DM87
SCHEDULING,
TIMETABLING AND ROUTING

Lecture 4
Constraint Programming, Heuristic Methods

Marco Chiarandini

Outline

1. Resume

2. Constraint Programming

3. Heuristic Methods
   Construction Heuristics and Local Search
   Solution Representations and Neighborhood Structures in LS
   Metaheuristics
      Metaheuristics for Construction Heuristics
      Metaheuristics for Local Search and Hybrids

Modeling: Mixed Integer Formulations

- Formulation for \( Qm|p_j = 1|\sum h_j(C_j) \) and relation with transportation problems
- Totally unimodular matrices and sufficient conditions for total unimodularity i) two ones per column and ii) consecutive 1’s property
- Formulation of \( 1|\text{prec}||\sum w_jC_j \) and \( Rm||\sum C_j \) as weighted bipartite matching and assignment problems.
- Formulation of set covering, set partitioning and set packing
- Formulation of Traveling Salesman Problem
- Formulation of \( 1|\text{prec}||\sum w_jC_j \) and how to deal with disjunctive constraints
- Graph coloring
Special Purpose Algorithms

- Dynamic programming
- Branch and Bound

Outline

1. Resume
2. Constraint Programming
3. Heuristic Methods
   - Construction Heuristics and Local Search
   - Solution Representations and Neighborhood Structures in LS
   - Metaheuristics
     - Metaheuristics for Construction Heuristics
     - Metaheuristics for Local Search and Hybrids

Constraint Satisfaction Problem

- **Input:** a set of variables $X_1, X_2, \ldots, X_n$ and a set of constraints. Each variable has a non-empty domain $D_i$ of possible values. Each constraint $C_i$ involves some subset of the variables and specify the allowed combination of values for that subset. A constraint $C$ on variables $X_i$ and $X_j$, $C(X_i, X_j)$, defines the subset of the Cartesian product of variable domains $D_iD_j$ of the consistent assignments of values to variables. A constraint $C$ on variables $X_i, X_j$ is satisfied by a pair of values $v_i, v_j$ if $(v_i, v_j) \in C(X_i, X_j)$.

- **Task:** find an assignment of values to all the variables $\{X_i = v_i, X_j = v_j, \ldots\}$ such that it is consistent, that is, it does not violate any constraints.

Search Problem

- **initial state:** the empty assignment $\{\}$ in which all variables are unassigned
- **successor function:** a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables
- **goal test:** the current assignment is complete
- **path cost:** a constant cost

Two search paradigms:

- search tree of depth $n$
- complete state formulation: local search
Types of Variables and Values

- Discrete variables with finite domain: complete enumeration is $O(d^n)$
- Discrete variables with infinite domains: Impossible by complete enumeration. Instead a constraint language (and constraint logic programming and constraint reasoning)
  Eg. project planning.
$$S_j + p_j \leq S_k$$
NB: if linear constraints, then integer linear programming
- Variables with continuous domains
  NB: if linear constraints or convex functions then mathematical programming

Types of constraints

- Unary constraint
- Binary constraints (constraint graph)
- Higher order (constraint hypergraph)
  Eg. AllDiff() Every higher order constraint can be reaged to binary (you may need auxiliary constraints)
- Preference constraints cost on individual variable assignments

General purpose solution algorithms

Search algorithms
tree with branching factor at the top level $nd$
at the next level $(n-1)d$.
The tree has $n! \cdot d^n$ even if only $d^n$ possible complete assignments.
- CSP is commutative in the order of application of any given set of action. (the order of the assignment does not influence)
- Hence we can consider search algts that generate successors by considering possible assignments for only a single variable at each node in the search tree.

Backtracking search
depth first search that chooses one variable at a time and backtracks when a variable has no legal values left to assign.
Backtrack Search

- No need to copy solutions all the times but rather extensions and undo extensions
- Since CSP is standard then the alg is also standard and can use general purpose algorithms for initial state, successor function and goal test.
- Backtracking is uninformed and complete. Other search algorithms may use information in form of heuristics.

General Purpose backtracking methods

1) Which variable should we assign next, and in what order should its values be tried?
   - Select-Unassigned-Variable
     Most constrained variable (DSATUR) = fail-first heuristic
     = Minimum remaining values (MRV) heuristic (speeds up pruning)
   - Select-Initial-Unassigned-Variable
     degree heuristic (reduces the branching factor) also used as tied breaker
   - Order-Domain-Values
     least-constraining-value heuristic (leaves maximum flexibility for subsequent variable assignments)

NB: If we search for all the solutions or a solution does not exists, then the ordering does not matter.

