Variable Neighbourhood Search (VNS)

Key Idea: systematically change neighbourhoods during search

Motivation:

- *recall*: changing neighbourhoods can help escape local optima
- a global optimum is locally optimal w.r.t. *all* neighbourhood structures
- *principle of VNS*: change the neighbourhood during the search

- main VNS variants
  - variable neighbourhood descent (VND, already discussed)
  - basic variable neighborhood search
  - reduced variable neighborhood search
  - variable neighborhood decomposition search

How to generate the various neighborhood structures?

- for many problems different neighborhood structures (local searches) exist / are in use
- use k-exchange neighborhoods; these can be naturally extended
- many neighborhood structures are associated with distance measures: define neighbourhoods in dependence of the distances between solutions

basic VNS

- uses neighborhood structures $\mathcal{N}_k, k = 1, \ldots, k_{max}$
- iterative improvement in $\mathcal{N}_1$
- other neighborhoods are explored only randomly
- exploration in other neighborhoods are perturbations in the ILS sense
- perturbation is systematically varied
- acceptance criterion Better$(s^*, s^{*\prime})$

Basic VNS — Procedural view

procedure *basic VNS*
  $s_0 \leftarrow$ GenerateInitialSolution, *choose* $\{\mathcal{N}_k\}, k = 1, \ldots, k_{\max}$
  repeat
    $s' \leftarrow RandomSolution(\mathcal{N}_k(s^*))$
    $s^{*\prime} \leftarrow \text{LocalSearch}(s')$ % *local search w.r.t.* $\mathcal{N}_1$
    if $f(s^{*\prime}) < f(s^*)$ **then**
      $s^* \leftarrow s^{*\prime}$
      $k \leftarrow 1$
    else
      $k \leftarrow k + 1$
  until *termination condition*
end

Basic VNS — variants

- order of the neighborhoods
  - forward VNS: start with $k = 1$ and increase k by one if no better solution is found; otherwise set $k \leftarrow 1$
  - backward VNS: start with $k = k_{max}$ and decrease k by one if no better solution is found
  - extended version: parameters k_{min} and k_{step}; set $k \leftarrow k_{min}$ and increase k by k_{step} if no better solution is found
- acceptance of worse solutions
  - Skewed VNS: accept if

$$f(s^{*\prime}) - \alpha d(s^*, s^{*\prime}) < f(s^*)$$

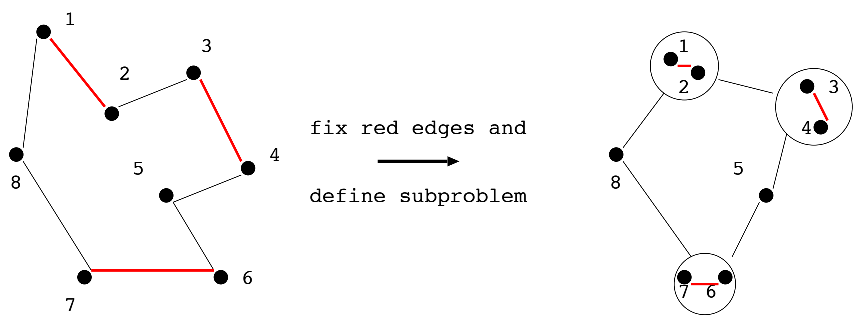
$d(s^*, s^{*\prime})$ measures distance between candidate solutions

Reduced VNS

- same as basic VNS except that no iterative improvement procedure is applied
- only explores randomly different neighborhoods
- goal: reach quickly good quality solutions for large instances

Variable Neighborhood Decomposition Search

- *central idea*
  - generate subproblems by keeping all but k solution components fixed
  - apply local search only to the k "free" components



- related approaches: POPMUSIC, MIMAUSA, etc.

