:

- Typical mechanism: increase weights of all solution components contained in candidate solution obtained from local search.
- Can also use aspects of search history; *e.g.*, current *incumbent candidate solution* can be used as basis for weight update for additional intensification.

Weight updating mechanism:

- Typical mechanism: increase weights of all solution components contained in candidate solution obtained from local search.
- Can also use aspects of search history; *e.g.*, current *incumbent candidate solution* can be used as basis for weight update for additional intensification.

Example: A simple AICS algorithm for the TSP (1)

(Based on Ant System for the TSP [Dorigo *et al.*, 1991].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate weight τ_{ij} with each edge (i, j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i, j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search* starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple AICS algorithm for the TSP (1)

(Based on Ant System for the TSP [Dorigo *et al.*, 1991].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate weight τ_{ij} with each edge (i, j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i, j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search* starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple AICS algorithm for the TSP (1)

(Based on Ant System for the TSP [Dorigo *et al.*, 1991].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate weight τ_{ij} with each edge (i,j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i,j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search* starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple AICS algorithm for the TSP (1)

(Based on Ant System for the TSP [Dorigo *et al.*, 1991].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate weight τ_{ij} with each edge (i, j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i, j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search* starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple AICS algorithm for the TSP (1)

(Based on Ant System for the TSP [Dorigo *et al.*, 1991].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate weight τ_{ij} with each edge (i, j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i, j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search* starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple AICS algorithm for the TSP (2)

- *Subsidiary local search* = iterative improvement based on standard 2-exchange neighbourhood (until local minimum is reached).
- *Weight update* according to

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(i, j, s')$$

where $\Delta(i, j, s') := 1/f(s')$, if edge (i, j) is contained in the cycle represented by s', and 0 otherwise.

- Criterion for weight increase is based on intuition that edges contained in short round trips should be preferably used in subsequent constructions.

Example: A simple AICS algorithm for the TSP (2)

- *Subsidiary local search* = iterative improvement based on standard 2-exchange neighbourhood (until local minimum is reached).
- *Weight update* according to

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(i, j, s')$$

  where $\Delta(i, j, s') := 1/f(s')$, if edge (i, j) is contained in the cycle represented by s', and 0 otherwise.
- Criterion for weight increase is based on intuition that edges contained in short round trips should be preferably used in subsequent constructions.

Example: A simple AICS algorithm for the TSP (2)

- *Subsidiary local search* = iterative improvement based on standard 2-exchange neighbourhood (until local minimum is reached).
- *Weight update* according to

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(i, j, s')$$

  where $\Delta(i, j, s') := 1/f(s')$, if edge (i, j) is contained in the cycle represented by s', and 0 otherwise.
- Criterion for weight increase is based on intuition that edges contained in short round trips should be preferably used in subsequent constructions.

Adaptive Iterated Construction Search ...

- models recent variants of constructive search, including:
  - stochastic tree search [Bresina, 1996],
  - Squeeky Wheel Optimisation [Joslin and Clements, 1999],
  - Adaptive Probing [Ruml, 2001];
- is a special case of Ant Colony Optimisation (which can be seen as population-based variant of AICS);
- has not (yet) been widely used as a general SLS technique.

Adaptive Iterated Construction Search ...

- models recent variants of constructive search, including:
  - stochastic tree search [Bresina, 1996],
  - Squeeky Wheel Optimisation [Joslin and Clements, 1999],
  - Adaptive Probing [Ruml, 2001];
- is a special case of Ant Colony Optimisation (which can be seen as population-based variant of AICS);
- has not (yet) been widely used as a general SLS technique.

Adaptive Iterated Construction Search ...

- models recent variants of constructive search, including:
  - stochastic tree search [Bresina, 1996],
  - Squeeky Wheel Optimisation [Joslin and Clements, 1999],
  - Adaptive Probing [Ruml, 2001];
- is a special case of Ant Colony Optimisation (which can be seen as population-based variant of AICS);
- has not (yet) been widely used as a general SLS technique.

Population-based SLS Methods

SLS methods discussed so far manipulate one candidate solution of given problem instance in each search step.

Straightforward extension: Use *population* (*i.e.*, set) of candidate solutions instead.

Note:

- The use of populations provides a generic way to achieve search diversification.
- Population-based SLS methods fit into the general definition from Chapter 1 by treating sets of candidate solutions as search positions.

Population-based SLS Methods

SLS methods discussed so far manipulate one candidate solution of given problem instance in each search step.

Straightforward extension: Use *population* (*i.e.*, set) of candidate solutions instead.

Note:

- The use of populations provides a generic way to achieve search diversification.
- Population-based SLS methods fit into the general definition from Chapter 1 by treating sets of candidate solutions as search positions.