2) What are the implications of the current variable assignments for the other unassigned variables?

   Propagating information through constraints
   - Implicit in Select-Unassigned-Variable
   - Forward checking (coupled with MRV)
   - Constraint propagation
     - arc consistency: force all (directed) arcs $uv$ to be consistent: $\exists$ a value in $D(v)$ : $\forall$ in values in $D(u)$, otherwise detects inconsistency
       can be applied as preprocessing or as propagation step after each assignment (MAC, Maintaining Arc Consistency)
       Applied repeatedly
     - $k$-consistency: if for any set of $k - 1$ variables, and for any consistent assignment to those variables, a consistent value can always be assigned to any $k$-th variable.
       determining the appropriate level of consistency checking is mostly an empirical science.
### Handling special constraints (higher order constraints)

Special purpose algorithms

- **Alldiff**
  - for \( m \) variables and \( n \) values cannot be satisfied if \( m > n \),
  - consider first singleton variables

- **Resource Constraint at most**
  - check the sum of minimum values of single domains
  - delete maximum values if not consistent with minimum values of others.
  - for large integer values not possible to represent the domain as a set of integers but rather as bounds.

Then bounds propagation: Eg,

- \( \text{Flight271} \in [0, 165] \) and \( \text{Flight272} \in [0, 385] \)
- \( \text{Flight271} + \text{Flight272} \in [420, 420] \)
- \( \text{Flight271} \in [35, 165] \) and \( \text{Flight272} \in [255, 385] \)

![Figure 5.6](image)

The progress of a map-coloring search with forward checking. **WA = red** is assigned first; then forward checking deletes **red** from the domains of the neighboring variables **NT** and **SA**. **After WA = green**, **green** is deleted from the domains of **NT**, **SA**, and **NSW**. **After V = blue**, **blue** is deleted from the domains of **NSW** and **SA**, leaving **SA** with no legal values.

3) When a path fails – that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?

**BACKTRACKING-SEARCH**

- chronological backtracking, the most recent decision point is revisited
- backjumping, backtracks to the most recent variable in the conflict set (set of previously assigned variables connected to \( X \) by constraints).

every branch pruned by backjumping is also pruned by forward checking idea remains: backtrack to reasons of failure.

---

**AC-3**

<table>
<thead>
<tr>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>G</td>
<td>B</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>R</td>
<td>B</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>G</td>
<td>B</td>
<td>R</td>
</tr>
</tbody>
</table>

**function AC-3**

**returns** the CSP, possibly with reduced domains

**inputs**: \( csp \), a binary CSP with variables \( \{X_1, X_2, \ldots, X_n\} \)

**local variables**: \( queue \), a queue of arcs, initially all the arcs in \( csp \)

**while** \( queue \) is not empty do

- \((X_i, X_j) \leftarrow \text{REMOVE-FIRST}(queue)\)
- if \( \text{REMOVE-INCORRECT-VALUES}(X_i, X_j) \) then
  - for each \( X_k \) in \( \text{NEIGHBORS}[X_i] \) do
    - add \((X_k, X_i)\) to \( queue \)

**function REMOVE-INCORRECT-VALUES**

**returns** true if we remove a value otherwise false

- for each \( x \) in \( \text{DOMAIN}[X_i] \) do
  - if no value \( y \) in \( \text{DOMAIN}[X_j] \) allows \((x, y)\) to satisfy the constraint between \( X_i \) and \( X_j \) then
    - delete \( x \) from \( \text{DOMAIN}[X_i] \); \( \text{removed} \leftarrow \text{true} \)

**return** \( \text{removed} \)
The structure of problems

- Decomposition in subproblems:
  - connected components in the constraint graph
  - $O(d^{n/c})$ vs $O(d^n)$

- Constraint graphs that are tree are solvable in poly time by reverse arc-consistency checks.

