DMP204
SCHEDULING,
TIMETABLING AND ROUTING

Lecture 9
Heuristics

Marco Chiarandini

Outline

Introduction

Heuristic methods make use of two search paradigms:

- construction rules (extends partial solutions)
- local search (modifies complete solutions)

These components are problem specific and implement informed search.

They can be improved by use of metaheuristics which are general-purpose guidance criteria for underlying problem specific components.

Final heuristic algorithms are often hybridization of several components.

Outline

Construction Heuristics

Heuristic: a common-sense rule (or set of rules) intended to increase the probability of solving some problem

Construction heuristics

(aka, single pass heuristics, dispatching rules, in scheduling)
They are closely related to tree search techniques but correspond to a single path from root to leaf

- search space = partial candidate solutions
- search step = extension with one or more solution components

Construction Heuristic (CH):
$s := \emptyset$
while s is not a complete solution **do**
  choose a solution component c
  add the solution component to s

Greedy best-first search

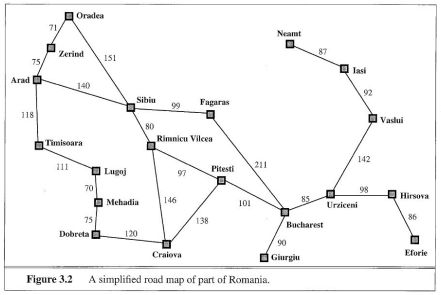


Figure 3.2 A simplified road map of part of Romania.

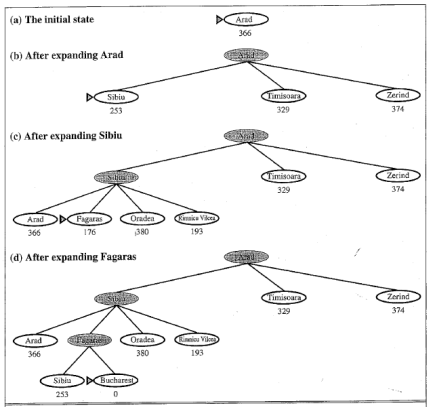


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- Sometimes greedy heuristics can be proved to be optimal (Minimum Spanning Tree, Single Source Shortest Path, $1||\sum w_j C_j$, $1||L_{max}$)

- Other times an approximation ratio can be prooved

Designing heuristics

- Same idea of (variable, value) selection in CP without backtracking

Variable

```
* INT_VAR_NONE: First unassigned

* INT_VAR_MIN_MIN: With smallest min
* INT_VAR_MIN_MAX: With largest min
* INT_VAR_MAX_MIN: With smallest max
* INT_VAR_MAX_MAX: With largest max

* INT_VAR_SIZE_MIN: With smallest domain size
* INT_VAR_SIZE_MAX: With largest domain size

* INT_VAR_DEGREE_MIN: With smallest degree The degree of a variable is defined as the number of dependant
       propagators. In case of ties, choose the variable with smallest domain.
* INT_VAR_DEGREE_MAX: With largest degree The degree of a variable is defined as the number of dependant
      propagators. In case of ties, choose the variable with smallest domain.
* INT_VAR_SIZE_DEGREE_MIN: With smallest domain size divided by degree
* INT_VAR_SIZE_DEGREE_MAX: With largest domain size divided by degree

* INT_VAR_REGRET_MIN_MIN: With smallest min-regret The min-regret of a variable is the difference between
       the smallest and second-smallest value still in the domain.
* INT_VAR_REGRET_MIN_MAX: With largest min-regret The min-regret of a variable is the difference between
      the smallest and second-smallest value still in the domain.
* INT_VAR_REGRET_MAX_MIN: With smallest max-regret The max-regret of a variable is the difference between
       the largest and second-largest value still in the domain.
* INT_VAR_REGRET_MAX_MAX: With largest max-regret The max-regret of a variable is the difference between
      the largest and second-largest
value still in the domain.
```

Designing heuristics

- Same idea of (variable, value) selection in CP without backtracking

Value

```
* INT_VAL_MIN: Select smallest value
* INT_VAL_MED: Select median value
* INT_VAL_MAX: Select maximal value

* INT_VAL_SPLIT_MIN: Select lower half of domain
* INT_VAL_SPLIT_MAX: Select upper half of domain
```

- Static vs Dynamic (➨ quality time tradeoff)

Dispatching Rules in Scheduling

	RULE	DATA	OBJECTIVES
Rules Dependent on Release Dates and Due Dates	ERD	r_j	Variance in Throughput Times
	EDD	d_j	Maximum Lateness
	MS	d_j	Maximum Lateness
Rules Dependent on Processing Times	LPT	p_j	Load Balancing over Parallel Machines
	SPT	p_j	Sum of Completion Times, WIP
	WSPT	p_j, w_j	Weighted Sum of Completion Times, WIP
	CP	p_j, $prec$	Makespan
	LNS	p_j, $prec$	Makespan
Miscellaneous	SIRO	-	Ease of Implementation
	SST	s_{jk}	Makespan and Throughput
	LFJ	M_j	Makespan and Throughput
	SQNO	-	Machine Idleness

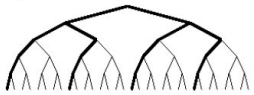
Truncated Search

They can be seen as form of Metaheuristics

Bounded-backtrack search:

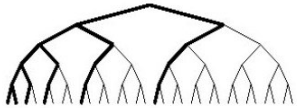
bbs(10)

Depth-bounded, then bounded-backtrack search:

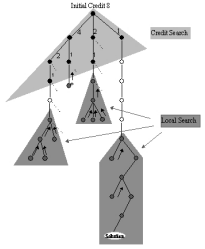


dbs(2, bbs(0))

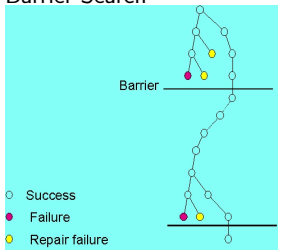
Limited Discrepancy Search (LDS)



Credit-based search



Barrier Search



A* best-first search

- The priority assigned to a node x is determined by the function

$$f(x) = g(x) + h(x)$$

$g(x)$: cost of the path so far
$h(x)$: heuristic estimate of the minimal cost to reach the goal from x.

