

**DM204, 2010**  
SCHEDULING, TIMETABLING AND ROUTING

Lecture 9  
**Heuristics**

Marco Chiarandini

Department of Mathematics & Computer Science  
University of Southern Denmark

# Course Overview

- ✓ Problem Introduction
  - ✓ Scheduling classification
  - ✓ Scheduling complexity
  - ✓ RCPSP
- General Methods
  - ✓ Integer Programming
  - ✓ Constraint Programming
    - Heuristics
    - Dynamic Programming and Branch and Bound
- Scheduling
  - Single Machine
  - Parallel Machine and Flow Shop Models
  - Job Shop
  - Resource Constrained Project Scheduling Model
- Timetabling
  - Reservations and Education
  - University Timetabling
  - Crew Scheduling
  - Public Transports
- Vehicle Routing
  - Capacited Models
  - Time Windows models
  - Rich Models

# Outline

## 1. Construction Heuristics

- General Principles

- Metaheuristics

  - CP like

  - Rollout

  - Beam Search

  - Iterated Greedy

  - GRASP

## 2. Local Search

- Components

- Beyond Local Optima

- Search Space Properties

- Neighborhood Representations

- Efficient Local Search

- Metaheuristics

  - Min Conflict Heuristic

  - Tabu Search

  - Iterated Local Search

# 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

## 1. Construction Heuristics

General Principles

Metaheuristics

CP like

Rollout

Beam Search

Iterated Greedy

GRASP

## 2. Local Search

Components

Beyond Local Optima

Search Space Properties

Neighborhood Representations

Efficient Local Search

Metaheuristics

Min Conflict Heuristic

Tabu Search

Iterated Local Search

# 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)

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

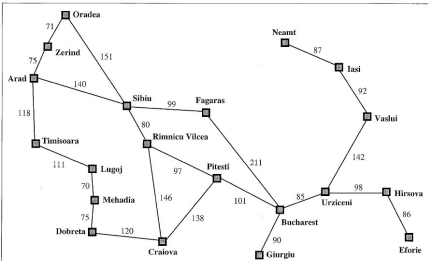


Figure 3.2 A simplified road map of part of Romania.

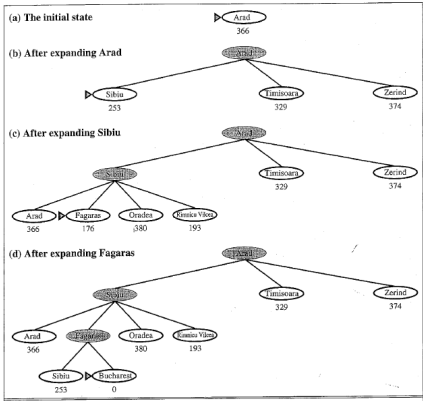


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic  $h_{SLD}$ . Nodes are labeled with their  $h$ -values.

- 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 proved



# 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

[Appendix C.2 of B1]

	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

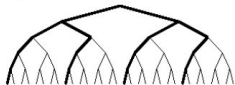
They can be seen as form of Metaheuristics

**Bounded-backtrack search:**



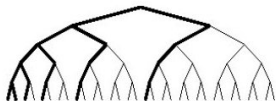
bbs(10)

**Depth-bounded, then bounded-backtrack search:**

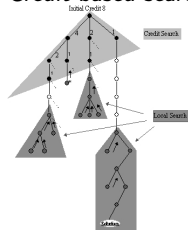


dbs(2, bbs(0))

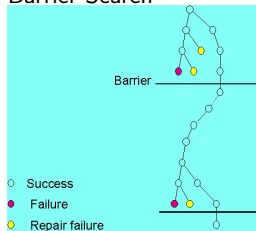
**Limited Discrepancy Search (LDS)**



**Credit-based search**



**Barrier Search**



# Rollout Method

(aka, pilot method)

[Bertsekas, Tsitsiklis, Cynara, JoH, 1997]

Derived from  $A^*$

- Each candidate solution is a collection of  $m$  components  
 $S = (s_1, s_2, \dots, s_m)$ .
- Master process adds components sequentially to a partial solution  
 $S_k = (s_1, s_2, \dots, s_k)$
- At the  $k$ -th iteration the master process evaluates seemingly 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) [Feo and Resende, 1989]

**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  $l$ . ( $\alpha$  is a parameter of the algorithm.)

# Outline

## 1. Construction Heuristics

- General Principles

- Metaheuristics

  - CP like

  - Rollout

  - Beam Search

  - Iterated Greedy

  - GRASP

## 2. Local Search

- Components

- Beyond Local Optima

- Search Space Properties

- Neighborhood Representations

- Efficient Local Search

- Metaheuristics

  - Min Conflict Heuristic

  - Tabu Search

  - Iterated Local Search

# 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):

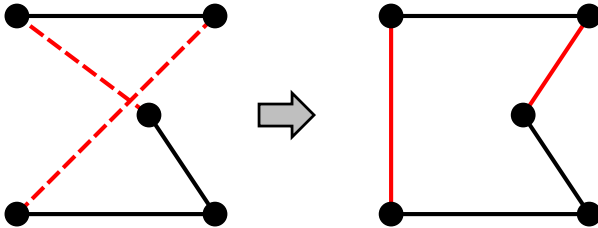
determine initial candidate solution  $s$