- Reduce constraint graph to tree:
  - removing nodes (cutset conditioning: find the smallest cycle cutset. It is NP-hard but good approximations exist)
  - collapsing nodes (tree decomposition)

  divide-and-conquer works well with small subproblems

Optimization Problems

Objective function $F(X_1, X_2, \ldots, X_n)$

- Solve a modified Constraint Satisfaction Problem by setting a (lower) bound $z^*$ in the objective function and increase if infeasible
- Dichotomic search: $U$ upper bound, $L$ lower bound

$$M = \frac{U + L}{2}$$

Constraint Logic Programming

Language is first-order logic.

- Syntax Language
  - Alphabet
  - Well-formed Expressions
    - E.g., $4X + 3Y = 10; 2X - Y = 0$

- Semantics Meaning
  - Interpretation
  - Logical Consequence

- Calculi Derivation
  - Inference Rule
  - Transition System
Logic Programming

A logic program is a set of axioms, or rules, defining relationships between objects.

A computation of a logic program is a deduction of consequences of the program.

A program defines a set of consequences, which is its meaning.

The art of logic programming is constructing concise and elegant programs that have desired meaning.

Sterling and Shapiro: The Art of Prolog, Page 1.

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.
Construction Heuristics (aka Dispatching Rules)

Closely related to search tree techniques
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:
1. choose a solution component \( c \)
2. add the solution component to \( s \)

An important class of Construction Heuristics are greedy algorithms
Always make the choice which is the best at the moment.
- Sometime it can be proved that they are optimal
  (Minimum Spanning Tree, Single Source Shortest Path, \( 1||\sum w_j C_j \), \( 1||L_{\text{max}} \))
- Other times it can be proved an approximation ratio

Another class can be derived by the (variable, value) selection rules in CP and removing backtracking (ex, MRV, least-constraining-values).
Examples of Dispatching Rules

<table>
<thead>
<tr>
<th>RULE Dependent on Release Dates and Due Dates</th>
<th>DATA</th>
<th>OBJECTIVES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERD r_j</td>
<td>Variance in Throughput Times</td>
<td></td>
</tr>
<tr>
<td>EDD d_j</td>
<td>Maximum Lateness</td>
<td></td>
</tr>
<tr>
<td>MS d_j</td>
<td>Maximum Lateness</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rules Dependent on Processing Times</th>
<th>DATA</th>
<th>OBJECTIVES</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPT p_j, p_j</td>
<td>Load Balancing over Parallel Machines</td>
<td></td>
</tr>
<tr>
<td>SPT p_j</td>
<td>Sum of Completion Times, WIP</td>
<td></td>
</tr>
<tr>
<td>WSPT p_j, v_j</td>
<td>Weighted Sum of Completion Times, WIP</td>
<td></td>
</tr>
<tr>
<td>CP p_j, pre</td>
<td>Makespan</td>
<td></td>
</tr>
<tr>
<td>LNS p_j, pre</td>
<td>Makekeep</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous</th>
<th>DATA</th>
<th>OBJECTIVES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIRO -</td>
<td>Ease of Implementation</td>
<td></td>
</tr>
<tr>
<td>SST i_k</td>
<td>Makespan and Throughput</td>
<td></td>
</tr>
<tr>
<td>LFJ M_t</td>
<td>Makespan and Throughput</td>
<td></td>
</tr>
<tr>
<td>SQNO -</td>
<td>Machine Idleness</td>
<td></td>
</tr>
</tbody>
</table>

Example: Local Search for CSP

```plaintext
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure

inputs: csp, a constraint satisfaction problem
max_steps, the number of steps allowed before giving up

current ← an initial complete assignment for csp
for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen, conflicted variable from VARIABLES[csp]
    value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current

return failure
```

Local Search

Components

- solution representation
- initial solution
- neighborhood structure
- acceptance criterion

Solution Representation

Determines the search space $S$

- permutations
  - linear (scheduling)
  - circular (routing)
- assignment arrays (timetabling)
- sets or lists (timetabling)
**Initial Solution**

- Random
- Construction heuristic

**Neighborhood Structure**

- **Neighborhood structure (relation):** equivalent definitions:
  - $\mathcal{N} : S \times S \rightarrow \{T, F\}$
  - $\mathcal{N} \subseteq S \times S$
  - $\mathcal{N} : S \rightarrow 2^S$

- **Neighborhood (set) of candidate solution $s$:** $\mathcal{N}(s) := \{s' \in S \mid \mathcal{N}(s, s')\}$

- A neighborhood structure is also defined as an operator. An operator $\Delta$ is a collection of operator functions $\delta : S \rightarrow S$ such that
  
  $s' \in \mathcal{N}(s) \iff \exists \delta \in \Delta \mid \delta(s) = s'$

**Example**

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

**Acceptance Criterion**

Defines how the neighborhood is searched and which neighbor is selected. Examples:

- uninformed random walk
- iterative improvement (hill climbing)
  - best improvement
  - first improvement

**Evaluation function**

- function $f(\pi) : S(\pi) \rightarrow \mathbb{R}$ 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).
Implementation Issues

At each iteration, the examination of the neighborhood must be fast!!