- It is optimal if $h(x)$ is an
  - admissible heuristic: *never overestimates* the cost to reach the goal
  - consistent: $h(n) \leq c(n, a, n') + h(n')$

A* best-first search

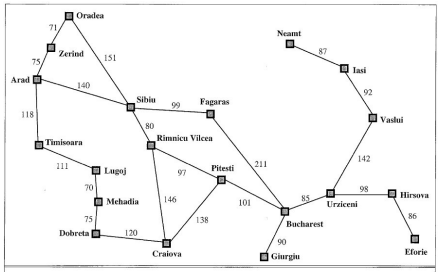


Figure 3.2 A simplified road map of part of Romania.

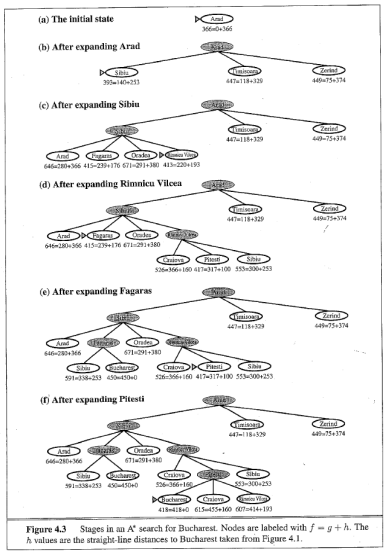


Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

A* best-first search

Possible choices for admissible heuristic functions

- optimal solution to an easily solvable **relaxed problem**
- optimal solution to an easily solvable **subproblem**
- preferred heuristics functions with higher values (provided they do not overestimate)
- if several heuristics available $h_1, h_2, \ldots, h_m$ and not clear which is the best then:

$$h(x) = \max\{h_1(x), \ldots, h_m(x)\}$$

A* best-first search

Drawbacks

- Time complexity: In the worst case, the number of nodes expanded is exponential, but it is polynomial when the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

  h^* is the optimal heuristic, the exact cost of getting from x to the goal.

- Memory usage: In the worst case, it must remember an exponential number of nodes.
  Several variants: including iterative deepening A* (IDA*), memory-bounded A* (MA*) and simplified memory bounded A* (SMA*) and recursive best-first search (RBFS)

Rollout Method

(aka, pilot method) [Bertsekas, Tsitsiklis, Cynara, JoH, 1997]
Derived from A*

- Each candidate solution is a collection of m components $S = (s_1, s_2, \ldots, s_m)$.
- Master process adds components sequentially to a partial solution $S_k = (s_1, s_2, \ldots s_k)$
- At the k-th iteration the master process evaluates seemly feasible components to add based on a look-ahead strategy based on heuristic algorithms.
- The evaluation function $H(S_{k+1})$ is determined by sub-heuristics that complete the solution starting from S_k
- Sub-heuristics are combined in $H(S_{k+1})$ by
  - weighted sum
  - minimal value

Rollout Method

Speed-ups:

- halt whenever cost of current partial solution exceeds current upper bound
- evaluate only a fraction of possible components

Beam Search

[Lowerre, Complex System, 1976]

Derived from A* and branch and bound

- maintains a set B of bw (beam width) partial candidate solutions
- at each iteration extend each solution from B in fw (filter width) possible ways
- rank each $bw \times fw$ candidate solutions and take the best bw partial solutions
- complete candidate solutions obtained by B are maintained in B_f
- Stop when no partial solution in B is to be extended

Iterated Greedy

Key idea: use greedy construction

- alternation of Construction and Deconstruction phases
- an acceptance criterion decides whether the search continues from the new or from the old solution.

Iterated Greedy (IG):
determine initial candidate solution s
while termination criterion is not satisfied **do**
  $r := s$
  greedily destruct part of s
  greedily reconstruct the missing part of s
  based on acceptance criterion,
    keep s or revert to $s := r$

GRASP

Greedy Randomized Adaptive Search Procedure (GRASP) []

Key Idea: Combine randomized constructive search with subsequent perturbative search.

Motivation:

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

Greedy Randomized "Adaptive" Search Procedure (GRASP):

While *termination criterion* is not satisfied:
- generate candidate solution s using subsidiary greedy randomized constructive search
- perform subsidiary perturbative search on s

Note:

- Randomization in *constructive search* ensures that a large number of good starting points for *subsidiary perturbative search* is obtained.
- Constructive search in GRASP is 'adaptive' (or dynamic): 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 search phase (aka *semi-greedy heuristics*) typically do not reach the performance of GRASP with perturbative search.

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.)

