DM841

Discrete Optimization: Heuristics

Metaheuristics

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Randomized (or Stochastic) Local Search

2. Guided Local Search

3. Simulated Annealing

4. Iterated Local Search

5. Tabu Search

6. Variable Neighborhood Search

Escaping Local Optima

Possibilities:

- Restart: re-initialize search whenever a local optimum is encountered.
  (Often rather ineffective due to cost of initialization.)
- Non-improving steps: in local optima, allow selection of candidate solutions with equal or worse evaluation function value, *e.g.*, using minimally worsening steps.
  (Can lead to long walks in *plateaus*, *i.e.*, regions of search positions with identical evaluation function.)
- Diversify the neighborhood

Note: None of these mechanisms is guaranteed to always escape effectively from local optima.

Diversification *vs* Intensification

- Goal-directed and randomized components of LS strategy need to be balanced carefully.
- Intensification: aims at greedily increasing solution quality, *e.g.*, by exploiting the evaluation function.
- Diversification: aims at preventing search stagnation, that is, the search process getting trapped in confined regions.

Examples:

- Iterative Improvement (II): *intensification* strategy.
- Uninformed Random Walk/Picking (URW/P): *diversification* strategy.

Balanced combination of intensification and diversification mechanisms forms the basis for advanced LS methods.

Outline

Greedy Local Search

Key idea: Best improvement = Iterative Improvement (or Hill Climber or Steepest Descent) + Sideways Moves + seldom worsening moves

Algorithm 6.1: GSAT (F)

```
Input      : A CNF formula F
Parameters : Integers MAX-FLIPS, MAX-TRIES
Output     : A satisfying assignment for F, or FAIL
begin
    for i ← 1 to MAX-TRIES do
        σ ← a randomly generated truth assignment for F
        for j ← 1 to MAX-FLIPS do
            if σ satisfies F then return σ                    // success
            v ← a variable flipping which results in the greatest decrease
                (possibly negative) in the number of unsatisfied clauses
            Flip v in σ
    return FAIL                            // no satisfying assignment found
end
```

- GSAT begins with a rapid greedy descent towards a better truth assignment
- then long sequences of sideways moves take place. Sideways moves are moves that do not increase or decrease the total number of unsatisfied clauses. They navigate through plateaux, which is SAT are many and large
- GSAT [Selman et al. 1992] at its times was able to beat complete search algorithms (they were not using CDC)

Randomized Iterative Improvement

aka, Stochastic Hill Climbing

Key idea: Allowed worsening moves: In each search step, with a fixed probability perform an uninformed random walk step instead of an iterative improvement step.
greedy + uniform random walk

Randomized Iterative Improvement (RII):

determine initial candidate solution s
while termination condition is not satisfied **do**
 With probability `wp`:
  choose a neighbor s' of s uniformly at random
 Otherwise:
  choose a neighbor s' of s such that $f(s') < f(s)$ or,
   if no such s' exists, choose s' such that $f(s')$ is minimal
 $s := s'$

Example: Randomized Iterative Improvement for SAT

```
procedure RIISAT(F, wp, maxSteps)
    input: a formula F, probability wp, integer maxSteps
    output: a model φ for F or ∅

    choose assignment φ for F uniformly at random;
    steps := 0;
    while not(φ is not proper) and (steps < maxSteps) do
        with probability wp do
            select x in X uniformly at random and flip;
        otherwise
            select x in X^c uniformly at random from those that
                maximally decrease number of clauses violated;
        change φ;
        steps := steps+1;
    end
    if φ is a model for F then return φ
    else return ∅
    end
end RIISAT
```

X^c set of variables in violated clauses

Note:

- No need to terminate search when local minimum is encountered

  Instead: Impose limit on number of search steps or CPU time, from beginning of search or after last improvement.

- Probabilistic mechanism permits arbitrary long sequences of random walk steps

  Therefore: When run sufficiently long, RII is guaranteed to find (optimal) solution to any problem instance with arbitrarily high probability.

- GWSAT [Selman et al., 1994], was at some point state-of-the-art for SAT.

SAT: an Insight

Theorem

For any satisfiable formula and starting from any truth assignment, there exists a sequence of flips using only variables from unsatisfied clauses such that one obtains a satisfying assignment.

Proof:

- fix a particular satisfying assignment $\bar{\sigma}$.
- Let σ be any truth assignment.
- Every clause not satisfied by σ must contain a variable whose truth value is different in σ and $\bar{\sigma}$.
- Flipping such a variable in σ brings it one step closer to $\bar{\sigma}$. (Note: it might have introduced new violated clauses)
- Repeating this at most n times makes σ identical to $\bar{\sigma}$, thereby turning σ into a satisfying assignment.