VNDS — Procedural view

procedure *VNDS*
  $s_0 \leftarrow$ GenerateInitialSolution, *choose* $\{\mathcal{N}_k\}, k = 1, \ldots, k_{\max}$
  repeat
    $s' \leftarrow$ *RandomSolution*$(\mathcal{N}_k(s))$
    $t \leftarrow$ *DetermineFreeComponents*(s', s)
    $t^* \leftarrow$ LocalSearch(t) % *local search w.r.t.* $\mathcal{N}_k$
    $s'' \leftarrow$ *InjectComponents*(t^*, s')
    if $f(s'') < f(s)$ **then**
      $s \leftarrow s''$
      $k \leftarrow 1$
    else
      $k \leftarrow k + 1$
  until *termination condition*
end

relationship between ILS and VNS

- the two SLS methods are based on different underlying "philosophies"
- they are similar in many respects
- ILS apears to be in literature more flexible w.r.t. optimization of the interaction of modules
- VNS gives place to approaches like VND for obtaining more powerful local search approaches

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.

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 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.)

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

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, path relinking and other SLS methods

Iterated Greedy

Key Idea: iterate over greedy construction heuristics through destruction and construction phases

Motivation:

- start solution construction from partial solutions to avoid reconstruction from scratch
  - keep features of the best solutions to improve solution quality
  - if few construction steps are to be executed, greedy heuristics are fast
- adding a subsidiary local search phase may further improve performance

Iterated Greedy (IG):

While *termination criterion* is not satisfied:
  generate candidate solution s using
    subsidiary greedy constructive search
While termination criterion is not satisfied:
  $r := s$
  apply *solution destruction* on s
  perform *subsidiary greedy constructive search* on s

  based on *acceptance criterion*,
    keep s or revert to $s := r$

Note:

- subsidiary local search after solution reconstruction can substantially improve performance

Iterated Greedy (IG):

While *termination criterion* is not satisfied:
    generate candidate solution s using
        subsidiary greedy constructive search
While termination criterion is not satisfied:
    $r := s$
    apply *solution destruction* on s
    perform *subsidiary greedy constructive search* on s
    perform *subsidiary local search* on s
    based on *acceptance criterion*,
        keep s or revert to $s := r$

Note:

- subsidiary local search after solution reconstruction can substantially improve performance

IG—main issues

- destruction phase
  - fixed vs. variable size of destruction
  - stochastic vs. deterministic destruction
  - uniform vs. biased destruction
- construction phase
  - not every construction heuristic is necessarily useful
  - typically, adaptive construction heuristics preferable
  - speed of the construction heuristic is an issue
- acceptance criterion
  - very much the same issue as in ILS

IG — enhancements

- usage of history information to bias destructive/constructive phase
- combination with local search in the constructive phase
- use local search to improve full solutions
  $\rightsquigarrow$ destruction / construction phases can be seen as a perturbation mechanism in ILS
- usage of elements of tree search algorithms such as lower bounds on the completion of a solution or constraint propagation techniques

Example: IG for SCP [Jacobs, Brusco, 1995]

- **Given:**
  - finite set $\mathbf{A} = \{a_1, \ldots, a_m\}$ of objects
  - family $\mathbf{B} = \{B_1, \ldots B_n\}$ of subsets of $\mathbf{A}$ that covers $\mathbf{A}$
  - weight function $w : \mathbf{B} \mapsto R^+$
- $\mathbf{C} \subseteq \mathbf{B}$ *covers* $\mathbf{A}$ if every element in $\mathbf{A}$ appears in at least one set in $\mathbf{C}$, i.e. if $\bigcup \mathbf{C} = \mathbf{A}$
- **Goal:**
  - find a subset $\mathbf{C}^* \subseteq \mathbf{B}$ of minimum total weight that covers $\mathbf{A}$.

Example: IG for SCP, continued ..

- assumption: all subsets from $\mathbf{B}$ are ordered according to nondecreasing costs
- construct initial solution using a greedy heuristic based on two steps
  - randomly select an uncovered object a_i
  - add the lowest cost subset that covers a_i
- the *destruction phase* removes a fixed number of $k_1|\mathbf{C}|$ subsets; k_1 is a parameter

- the *construction phase* proceeds as
  - build a candidate set containing subsets with cost of less than $k_2 \cdot f(\mathbf{C})$
  - compute cover value $\gamma_j = w_j/d_j$
    d_j: number of additional objects covered by adding subset b_j
  - add a subset with minimum cover value
- complete solution is post-processed by removing redundant subsets
- *acceptance criterion*: Metropolis condition from PII
- computational experience
  - good performance with this simple approach
  - more recent IG variants are state-of-the-art algorithms for SCP

- IG has been re-invented several times; names include
  - simulated annealing, ruin–and–recreate, iterative flattening, iterative construction search, large neighborhood search, ..
- close relationship to iterative improvement in large neighbourhoods
- analogous extension to greedy heuristics as ILS to local search
- for some applications so far excellent results
- can give lead to effective combinations of tree search and local search heuristics

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 constructive approach where a population of agents – *(artificial) ants* – communicate via common memory – *(simulated) pheromone trails*.