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 perturbative search:**
  - iterative best improvement using 1-flip neighborhood
  - 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
- various scheduling problems

Extensions and improvements of GRASP:

- reactive GRASP (*e.g.*, dynamic adaptation of α during search)

Outline

Local Search Paradigm

- search space = complete candidate solutions
- search step = modification of one or more solution components
- iteratively generate and evaluate candidate solutions
  - decision problems: evaluation = test if solution
  - optimization problems: evaluation = check objective function value
- evaluating candidate solutions is typically computationally much cheaper than finding (optimal) solutions

Iterative Improvement (II):
determine initial candidate solution s
while s has better neighbors **do**
  choose a neighbor s' of s such that $f(s') < f(s)$
  $s := s'$

Local Search Algorithm (1)

Given a (combinatorial) optimization problem Π and one of its instances π:

- search space $S(\pi)$
  specified by candidate solution representation:
  discrete structures: sequences, permutations, graphs, partitions
  (*e.g.*, for SAT: array (sequence of all truth assignments to propositional variables)

  Note: solution set $S'(\pi) \subseteq S(\pi)$
  (*e.g.*, for SAT: models of given formula)
- evaluation function $f(\pi) : S(\pi) \mapsto \mathbf{R}$
  (*e.g.*, for SAT: number of false clauses)
- neighborhood function, $\mathcal{N}(\pi) : S \mapsto 2^{S(\pi)}$
  (*e.g.*, for SAT: neighboring variable assignments differ in the truth value of exactly one variable)

Local Search Algorithm (2)

- set of memory states $M(\pi)$
  (may consist of a single state, for LS algorithms that do not use memory)
- initialization function `init` : $\emptyset \mapsto \mathcal{P}(S(\pi) \times M(\pi))$
  (specifies probability distribution over initial search positions and memory states)
- step function `step` : $S(\pi) \times M(\pi) \mapsto \mathcal{P}(S(\pi) \times M(\pi))$
  (maps each search position and memory state onto probability distribution over subsequent, neighboring search positions and memory states)
- termination predicate `terminate` : $S(\pi) \times M(\pi) \mapsto \mathcal{P}(\{\top, \bot\})$
  (determines the termination probability for each search position and memory state)

Local Search Algorithm

For given problem instance π:

- search space (solution representation) $S(\pi)$
- neighborhood relation $\mathcal{N}(\pi) \subseteq S(\pi) \times S(\pi)$
- evaluation function $f(\pi) : S \mapsto \mathbf{R}$
- set of memory states $M(\pi)$
- initialization function $\texttt{init} : \emptyset \mapsto \mathcal{P}(S(\pi) \times M(\pi))$
- step function $\texttt{step} : S(\pi) \times M(\pi) \mapsto \mathcal{P}(S(\pi) \times M(\pi))$
- termination predicate $\texttt{terminate} : S(\pi) \times M(\pi) \mapsto \mathcal{P}(\{\top, \bot\})$

LS Algorithm Components

Search Space

Defined by the solution representation:

- permutations
  - linear (scheduling)
  - circular (TSP)
- arrays (assignment problems: GCP)
- sets or lists (partition problems: Knapsack)

LS Algorithm Components

Neighborhood function $\mathcal{N}(\pi) : S(\pi) \mapsto 2^{S(\pi)}$

Also defined as: $\mathcal{N} : S \times S \to \{T, F\}$ or $\mathcal{N} \subseteq S \times S$

- neighborhood (set) of candidate solution s: $N(s) := \{s' \in S \mid \mathcal{N}(s, s')\}$
- neighborhood size is $|N(s)|$
- neighborhood is symmetric if: $s' \in N(s) \Rightarrow s \in N(s')$
- neighborhood graph of (S, N, π) is a directed vertex-weighted graph: $G_{\mathcal{N}}(\pi) := (V, A)$ with $V = S(\pi)$ and $(uv) \in A \Leftrightarrow v \in N(u)$ (if symmetric neighborhood $\Rightarrow$ undirected graph)

Note on notation: N when set, $\mathcal{N}$ when collection of sets or function

A neighborhood function is also defined by means of an operator.

An operator Δ is a collection of operator functions $\delta : S \to S$ such that

$$s' \in N(s) \iff \exists\, \delta \in \Delta, \delta(s) = s'$$

Definition

k-exchange neighborhood: candidate solutions s, s' are neighbors iff s differs from s' in at most k solution components

Examples:

- 1-exchange (flip) neighborhood for SAT
  (solution components = single variable assignments)
- 2-exchange neighborhood for TSP
  (solution components = edges in given graph)

LS Algorithm Components

Note:

- Local search implements a walk through the neighborhood graph
- Procedural versions of `init`, `step` and `terminate` implement sampling from respective probability distributions.
- Memory state m can consist of multiple independent attributes, *i.e.*, $M(\pi) := M_1 \times M_2 \times \ldots \times M_{l(\pi)}$.
- Local search algorithms are Markov processes: behavior in any search state $\{s, m\}$ depends only on current position s and (limited) memory m.

LS Algorithm Components

Search step (or move):
pair of search positions s, s' for which
s' can be reached from s in one step, *i.e.*, $\mathcal{N}(s, s')$ and
$\texttt{step}(\{s, m\}, \{s', m'\}) > 0$ for some memory states $m, m' \in M$.

- Search trajectory: finite sequence of search positions $< s_0, s_1, \ldots, s_k >$ such that (s_{i-1}, s_i) is a *search step* for any $i \in \{1, \ldots, k\}$ and the probability of initializing the search at s_0 is greater zero, *i.e.*, $\texttt{init}(\{s_0, m\}) > 0$ for some memory state $m \in M$.
- Search strategy: specified by `init` and `step` function; to some extent independent of problem instance and other components of LS algorithm.
  - random
  - based on evaluation function
  - based on memory

Uninformed Random Picking

- $\mathcal{N} := S \times S$
- does not use memory and evaluation function
- `init`, `step`: uniform random choice from S, *i.e.*, for all $s, s' \in S$, $\texttt{init}(s) := \texttt{step}(\{s\}, \{s'\}) := 1/|S|$

Uninformed Random Walk

- does not use memory and evaluation function
- `init`: uniform random choice from S
- `step`: uniform random choice from current neighborhood, *i.e.*, for all $s, s' \in S$, $\texttt{step}(\{s\}, \{s'\}) := \begin{cases} 1/|N(s)| & \text{if } s' \in N(s) \\ 0 & \text{otherwise} \end{cases}$

Note: These uninformed LS strategies are quite ineffective, but play a role in combination with more directed search strategies.