Focused Local Search: WalkSAT

```
procedure WalkSAT (F, maxTries, maxSteps, slc)
    input: CNF formula F, positive integers maxTries and maxSteps,
        heuristic function slc
    output: model of F or 'no solution found'
    for try := 1 to maxTries do
        a := randomly chosen assignment of the variables in formula F;
        for step := 1 to maxSteps do
            if a satisfies F then return a end
            c := randomly selected clause unsatisfied under a;
            x := variable selected from c according to heuristic function slc;
            a := a with x flipped;
        end
    end
    return 'no solution found'
end WalkSAT
```

Example of *slc* heuristic: with prob. *wp* select a random move, with prob. $1 - wp$ select the best

Focused local search: WalkSAT

Algorithm 6.2: `Walksat` (F)

```
Input       : A CNF formula F
Parameters  : Integers MAX-FLIPS, MAX-TRIES; noise parameter p ∈ [0,1]
Output      : A satisfying assignment for F, or FAIL
begin
  for i ← 1 to MAX-TRIES do
    σ ← a randomly generated truth assignment for F
    for j ← 1 to MAX-FLIPS do
      if σ satisfies F then return σ                        // success
      C ← an unsatisfied clause of F chosen at random
      if ∃ variable x ∈ C with break-count = 0 then
        v ← x                                               // freebie move
      else
        With probability p:                                 // random walk move
            v ← a variable in C chosen at random
        With probability 1 − p:                             // greedy move
            v ← a variable in C with the smallest break-count
      Flip v in σ
  return FAIL                          // no satisfying assignment found
end
```

Extension to CSP. Recall the Definitions

Constraint Satisfaction Problem (CSP)

A CSP is a finite set of variables X, together with a finite set of constraints C, each on a subset of X. A **solution** to a CSP is an assignment of a value $d \in D(x)$ to each $x \in X$, such that all constraints are satisfied simultaneously.

Constraint Optimization Problem (COP)

A COP is a CSP P defined on the variables $x_1, \ldots, x_n$, together with an objective function $f : D(x_1) \times \cdots \times D(x_n) \to Q$ that assigns a value to each assignment of values to the variables. An **optimal solution** to a minimization (maximization) COP is a solution d to P that minimizes (maximizes) the value of $f(d)$.

$\rightsquigarrow$ Constraints in a CSP can be relaxed and their violations determine the objective function. This is the most common approach in LS

Min-Conflict Heuristic

procedure *MCH* (*P*, *maxSteps*)
  input: *CSP instance P, positive integer maxSteps*
  output: *solution of P or* "no solution found"
  a := randomly chosen assignment of the variables in P;
  for *step* := 1 **to** *maxSteps* **do**
    if a satisfies all constraints of P **then** **return** a **end**
    x := randomly selected variable from conflict set $K(a)$;
    v := randomly selected value from the domain of x such that
      setting x to v minimises the number of unsatisfied constraints;
    a := a with x set to v;
  end
  return "no solution found"
end *MCH*

Min-Conflict Heuristic for n-Queens Problem

```
var{int} queen[Size](m,Size) := distr.get();

ConstraintSystem S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  select(q in Size : S.violations(queen[q])>0) {
    selectMin(v in Size)(S.getAssignDelta(queen[q],v)) {
      queen[q] := v;
    }
    it = it + 1;
  }
}
cout << queen << endl;
```

Evaluation Measures of LS Algorithms

- depth
- mobility
- coverage

Outline

1. Randomized (or Stochastic) Local Search
2. Guided Local Search
3. Simulated Annealing
4. Iterated Local Search
5. Tabu Search
6. Variable Neighborhood Search

Guided Local Search

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

Guided Local Search (GLS):

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

Guided Local Search

- **Modified evaluation function:**

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

  where $SC(s)$ is the set of solution components used in candidate solution s.

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

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

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

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

Possible solutions:

A: Occasional decreases/smoothing of penalties.

B: Only increase penalties of solution components that are least likely to occur in (optimal) solutions.

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

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

where $f_i(s)$ is the solution quality contribution of i in s.

Example: Guided Local Search (GLS) for the TSP

[Voudouris and Tsang 1995; 1999]

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

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

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

Guided Local Search for SAT

- Assign a positive weight to each clause
- attempt to minimize the sum of the weights of the unsatisfied clauses.
- The clause weights are dynamically modified (additively or multiplicatively) as the search progresses, increasing the weight of the clauses that are currently unsatisfied.
- Depends on:
  how often and by how much the weights of unsatisfied clauses are increased, and
  how are all weights periodically decreased in order to prevent certain weights from becoming dis-proportionately high.