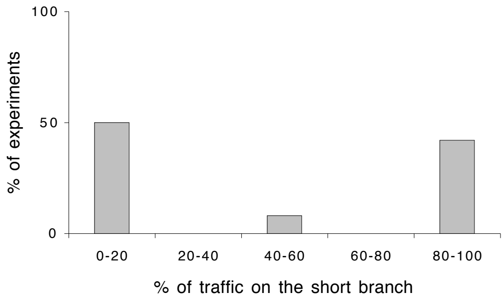
Inspired by foraging behaviour of real ants:

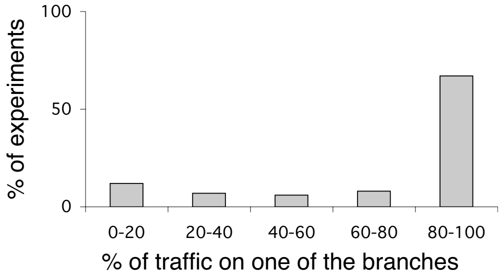
- Ants often communicate via chemicals known as *pheromones*, which are deposited on the ground in the form of trails. (This is a form of *stigmergy*: indirect communication via manipulation of a common environment.)
- Pheromone trails provide the basis for (stochastic) trail-following behaviour underlying,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

Double bridge experiments Deneubourg++

- laboratory colonies of *Iridomyrmex humilis*
- ants deposit pheromone while walking from food sources to nest *and* vice versa
- ants tend to choose, in probability, paths marked by strong pheromone concentrations

- equal length bridges: convergence to a single path
- different length paths: convergence to short path





- a stochastic model was derived from the experiments and verified in simulations
- functional form of transition probability

$$p_{i,a} = \frac{(k + \tau_{i,a})^\alpha}{(k + \tau_{i,a})^\alpha + (k + \tau_{i,a'})^\alpha}$$

- $p_{i,a}$: probability of choosing branch a when being at decision point i
  $\tau_{i,a}$: corresponding pheromone concentration
- good fit to experimental data with $\alpha = 2$

Towards artificial ants

- real ant colonies are solving *shortest path problems*
- ACO takes elements from real ant behavior to solve more complex problems than real ants
- In ACO, artificial ants are *stochastic solution construction procedures* that probabilistically build solutions exploiting
  - (artificial) *pheromone trails* that change at run time to reflect the agents' acquired search experience
  - *heuristic information* on the problem instance being solved

Ant Colony Optimisation

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):