**while**  $s$  has better neighbors **do**

┌ choose a neighbor  $s'$  of  $s$  such that  $f(s') < f(s)$   
└  $s := s'$

# Example: Travelling Salesperson Problem

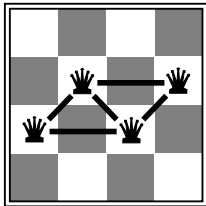
Start with any complete tour, perform pairwise exchanges



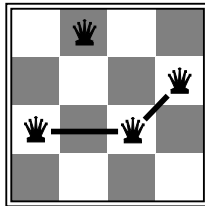
Variants of this approach get within 1% of optimal very quickly with thousands of cities

## Example: $n$ -queens

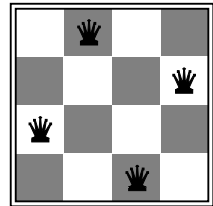
Put  $n$  queens on an  $n \times n$  board with no two queens on the same row, column, or diagonal  
Move a queen to reduce number of conflicts



$h = 5$











$h = 2$



$h = 0$

Almost always solves  $n$ -queens problems almost instantaneously  
for very large  $n$ , e.g.,  $n = 1\text{million}$

## Example: $n$ -queens

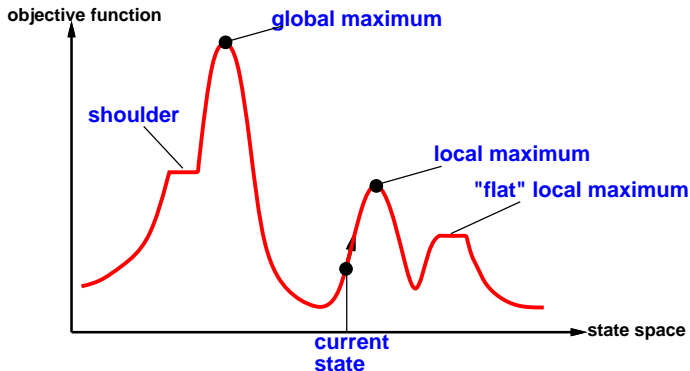
18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14		13	16	13	16
	14	17	15		14	16	16
17		16	18	15		15	
18	14		15	15	14		16
14	14	13	17	12	14	12	18

- Current cost 17
- 8 possible successor



# Hill-climbing contd.

Useful to think of **state space landscape**



Random-restart hill climbing overcomes local maxima—trivially complete  
Random sideways moves 😊 escape from shoulders 😞 loop on flat maxima

# Local Search Algorithm (1)

[Hoos and Stützle, 2005]

Given a (combinatorial) optimization problem  $\Pi$  and one of its instances  $\pi$ :

- search space  $S(\pi)$   
specified by candidate solution representation:  
discrete structures: sequences, permutations, graphs, partitions  
(e.g., for Queens: array of column assignment)  
  
Note: solution set  $S'(\pi) \subseteq S(\pi)$   
(e.g., for Queens: not all assignments satisfy all constraints)
- evaluation function  $f(\pi) : S(\pi) \mapsto \mathbf{R}$   
(e.g., for Queens: number of offending Queens)
- neighborhood function,  $\mathcal{N}(\pi) : S \mapsto 2^{S(\pi)}$   
(e.g., for Queens: neighboring variable assignments differ  
in the value of exactly one queen)

## 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  $\text{init} : \emptyset \mapsto \mathcal{P}(S(\pi) \times M(\pi))$   
(specifies probability distribution over initial search positions and memory states)
- step function  $\text{step} : S(\pi) \times M(\pi) \mapsto \mathcal{P}(S(\pi) \times M(\pi))$   
(maps each search position and memory state onto probability distribution over subsequent, neighboring search positions and memory states)
- termination predicate  $\text{terminate} : S(\pi) \times M(\pi) \mapsto \mathcal{P}(\{T, \perp\})$   
(determines the termination probability for each search position and memory state)

# Local Search Algorithm

For given problem instance  $\pi$ :

- 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  $\text{init} : \emptyset \mapsto \mathcal{P}(S(\pi) \times M(\pi))$
- step function  $\text{step} : S(\pi) \times M(\pi) \mapsto \mathcal{P}(S(\pi) \times M(\pi))$
- termination predicate  $\text{terminate} : S(\pi) \times M(\pi) \mapsto \mathcal{P}(\{\top, \perp\})$