Discrete Lagrangian Method

- Change the objective function bringing constraints g_i into it

$$L(\vec{s}, \vec{\lambda}) = f(\vec{s}) + \sum_i \lambda_i g_i(\vec{s})$$

- λ_i are continous variables called Lagrangian Multipliers
- $L(\vec{s}^*, \lambda) \leq L(\vec{s}^*, \vec{\lambda}^*) \leq L(\vec{s}, \vec{\lambda}^*)$
- Alternate optimizations in $\vec{s}$ and in $\vec{\lambda}$

Discrete Lagrangian Method for SAT

let $U_i(x)$ be a function that is 0 if C_i is satisfied by a solution x, and 1 otherwise.

$$\text{minimize } N(x) = \sum_{i=1}^{m} U_i(x)$$
$$\text{s.t.} U_i(x) = 0 \qquad \forall i \in \{1, 2, \ldots, m\}$$

Discrete Lagrangian Function:

$$L_d(x, \lambda) = N(x) + \sum_{i=1}^{m} \lambda_i U_i(x)$$

Probabilistic Iterative Improv.

Key idea: Accept worsening steps with probability that depends on respective deterioration in evaluation function value: bigger deterioration $\cong$ smaller probability

Realization:

- Function $p(f, s)$: determines probability distribution over neighbors of s based on their values under evaluation function f.
- Let $\texttt{step}(s, s') := p(f, s, s')$.

Note:

- Behavior of PII crucially depends on choice of p.
- II and RII are special cases of PII.

Example: Metropolis PII for the TSP

- **Search space S:** set of all Hamiltonian cycles in given graph G.
- **Solution set:** same as S
- **Neighborhood relation $\mathcal{N}(s)$:** 2-edge-exchange
- **Initialization:** an Hamiltonian cycle uniformly at random.
- **Step function:** implemented as 2-stage process:

  1. select neighbor $s' \in N(s)$ uniformly at random;
  2. accept as new search position with probability:

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

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

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

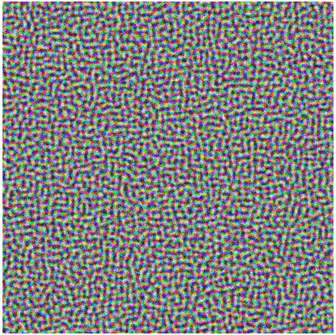
Outline

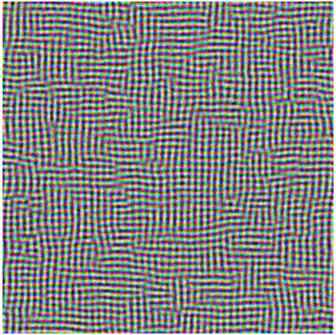
1. Randomized (or Stochastic) Local Search
2. Guided Local Search
3. Simulated Annealing
4. Iterated Local Search
5. Tabu Search
6. Variable Neighborhood Search

Inspired by statistical mechanics in matter physics:

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

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





Simulated Annealing

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

Simulated Annealing (SA):

determine initial candidate solution s
set initial temperature T according to annealing schedule
while termination condition is not satisfied: **do**
  while maintain same temperature T according to annealing schedule **do**
    probabilistically choose a neighbor s' of s using proposal mechanism
    if s' satisfies probabilistic acceptance criterion (depending on T) **then**
      $s := s'$
  update T according to annealing schedule

- 2-stage step function based on
  - proposal mechanism (often uniform random choice from $N(s)$)
  - acceptance criterion (often *Metropolis condition*)
- Annealing schedule
  (function mapping run-time t onto temperature $T(t)$):
  - initial temperature T_0
    (may depend on properties of given problem instance)
  - temperature update scheme
    (*e.g.*, linear cooling: $T_{i+1} = T_0(1 - i/I_{max})$,
    geometric cooling: $T_{i+1} = \alpha \cdot T_i$)
  - number of search steps to be performed at each temperature
    (often multiple of neighborhood size)
  - may be *static* or *dynamic*
  - seek to balance moderate execution time with asymptotic behavior properties
- Termination predicate: often based on *acceptance ratio*,
  i.e., ratio accepted / proposed steps *or* number of idle iterations