- 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.

- Special algorithms for solving efficiently the neighborhood search problem

Local Optima

Definition:

- **Local minimum:** search position without improving neighbors w.r.t. given evaluation function $f$ and neighborhood $N$, 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.

Example: Iterative Improvement

First improvement for TSP

```plaintext
procedure TSP-2opt-first(s)
    input: an initial candidate tour $s \in S(\varepsilon)$
    $\Delta = 0$;
    Improvement=FALSE;
    do
        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
        end
    until Improvement==TRUE;
    return: a local optimum $s \in S(\pi)$
end TSP-2opt-first
```

Outline

1. Resume
2. Constraint Programming
3. Heuristic Methods
   - Construction Heuristics and Local Search
   - Solution Representations and Neighborhood Structures in LS
   - Metaheuristics
     - Metaheuristics for Construction Heuristics
     - Metaheuristics for Local Search and Hybrids
### Permutations

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

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

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

1. \( \pi_i \) is the element at position \( i \)
2. \( pos_{\pi}(i) \) 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 \( \pi \) there exists a permutation such that \( \pi^{-1} \cdot \pi = \iota \)

\[ \Delta_N \subset \Pi \]

### Neighborhood Operators for Circular Permutations

**Reversal (2-edge-exchange)**

\[ \Delta_R = \{ \delta^R_{ij} | 1 \leq i < j \leq n \} \]

\[ \delta^R_{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \pi_{i+1} \ldots \pi_n) \]

**Block moves (3-edge-exchange)**

\[ \Delta_B = \{ \delta^B_{ijk} | 1 \leq i < j < k \leq n \} \]

\[ \delta^B_{ijk}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \pi_k \pi_i \pi_{j+1} \ldots \pi_{k+1} \ldots \pi_n) \]

**Short block move (Or-edge-exchange)**

\[ \Delta_{SB} = \{ \delta^SB_{ij} | 1 \leq i < j \leq n \} \]

\[ \delta^SB_{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \pi_{j+1} \pi_{i+2} \pi_i \ldots \pi_{j-1} \pi_{j+3} \ldots \pi_n) \]

### Neighborhood Operators for Linear Permutations

**Swap operator**

\[ \Delta_S = \{ \delta^S_{i} | 1 \leq i \leq n \} \]

\[ \delta^S_{i}(\pi_1 \ldots \pi_i \pi_{i+1} \ldots \pi_n) = (\pi_1 \ldots \pi_{i+1} \pi_i \pi_{i+2} \ldots \pi_n) \]

**Interchange operator**

\[ \Delta_X = \{ \delta^X_{ij} | 1 \leq i < j \leq n \} \]

\[ \delta^X_{ij}(\pi) = (\pi_1 \ldots \pi_{i-1} \pi_j \pi_{i+1} \ldots \pi_{j-1} \pi_{i+1} \ldots \pi_n) \]

**Insert operator**

\[ \Delta_I = \{ \delta^I_{ij} | 1 \leq i \leq n, 1 \leq j \leq n, j \neq i \} \]

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

### 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**

\[ \Delta_{1E} = \{ \delta^1_{iE} | 1 \leq i \leq n, 1 \leq l \leq k \} \]

\[ \delta^1_{iE}(\sigma) = \{ \sigma : \sigma'(X_i) = v_l \text{ and } \sigma'(X_j) = \sigma(X_j) \forall j \neq i \} \]

**Two exchange operator**

\[ \Delta_{2E} = \{ \delta^2_{ij} | 1 \leq i < j \leq n \} \]

\[ \delta^2_{ij}(\sigma : \sigma'(X_i) = \sigma(X_j), \sigma'(X_j) = \sigma(X_i) \text{ and } \sigma'(X_l) = \sigma(X_l) \forall l \neq i, j \} \]
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

$$\Delta_1E = \{\delta^v_1E | v \in C\}$$

$$\delta^v_1E(s) = \{s : C' = C \cup v \text{ and } \overline{C'} = \overline{C} \setminus v\}$$

