Outline

1. Introduction
2. A didactic optimization problem (dTP)
   - Motivations for cooperation
3. A zoo of CP / LS hybrids
   - Sequential combination
   - Master / sub-problem decomposition
   - Improved neighborhood exploration
   - CP Neighborhood search
   - Large neighborhood search
   - Local moves during construction
   - Local moves over a heuristic
What this tutorial addresses

- Solving large hard combinatorial optimization problems
- Systematic description of ways of combining LS and CP techniques
- Goal: provide a check-list of recipes that can be tried when tackling a new optimization application
- Illustrated on a didactic problem
When should you enquire about CP / LS hybrids?

- When you have:
  - A large complex optimization problem
  - No solution neither with CP nor with LS
  - The problem specification may change over time

- Best in case of strong execution requirements
  - Limited planning resource
  - On-line optimization
When can’t it help?

- When modeling is the issue
- When optimization is the single difficulty
- When thousands of man.year have been spent studying your very problem
  
  => useless for solving a 1M node TSP
Comparing CP and LS

- **Constraint Programming**
  - Solves complex problems
  - Models capturing many side constraints
  - Solves by global search and propagation

- **Local search**
  - Solves problems with simple models
  - Efficiency: quick first solution, rapid early convergence
Opportunities for collaboration

- Expected combination of:
  - Generality (solve complex problems)
    - Nice modeling
    - Generic methods from the model
    - Easy to add/modify constraints
  - Efficiency (solve them fast)
    - Initial solution
    - Quick convergence
  - Address both feasibility and optimization issues
    - Keep constraints hard
- Difficulty to combine:
  - Monotonic reasoning (CP)
  - Non-monotonic modifications (LS)
Outline

1. Introduction
2. A didactic optimization problem (dTP)
   - Motivations for cooperation
3. A zoo of CP / LS hybrids
   - Sequential combination
   - Master / sub-problem decomposition
   - Improved neighborhood exploration
   - CP Neighborhood search
   - Large neighborhood search
   - Local moves during construction
   - Local moves over a heuristic
A didactic transportation problem

Collect goods from clients
- Set of trucks located in a depot
- Each truck can carry two bins
- Each bin may contain only goods from the same type
- Clients have time window constraints
- Bins have capacity constraints
A simple model for dTP

\[ i, j \in \{1, \ldots, n\}: \text{clients (their locations)} \]

\[ k \in \{1, \ldots, M\}: \text{trucks} \]

\[ h \in \{1, \ldots, 2M\}: \text{bins} \]

\[ l \in \{1, \ldots, P\}: \text{types of goods} \]
Minimize \( \text{totCost} = \sum_{k=1}^{M} \text{cost}_k \)

On \( \forall k, \text{cost}_k \geq 0 \)

\( \text{truck}_k: \text{UnaryResource}(tt,c,\text{cost}_k) \)

\( \forall h, \text{collects}_h \in [1 .. P] \)

\( \forall i, \text{start}_i \in [a_i .. b_i] \)

\( \text{service}_i: \text{Activity}(<\text{start}_i, d_i, i>) \)

\( \text{visitedBy}_i \in [1 .. M] \)

\( \text{collectedIn}_i \in [1 .. 2M] \)
Subject to

\[ \forall i, \text{service}_i \text{ requires truck}[\text{visitedBy}_i] \]
\[ \forall h, \sum_{i | \text{collectedIn}_i = h} q_i \leq C \]
\[ \forall i, \text{collects}[\text{collectedIn}_i] = \text{type}_i \]
\[ \forall i, \text{visitedBy}_i = \left\lceil \frac{\text{collectedIn}_i}{2} \right\rceil \]
A CP approach

- Strengthen the model
  - Add redundant constraints
  - Add global constraints
  - Add constraints evaluating the cost of the solutions
  - Symmetry breaking (dominance) constraints

- Find a search heuristic
  - Variable / value orderings
  - Explore part of the search tree through Branch and Bound
A CP approach

Redundant models for stronger propagation

Example: redundant routing model

\[ \forall k, \quad first_k \in [1 .. N] \]
\[ \forall i, \quad next_i \in [1 .. N+M] \]
\[ succ_i \in [\{\} .. \{1, ..., N\}] \]

multiPath(first,next,succ,visitedBy)

costPaths(first,next,succ,c,totCost)

\[ \forall i,j, \quad j \in succ_i \iff \]
\[ \left( \text{visitedBy}_i = \text{visitedBy}_j \land start_i < start_j \right) \]
Solving through CP