Population-based SLS Methods

SLS methods discussed so far manipulate one candidate solution of given problem instance in each search step.

Straightforward extension: Use *population* (*i.e.*, set) of candidate solutions instead.

Note:

- The use of populations provides a generic way to achieve search diversification.
- Population-based SLS methods fit into the general definition from Chapter 1 by treating sets of candidate solutions as search positions.

Population-based SLS Methods

SLS methods discussed so far manipulate one candidate solution of given problem instance in each search step.

Straightforward extension: Use *population* (*i.e.*, set) of candidate solutions instead.

Note:

- The use of populations provides a generic way to achieve search diversification.
- Population-based SLS methods fit into the general definition from Chapter 1 by treating sets of candidate solutions as search positions.

Ant Colony Optimisation (1)

Key idea: Can be seen as population-based extension of AICS where population of agents – *(artificial) ants* – communicate via common memory – *(simulated) pheromone trails*.

Inspired by foraging behaviour of real ants:

- Ants often communicate via chemicals known as *pheromones*, which are deposited on the ground in the form of trails. (This is a form of *stigmergy*: indirect communication via manipulation of a common environment.)
- Pheromone trails provide the basis for (stochastic) trail-following behaviour underlying, *e.g.*, the collective ability to find shortest paths between a food source and the nest.

Ant Colony Optimisation (1)

Key idea: Can be seen as population-based extension of AICS where population of agents – *(artificial) ants* – communicate via common memory – *(simulated) pheromone trails*.

Inspired by foraging behaviour of real ants:

- Ants often communicate via chemicals known as *pheromones*, which are deposited on the ground in the form of trails. (This is a form of *stigmergy*: indirect communication via manipulation of a common environment.)
- Pheromone trails provide the basis for (stochastic) trail-following behaviour underlying, *e.g.*, the collective ability to find shortest paths between a food source and the nest.

Ant Colony Optimisation (1)

Key idea: Can be seen as population-based extension of AICS where population of agents – *(artificial) ants* – communicate via common memory – *(simulated) pheromone trails*.

Inspired by foraging behaviour of real ants:

- Ants often communicate via chemicals known as *pheromones*, which are deposited on the ground in the form of trails. (This is a form of *stigmergy*: indirect communication via manipulation of a common environment.)
- Pheromone trails provide the basis for (stochastic) trail-following behaviour underlying, *e.g.*, the collective ability to find shortest paths between a food source and the nest.

Ant Colony Optimisation (2)

Application to combinatorial problems:
[Dorigo et al. 1991, 1996]

- Ants iteratively construct candidate solutions.
- Solution construction is probabilistically biased by pheromone trail information, heuristic information and partial candidate solution of each ant.
- Pheromone trails are modified during the search process to reflect collective experience.

Ant Colony Optimisation (2)

Application to combinatorial problems:
[Dorigo et al. 1991, 1996]

- Ants iteratively construct candidate solutions.
- Solution construction is probabilistically biased by pheromone trail information, heuristic information and partial candidate solution of each ant.
- Pheromone trails are modified during the search process to reflect collective experience.

Ant Colony Optimisation (2)

Application to combinatorial problems:
[Dorigo et al. 1991, 1996]

- Ants iteratively construct candidate solutions.
- Solution construction is probabilistically biased by pheromone trail information, heuristic information and partial candidate solution of each ant.
- Pheromone trails are modified during the search process to reflect collective experience.

Ant Colony Optimisation (ACO):

initialise pheromone trails

While termination criterion is not satisfied:

generate population *sp* of candidate solutions
using *subsidiary randomised constructive search*

perform *subsidiary local search* on *sp*

update pheromone trails based on *sp*

Ant Colony Optimisation (ACO):

initialise pheromone trails

While termination criterion is not satisfied:

generate population *sp* of candidate solutions
using *subsidiary randomised constructive search*

perform *subsidiary local search* on *sp*

update pheromone trails based on *sp*

Ant Colony Optimisation (ACO):

initialise pheromone trails

While termination criterion is not satisfied:

- generate population *sp* of candidate solutions using *subsidiary randomised constructive search*
- perform *subsidiary local search* on *sp*
- *update pheromone trails* based on *sp*

Note:

- In each cycle, each ant creates one candidate solution using a *constructive search procedure*.
- *Subsidiary local search* is applied to individual candidate solutions. (Some ACO algorithms do not use a subsidiary local search procedure.)
- All *pheromone trails* are initialised to the same value, τ_0.
- *Pheromone update* typically comprises uniform decrease of all trail levels (*evaporation*) and increase of some trail levels based on candidate solutions obtained from construction + local search.
- *Termination criterion* can include conditions on make-up of current population, *e.g.*, variation in solution quality or distance between individual candidate solutions.