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

$$\Delta_1E = \{\delta^v_1E | v \in C, u \in C\}$$

$$\delta^v_1E(s) = \{s : C' = C \cup u \setminus v \text{ and } \overline{C'} = \overline{C} \cup v \setminus u\}$$

Outline

1. Resume
2. Constraint Programming
3. Heuristic Methods
   - Construction Heuristics and Local Search
   - Solution Representations and Neighborhood Structures in LS
   - Metaheuristics
      - Metaheuristics for Construction Heuristics
      - Metaheuristics for Local Search and Hybrids

Bounded-backtrack search:

```
/\     \    
/     /\   
```

Depth-bounded, then bounded-backtrack search:

```
/ \ / \     
/  /  /\    
```
Credit-based search:

Limited Discrepancy Search:

Limited Discrepancy Search
▶ A discrepancy is a branch against the value of an heuristic
▶ Ex: count one discrepancy if second best is chosen
  count two discrepancies either if third best is chosen or twice the second best is chosen
▶ Explore the tree in order of an increasing number of discrepancies

Rollout/Pilot Method

Derived from A*  
▶ Each candidate solution is a collection of $m$ components $s = (s_1, s_2, \ldots, s_m)$.  
▶ Master process add 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

Figure 3.2 A simplified road map of part of Romania.
Speed-ups:
- halt whenever cost of current partial solution exceeds current upper bound
- evaluate only a fraction of possible components

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

Possible choices for admissible heuristic functions
- optimal solution to an easily solvable relaxed problem
- optimal solution to an easily solvable subproblem
- learning from experience by gathering statistics on state features
- 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:
  $$h(x) = \max\{h_1(x), \ldots, h_m(x)\}$$

Beam Search

Possible extension of tree based construction heuristics:
- 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 extend
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 \( s := r \)

Greedy Randomized Adaptive Search Procedure (GRASP)

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

Greedy Randomized Adaptive Search Procedure (GRASP):

| While termination criterion is not satisfied: |
| generate candidate solution \( s \) using subsidiary greedy randomized constructive search |
| perform subsidiary local search on \( s \) |

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 \( h(l) \leq h_{\text{min}} + \alpha \cdot (h_{\text{max}} - h_{\text{min}}) \),
    where \( h_{\text{min}} \) = minimal value of \( h \) and \( h_{\text{max}} \) = maximal value of \( h \) for any \( l \). (\( \alpha \) is a parameter of the algorithm.)

Outline

1. Resume
2. Constraint Programming
3. Heuristic Methods
   - Construction Heuristics and Local Search
   - Solution Representations and Neighborhood Structures in LS
   - Metaheuristics
     - Metaheuristics for Construction Heuristics
     - Metaheuristics for Local Search and Hybrids
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:
  - While maintain same temperature \( T \) according to *annealing schedule*:
    - probabilistically choose a neighbor \( s' \) of \( s \)
      - using *proposal mechanism*
      - If \( s' \) satisfies probabilistic *acceptance criterion* (depending on \( T \)):
        - \( s := s' \)
      - update \( T \) according to *annealing schedule*

**Example: Simulated Annealing for the 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 \( \exp \left( \frac{(f(s) - f(s'))}{T} \right) \));
- *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)

**Note:**
- 2-stage neighbor selection procedure
  - *proposal mechanism* (often uniform random choice from \( N(s) \))
  - *acceptance criterion* (often *Metropolis condition*)
    
    \[
    p(T, s, s') := \begin{cases} 
    1 & \text{if } g(s') \leq f(s) \\
    \exp \left( \frac{f(s) - f(s')}{T} \right) & \text{otherwise}
    \end{cases}
    \]
- *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: \( T_{i+1} = T_i (1 - i/I_{max}) \),
      geometric cooling: \( T_{i+1} = \alpha \cdot T_i \))
  - number of search steps to be performed at each temperature
    - (often multiple of neighborhood size)
- *Termination predicate:* often based on *acceptance ratio*, i.e., ratio of proposed vs accepted steps or number of idle iterations

---

**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.

---

**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 $s := r$

---

**Memetic Algorithm**

**Population based method inspired by evolution**

determine initial population $sp$
perform **subsidiary local search** on $sp$

While **termination criterion** is not satisfied:

- generate set $spr$ of new candidate solutions by **recombination**
- perform **subsidiary local search** on $spr$
- generate set $spm$ of new candidate solutions from $spr$ and $sp$ by **mutation**
- perform **subsidiary local search** on $spm$
- select new population $sp$ from candidate solutions in $sp$, $spr$, and $spm$
Selection

Main idea: selection should be related to fitness

▶ Fitness proportionate selection (Roulette-wheel method)

\[ p_i = \frac{f_i}{\sum_j f_j} \]

▶ Tournament selection: a set of chromosomes is chosen and compared and the best chromosome chosen.