LS Algorithm Components

Evaluation (or cost) function:

- function $f(\pi) : S(\pi) \mapsto \mathbf{R}$ that maps candidate solutions of a given problem instance π onto real numbers, such that global optima correspond to solutions of π;
- used for ranking or assessing neighbors of current search position to provide guidance to search process.

Evaluation *vs* objective functions:

- *Evaluation function*: part of LS algorithm.
- *Objective function*: integral part of optimization problem.
- Some LS methods use evaluation functions different from given objective function (*e.g.*, dynamic local search).

Iterative Improvement

- does not use memory
- `init`: uniform random choice from S
- `step`: uniform random choice from improving neighbors,
  i.e., $\texttt{step}(\{s\}, \{s'\}) := 1/|I(s)|$ if $s' \in I(s)$, and 0 otherwise,
  where $I(s) := \{s' \in S \mid \mathcal{N}(s, s') \text{ and } f(s') < f(s)\}$
- terminates when no improving neighbor available
  (to be revisited later)
- different variants through modifications of step function
  (to be revisited later)

Note: II is also known as *iterative descent* or *hill-climbing*.

Example: Iterative Improvement for SAT

- **search space S:** set of all truth assignments to variables
  in given formula F
  (**solution set S':** set of all models of F)
- **neighborhood function $\mathcal{N}$:** 1-flip neighborhood
  (as in Uninformed Random Walk for SAT)
- **memory:** not used, *i.e.*, $M := \{0\}$
- **initialization:** uniform random choice from S, *i.e.*, $\texttt{init}(\emptyset, \{a'\}) := 1/|S|$
  for all assignments a'
- **evaluation function:** $f(a) :=$ number of clauses in F
  that are *unsatisfied* under assignment a
  (*Note:* $f(a) = 0$ iff a is a model of F.)
- **step function**: uniform random choice from improving neighbors, *i.e.*,
  $\texttt{step}(a, a') := 1/\#I(a)$ if $s' \in I(a)$,
  and 0 otherwise, where $I(a) := \{a' \mid \mathcal{N}(a, a') \wedge f(a') < f(a)\}$
- **termination**: when no improving neighbor is available
  i.e., $\texttt{terminate}(a, \top) := 1$ if $I(a) = \emptyset$, and 0 otherwise.

Definition:

- Local minimum: search position without improving neighbors w.r.t. given evaluation function f and neighborhood $\mathcal{N}$,
  i.e., position $s \in S$ such that $f(s) \leq f(s')$ for all $s' \in N(s)$.
- Strict local minimum: search position $s \in S$ such that
  $f(s) < f(s')$ for all $s' \in N(s)$.
- *Local maxima* and *strict local maxima*: defined analogously.

There might be more than one neighbor that have better cost.

Pivoting rule decides which to choose:

- Best Improvement (aka *gradient descent*, *steepest descent*, *greedy hill-climbing*): Choose maximally improving neighbor,
  i.e., randomly select from $I^*(s) := \{s' \in N(s) \mid f(s') = f^*\}$,
  where $f^* := \min\{f(s') \mid s' \in N(s)\}$.

  Note: Requires evaluation of all neighbors in each step.

- First Improvement: Evaluate neighbors in fixed order,
  choose first improving step encountered.

  Note: Can be much more efficient than Best Improvement; order of evaluation can have significant impact on performance.

Example: Iterative Improvement for TSP (2-opt)

procedure *TSP-2opt-first*(s)
  input: *an initial candidate tour* $s \in S(\in)$
  output: *a local optimum* $s \in S(\pi)$
  $\Delta = 0$;
  do
    Improvement=FALSE;
    for $i = 1$ **to** $n - 2$ **do**
    if $i = 1$ **then** $n' = n - 1$ **else**$n' = n$
      for $j = i + 2$ **to** n' **do**
        $\Delta_{ij} = d(c_i, c_j) + d(c_{i+1}, c_{j+1}) - d(c_i, c_{i+1}) - d(c_j, c_{j+1})$
        if $\Delta_{ij} < 0$ **then**
          UpdateTour(s,i,j);
          Improvement=TRUE;
      end
    end
  until Improvement==FALSE;
end *TSP-2opt-first*

➤ Are we in a local optimum when it terminates?

A note on terminology

Heuristic Methods ≡ Metaheuristics ≡ Local Search Methods ≡ Stochastic Local Search Methods ≡ Hybrid Metaheuristics

Method ≠ Algorithm

Stochastic Local Search (SLS) algorithms allude to:

- Local Search: informed search based on *local* or incomplete knowledge as opposed to systematic search
- Stochastic: use *randomized choices* in generating and modifying candidate solutions. They are introduced whenever it is unknown which deterministic rules are profitable for all the instances of interest.

Escaping from Local Optima

- Enlarge the neighborhood
- 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.)

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 to greedily increase solution quality or probability, *e.g.*, by exploiting the evaluation function.
- Diversification: aim to prevent search stagnation by preventing search process from 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.

Learning goals of this section

- Review basic theoretical concepts
- Learn about techniques and goals of experimental search space analysis.
- Develop intuition on which features of local search are adequate to contrast a specific situation.