- Instantiate $visitedBy_i$
- Rank all activities on the routes (instantiate $next_i / succ_i$)
- Instantiate $start_i$ to their earliest possible value
Difficulties with CP

- Poor global reasoning
- Poor cost anticipation
- Goes backtracking « forever »
- As propagation is strengthened, the model is slowed down
A local search approach

- Two possibilities:
  - Work in the space of feasible solutions
  - Accept infeasible solutions by turning constraints into penalties

- Possible combinations, work with feasible but add, if needed, extra resources (trucks and bins)
Local search for dTP

- Generate an initial solution
  - Select clients $i$ in random order
  - Assign it to a truck that has a bin of $type_i$, or to a truck that can be added an extra bin of $type_i$

- Move from a solution to one of its neighbors, in order to improve the objective
Neighborhoods for dTP

- **Node transfer:**
  - Change values of $visitedBy_i$ and $collectedIn_i$ for some $i$

- **Bin swap:**
  - Select bins $h_1$, $h_2$ on trucks $k_1 = \lceil h_1/2 \rceil$, $k_2 = \lceil h_2/2 \rceil$
  - For all clients $i$,
    - $collectedIn_i = h_1 \Rightarrow collectedIn_i = h_2$, $visitedBy_i = k_2$
    - $collectedIn_i = h_2 \Rightarrow collectedIn_i = h_1$, $visitedBy_i = k_1$
  - Swap $collects_{h_1}$ and $collects_{h_2}$
Neighborhoods

- **k-opt:**
  - select $i_1$, $i_2$, $i_3$ such that
    \[ \text{visitedBy}_{i_1} = \text{visitedBy}_{i_2} = \text{visitedBy}_{i_3} \]
  - Exchange edges:
    - Replace $\text{next}_{i_1} = j_1$, $\text{next}_{i_2} = j_2$, $\text{next}_{i_3} = j_3$
    - By $\text{next}_{i_1} = j_2$, $\text{next}_{i_2} = j_3$, $\text{next}_{i_3} = j_1$
Driving the local search process

- **Main iteration:**
  - Until a global stopping criterion is met:
    - generate a new initial solution
    - perform a local walk

- **Each walk:**
  - Until a local criterion is met:
    - Iterate the neighborhood, until a neighbor satisfying all constraints as well as the acceptance criterion is found
    - Perform the move
Difficulties with LS

As the problem gets more constrained...

- Generating a good feasible first solution becomes harder
- Exploring neighborhoods
  - takes longer: constraints checks
  - is less interesting: fewer valid nodes
  - more local optima appear
Conclusion

Neither of the “pure” approaches works

- Need for hybridization with other techniques
  - Try a cooperation between CP and LS
  - Expect to retain:
    - *Good sides of CP*: handling side constraints, building valid solutions, systematic search
    - *Good sides of LS*: quick easy improvements, quick convergence.
Outline

1. Introduction
2. A didactic optimization problem (dTP)
   - Motivations for cooperation
3. A zoo of CP / LS hybrids
   - Sequential combination
   - Master / sub-problem decomposition
   - Improved neighborhood exploration
   - CP Neighborhood search
   - Large neighborhood search
   - Local moves during construction
   - Local moves over a heuristic
Sequential combination: LS-CP

- Use local search for the beginning of the optimization descent – switch to CP at time $t_0$
Discussion

- A good idea
  - When the feasibility problem is easy
  - For time-constrained optimization

- But, the switch from LS to CP is not immediate
  - CP starts with a good upper bound, but without no-goods

- On the didactic Transportation Problem (dTP)
  - Lack of good lower bounds
    => Systematic CP search gets stuck near the optimal region
Sequential combination: CP-LS

- Build a first feasible solution with CP
  - Greedy heuristic
- Try to improve it through LS
  - Constraints can be softened to support dense neighborhoods
Greedy insertion algorithm

- At each choice point a function $h$ is evaluated for all possible choices
- The choice that minimizes $h$ is considered as preferred decision
- The preferred decision is taken
Greedy insertion for dTP
Discussion

- CP then LS: can be interesting for dTP
  - In particular in case of tight side constraints
- « One-shot » use of CP:
  - as long as no valid solution has been found, we look for one
  - Enables to start LS with a valid solution
A systematic combination

- Solve the problem through CP (global search tree)
- Try to improve each solution found through local search
- Improve the optimization cuts
Discussion

- Local moves should change the assignment of « early » variables
  - Avoid visiting the same region as with backtracking

- Especially interesting in case of incomplete search
  - CP provides a set of diversified seeds for local search
Outline