▶ Rank based and selection pressure

Recombination (Crossover)

▶ Binary or assignment representations

▶ one-point, two-point, m-point (preference to positional bias w.r.t. distributional bias

▶ uniform cross over (through a mask controlled by a Bernoulli parameter \( p \))

▶ Non-linear representations

▶ (Permutations) Partially mapped crossover

▶ (Permutations) mask based

▶ More commonly ad hoc crossovers are used as this appears to be a crucial feature of success

▶ Two off-springs are generally generated

▶ Crossover rate controls the application of the crossover. May be adaptive: high at the start and low when convergence

Example: crossovers for binary representations

Mutation

▶ Goal: Introduce relatively small perturbations in candidate solutions in current population + offspring obtained from recombination.

▶ Typically, perturbations are applied stochastically and independently to each candidate solution; amount of perturbation is controlled by mutation rate.

▶ Mutation rate controls the application of bit-wise mutations. May be adaptive: low at the start and high when convergence

▶ Possible implementation through Poisson variable which determines the \( m \) genes which are likely to change allele.

▶ Can also use subsidiary selection function to determine subset of candidate solutions to which mutation is applied.

▶ The role of mutation (as compared to recombination) in high-performance evolutionary algorithms has been often underestimated
New Population

- Determines population for next cycle (generation) of the algorithm by selecting individual candidate solutions from current population + new candidate solutions obtained from recombination, mutation (+ subsidiary local search). \((\lambda, \mu) (\lambda + \mu)\)
- **Goal:** Obtain population of high-quality solutions while maintaining population diversity.
- Selection is based on evaluation function (fitness) of candidate solutions such that better candidate solutions have a higher chance of ‘surviving’ the selection process.
- It is often beneficial to use elitist selection strategies, which ensure that the best candidate solutions are always selected.
- Most commonly used: steady state in which only one new chromosome is generated at each iteration
- Diversity is checked and duplicates avoided

Ant Colony Optimization

The Metaheuristic

- The optimization problem is transformed into the problem of finding the best path on a weighted graph \(G(V,E)\) called construction graph
- The artificial ants incrementally build solutions by moving on the graph.
- The solution construction process is
  - stochastic
  - biased by a pheromone model, that is, a set of parameters associated with graph components (either nodes or edges) whose values are modified at runtime by the ants.
- All pheromone trails are initialized to the same value, \(\tau_0\).
- At each iteration, pheromone trails are updated by decreasing (evaporation) or increasing (reinforcement) some trail levels on the basis of the solutions produced by the ants

Example: A simple ACO algorithm for the TSP

- **Construction graph**
- To each edge \(ij\) in \(G\) associate
  - pheromone trails \(\tau_{ij}\)
  - heuristic values \(\eta_{ij} := \frac{1}{c_{ij}}\)
- Initialize pheromones
- **Constructive search:**
  \[
p_{ij} = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha \cdot [\eta_{il}]^\beta},
\]
- Update pheromone trail levels
  \[
  \tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \rho \cdot \text{Reward}
  \]
  \(\alpha\) and \(\beta\) are parameters.
Example: A simple ACO algorithm for the TSP (2)

- **Subsidiary local search**: Perform iterative improvement based on standard 2-exchange neighborhood on each candidate solution in population (until local minimum is reached).

- **Update pheromone trail levels** according to
  \[
  \tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \sum_{s \in sp'} \Delta_{ij}(s)
  \]
  where \(\Delta_{ij}(s) := 1/C^s\)
  if edge \((i, j)\) is contained in the cycle represented by \(s'\), and 0 otherwise.

**Motivation**: Edges belonging to highest-quality candidate solutions and/or that have been used by many ants should be preferably used in subsequent constructions.

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