Definitions

- Search space S
- Neighborhood function $\mathcal{N} : S \subseteq 2^S$
- Evaluation function $f(\pi) : S \mapsto \mathbf{R}$
- Problem instance π

Definition:

The **search landscape** L is the vertex-labeled neighborhood graph given by the triplet $\mathcal{L} = (S(\pi), N(\pi), f(\pi))$.

Fundamental Search Space Properties

The behavior and performance of an LS algorithm on a given problem instance crucially depends on properties of the respective search space.

Simple properties of search space S:

- search space size $|S|$
- reachability: solution j is reachable from solution i if neighborhood graph has a path from i to j.
  - strongly connected neighborhood graph
  - weakly optimally connected neighborhood graph
- search space diameter $\texttt{diam}(G_{\mathcal{N}})$
  (= maximal distance between any two candidate solutions)
  Note: Diameter of $G_{\mathcal{N}}$ = worst-case lower bound for number of search steps required for reaching (optimal) solutions.
  Maximal shortest path between any two vertices in the neighborhood graph.

Solution Representations and Neighborhoods

Three different types of solution representations:

- Permutation
  - linear permutation: Single Machine Total Weighted Tardiness Problem
  - circular permutation: Traveling Salesman Problem
- Assignment: Graph Coloring Problem, SAT, CSP
- Set, Partition: Knapsack, Max Independent Set

A neighborhood function $\mathcal{N} : S \to S \times S$ is also defined through an operator.
An operator Δ is a collection of operator functions $\delta : S \to S$ such that

$$s' \in N(s) \iff \exists \delta \in \Delta \mid \delta(s) = s'$$

Permutations

$\Pi(n)$ indicates the set all permutations of the numbers $\{1, 2, \ldots, n\}$

$(1, 2 \ldots, n)$ is the identity permutation ι.

If $\pi \in \Pi(n)$ and $1 \leq i \leq n$ then:

- π_i is the element at position i
- $pos_\pi(i)$ is the position of element i

Alternatively, a permutation is a bijective function $\pi(i) = \pi_i$

the permutation product $\pi \cdot \pi'$ is the composition $(\pi \cdot \pi')_i = \pi'(\pi(i))$

For each π there exists a permutation such that $\pi^{-1} \cdot \pi = \iota$

$$\Delta_N \subset \Pi$$

Neighborhood Operators for Linear Permutations

Swap operator

$$\Delta_S = \{\delta_S^i | 1 \leq i \leq n\}$$

$$\delta_S^i(\pi_1 \ldots \pi_i \pi_{i+1} \ldots \pi_n) = (\pi_1 \ldots \pi_{i+1} \pi_i \ldots \pi_n)$$

Interchange operator

$$\Delta_X = \{\delta_X^{ij} | 1 \leq i < j \leq n\}$$

$$\delta_X^{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \pi_{i+1} \ldots \pi_{j-1} \pi_i \pi_{j+1} \ldots \pi_n)$$

($\equiv$ set of all transpositions)

Insert operator

$$\Delta_I = \{\delta_I^{ij} | 1 \leq i \leq n, 1 \leq j \leq n, j \neq i\}$$

$$\delta_I^{ij}(\pi) = \begin{cases} (\pi_1 \ldots \pi_{i-1} \pi_{i+1} \ldots \pi_j \pi_i \pi_{j+1} \ldots \pi_n) & i < j \\ (\pi_1 \ldots \pi_j \pi_i \pi_{j+1} \ldots \pi_{i-1} \pi_{i+1} \ldots \pi_n) & i > j \end{cases}$$

Neighborhood Operators for Circular Permutations

Reversal (2-edge-exchange)

$$\Delta_R = \{\delta_R^{ij} | 1 \leq i < j \leq n\}$$

$$\delta_R^{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \ldots \pi_i \pi_{j+1} \ldots \pi_n)$$

Block moves (3-edge-exchange)

$$\Delta_B = \{\delta_B^{ijk} | 1 \leq i < j < k \leq n\}$$

$$\delta_B^{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \ldots \pi_k \pi_i \ldots \pi_{j-1} \pi_{k+1} \ldots \pi_n)$$

Short block move (Or-edge-exchange)

$$\Delta_{SB} = \{\delta_{SB}^{ij} | 1 \leq i < j \leq n\}$$

$$\delta_{SB}^{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \pi_{j+1} \pi_{j+2} \pi_i \ldots \pi_{j-1} \pi_{j+3} \ldots \pi_n)$$

Neighborhood Operators for Assignments

An assignment can be represented as a mapping $\sigma : \{X_1 \ldots X_n\} \rightarrow \{v : v \in D, |D| = k\}$:

$$\sigma = \{X_i = v_i, X_j = v_j, \ldots\}$$

One-exchange operator

$$\Delta_{1E} = \{\delta_{1E}^{il} | 1 \leq i \leq n, 1 \leq l \leq k\}$$

$$\delta_{1E}^{il}(\sigma) = \{\sigma : \sigma'(X_i) = v_l \text{ and } \sigma'(X_j) = \sigma(X_j) \ \forall j \neq i\}$$

Two-exchange operator

$$\Delta_{2E} = \{\delta_{2E}^{ij} | 1 \leq i < j \leq n\}$$

$$\delta_{2E}^{ij}\{\sigma : \sigma'(X_i) = \sigma(X_j), \ \sigma'(X_j) = \sigma(X_i) \text{ and } \sigma'(X_l) = \sigma(X_l) \ \forall l \neq i, j\}$$