Note:

- In each cycle, each ant creates one candidate solution using a *constructive search procedure*.
- *Subsidiary local search* is applied to individual candidate solutions. (Some ACO algorithms do not use a subsidiary local search procedure.)
- All *pheromone trails* are initialised to the same value, τ_0.
- *Pheromone update* typically comprises uniform decrease of all trail levels (*evaporation*) and increase of some trail levels based on candidate solutions obtained from construction + local search.
- *Termination criterion* can include conditions on make-up of current population, *e.g.*, variation in solution quality or distance between individual candidate solutions.

Note:

- In each cycle, each ant creates one candidate solution using a *constructive search procedure*.
- *Subsidiary local search* is applied to individual candidate solutions. (Some ACO algorithms do not use a subsidiary local search procedure.)
- All *pheromone trails* are initialised to the same value, τ_0.
- *Pheromone update* typically comprises uniform decrease of all trail levels (*evaporation*) and increase of some trail levels based on candidate solutions obtained from construction + local search.
- *Termination criterion* can include conditions on make-up of current population, *e.g.*, variation in solution quality or distance between individual candidate solutions.

Note:

- In each cycle, each ant creates one candidate solution using a *constructive search procedure*.
- *Subsidiary local search* is applied to individual candidate solutions. (Some ACO algorithms do not use a subsidiary local search procedure.)
- All *pheromone trails* are initialised to the same value, τ_0.
- *Pheromone update* typically comprises uniform decrease of all trail levels (*evaporation*) and increase of some trail levels based on candidate solutions obtained from construction + local search.
- *Termination criterion* can include conditions on make-up of current population, *e.g.*, variation in solution quality or distance between individual candidate solutions.

Note:

- In each cycle, each ant creates one candidate solution using a *constructive search procedure*.
- *Subsidiary local search* is applied to individual candidate solutions. (Some ACO algorithms do not use a subsidiary local search procedure.)
- All *pheromone trails* are initialised to the same value, τ_0.
- *Pheromone update* typically comprises uniform decrease of all trail levels (*evaporation*) and increase of some trail levels based on candidate solutions obtained from construction + local search.
- *Termination criterion* can include conditions on make-up of current population, *e.g.*, variation in solution quality or distance between individual candidate solutions.

Example: A simple ACO algorithm for the TSP (1)

(Variant of Ant System for the TSP [Dorigo *et al.*, 1991; 1996].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate pheromone trails τ_{ij} with each edge (i,j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i,j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search:* Each ant starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple ACO algorithm for the TSP (1)

(Variant of Ant System for the TSP [Dorigo *et al.*, 1991; 1996].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate pheromone trails τ_{ij} with each edge (i,j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i,j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search:* Each ant starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple ACO algorithm for the TSP (1)

(Variant of Ant System for the TSP [Dorigo *et al.*, 1991; 1996].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate pheromone trails τ_{ij} with each edge (i,j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i,j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search:* Each ant starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple ACO algorithm for the TSP (1)

(Variant of Ant System for the TSP [Dorigo *et al.*, 1991; 1996].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate pheromone trails τ_{ij} with each edge (i, j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i, j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search:* Each ant starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{l \in N'(i)} [\tau_{il}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}$$

Example: A simple ACO algorithm for the TSP (1)

(Variant of Ant System for the TSP [Dorigo *et al.*, 1991; 1996].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G).
- Associate pheromone trails τ_{ij} with each edge (i,j) in G.
- Use heuristic values $\eta_{ij} := 1/w((i,j))$.
- Initialise all weights to a small value τ_0 (parameter).
- *Constructive search:* Each ant starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{l \in N'(i)} [\tau_{il}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}$$

Example: A simple ACO algorithm for the TSP (2)

- *Subsidiary local search:* Perform iterative improvement based on standard 2-exchange neighbourhood on each candidate solution in population (until local minimum is reached).
- *Update pheromone trail levels* according to

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \sum_{s' \in sp'} \Delta(i, j, s')$$

where $\Delta(i, j, s') := 1/f(s')$ if edge (i, j) is contained in the cycle represented by s', and 0 otherwise.

Motivation: Edges belonging to highest-quality candidate solutions and/or that have been used by many ants should be preferably used in subsequent constructions.

Example: A simple ACO algorithm for the TSP (2)

- *Subsidiary local search:* Perform iterative improvement based on standard 2-exchange neighbourhood on each candidate solution in population (until local minimum is reached).
- *Update pheromone trail levels* according to

$$\tau_{ij} := (1-\rho)\cdot\tau_{ij} + \sum_{s' \in sp'} \Delta(i,j,s')$$

where $\Delta(i,j,s') := 1/f(s')$ if edge (i,j) is contained in the cycle represented by s', and 0 otherwise.