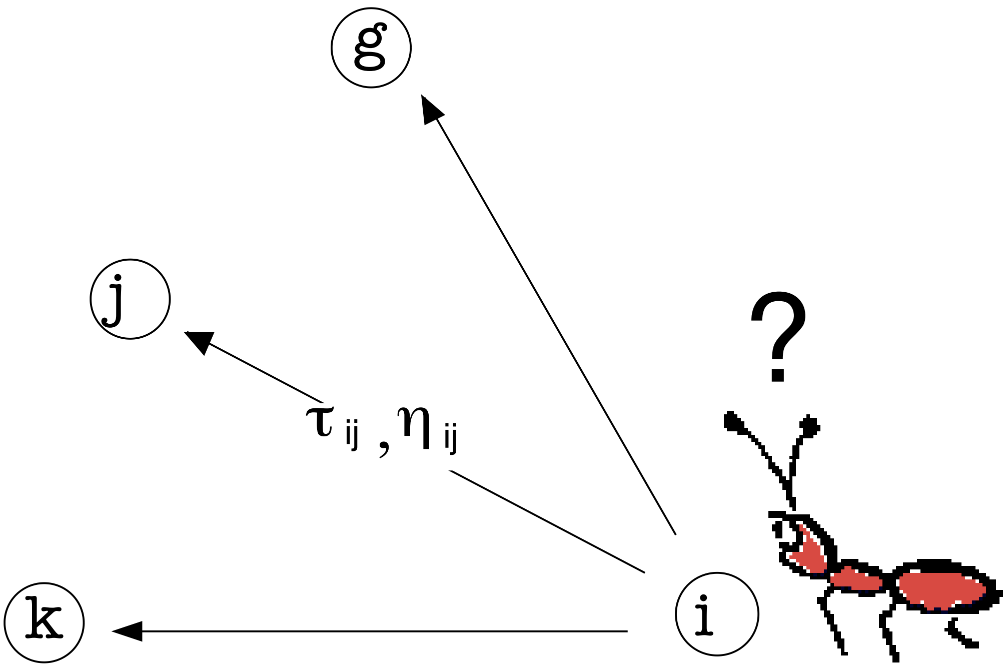
initialise pheromone trails

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*

Note:

- In each cycle, each ant creates one candidate solution using a *constructive search procedure*.
- *Subsidiary local search* is applied to individual candidate solutions.
- 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 (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 iterations ($=$ 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)

ACO algorithm	*Authors*	*Year*	*TSP*
Ant System	Dorigo, Maniezzo, Colorni	1991	yes
Elitist AS	Dorigo	1992	yes
Ant-Q	Gambardella & Dorigo	1995	yes
Ant Colony System	Dorigo & Gambardella	1996	yes
$\mathcal{MMAS}$	Stützle & Hoos	1996	yes
Rank-based AS	Bullnheimer, Hartl, Strauss	1997	yes
ANTS	Maniezzo	1998	no
Best-Worst AS	Cordón, et al.	2000	yes
Hyper-cube ACO	Blum, Roli, Dorigo	2001	no
Population-based ACO	Guntsch, Middendorf	2002	yes
Beam-ACO	Blum	2004	no

$\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

- extension of Ant System with stronger exploitation of best solutions and additional mechanism to avoid search stagnation
- *exploitation*: only the iteration-best or best-so-far ant deposit pheromone

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \Delta\tau_{ij}^{best}$$

- frequently, a schedule for choosing between iteration-best and best-so-far update is used

- *stagnation avoidance:* additional limits on the feasible pheromone trails
  - for all $\tau_{ij}(t)$ we have: $\tau_{min} \leq \tau_{ij}(t) \leq \tau_{max}$
  - counteracts stagnation of search through aggressive pheromone update
  - heuristics for determining τ_{min} and τ_{max}
- *stagnation avoidance 2:* occasional pheromone trail re-initialization when $\mathcal{MM}$AS has converged
- *increase of exploration:* pheromone values are initialized to τ_{max} to have less pronounced differences in selection probabilities

Ant Colony System (ACS) Gambardella, Dorigo 1996, 1997

- pseudo-random proportional action choice rule
  - with probability q_0 an ant k located at city i chooses successor city j with maximal $\tau_{ij}(t) \cdot [\eta_{ij}]^{\beta}$ (*exploitation*)
  - with probability $1 - q_0$ an ant k chooses the successor city j according to action choice rule used in Ant System (*biased exploration*)

- global pheromone update rule (*exploitation*)
  - best-so-far solution T^{gb} deposits pheromone after each iteration

$$\tau_{ij}(t) = (1-\rho) \cdot \tau_{ij}(t-1) + \rho \cdot \Delta\tau_{ij}^{gb}(t-1),$$

    where $\Delta\tau_{ij}^{gb}(t) = 1/L^{gb}$.
  - pheromone update only affects edges in T^{gb}!
  - limited tests with iteration-best solution; however, best-so-far update is recommended