Neighborhood Operators for Partitions or Sets

An assignment can be represented as a partition of objects selected and not selected $s : \{X\} \rightarrow \{C, \overline{C}\}$
(it can also be represented by a bit string)

One-addition operator

$$\Delta_{1E} = \{\delta^v_{1E} | v \in \overline{C}\}$$
$$\delta^v_{1E}(s) = \{s : C' = C \cup v \text{ and } \overline{C}' = \overline{C} \setminus v\}$$

One-deletion operator

$$\Delta_{1E} = \{\delta^v_{1E} | v \in C\}$$
$$\delta^v_{1E}(s) = \{s : C' = C \setminus v \text{ and } \overline{C}' = \overline{C} \cup v\}$$

Swap operator

$$\Delta_{1E} = \{\delta^v_{1E} | v \in C, u \in \overline{C}\}$$
$$\delta^v_{1E}(s) = \{s : C' = C \cup u \setminus v \text{ and } \overline{C}' = \overline{C} \cup v \setminus u\}$$

Distances

Set of paths in $G_{\mathcal{N}}$ with $s, s' \in S$:

$$\Phi(s, s') = \{(s_1, \ldots, s_h) | s_1 = s, s_h = s' \, \forall i : 1 \leq i \leq h-1, \langle s_i, s_{i+1} \rangle \in E_N\}$$

If $\phi = (s_1, \ldots, s_h) \in \Phi(s, s')$ let $|\phi| = h$ be the length of the path; then the distance between any two solutions s, s' is the length of shortest path between s and s' in $G_{\mathcal{N}}$:

$$d_{\mathcal{N}}(s, s') = \min_{\phi \in \Phi(s,s')} |\Phi|$$

$\texttt{diam}(G_{\mathcal{N}}) = \max\{d_{\mathcal{N}}(s, s') \mid s, s' \in S\}$

Note: with permutations it is easy to see that:

$$d_{\mathcal{N}}(\pi, \pi') = d_{\mathcal{N}}(\pi^{-1} \cdot \pi', \iota)$$

Distances for Linear Permutation Representations

- Swap neighborhood operator
  computable in $O(n^2)$ by the precedence based distance metric:
  $d_S(\pi, \pi') = \#\{\langle i, j \rangle | 1 \leq i < j \leq n, pos_{\pi'}(\pi_j) < pos_{\pi'}(\pi_i)\}$.
  $\texttt{diam}(G_{\mathcal{N}}) = n(n-1)/2$

- Interchange neighborhood operator
  Computable in $O(n) + O(n)$ since
  $d_X(\pi, \pi') = d_X(\pi^{-1} \cdot \pi', \iota) = n - c(\pi^{-1} \cdot \pi')$
  where $c(\pi)$ is the number of disjoint cycles that decompose a permutation.
  $\texttt{diam}(G_{\mathcal{N}_X}) = n - 1$

- Insert neighborhood operator
  Computable in $O(n) + O(n\,log(n))$ since
  $d_I(\pi, \pi') = d_I(\pi^{-1} \cdot \pi', \iota) = n - |lis(\pi^{-1} \cdot \pi')|$ where $lis(\pi)$ denotes the length of the longest increasing subsequence.
  $\texttt{diam}(G_{\mathcal{N}_I}) = n - 1$

Distances for Circular Permutation Representations

- Reversal neighborhood operator
  sorting by reversal is known to be NP-hard
  surrogate in TSP: bond distance

- Block moves neighborhood operator
  unknown whether it is NP-hard but there does not exist a proved polynomial-time algorithm

Distances for Assignment Representations

- Hamming Distance
- An assignment can be seen as a partition of n in k mutually exclusive non-empty subsets

  One-exchange neighborhood operator
  The *partition-distance* $d_{1E}(\mathcal{P}, \mathcal{P}')$ between two partitions $\mathcal{P}$ and $\mathcal{P}'$ is the minimum number of elements that must be moved between subsets in $\mathcal{P}$ so that the resulting partition equals $\mathcal{P}'$.

  The partition-distance can be computed in polynomial time by solving an assignment problem. Given the assignment matrix M where in each cell (i, j) it is $|S_i \cap S'_j|$ with $S_i \in \mathcal{P}$ and $S'_j \in \mathcal{P}'$ and defined $A(\mathcal{P}, \mathcal{P}')$ the assignment of maximal sum then it is $d_{1E}(\mathcal{P}, \mathcal{P}') = n - A(\mathcal{P}, \mathcal{P}')$

Example: Search space size and diameter for the TSP

- Search space size = $(n-1)!/2$
- Insert neighborhood
  size = $(n-3)n$
  diameter = $n-2$
- 2-exchange neighborhood
  size = $\binom{n}{2} = n \cdot (n-1)/2$
  diameter in $[n/2, n-2]$
- 3-exchange neighborhood
  size = $\binom{n}{3} = n \cdot (n-1) \cdot (n-2)/6$
  diameter in $[n/3, n-1]$

Example: Search space size and diameter for SAT

SAT instance with n variables, 1-flip neighborhood:
$G_{\mathcal{N}} = n$-dimensional hypercube; diameter of $G_{\mathcal{N}} = n$.

Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two different neighborhood functions for the same instance (S, f, π) of a combinatorial optimization problem.

If for all solutions $s \in S$ we have $N_1(s) \subseteq N_2(s')$ then we say that $\mathcal{N}_2$ dominates $\mathcal{N}_1$

Example:

In TSP, 1-insert is domnated by 3-exchange.
(1-insert corresponds to 3-exchange and there are 3-exchnages that are not 1-insert)

Efficiency vs Effectiveness

The performance of local search is determined by:

1. quality of local optima (effectiveness)
2. time to reach local optima (efficiency):
   A. time to move from one solution to the next
   B. number of solutions to reach local optima

Note:

- Local minima depend on g and neighborhood function $\mathcal{N}$.
- Larger neighborhoods $\mathcal{N}$ induce
  - neighborhood graphs with smaller diameter;
  - fewer local minima.