1. Introduction
2. A didactic optimization problem (dTP)
   - Motivations for cooperation
3. A zoo of CP / LS hybrids
   - Sequential combination
   - Master / sub-problem decomposition
   - Improved neighborhood exploration
   - CP Neighborhood search
   - Large neighborhood search
   - Local moves during construction
   - Local moves over a heuristic
Master / sub-problem decomposition

- Idea: identify two sub-problems and solve them by different techniques
  - Master problem
  - Induced sub-problem

- Decomposition: the sub-problem can only be stated once the master problem is solved.
Purpose of decomposition

- Decompose into easier problems
  - Smaller size
  - Simpler models
  - Well known structure

- Traditional approach with exact methods (Dantzig, Lagrangean, Benders)
A decomposition on dTP

- **Master Problem:**
  - Assignment of clients to trucks (*visitedBy*)

- **Induced sub-problem:**
  - Traveling salesman with time windows

- **Algorithms:**
  - Assess a cost for each client (e.g. distance to neighbor), solve assignment with some method
  - Solve small TSPs with CP
  - Analyze TSPs, re-assess client cost and try improving local moves on the master problem.
Discussion

- Decomposition makes the problem easier to solve
- Estimating the cost in the master problem may be difficult
- Try local changes on the evaluated cost of the master problem
  - improve subsequent optimization (feedback from the sub-problem)
Outline

1. Introduction
2. A didactic optimization problem (dTP)
   – Motivations for cooperation
3. A zoo of CP / LS hybrids
   – Sequential combination
   – Master / sub-problem decomposition
   – Improved neighborhood exploration
   – CP Neighborhood search
   – Large neighborhood search
   – Local moves during construction
   – Local moves over a heuristic
Constrained local search

- Small neighborhoods
  - A neighbor solution $S_1$ can be reached from a given solution $S^*$ by performing “simple” modifications of $S^*$.
  - Examples:
    - Choose two visits $i_1$ and $i_2$, remove $i_1$ from its current position and reinsert it after $i_2$
    - Choose two visits $i_1$ and $i_2$ and exchange their positions
Constrained local search

- **Node Exchange**
  - Choose two visits $i_1$ and $i_2$ assigned to different trucks and exchange their positions
  - Accept the first exchange improving the cost
Node Exchange: version 1

Procedure exchange(P,S)
forall nodes i_1
forall nodes i_2 | (svisitedBy_{i_1} ≠ svisitedBy_{i_2})
exchangeInstantiate(P,S,i_1,i_2)
// check feasibility and check cost function
if (propagate(P) && improving(P,S))
storeSolution(P,S)
resetProblem(P)
exit iterations
// reinitialize the domain variables
resetProblem(P)
Procedure exchange\(\text{Instantiate}(P, S, i_1, i_2)\)

// exchange \(i_1\) and \(i_2\)
\[
\begin{align*}
\text{next}[\text{sprev}_{i_1}] &= i_2; \\
\text{next}[i_2] &= \text{snext}_{i_1}; \\
\text{next}[\text{sprev}_{i_2}] &= i_1; \\
\text{next}[i_1] &= \text{snext}_{i_2};
\end{align*}
\]

// restore the rest
forall \(k \notin \{i_1, i_2, \text{sprev}_{i_1}, \text{sprev}_{i_2}\}\)
\[
\text{next}[k] = \text{snext}_k;
\]
Node Exchange: version 1

- **Pros:**
  - Independent from side-constraints

- **Cons:**
  - CP imposes monotonic changes
    - while moving from one neighbor to the next one all problem variables are un-instantiated and re-instantiated
  - Constraints are checked in “generate and test”
    - inefficient
Node Exchange: version 2

Add inlined constraint checks

Procedure exchange\( (P, S) \)

for all nodes \( i_1 \)

for all trucks \( k \) \( (k \neq \text{svistedBy}_{i_1}) \)

if (not binCompatible\( (P, S, \text{svistedBy}_{i_1}, k) \)) continue

for all nodes \( i_2 \) \( (\text{svistedBy}_{i_2} = k) \)

if (not timeWindowCompatible\( (P, S, i_1, i_2) \)) continue

if (not improving\( (P, S, i_1, i_2) \)) continue

exchangeInstantiate\( (P, S, i_1, i_2) \)

// check feasibility && check cost function

if (propagate\( (P) \) && improving\( (P, S) \))

storeSolution\( (P, S) \)

resetProblem\( (P) \)

exit iterations

resetProblem\( (P); \) // reinitialize the domains page 44
Node Exchange: version 2