Example: Simulated Annealing for TSP

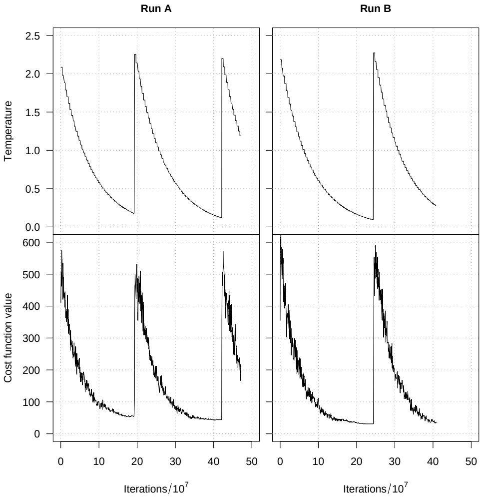
Extension of previous PII algorithm for the TSP, with

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

Improvements:

- neighborhood pruning (*e.g.*, candidate lists for TSP)
- greedy initialization (*e.g.*, by using NNH for the TSP)
- *low temperature starts* (to prevent good initial candidate solutions from being too easily destroyed by worsening steps)

Profiling


Run A
Run B
Temperature
2.5
2.0
1.5
1.0
0.5
0.0
Cost function value
600
500
400
300
200
100
0
0
10
20
30
40
50
Iterations/10^7
Iterations/10^7

Outline

1. Randomized (or Stochastic) Local Search
2. Guided Local Search
3. Simulated Annealing
4. Iterated Local Search
5. Tabu Search
6. Variable Neighborhood Search

Iterated Local Search

Key Idea: Use two types of LS 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 behavior.

Iterated Local Search (ILS):
determine initial candidate solution s
perform subsidiary local search on s
while termination criterion is not satisfied **do**
  $r := s$
  perform perturbation on s
  perform subsidiary local search on s
  based on acceptance criterion,
    keep s or revert to $s := r$

Note:

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

Components

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 LS methods can be used.
  (*e.g.*, Tabu Search).

Components

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 neighborhood.)
  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 $\Rightarrow$ short subsequent local search phase;
  but: risk of revisiting current local minimum.
- Strong perturbation $\Rightarrow$ 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.

Components

Acceptance criteria:

- Always accept the best of the two candidate solutions

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

- Always accept the most recent of the two candidate solutions

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

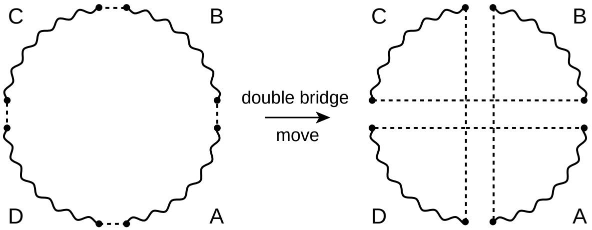
- Intermediate behavior: select between the two candidate solutions based on the *Metropolis criterion* (*e.g.*, used in *Large Step Markov Chains* [Martin et al., 1991].

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

Examples

Example: Iterated Local Search for the TSP (1)

- **Given:** TSP instance π.
- **Search space:** Hamiltonian cycles in π.
- **Subsidiary local search:** Lin-Kernighan variable depth search algorithm
- **Perturbation mechanism:**
  'double-bridge move' = particular 4-exchange step:



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

Outline

1. Randomized (or Stochastic) Local Search
2. Guided Local Search
3. Simulated Annealing
4. Iterated Local Search
5. Tabu Search
6. Variable Neighborhood Search

Tabu Search

Key idea: Avoid repeating history (memory)
How can we remember the history without

- memorizing full solutions (space)
- computing hash functions (time)

$\rightsquigarrow$ use attributes

Tabu Search

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

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

Tabu Search (TS):

determine initial candidate solution s
While *termination criterion* is not satisfied:
  determine set N' of non-tabu neighbors of s
  choose a best candidate solution s' in N'

  update tabu attributes based on s'
  $s := s'$

Example: Tabu Search for CSP

- **Search space:** set of all complete assignments of X.
- **Solution set:** assignments that satisfy all constraints
- **Neighborhood relation:** one exchange
- **Memory:** Associate tabu status (Boolean value) with each pair (variable,value) (x, val).
- **Initialization:** a random assignment
- **Search steps:**
  - pairs (x, v) are tabu if they have been changed in the last `tt` steps;
  - neighboring assignments are admissible if they can be reached by changing a non-tabu pair or have fewer unsatisfied constraints than the best assignments seen so far (aspiration criterion);
  - choose uniformly at random admissible neighbors with minimal number of unsatisfied constraints.
- **Termination:** upon finding a feasible assignment *or*
  after given bound on number of search steps has been reached *or*
  after a number of idle iterations

Note:

- Admissible neighbors of s: Non-tabu search positions in $N(s)$
- Tabu tenure: a fixed number of subsequent search steps for which the last search position or the solution components just added/removed from it are declared tabu
- Aspiration criterion (often used): specifies conditions under which tabu status may be overridden (*e.g.*, if considered step leads to improvement in incumbent solution).
- Crucial for efficient implementation:
  - efficient best improvement local search
    $\leadsto$ pruning, delta updates, (auxiliary) data structures
  - efficient determination of tabu status:
    store for each variable x the number of the search step when its value was last changed it_x; x is tabu if $it - it_x < tt$, where it = current search step number.