Ideal case: exact neighborhood, *i.e.*, neighborhood function for which any local optimum is also guaranteed to be a global optimum.

- Typically, exact neighborhoods are too large to be searched effectively (exponential in size of problem instance).
- *But:* exceptions exist, *e.g.*, polynomially searchable neighborhood in Simplex Algorithm for linear programming.

Trade-off (to be assessed experimentally):

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

Speedups Techniques for Efficient Neighborhood Search

1) Incremental updates

2) Neighborhood pruning

Speedups in Neighborhood Examination

1) Incremental updates (aka delta evaluations)

- **Key idea:** calculate effects of differences between current search position s and neighbors s' on evaluation function value.
- Evaluation function values often consist of independent contributions of solution components; hence, $f(s)$ can be efficiently calculated from $f(s')$ by differences between s and s' in terms of solution components.
- Typically crucial for the efficient implementation of II algorithms (and other LS techniques).

Example: Incremental updates for TSP

- solution components = edges of given graph G
- standard 2-exchange neighborhood, *i.e.*, neighboring round trips p, p' differ in two edges
- $w(p') := w(p)$ $-$ edges in p but not in p'
  $+$ edges in p' but not in p

Note: Constant time (4 arithmetic operations), compared to linear time (n arithmetic operations for graph with n vertices) for computing $w(p')$ from scratch.

2) Neighborhood Pruning

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

Example: Heuristic candidate lists for the TSP

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

Overview

Delta evaluations and neighborhood examinations in:

- Permutations
  - TSP
  - SMTWTP
- Assignments
  - SAT
- Sets
  - Max Independent Set

Local Search for TSP

- k-exchange heuristics
  - 2-opt
  - 2.5-opt
  - Or-opt
  - 3-opt
- complex neighborhoods
  - Lin-Kernighan
  - Helsgaun's Lin-Kernighan
  - Dynasearch
  - ejection chains approach

Implementations exploit speed-up techniques

1. neighborhood pruning: fixed radius nearest neighborhood search
2. neighborhood lists: restrict exchanges to most interesting candidates
3. don't look bits: focus perturbative search to "interesting" part
4. sophisticated data structures

TSP data structures

Tour representation:

- determine pos of v in π
- determine succ and prec
- check whether u_k is visited between u_i and u_j
- execute a k-exchange (reversal)

Possible choices:

- $|V| < 1.000$ array for π and π^{-1}
- $|V| < 1.000.000$ two level tree
- $|V| > 1.000.000$ splay tree

Moreover static data structure:

- priority lists
- k-d trees

SMTWTP

- Interchange: size $\binom{n}{2}$ and $O(|i-j|)$ evaluation each
  - first-improvement: π_j, π_k
    - $p_{\pi_j} \leq p_{\pi_k}$ for improvements, $w_j T_j + w_k T_k$ must decrease because jobs in $\pi_j, \ldots, \pi_k$ can only increase their tardiness.
    - $p_{\pi_j} \geq p_{\pi_k}$ possible use of auxiliary data structure to speed up the computation
  - first-improvement: π_j, π_k
    - $p_{\pi_j} \leq p_{\pi_k}$ for improvements, $w_j T_j + w_k T_k$ must decrease at least as the best interchange found so far because jobs in $\pi_j, \ldots, \pi_k$ can only increase their tardiness.
    - $p_{\pi_j} \geq p_{\pi_k}$ possible use of auxiliary data structure to speed up the computation
- Swap: size $n-1$ and $O(1)$ evaluation each
- Insert: size $(n-1)^2$ and $O(|i-j|)$ evaluation each
  But possible to speed up with systematic examination by means of swaps: an interchange is equivalent to $|i-j|$ swaps hence overall examination takes $O(n^2)$

LS for GCP

- **search space** S: set of all k-colorings of G
- **solution set** S': set of all proper k-coloring of F
- **neighborhood function** $\mathcal{N}$: 1-exchange neighborhood (as in Uninformed Random Walk)
- **memory:** not used, *i.e.*, $M := \{0\}$
- **initialization:** uniform random choice from S, *i.e.*, $\texttt{init}\{\emptyset, \varphi'\} := 1/|S|$ for all colorings φ'
- **step function**:
  - **evaluation function:** $g(\varphi) :=$ number of edges in G whose ending vertices are assigned the same color under assignment φ (*Note:* $g(\varphi) = 0$ iff φ is a proper coloring of G.)
  - **move mechanism:** uniform random choice from improving neighbors, *i.e.*, $\texttt{step}\{\varphi, \varphi'\} := 1/|I(\varphi)|$ if $s' \in I(\varphi)$, and 0 otherwise, where $I(\varphi) := \{\varphi' \mid \mathcal{N}(\varphi, \varphi') \wedge g(\varphi') < g(\varphi)\}$
- **termination**: when no improving neighbor is available *i.e.*, $\texttt{terminate}\{\varphi, \top\} := 1$ if $I(\varphi) = \emptyset$, and 0 otherwise.