Motivation: Edges belonging to highest-quality candidate solutions and/or that have been used by many ants should be preferably used in subsequent constructions.

Example: A simple ACO algorithm for the TSP (2)

- *Subsidiary local search:* Perform iterative improvement based on standard 2-exchange neighbourhood on each candidate solution in population (until local minimum is reached).
- *Update pheromone trail levels* according to

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \sum_{s' \in sp'} \Delta(i, j, s')$$

where $\Delta(i, j, s') := 1/f(s')$ if edge (i, j) is contained in the cycle represented by s', and 0 otherwise.

Motivation: Edges belonging to highest-quality candidate solutions and/or that have been used by many ants should be preferably used in subsequent constructions.

Example: A simple ACO algorithm for the TSP (3)

- *Termination:* After fixed number of cycles (= construction + local search phases).

Note:

- Ants can be seen as walking along edges of given graph (using memory to ensure their tours correspond to Hamiltonian cycles) and depositing pheromone to reinforce edges of tours.
- Original Ant System did not include subsidiary local search procedure (leading to worse performance compared to the algorithm presented here)

Example: A simple ACO algorithm for the TSP (3)

- *Termination:* After fixed number of cycles ($=$ construction $+$ local search phases).

Note:

- Ants can be seen as walking along edges of given graph (using memory to ensure their tours correspond to Hamiltonian cycles) and depositing pheromone to reinforce edges of tours.
- Original Ant System did not include subsidiary local search procedure (leading to worse performance compared to the algorithm presented here)

Example: A simple ACO algorithm for the TSP (3)

- *Termination:* After fixed number of cycles
(= construction + local search phases).

Note:

- Ants can be seen as walking along edges of given graph (using memory to ensure their tours correspond to Hamiltonian cycles) and depositing pheromone to reinforce edges of tours.
- Original Ant System did not include subsidiary local search procedure (leading to worse performance compared to the algorithm presented here)

Enhancements:

- use of look-ahead in construction phase;
- pheromone updates during construction phase;
- bounds on range and smoothing of pheromone levels.

Advanced ACO methods:

- Ant Colony System [Dorigo and Gambardella, 1997]
- $\mathcal{MAX}-\mathcal{MIN}$ Ant System [Stützle and Hoos, 1997; 2000]
- the ANTS Algorithm [Maniezzo, 1999]

Enhancements:

- use of look-ahead in construction phase;
- pheromone updates during construction phase;
- bounds on range and smoothing of pheromone levels.

Advanced ACO methods:

- Ant Colony System [Dorigo and Gambardella, 1997]
- $\mathcal{MAX}-\mathcal{MIN}$ Ant System [Stützle and Hoos, 1997; 2000]
- the ANTS Algorithm [Maniezzo, 1999]

Enhancements:

- use of look-ahead in construction phase;
- pheromone updates during construction phase;
- bounds on range and smoothing of pheromone levels.

Advanced ACO methods:

- Ant Colony System [Dorigo and Gambardella, 1997]
- $\mathcal{MAX}-\mathcal{MIN}$ Ant System [Stützle and Hoos, 1997; 2000]
- the ANTS Algorithm [Maniezzo, 1999]

Enhancements:

- use of look-ahead in construction phase;
- pheromone updates during construction phase;
- bounds on range and smoothing of pheromone levels.

Advanced ACO methods:

- Ant Colony System [Dorigo and Gambardella, 1997]
- $\mathcal{MAX}-\mathcal{MIN}$ Ant System [Stützle and Hoos, 1997; 2000]
- the ANTS Algorithm [Maniezzo, 1999]

Ant Colony Optimisation ...

- has been applied very successfully to a wide range of combinatorial problems, including
  - the Open Shop Scheduling Problem,
  - the Sequential Ordering Problem, and
  - the Shortest Common Supersequence Problem;
- underlies new high-performance algorithms for *dynamic optimisation problems*, such as routing in telecommunications networks [Di Caro and Dorigo, 1998].

Ant Colony Optimisation ...

- has been applied very successfully to a wide range of combinatorial problems, including
  - the Open Shop Scheduling Problem,
  - the Sequential Ordering Problem, and
  - the Shortest Common Supersequence Problem;
- underlies new high-performance algorithms for *dynamic optimisation problems*, such as routing in telecommunications networks [Di Caro and Dorigo, 1998].

Note:

A general algorithmic framework for solving static and dynamic combinatorial problems using ACO techniques is provided by the *ACO metaheuristic* [Dorigo and Di Caro, 1999; Dorigo et al., 1999].

For further details on Ant Colony Optimisation, see the book by Dorigo and Stützle [2004].

Evolutionary Algorithms

Key idea: Iteratively apply *genetic operators* *mutation*, *recombination*, *selection* to a population of candidate solutions.