- **Pros:**
  - “Almost” independent from side-constraints
  - Some constraints are tested before performing the move
    ➤ much more efficient

- **Cons:**
  - CP imposes monotonic changes
  - Some constraints are still checked in “generate and test”
Outline

1. Introduction
2. A didactic optimization problem (dTP)
   - Motivations for cooperation
3. A zoo of CP / LS hybrids
   - Sequential combination
   - *Master / sub-problem decomposition*
   - *Improved neighborhood exploration*
   - CP Neighborhood search
   - Large neighborhood search
   - Local moves during construction
   - *Local moves over a heuristic*
CP Based Operators

- Operators define neighborhoods
- Finding the best solution in a neighborhood is an optimization problem
- Which can be solved with constraint programming
- Neighborhood search problem can be expressed:
  - With a specific model and interface constraints
  - With the original model and additional constraints
Specific Model

- A special model is developed to represent the neighborhood
- Interface constraints link the new model to the original model
- All the constraints stated in the original model are enforced in the specific model via the interface constraints
- During search, constraint propagation allows to prune (via the interface) regions of the neighborhood
- No restrictions on the neighborhood which can be defined
The neighborhood is defined simply by adding additional constraints to the original model.
No need to define a new model and interface constraints.
All the constraints in the original model are naturally enforced.
During search, constraint propagation allows to prune directly large regions of the neighborhood.
Not all neighborhoods can defined inside the original model (i.e. GENeralized Insertion).
The neighborhood of a solution $S$ for a problem $P$ is defined by a constraint problem

$$NP(P, S) :: [{I_1, \ldots, I_n}, {C_1, \ldots, C_m}]$$

- Each solution of $NP$ represents a neighbor of $S$ for $P$
Node Exchange: nhood model

- Variables: $I::[0..n-1]$, $J::[0..n-1]$, $DCost::[-\infty..0]$
  - $I, J$ are domain-variables representing the nodes $i,j$ that we want to exchange.

- Constraints:
  
  // neighborhood cst
  $I > J$
  $s\text{visitedBy}[I] \neq s\text{visitedBy}[J]$
  $next[I] = s\text{next}[J]$
  $next[J] = s\text{next}[I]$
  $next[s\text{prev}[I]] = J$
  $next[s\text{prev}[J]] = I$

  // interface cst
  $\forall k, (k \neq I \land k \neq J)$
  $\Rightarrow next[k] = snext[k], visitedBy[k] = s\text{visitedBy}[k]$
Node Exchange: nhood model

- **DCost** represents the gain w.r.t S:

\[
DCost = \text{cost}[\text{sprev}[J], I] + \text{cost}[I, \text{snext}[J]] + \\
\text{cost}[\text{sprev}[I], J] + \text{cost}[J, \text{snext}[I]] - \\
\text{cost}[\text{sprev}[I], I] - \text{cost}[I, \text{snext}[I]] - \\
\text{cost}[\text{sprev}[J], J] - \text{cost}[J, \text{snext}[J]]
\]

// improving cst
DCost < 0

- **Search** (explore via tree search):
  instantiate(I) && instantiate(J)
Node Exchange: nhood model

- Search: instantiate(I) && instantiate(J)
- Each leaf defines a feasible exchange
Suppose that in S:
- clients 1,2 are visited by truck 1,
- clients 3,4 are visited by truck 2

\[ \text{svisitedBy}[I] \neq \text{svisitedBy}[J] \]
Pros:
- Independent from side-constraints
- Constraint Propagation removes infeasible neighbors a priori.
  - efficient when many side constraints
  - efficient when large neighborhoods
- May freely mix tree search and local search

Cons:
- Overhead due to tree search
Node Exchange: nhood model

- Overhead due to tree search

  - Often most problem variables are instantiated by the interface constraints only when ALL neighborhood variables are instantiated (at every leaf of the nhood tree search)

  - In this case the nhood tree search keeps “doing” and “undoing” the instantiations of ALL the problem variables
Local Search via solution deltas

- Goal: avoid instantiating and un-instantiating ALL problem variables while moving from one neighbor to the other
  - A neighbor is identified by the modification over the original solution S. This modification is defined \textit{solution delta}.
  - A neighborhood is an array of \textit{deltas}.
  - The exploration of the neighborhood takes place on a tree search.
Procedure exchange\((P, S)\)

SolutionArray neighborArray

forall nodes \(i_1\)

forall nodes \(i_2\) \(\mid (s\text{visitedBy}_{i_1} \neq s\text{visitedBy}_{i_2})\)