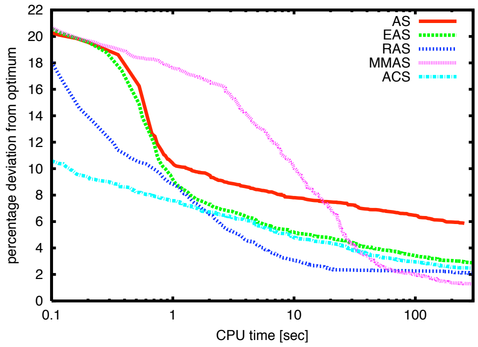
- local pheromone update rule
  - ants "eat away" pheromone on the edges just crossed

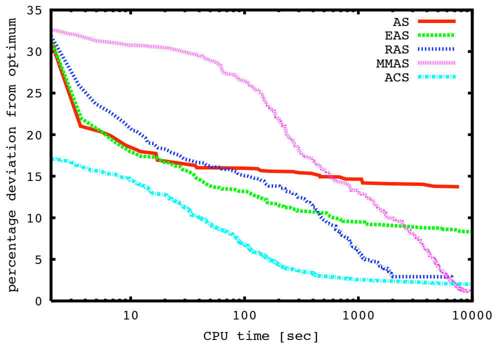
$$\tau_{ij} = (1-\xi) \cdot \tau_{ij} + \xi \cdot \tau_0$$

  - τ_0 is some small constant value
  - increases exploration by reducing the desirability of frequently used edges
  - in Ant-Q (predecessor of ACS): $\tau_0 = \gamma \cdot \max_{l \in \mathcal{N}_j^k}\{\tau_{jl}\}$

solution quality versus time

d198 rat783

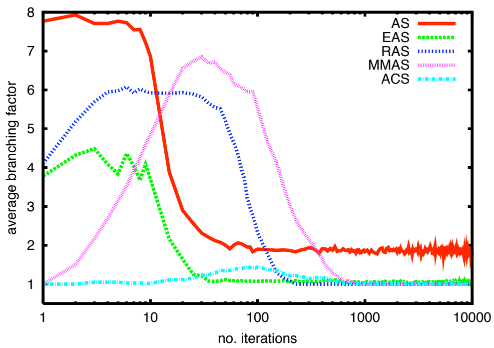


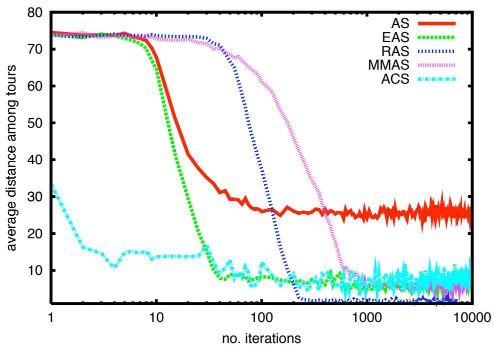


(typical parameter settings for high final solution quality)