Inspired by simple model of biological evolution:

- *Mutation* introduces random variation in the genetic material of individuals.
- *Recombination* of genetic material during sexual reproduction produces *offspring* that combines features inherited from both *parents*.
- Differences in *evolutionary fitness* lead *selection* of genetic traits ('survival of the fittest').

Evolutionary Algorithms

Key idea: Iteratively apply *genetic operators* *mutation*, *recombination*, *selection* to a population of candidate solutions.

Inspired by simple model of biological evolution:

- *Mutation* introduces random variation in the genetic material of individuals.
- *Recombination* of genetic material during sexual reproduction produces *offspring* that combines features inherited from both *parents*.
- Differences in *evolutionary fitness* lead *selection* of genetic traits ('survival of the fittest').

Evolutionary Algorithms

Key idea: Iteratively apply *genetic operators* *mutation*, *recombination*, *selection* to a population of candidate solutions.

Inspired by simple model of biological evolution:

- *Mutation* introduces random variation in the genetic material of individuals.
- *Recombination* of genetic material during sexual reproduction produces *offspring* that combines features inherited from both *parents*.
- Differences in *evolutionary fitness* lead *selection* of genetic traits ('survival of the fittest').

Evolutionary Algorithms

Key idea: Iteratively apply *genetic operators* *mutation*, *recombination*, *selection* to a population of candidate solutions.

Inspired by simple model of biological evolution:

- *Mutation* introduces random variation in the genetic material of individuals.
- *Recombination* of genetic material during sexual reproduction produces *offspring* that combines features inherited from both *parents*.
- Differences in *evolutionary fitness* lead *selection* of genetic traits ('survival of the fittest').

Evolutionary Algorithm (EA):

determine initial population *sp*

While *termination criterion* is not satisfied:

generate set *spr* of new candidate solutions
by *recombination*

generate set *spm* of new candidate solutions
from *spr* and *sp* by *mutation*

select new population *sp* from
candidate solutions in *sp*, *spr*, and *spm*

Evolutionary Algorithm (EA):

determine initial population *sp*

While *termination criterion* is not satisfied:

generate set *spr* of new candidate solutions
by *recombination*

generate set *spm* of new candidate solutions
from *spr* and *sp* by *mutation*

select new population *sp* from
candidate solutions in *sp*, *spr*, and *spm*

Evolutionary Algorithm (EA):

determine initial population *sp*

While *termination criterion* is not satisfied:

generate set *spr* of new candidate solutions
by *recombination*

generate set *spm* of new candidate solutions
from *spr* and *sp* by *mutation*

select new population *sp* from
candidate solutions in *sp*, *spr*, and *spm*

Evolutionary Algorithm (EA):

determine initial population *sp*

While *termination criterion* is not satisfied:

- generate set *spr* of new candidate solutions by *recombination*
- generate set *spm* of new candidate solutions from *spr* and *sp* by *mutation*
- *select* new population *sp* from candidate solutions in *sp*, *spr*, and *spm*

Problem: Pure evolutionary algorithms often lack capability of sufficient *search intensification*.

Solution: Apply subsidiary local search after initialisation, mutation and recombination.

⇒ *Memetic Algorithms* (aka *Genetic Local Search*)

Problem: Pure evolutionary algorithms often lack capability of sufficient *search intensification*.

Solution: Apply subsidiary local search after initialisation, mutation and recombination.

$\Rightarrow$ *Memetic Algorithms* (aka *Genetic Local Search*)

Evolutionary Algorithm (EA):

determine initial population *sp*

perform *subsidiary local search* on *sp*

While *termination criterion* is not satisfied:

generate set *spr* of new candidate solutions
by *recombination*

perform *subsidiary local search* on *spr*

generate set *spm* of new candidate solutions
from *spr* and *sp* by *mutation*

perform *subsidiary local search* on *spm*

select new population *sp* from
candidate solutions in *sp*, *spr*, and *spm*

Memetic Algorithm (MA):

determine initial population *sp*

perform *subsidiary local search* on *sp*

While *termination criterion* is not satisfied:

- generate set *spr* of new candidate solutions by *recombination*
- perform *subsidiary local search* on *spr*
- generate set *spm* of new candidate solutions from *spr* and *sp* by *mutation*
- perform *subsidiary local search* on *spm*
- *select* new population *sp* from candidate solutions in *sp*, *spr*, and *spm*

Initialisation

- *Often:* independent, uninformed random picking from given search space.
- *But:* can also use multiple runs of construction heuristic.

Recombination

- Typically repeatedly selects a set of *parents* from current population and generates *offspring* candidate solutions from these by means of *recombination operator*.
- *Recombination operators* are generally based on *linear representation* of candidate solutions and piece together *offspring* from fragments of *parents*.

