Faculty of Information Technology

Hybrid Stochastic Local Search Methods

FIT4012 Advanced topics in computational science

This material is based on the book 'Stochastic Local Search: Foundations and Applications' by Holger H. Hoos and Thomas Stützle (Morgan Kaufmann, 2004) - see www.sls-book.net for further information.

Neighbourhood

- Local minima depend on g & neighbourhood relation N.
- Larger neighbourhoods $N(s)$ induce
  - neighbourhood 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.

Neighbourhood

- Local minima depend on g & neighbourhood relation N.
- Larger neighbourhoods $N(s)$ induce
  - neighbourhood 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.

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

Neighbourhood Pruning

- *Idea:* Reduce size of neighbourhoods by excluding neighbours that are likely (or guaranteed) not to yield improvements in g.
- Crucial for large neighbourhoods, but can also be very useful for small neighbourhoods (linear in instance size).

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

Neighbourhood Pruning

- *Idea:* Reduce size of neighbourhoods by excluding neighbours that are likely (or guaranteed) not to yield improvements in g.
- Crucial for large neighbourhoods, but can also be very useful for small neighbourhoods (linear in instance size).

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

Neighbourhood Pruning

- *Idea:* Reduce size of neighbourhoods by excluding neighbours that are likely (or guaranteed) not to yield improvements in g.
- Crucial for large neighbourhoods, but can also be very useful for small neighbourhoods (linear in instance size).

Example: Candidate lists for the TSP

- *Intuition:* High-quality solutions are likely to include short edges.
- *Candidate list* of vertex v: list of v's nearest neighbours 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 (*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 $g^* := \min\{g(s') \mid s' \in N(s)\}$.

- Requires evaluation of all neighbours in each step.
- *First Improvement:* Evaluate neighbours in fixed order, choose first improving step encountered.
- Can be much more efficient than Best Improvement; order of evaluation can have significant impact on performance.

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 neighbourhood relation.
- Use k neighbourhood relations $N_1, \ldots, N_k$, (typically) ordered according to increasing neighbourhood size.
- Upon termination, candidate solution is locally optimal with respect to all neighbourhoods.

Note:

- VND often performs substantially better than simple II or II in large neighbourhoods [Hansen and Mladenović, 1999]
- Many variants exist that switch between neighbourhoods 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

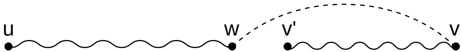
- **Key idea:** *Complex steps* in large neighbourhoods = variable-length sequences of *simple steps* in small neighbourhood.

Example: The Lin-Kernighan (LK) for TSP (1)[1]

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

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

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



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

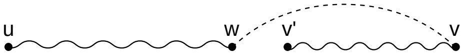
[1] http://www.akira.ruc.dk/ keld/research/LKH/

Example: The Lin-Kernighan (LK) for TSP (1)[1]

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

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

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



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

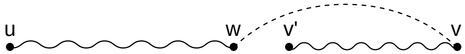
[1] http://www.akira.ruc.dk/ keld/research/LKH/

Example: The Lin-Kernighan (LK) for TSP (1)[1]

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

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

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



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

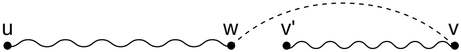
[1] http://www.akira.ruc.dk/ keld/research/LKH/

Example: The Lin-Kernighan (LK) for TSP (1)[1]

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

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

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



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

[1] http://www.akira.ruc.dk/ keld/research/LKH/

Application of VD search algorithms

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].

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 (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 $s := r$

- *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.
- 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

- 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.
- 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

- 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.
- 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

- 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.
- 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

- 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.
- 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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

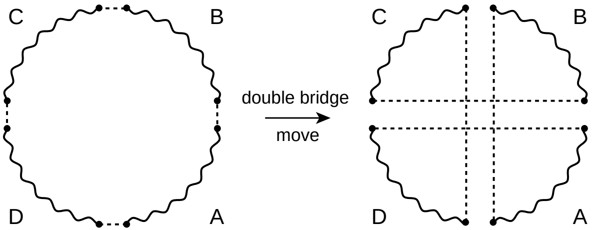
- Always accept the *better* of the two candidate solutions $\Rightarrow$ ILS performs Iterative Improvement in the space of local optima reached by subsidiary local search.
- Always accept the *more recent* of the two candidate solutions $\Rightarrow$ ILS performs random walk in the space of local optima 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

- **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

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



- 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.
- 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.
- 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]

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 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 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 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*

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*, add a solution component selected uniformly at random from a *restricted candidate list (RCL)*.
- RCLs are constructed in each step using 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*, add a solution component selected uniformly at random from a *restricted candidate list (RCL)*.
- RCLs are constructed in each step using 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*, add a solution component selected uniformly at random from a *restricted candidate list (RCL)*.
- RCLs are constructed in each step using 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*, add a solution component selected uniformly at random from a *restricted candidate list (RCL)*.
- RCLs are constructed in each step using 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).

- 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 (*semi-greedy heuristics*) typically do not reach the performance of GRASP with perturbative local search.

- 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 (*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 an 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)

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

Given: CNF formula F over variables $x_1, \ldots, x_n$

Subsidiary constructive search:

- start from an 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)

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

Given: CNF formula F over variables $x_1, \ldots, x_n$

Subsidiary constructive search:

- start from an 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)

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

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]

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 (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

(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

(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

(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

(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

(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

- *Subsidiary local search*: iterative improvement based on standard 2-exchange neighbourhood (until local min.).