Design Choices

Design choices:

- Neighborhood exploration:
  - no reduction
  - min-conflict heuristic
- Prohibition power for `move = <x,new_v,old_v>`
  - `<x,-,->`
  - `<x,-,old_v>`
  - `<x,new_v,old_v>`, `<x,old_v,new_v>`
- Tabu list dynamics:
  - Interval: `tt` $\in [t_b, t_b + w]$
  - Adaptive: `tt` $= \lfloor \alpha \cdot c \rfloor + \texttt{RandU}(0, t_b)$

Outline

1. Randomized (or Stochastic) Local Search
2. Guided Local Search
3. Simulated Annealing
4. Iterated Local Search
5. Tabu Search
6. Variable Neighborhood Search

Variable Neighborhood Search

Variable Neighborhood Search is a method based on the systematic change of the neighborhood during the search.

Central observations

- a local minimum w.r.t. one neighborhood function is not necessarily locally minimal w.r.t. another neighborhood function
- a global optimum is locally optimal w.r.t. all neighborhood functions

Key principle: change the neighborhood during the search

- Several adaptations of this central principle
  - (Basic) Variable Neighborhood Descent (VND)
  - Variable Neighborhood Search (VNS)
  - Reduced Variable Neighborhood Search (RVNS)
  - Variable Neighborhood Decomposition Search (VNDS)
  - Skewed Variable Neighborhood Search (SVNS)
- Notation
  - N_k, $k = 1, 2, \ldots, k_m$ is a set of neighborhood functions
  - $N_k(s)$ is the set of solutions in the k-th neighborhood of s

How to generate the various neighborhood functions?

- for many problems different neighborhood functions (local searches) exist / are in use
- change parameters of existing local search algorithms
- use k-exchange neighborhoods; these can be naturally extended
- many neighborhood functions are associated with distance measures; in this case increase the distance

Basic Variable Neighborhood Descent

Procedure BVND
input : N_k, $k = 1, 2, \ldots, k_{max}$, and an initial solution s
output: a local optimum s for N_k, $k = 1, 2, \ldots, k_{max}$

```
k ← 1
repeat
    s' ← FindBestNeighbor(s, N_k)
    if f(s') < f(s) then
        s ← s'
        (k ← 1)
    else
        k ← k + 1
until k = k_max;
```

Variable Neighborhood Descent

Procedure VND
input : N_k, $k = 1, 2, \ldots, k_{max}$, and an initial solution s
output: a local optimum s for N_k, $k = 1, 2, \ldots, k_{max}$

```
k ← 1
repeat
    s' ← IterativeImprovement(s,N_k)
    if f(s') < f(s) then
        s ← s'
        k ← 1
    else
        k ← k + 1
until k = k_max;
```

- Final solution is locally optimal w.r.t. all neighborhoods

- First improvement may be applied instead of best improvement

- Typically, order neighborhoods from smallest to largest

- If iterative improvement algorithms II_k, $k = 1, \ldots, k_{max}$ are available as black-box procedures:
  - order black-boxes
  - apply them in the given order
  - possibly iterate starting from the first one
  - order chosen by: *solution quality* and *speed*

Basic Variable Neighborhood Search

```
Procedure BVNS
input  : N_k, k = 1, 2, ..., k_max, and an initial solution s
output: a local optimum s for N_k, k = 1, 2, ..., k_max
repeat
    k ← 1
    repeat
        s' ← RandomPicking(s, N_k)
        s'' ← IterativeImprovement(s', N_k)
        if f(s'') < f(s) then
            s ← s''
            k ← 1
        else
            k ← k + 1
    until k = k_max;
until Termination Condition;
```

To decide:

- which neighborhoods
- how many
- which order
- which change strategy

- Extended version: parameters k_{min} and k_{step}; set $k \leftarrow k_{min}$ and increase by k_{step} if no better solution is found (achieves diversification)

Summary

1. Randomized (or Stochastic) Local Search
2. Guided Local Search
3. Simulated Annealing
4. Iterated Local Search
5. Tabu Search
6. Variable Neighborhood Search