Initialisation

- *Often:* independent, uninformed random picking from given search space.
- *But:* can also use multiple runs of construction heuristic.

Recombination

- Typically repeatedly selects a set of *parents* from current population and generates *offspring* candidate solutions from these by means of *recombination operator*.
- *Recombination operators* are generally based on *linear representation* of candidate solutions and piece together *offspring* from fragments of *parents*.

Initialisation

- *Often:* independent, uninformed random picking from given search space.
- *But:* can also use multiple runs of construction heuristic.

Recombination

- Typically repeatedly selects a set of *parents* from current population and generates *offspring* candidate solutions from these by means of *recombination operator*.
- *Recombination operators* are generally based on *linear representation* of candidate solutions and piece together *offspring* from fragments of *parents*.

Initialisation

- *Often:* independent, uninformed random picking from given search space.
- *But:* can also use multiple runs of construction heuristic.

Recombination

- Typically repeatedly selects a set of *parents* from current population and generates *offspring* candidate solutions from these by means of *recombination operator*.
- *Recombination operators* are generally based on *linear representation* of candidate solutions and piece together *offspring* from fragments of *parents*.

Example: One-point binary crossover operator

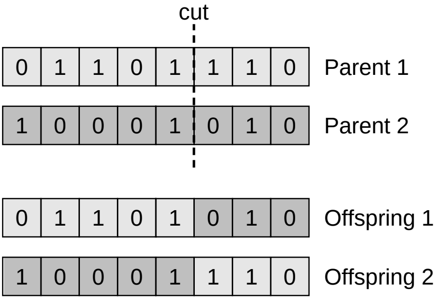
Given two parent candidate solutions $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_n$:

1. choose index i from set $\{2, \ldots, n\}$ uniformly at random;
2. define offspring as $x_1 \ldots x_{i-1} y_i \ldots y_n$ and $y_1 \ldots y_{i-1} x_i \ldots x_n$.

Example: One-point binary crossover operator

Given two parent candidate solutions $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_n$:

1. choose index i from set $\{2, \ldots, n\}$ uniformly at random;
2. define offspring as $x_1 \ldots x_{i-1} y_i \ldots y_n$ and $y_1 \ldots y_{i-1} x_i \ldots x_n$.



Mutation

- *Goal:* Introduce relatively small perturbations in candidate solutions in current population + offspring obtained from *recombination*.
- Typically, perturbations are applied stochastically and independently to each candidate solution; amount of perturbation is controlled by *mutation rate*.
- Can also use *subsidiary selection function* to determine subset of candidate solutions to which mutation is applied.
- In the past, the role of mutation (as compared to recombination) in high-performance evolutionary algorithms has been often underestimated [Bäck, 1996].

Mutation

- *Goal:* Introduce relatively small perturbations in candidate solutions in current population + offspring obtained from *recombination*.
- Typically, perturbations are applied stochastically and independently to each candidate solution; amount of perturbation is controlled by *mutation rate*.
- Can also use *subsidiary selection function* to determine subset of candidate solutions to which mutation is applied.
- In the past, the role of mutation (as compared to recombination) in high-performance evolutionary algorithms has been often underestimated [Bäck, 1996].

Mutation

- *Goal:* Introduce relatively small perturbations in candidate solutions in current population + offspring obtained from *recombination*.
- Typically, perturbations are applied stochastically and independently to each candidate solution; amount of perturbation is controlled by *mutation rate*.
- Can also use *subsidiary selection function* to determine subset of candidate solutions to which mutation is applied.
- In the past, the role of mutation (as compared to recombination) in high-performance evolutionary algorithms has been often underestimated [Bäck, 1996].

Mutation

- *Goal:* Introduce relatively small perturbations in candidate solutions in current population + offspring obtained from *recombination*.
- Typically, perturbations are applied stochastically and independently to each candidate solution; amount of perturbation is controlled by *mutation rate*.
- Can also use *subsidiary selection function* to determine subset of candidate solutions to which mutation is applied.
- In the past, the role of mutation (as compared to recombination) in high-performance evolutionary algorithms has been often underestimated [Bäck, 1996].

Selection (1)

- Determines population for next cycle (*generation*) of the algorithm by selecting individual candidate solutions from current population + new candidate solutions obtained from *recombination*, *mutation* (+ *subsidiary local search*).
- *Goal:* Obtain population of high-quality solutions while maintaining *population diversity*.
- Selection is based on evaluation function (*fitness*) of candidate solutions such that better candidate solutions have a higher chance of 'surviving' the selection process.

Selection (1)

- Determines population for next cycle (*generation*) of the algorithm by selecting individual candidate solutions from current population + new candidate solutions obtained from *recombination*, *mutation* (+ *subsidiary local search*).
- *Goal:* Obtain population of high-quality solutions while maintaining *population diversity*.
- Selection is based on evaluation function (*fitness*) of candidate solutions such that better candidate solutions have a higher chance of 'surviving' the selection process.