- *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.

- Decay mechanism ($0 < \rho \leq 1$) helps to avoid unlimited growth of weights and lets algorithm forget past experience reflected in weights.

Example: A simple AICS algorithm for the TSP

- *Subsidiary local search*: iterative improvement based on standard 2-exchange neighbourhood (until local min.).

- *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.

- Decay mechanism ($0 < \rho \leq 1$) helps to avoid unlimited growth of weights and lets algorithm forget past experience reflected in weights.

Example: A simple AICS algorithm for the TSP

- *Subsidiary local search*: iterative improvement based on standard 2-exchange neighbourhood (until local min.).

- *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.

- Decay mechanism ($0 < \rho \leq 1$) helps to avoid unlimited growth of weights and lets algorithm forget past experience reflected in weights.

Example: A simple AICS algorithm for the TSP

- *Subsidiary local search*: iterative improvement based on standard 2-exchange neighbourhood (until local min.).
- *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.
- Decay mechanism ($0 < \rho \leq 1$) helps to avoid unlimited growth of weights and lets algorithm forget past experience reflected in weights.

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.

- The use of populations provides a generic way to achieve search diversification.
- Population-based SLS methods treat 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.

- The use of populations provides a generic way to achieve search diversification.
- Population-based SLS methods treat 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.

- The use of populations provides a generic way to achieve search diversification.
- Population-based SLS methods treat 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.

- The use of populations provides a generic way to achieve search diversification.
- Population-based SLS methods treat sets of candidate solutions as search positions.

Ant Colony Optimisation

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

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

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

- 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.

- 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.

- 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.

- 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.

- 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

(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

(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

(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

(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

(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

- *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

- *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

- *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

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

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

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

These and other enhancements provide the basis for advanced ACO methods, such as:

- 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].
- 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].

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].
- 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*

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

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

$\Rightarrow$ *Memetic Algorithms* (*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* (*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

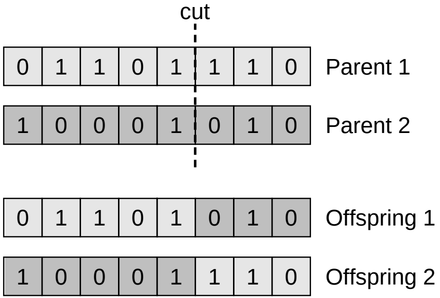
Given two parent candidate solutions $x_1 x_2 \dots x_n$ and $y_1 y_2 \dots y_n$:

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

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 often been 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 often been 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 often been 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 often been underestimated [Bäck, 1996].

Selection

- 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

- 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

- 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

- 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.

Selection

- 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.

Selection

- 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.

Selection

- 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.

Example: A memetic algorithm for SAT

- *Search space:* set of all truth assignments for propositional variables in given CNF formula F;
- *solution set:* satisfying assignments of F
- *neighbourhood relation:* 1-flip
- *evaluation function:* number of unsatisfied clauses in F.
- 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

- *Search space:* set of all truth assignments for propositional variables in given CNF formula F;
- *solution set:* satisfying assignments of F
- *neighbourhood relation:* 1-flip
- *evaluation function:* number of unsatisfied clauses in F.
- 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

- *Search space:* set of all truth assignments for propositional variables in given CNF formula F;
- *solution set:* satisfying assignments of F
- *neighbourhood relation:* 1-flip
- *evaluation function:* number of unsatisfied clauses in F.
- 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

- *Search space:* set of all truth assignments for propositional variables in given CNF formula F;
- *solution set:* satisfying assignments of F
- *neighbourhood relation:* 1-flip
- *evaluation function:* number of unsatisfied clauses in F.
- 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

- *Search space:* set of all truth assignments for propositional variables in given CNF formula F;
- *solution set:* satisfying assignments of F
- *neighbourhood relation:* 1-flip
- *evaluation function:* number of unsatisfied clauses in F.
- 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

- *Search space:* set of all truth assignments for propositional variables in given CNF formula F;
- *solution set:* satisfying assignments of F
- *neighbourhood relation:* 1-flip
- *evaluation function:* number of unsatisfied clauses in F.
- 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

- *Search space:* set of all truth assignments for propositional variables in given CNF formula F;
- *solution set:* satisfying assignments of F
- *neighbourhood relation:* 1-flip
- *evaluation function:* number of unsatisfied clauses in F.
- 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

- **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

- **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

- **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

- **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

- **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

- **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

- *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 disadvantageous for combinatorial problems such as the TSP.

Note: There are some interesting theoretical results for GAs (*e.g.*, *Schema Theorem*).

Types of evolutionary algorithms

- *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 disadvantageous for combinatorial problems such as the TSP.

Note: There are some interesting theoretical results for GAs (*e.g.*, *Schema Theorem*).

Types of evolutionary algorithms

- *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 disadvantageous for combinatorial problems such as the TSP.

Note: There are some interesting theoretical results for GAs (*e.g.*, *Schema Theorem*).

Types of evolutionary algorithms

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
  - originally 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, but does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.

Types of evolutionary algorithms

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
  - originally 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, but does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.

Types of evolutionary algorithms

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
    - originally 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, but does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.

Types of evolutionary algorithms

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
  - originally 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, but does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.

Types of evolutionary algorithms

- *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981]:
  - originally 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, but does not make use of *recombination* and uses *stochastic selection* based on *tournament mechanisms*.