# LS Algorithm Components

## Search Space

Defined by the solution representation:

- permutations
  - linear (scheduling)
  - circular (TSP)
- arrays (assignment problems: Timetabling)
- sets or lists (partition problems: SCP, VRP)

# LS Algorithm Components

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

Also defined as:  $\mathcal{N} : S \times S \rightarrow \{T, F\}$  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  $(S, N, \pi)$  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  $\Delta$  is a collection of operator functions  $\delta : S \rightarrow S$  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 Queens  
(solution components = single variable assignments)
- 2-exchange neighborhood for TSP  
(solution components = edges in given graph)

# 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

$\text{step}(\{s, m\}, \{s', m'\}) > 0$  for some memory states  $m, m' \in M$ .

- Search strategy: specified by **init** and **step** function; to some extent independent of problem instance and other components of LS algorithm.
- Search trajectory: finite sequence of search positions  $\langle s_0, s_1, \dots, s_k \rangle$  such that  $(s_{i-1}, s_i)$  is a *search step* for any  $i \in \{1, \dots, k\}$ 
  - random
  - based on evaluation function
  - based on memory



# LS Algorithm Components

## Evaluation (or cost) function:

- function  $f(\pi) : S(\pi) \mapsto \mathbf{R}$  that maps candidate solutions of a given problem instance  $\pi$  onto real numbers, such that global optima correspond to solutions of  $\pi$ ;
- 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.*,  $\text{step}(\{s\}, \{s'\}) := 1/|I(s)|$  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:** It is also known as *iterative descent* or *hill-climbing*.

## 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  $I^*(s) := \{s' \in N(s) | f(s') = f^*\}$ , where  $f^* := \min\{f(s') | s' \in N(s)\}$ .

*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.

# A note on terminology

Method  $\neq$  Algorithm

Heuristic Methods  $\equiv$  Metaheuristics  $\equiv$  Local Search Methods  $\equiv$  Stochastic  
Local Search Methods  $\equiv$  Hybrid Metaheuristics

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**: greedily increase solution quality or probability, e.g., by exploiting the evaluation function.
- **Diversification**: prevent search stagnation by avoiding 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.

# Definitions

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

## Definition:

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



# 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

# Solution Representations

Three different types of solution representations:

- **Permutation**
  - **linear permutation**: Single Machine Total Weighted Tardiness Problem
  - **circular permutation**: Traveling Salesman Problem
- **Assignment**: CSP, Timetabling
- **Set, Partition**: Crew Scheduling, Vehicle Routing

A neighborhood function  $\mathcal{N} : S \rightarrow S \times S$  is also defined through an operator.  
An **operator**  $\Delta$  is a collection of operator functions  $\delta : S \rightarrow S$  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  $\{1, 2, \dots, n\}$

$(1, 2, \dots, n)$  is the identity permutation  $\iota$ .

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

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

$$\Delta_N \subset \Pi$$

# Operators for Linear Permutations

Swap operator

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

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

Interchange operator

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

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

( $\equiv$  set of all transpositions)

Insert operator

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

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

# Operators for Circular Permutations

Reversal (2-edge-exchange)

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

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

Block moves (3-edge-exchange)

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

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

Short block move (Or-edge-exchange)

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

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

# Operators for Assignments

An assignment can be represented as a mapping

$$\sigma : \{X_1 \dots X_n\} \rightarrow \{v : v \in D, |D| = k\}:$$

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

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

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

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

# Operators for Partitions or Sets

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

One-addition operator

$$\Delta_{1E} = \{\delta_{1E}^v \mid v \in \bar{C}\}$$

$$\delta_{1E}^v(s) = \{s : C' = C \cup v \text{ and } \bar{C}' = \bar{C} \setminus v\}$$

One-deletion operator

$$\Delta_{1E} = \{\delta_{1E}^v \mid v \in C\}$$

$$\delta_{1E}^v(s) = \{s : C' = C \setminus v \text{ and } \bar{C}' = \bar{C} \cup v\}$$

Swap operator

$$\Delta_{1E} = \{\delta_{1E}^{v,u} \mid v \in C, u \in \bar{C}\}$$

$$\delta_{1E}^{v,u}(s) = \{s : C' = C \cup u \setminus v \text{ and } \bar{C}' = \bar{C} \cup v \setminus u\}$$

# 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  $f$  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.

# Min Conflict Heuristic

- A variable  $x$  is selected uniformly at random from the **conflict set**  $K(a)$
- A value  $v$  is chosen from the domain of  $x$ , such that assigning  $v$  to  $x$ , the number of conflicts is minimized. Break ties at random.

# 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*

# 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  $s'$   
     $s := s'$



- 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  $\Rightarrow$  search stagnates due to inability to escape from local minima;
- $tt$  too high  $\Rightarrow$  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  $s := r$