behavior of ACO algorithms

average λ-branching factor average distance among tours



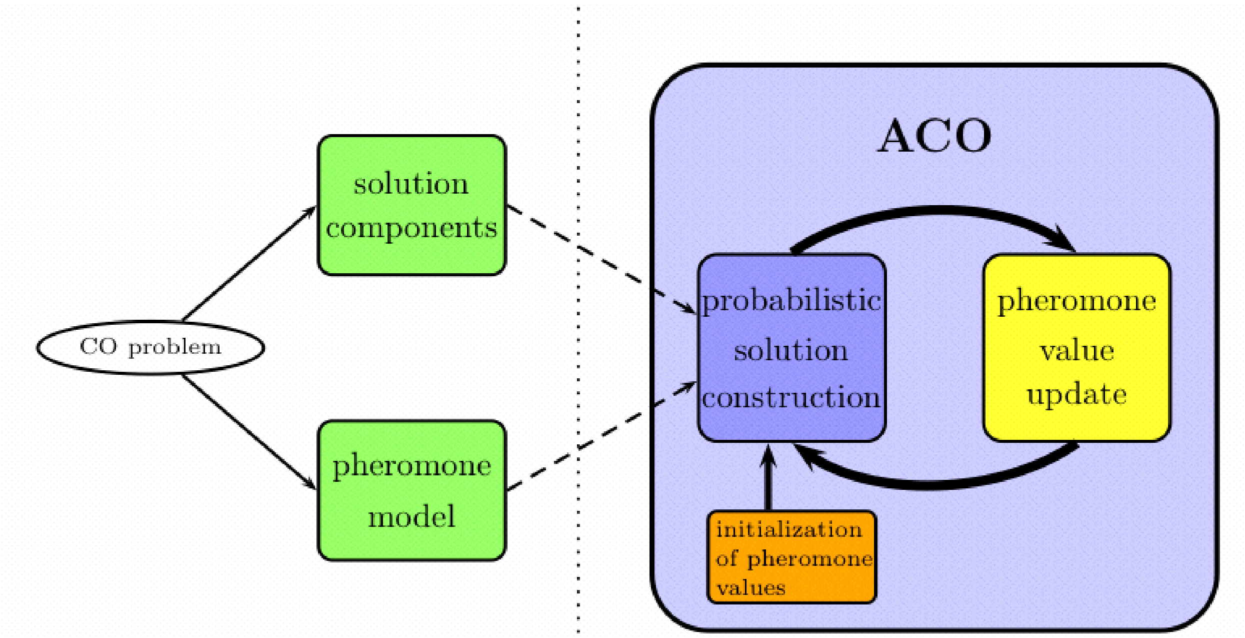


(typical parameter settings for high final solution quality)

Enhancements:

- use of look-ahead in construction phase;
- start of solution construction from partial solutions (memory-based schemes, ideas gleaned from iterated greedy);
- combination of ants with techniques from tree search such as
  - lower bounding information
  - combination with beam search
  - constraint programming (constraint propagation)

Applying ACO



Ant Colony Optimisation . . .

- has been applied very successfully to a wide range of combinatorial problems, including
  - various scheduling problems,
  - various vehicle routing problems, and
  - various bio-informatics problems such as protein–ligand docking
- 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 course on Swarm Intelligence or 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 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.

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

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.

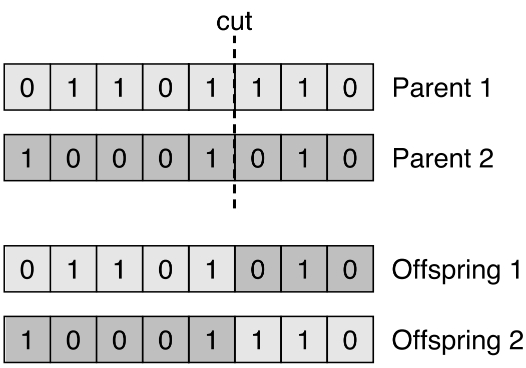
Recombination

- 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*.

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 $y_1 \ldots y_{i-1} x_i \ldots x_n$.



Generalization: two-point, k-point, uniform crossover

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

- *Selection for variation*: determines which of the individual candidate solutions of the current population are chosen to undergo recombination and/or mutation
- *Selection for survival*: 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 (general)

- Many selection schemes involve probabilistic choices, using the idea that better candidate solutions have a higher probability of being chosen.
- examples
  - roulette wheel selection (probability of selecting a candidate solution s is proportional to its fitness value, $g(s)$)
  - tournament selection (choose best of k randomly sampled candidate solutions)
  - rank-based computation of selection probabilities
- the strength of the probabilistic bias determines the *selection pressure*

Selection (survival)

- *generational* replacement versus *overlapping populations*; (extreme case, *steady-state selection*)
- It is often beneficial to use *elitist selection strategies*, which ensure that the best candidate solutions are always selected.
- probabilistic versus deterministic replacement
- quasi-deterministic replacement strategies implemented by classical selections schemes from evolution strategies (a particular type of EAs)
  - λ: number offspring, μ: number parent candidate solutions
  - (μ, λ) strategy: choose best μ of $\lambda > \mu$ offspring
  - $(\mu + \lambda)$ strategy: choose best μ of $\mu + \lambda$ candidate solutions

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 (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).