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 improving candidate solution s' in N'
  update tabu attributes based on s'
  $s := s'$

Note:

- Non-tabu search positions in $N(s)$ are called *admissible neighbors of* s.
- After a search step, the current search position or the solution components just added/removed from it are declared *tabu* for a fixed number of subsequent search steps (*tabu tenure*).
- Often, an additional *aspiration criterion* is used: this specifies conditions under which tabu status may be overridden (*e.g.*, if considered step leads to improvement in incumbent solution).
- Crucial for efficient implementation:
  - keep time complexity of search steps minimal by using special data structures, incremental updating and caching mechanism for evaluation function values;
  - efficient determination of tabu status: store for each variable x the number of the search step when its value was last changed it_x; x is tabu if $it - it_x < tt$, where it = current search step number.

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

- *tt* too low ⇒ search stagnates due to inability to escape from local minima;
- *tt* too high ⇒ search becomes ineffective due to overly restricted search path (admissible neighborhoods too small)

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:
  $r := s$
  perform *perturbation* on s
  perform *subsidiary local search* on s
  based on *acceptance criterion*, keep s or revert to $s := r$

Outline

Software Tools

- Modeling languages
  interpreted languages with a precise syntax and semantics
- Software libraries
  collections of subprograms used to develop software
- Software frameworks
  set of abstract classes and their interactions
  - *frozen spots* (remain unchanged in any instantiation of the framework)
  - *hot spots* (parts where programmers add their own code)

No well established software tool for Local Search:

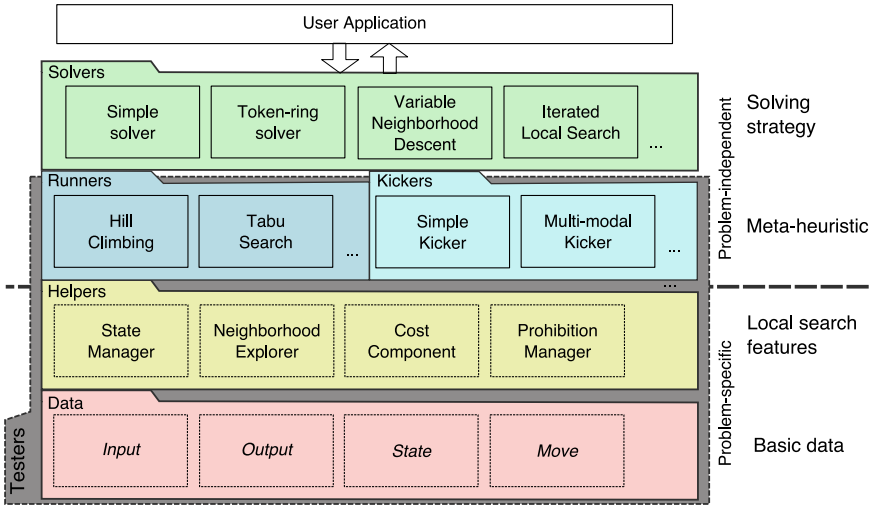
- the apparent simplicity of Local Search induces to build applications from scratch.
- crucial roles played by delta/incremental updates which is problem dependent
- the development of Local Search is in part a craft, beside engineering and science.
- lack of a unified view of Local Search.

Software tools for Local Search and Metaheuristics

Tool	Reference	Language	Type
ILOG	?	C++, Java, .NET	LS
GAlib	?	C++	GA
GAUL	?	C	GA
Localizer++	?	C++	Modeling
HotFrame	?	C++	LS
EasyLocal++	?	C++, Java	LS
HSF	?	Java	LS, GA
ParadisEO	?	C++	EA, LS
OpenTS	?	Java	TS
MDF	?	C++	LS
TMF	?	C++	LS
SALSA	?	—	Language
Comet	?	—	Language

table prepared by L. Di Gaspero

Separation of Concepts in Local Search Algorithms



implemented in EasyLocal++

Input (util.h, util.c)

```
typedef struct {
  long int number_jobs; /* number of jobs in instance */
  long int release_date[MAX_JOBS]; /*there is no release date for these instances*/
  long int proc_time[MAX_JOBS];
  long int weight[MAX_JOBS];
  long int due_date[MAX_JOBS];
} instance_type;

instance_type instance;

void read_problem_size (char name[100])
void read_instances (char input_file_name[100])
```


State/Solution (util.h)

```
typedef struct {
  long int job_at_pos[MAX_JOBS]; /* Gives the job at a certain pos */
  long int pos_of_job[MAX_JOBS]; /* Gives the position of a specific job */
  long int completion_time_job[MAX_JOBS]; /* Gives C_j of job j */
  long int start_time_job[MAX_JOBS]; /* Gives start time of job j */
  long int tardiness_job[MAX_JOBS]; /* Gives T_j of job j */
  long int value; /* Objective function value */
} sol_representation;

sol_representation sequence;
```

Output (util.c)

```
void print_sequence (long int k)
void print_completion_times ()
```

State Manager (util.c)

```
void construct_sequence_random ()
void construct_sequence_canonical ()
long int evaluate ()
```


Random Generator (random.h, random.c)

```
void set_seed (double arg)
double MRG32k3a (void)
double ranU01 (void)
int ranUint (int i, int j)
void shuffle (int *X, int size)
```

Timer (timer.c)

```
double getCurrentTime ()
```


- Implement two basic local search procedures that return a local optimum:

```
void ls_swap_first( ) {};
void ls_interchange_first( ) {};
```

- Implement the other neighborhood for permutation representation mentioned at the lecture from one of the two previous neighborhoods.
- Provide computational analysis of the LS implemented. Consider:
  - size of the neighborhood
  - diameter of neighborhood
  - complete neighborhood examination
  - local optima attainment
- Devise speed ups to reduce the computational complexity of the LS implemented
- Improve your heuristic in order to find solutions of better quality. (Hint: use a construction heuristic and/or a metaheuristic)