Selection (1)

- Determines population for next cycle (*generation*) of the algorithm by selecting individual candidate solutions from current population + new candidate solutions obtained from *recombination*, *mutation* (+ *subsidiary local search*).
- *Goal:* Obtain population of high-quality solutions while maintaining *population diversity*.
- Selection is based on evaluation function (*fitness*) of candidate solutions such that better candidate solutions have a higher chance of 'surviving' the selection process.

Selection (2)

- Many selection schemes involve probabilistic choices, *e.g.*, *roulette wheel selection*, where the probability of selecting any candidate solution s is proportional to its fitness value, $g(s)$.
- It is often beneficial to use *elitist selection strategies*, which ensure that the best candidate solutions are always selected.

Subsidiary local search

- Often useful and necessary for obtaining high-quality candidate solutions.
- Typically consists of selecting some or all individuals in the given population and applying an *iterative improvement procedure* to each element of this set independently.

Selection (2)

- Many selection schemes involve probabilistic choices, *e.g.*, *roulette wheel selection*, where the probability of selecting any candidate solution s is proportional to its fitness value, $g(s)$.
- It is often beneficial to use *elitist selection strategies*, which ensure that the best candidate solutions are always selected.

Subsidiary local search

- Often useful and necessary for obtaining high-quality candidate solutions.
- Typically consists of selecting some or all individuals in the given population and applying an *iterative improvement procedure* to each element of this set independently.

Selection (2)

- Many selection schemes involve probabilistic choices, *e.g.*, *roulette wheel selection*, where the probability of selecting any candidate solution s is proportional to its fitness value, $g(s)$.
- It is often beneficial to use *elitist selection strategies*, which ensure that the best candidate solutions are always selected.

Subsidiary local search

- Often useful and necessary for obtaining high-quality candidate solutions.
- Typically consists of selecting some or all individuals in the given population and applying an *iterative improvement procedure* to each element of this set independently.

Selection (2)

- Many selection schemes involve probabilistic choices, *e.g.*, *roulette wheel selection*, where the probability of selecting any candidate solution s is proportional to its fitness value, $g(s)$.
- It is often beneficial to use *elitist selection strategies*, which ensure that the best candidate solutions are always selected.

Subsidiary local search

- Often useful and necessary for obtaining high-quality candidate solutions.
- Typically consists of selecting some or all individuals in the given population and applying an *iterative improvement procedure* to each element of this set independently.

Example: A memetic algorithm for SAT (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*; *evaluation function:* number of unsatisfied clauses in F.
- *Note:* truth assignments can be naturally represented as bit strings.
- Use population of k truth assignments; *initialise* by (independent) Uninformed Random Picking.

Example: A memetic algorithm for SAT (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*; *evaluation function:* number of unsatisfied clauses in F.
- *Note:* truth assignments can be naturally represented as bit strings.
- Use population of k truth assignments; *initialise* by (independent) Uninformed Random Picking.

Example: A memetic algorithm for SAT (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*; *evaluation function:* number of unsatisfied clauses in F.
- *Note:* truth assignments can be naturally represented as bit strings.
- Use population of k truth assignments; *initialise* by (independent) Uninformed Random Picking.

Example: A memetic algorithm for SAT (2)

- **Recombination:** Add offspring from $n/2$ (independent) one-point binary crossovers on pairs of randomly selected assignments from population to current population ($n =$ number of variables in F).
- **Mutation:** Flip μ randomly chosen bits of each assignment in current population (*mutation rate* μ: parameter of the algorithm); this corresponds to μ steps of Uninformed Random Walk; mutated individuals are added to current population.
- **Selection:** Selects the k best assignments from current population (simple *elitist selection mechanism*).

Example: A memetic algorithm for SAT (2)

- **Recombination:** Add offspring from $n/2$ (independent) one-point binary crossovers on pairs of randomly selected assignments from population to current population (n = number of variables in F).
- **Mutation:** Flip μ randomly chosen bits of each assignment in current population (*mutation rate* μ: parameter of the algorithm); this corresponds to μ steps of Uninformed Random Walk; mutated individuals are added to current population.
- **Selection:** Selects the k best assignments from current population (simple *elitist selection mechanism*).

Example: A memetic algorithm for SAT (2)

- **Recombination:** Add offspring from $n/2$ (independent) one-point binary crossovers on pairs of randomly selected assignments from population to current population ($n =$ number of variables in F).
- **Mutation:** Flip μ randomly chosen bits of each assignment in current population (*mutation rate* μ: parameter of the algorithm); this corresponds to μ steps of Uninformed Random Walk; mutated individuals are added to current population.
- **Selection:** Selects the k best assignments from current population (simple *elitist selection mechanism*).