Problem representation and operators

- simplest choice of candidate solution representation: bitstrings
- advantage: application of simple recombination, mutation operators
- problems with this arise in case of
  - problem constraints (e.g. set covering, graph bi-partitioning)
  - "richer" problem representations are much better suited (e.g. TSP)
- *possible solutions*
  - application of representation- (and problem-) specific recombination and mutation operators
  - application of repair mechanisms to reestablish feasibility of candidate solutions

Memetic algorithm by Merz and Freisleben (MA-MF)

- one of the best studied MAs for the TSP
- first versions proposed in 1996 and further developed until 2001
- main characteristics
  - population initialisation by constructive search
  - exploits an effective LK implementation
  - specialised recombination operator
  - restart operators in case the search is deemed to stagnate
  - standard selection and mutation operators

MA-MF: population initialisation

- each individual of initial population constructed by a randomised variant of the greedy heuristic

  Step 1: choose $n/4$ edges by the following two steps
    - select a vertex $v \in V$ uniformly at random among those that are not yet in partial tour
    - insert shortest (second-shortest) feasible edge incident to v with a probability of $2/3$ $(1/3)$

  Step 2: complete tour using the greedy heuristic

- locally optimise initial tours by LK

MA-MF: recombination

- various specialised crossover operators examined (distance-preserving crossover, greedy crossover)
- crossover operators generate feasible offspring
- best performance by greedy crossover that borrows ideas from greedy heuristic
- one offspring is generated from two parents:
  1. copy fraction of p_e common edges to offspring
  2. add fraction of p_n new short edges not contained in any of the parents
  3. add fraction of p_c shortest edges from parents
  4. complete tour by greedy heuristic
- best performance for $p_e = 1$; $\mu/2$ pairs of tours are chosen uniformly at random from population for recombination

MA-MF: mutation, selection, restart, results

- *mutation* by (usual) double-bridge move
- *selection* done by usual $(\mu + \lambda)$ strategy
  - μ: population size, λ: number of new offspring generated
  - select μ lowest weight tours among $\mu + \lambda$ current tours for next iteration
  - take care that no duplicate tours occur in population
- *partial restart* by strong mutation
  - if average distance is below 10 or did not change for 30 iterations, apply random k-exchange move ($k = 0.1 \cdot n$) plus local search to all individuals except population-best one
- *results:* high solution quality reachable, though not fully competitive with state-of-the-art ILS algorithms for TSP

Memetic algorithm by Walters (MA-W)

- differs in many aspects from other MAs for the TSP
- main differences concern
  - solution representation by nearest neighbour indexing instead of permutation representation
  - usage of general-purpose recombination operators that may generate infeasible offspring
  - repair mechanism is used to restore valid tours from infeasible offspring
  - uses "only" a 3–opt algorithm as subsidiary local search

MA-W: solution representation, initialisation, mutation

- *solution representation* through nearest neighbour indexing
  - tour p represented as vector $s := (s_1, \ldots, s_n)$ such that $s_i = k$ if, and only if, the successor of vertex u_i in p is kth nearest neighbour of u_i
  - leads, however, to some redundancies for symmetric TSPs
- *population initialisation* by choosing randomly nearest neighbour indices
  - three nearest neighbours selected with probability of $0.45, 0.25, 0.15$, respectively
  - in remaining cases index between four and ten chosen uniformly at random
- *mutation* modifies nearest neighbour indices of randomly chosen vertices according to same probability distribution

MA-W: recombination, repair mechanism, results

- *recombination* is based on a slight variation of standard two-point crossover operator
- infeasible candidate solutions from crossover and mutation are repaired
- *repair mechanism* tries to preserve as many edges as possible and replaces an edge e by an edge e' such that $|w(e) - w(e')|$ is minimal
- *results* are interesting considering that "only" 3–opt was used as subsidiary local search; however, worse than state-of-the-art ILS algorithms

Tour merging

- can be seen as an extreme case of MAs
- exploits information collected by high-quality solutions from various ILS runs in a two phases approach
- *phase one*
  - generate a set T of very high quality tours for $G = (V, E, w)$
  - define subgraph $G' = (V, E', w')$, where E' contains all edges in at least one $t \in T$ and w' is original w restricted to E'
- *phase two*
  - determine optimal tour in G'
  - *Note:* general-purpose or specialised algorithm that exploit characteristics of T are applicable

- very high quality solutions can be obtained
  - optimal solution to TSPLIB instance `d15112` in 22 days on a 500 MHz Alpha processor
  - new best-known solutions to instances `brd14051` and `d18512`

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*.