Example: A memetic algorithm for SAT (3)

- **Subsidiary local search:** Applied after *initialisation*, *recombination* and *mutation*; performs *iterative best improvement* search on each individual assignment independently until local minimum is reached.
- **Termination:** upon finding model of F or after bound on number of cycles (*generations*) is reached.

Note: This algorithm does not reach state-of-the-art performance, but many variations are possible (few of which have been explored).

Example: A memetic algorithm for SAT (3)

- **Subsidiary local search:** Applied after *initialisation*, *recombination* and *mutation*; performs *iterative best improvement* search on each individual assignment independently until local minimum is reached.

- **Termination:** upon finding model of F or after bound on number of cycles (*generations*) is reached.

Note: This algorithm does not reach state-of-the-art performance, but many variations are possible (few of which have been explored).

Example: A memetic algorithm for SAT (3)

- **Subsidiary local search:** Applied after *initialisation*, *recombination* and *mutation*; performs *iterative best improvement* search on each individual assignment independently until local minimum is reached.
- **Termination:** upon finding model of F or after bound on number of cycles (*generations*) is reached.

Note: This algorithm does not reach state-of-the-art performance, but many variations are possible (few of which have been explored).

Types of evolutionary algorithms (1)

- *Genetic Algorithms (GAs)* [Holland, 1975; Goldberg, 1989]:
  - have been applied to a very broad range of (mostly discrete) combinatorial problems;
  - often encode candidate solutions as bit strings of fixed length, which is now known to be disadvantagous for combinatorial problems such as the TSP.

Note: There are some interesting theoretical results for GAs (*e.g.*, *Schema Theorem*), but – as for SA – their practical relevance is rather limited.

Types of evolutionary algorithms (1)

- *Genetic Algorithms (GAs)* [Holland, 1975; Goldberg, 1989]:
  - have been applied to a very broad range of (mostly discrete) combinatorial problems;
  - often encode candidate solutions as bit strings of fixed length, which is now known to be disadvantagous for combinatorial problems such as the TSP.

Note: There are some interesting theoretical results for GAs (*e.g.*, *Schema Theorem*), but – as for SA – their practical relevance is rather limited.

Types of evolutionary algorithms (1)

- *Genetic Algorithms (GAs)* [Holland, 1975; Goldberg, 1989]:
  - have been applied to a very broad range of (mostly discrete) combinatorial problems;
  - often encode candidate solutions as bit strings of fixed length, which is now known to be disadvantagous for combinatorial problems such as the TSP.

Note: There are some interesting theoretical results for GAs (*e.g.*, *Schema Theorem*), but – as for SA – their practical relevance is rather limited.

Types of evolutionary algorithms (2)

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
  - orginally developed for (continuous) numerical optimisation problems;
  - operate on more natural representations of candidate solutions;
  - use *self-adaptation* of perturbation strength achieved by *mutation*;
  - typically use *elitist deterministic selection*.
- *Evolutionary Programming* [Fogel *et al.*, 1966]:
  - similar to Evolution Strategies (developed independently), but typically does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.

Types of evolutionary algorithms (2)

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
  - orginally developed for (continuous) numerical optimisation problems;
  - operate on more natural representations of candidate solutions;
  - use *self-adaptation* of perturbation strength achieved by *mutation*;
  - typically use *elitist deterministic selection*.
- *Evolutionary Programming* [Fogel *et al.*, 1966]:
  - similar to Evolution Strategies (developed independently), but typically does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.

Types of evolutionary algorithms (2)

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
  - orginally developed for (continuous) numerical optimisation problems;
  - operate on more natural representations of candidate solutions;
  - use *self-adaptation* of perturbation strength achieved by *mutation*;
  - typically use *elitist deterministic selection*.
- *Evolutionary Programming* [Fogel *et al.*, 1966]:
  - similar to Evolution Strategies (developed independently), but typically does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.

Types of evolutionary algorithms (2)

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
  - orginally developed for (continuous) numerical optimisation problems;
  - operate on more natural representations of candidate solutions;
  - use *self-adaptation* of perturbation strength achieved by *mutation*;
  - typically use *elitist deterministic selection*.
- *Evolutionary Programming* [Fogel *et al.*, 1966]:
  - similar to Evolution Strategies (developed independently), but typically does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.

Types of evolutionary algorithms (2)

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
  - orginally developed for (continuous) numerical optimisation problems;
  - operate on more natural representations of candidate solutions;
  - use *self-adaptation* of perturbation strength achieved by *mutation*;
  - typically use *elitist deterministic selection*.
- *Evolutionary Programming* [Fogel *et al.*, 1966]:
  - similar to Evolution Strategies (developed independently), but typically does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.