Solution delta = \{(next[sprev[i_2]] = i_1),
(next[i_1] = snext[i_2]),
(next[sprev[i_1]] = i_2),
(next[i_2] = snext[i_1])\}

neighborArray.add(delta)

exploreNeighborhood\((P, S, \text{neighborArray})\)
LS via solution deltas: explore the neighborhood

- Map the array of deltas in a tree search
  - recursively split the array of deltas in two parts
  - a split correspond to a branching node in the tree search
  - each feasible neighbor is a leaf of the tree
  - at each node restore the fraction of S that is shared by all neighbors in that node
Exploring the neighborhood through solution deltas

Example: $deltas = [d_1,d_2,d_3,d_4,d_5,d_6]$
Local Search via solution deltas

- **Pros:**
  - Independent from side-constraints
  - Constraint Propagation removes infeasible neighbors a priori.
    - Efficient when many side constraints
    - Efficient when large neighborhood
  - May freely mix tree search and local search
  - Reduced overhead of the tree search

- **Cons:**
  - Requires an explicit generation of the neighborhood
  - Requires to fully specify each move
Outline

1. Introduction
2. A didactic optimization problem (dTP)
   - Motivations for cooperation
3. A zoo of CP / LS hybrids
   - Sequential combination
   - Master / sub-problem decomposition
   - Improved neighborhood exploration
   - CP Neighborhood search
   - Large neighborhood search
   - Local moves during construction
   - Local moves over a heuristic
A local minimum is reached when no solutions in the neighborhood is better than the current solution.

Usual solution is to use metaheuristics to allow a temporary degradations of the objective.
Large Neighborhoods: Gains

A larger neighborhood means:

- More solutions are considered
- Better chance of avoiding local minima

- Can still use metaheuristics
Large Neighborhoods: loss

- A larger neighborhood also means:
  - More solutions need to be evaluated
  - The complexity of evaluating all solutions makes having neighborhoods too large unattractive

- Unless we don’t evaluate all the solutions!
  - This is where Constraint Programming is useful
Large neighborhood search

Idea: partition the variables of the current solution into two subsets
- A fragment: assignments are kept as they are
- A shuffle set: assignments may be changed
Large neighborhood search on dTP

- From a solution
Large neighborhood search on dTP

- Select a shuffle set

Select a subset of clients $i_1, i_2, \ldots, i_k \subset C$
Large neighborhood search on dTP

Example on dTP

For all clients $i$ from the shuffle set:

- Unassign variable:
  - $visitedBy_i$, $collectedIn_i$
  - $start_i$
- Undo all ordering decisions between $i$ and other clients $j$
- Unassign all cost variables
- Post a cost improvement cut

$cost \leq getValue(cost, S)$
Large neighborhood search on dTP

- Look for a new solution by solving the remaining sub-problem
Large neighborhood search

Exploring the neighborhood

Procedure moveLNS(P,S,algorithm)

// define current sub-problem
Problem P' = P \land (cost \leq getValue(cost,S) - \varepsilon)
propagate(P')
while (not stop())
    VariableSet shuffleSet = defineShuffleSet(P,S)
    foreach variable X | X \notin shuffleSet
        P' = P' \land (X = getValue(X,S))
    if solve(P',algorithm,S) succeeds
        stop iteration
Selecting shuffle sets

- Select a set:
  - large enough to introduce enough flexibility
  - small enough to reduce the overall problem
  - of inter-dependent variables
  - of ill-assigned variables (an improvement can be expected)

- Vary the types of sets that are shuffled
- Vary the size of sets that are shuffled
  - variable neighborhood search
Shuffle sets for dTP

- A set of clients that are
  - Within short distance of some specific client
  - Visited by the same truck
  - Sharing a common type of goods
  - Visited within a common time frame
  - ...
A few hints for LNS with CP

- Use incomplete tree search to speedup the sub-problem solution (e.g. LDS)
- Use strong constraint propagation to reduce the neighborhood exploration
- Compute relaxations to prune non-improving neighbors
- Rather switch neighborhood than fully explore one by backtracking
Outline

1. Introduction
2. A didactic optimization problem (dTP)
   - Motivations for cooperation
3. A zoo of CP / LS hybrids
   - Sequential combination
   - *Master / sub-problem decomposition*
   - Improved neighborhood exploration
   - CP Neighborhood search
   - Large neighborhood search
   - Local moves during construction
   - Local moves over a heuristic
Local Search and Greedy Construction

- Local search is most often applied to complete solutions
- First build a solution, then improve it
- Idea: better repair while building than afterwards.

=> Incremental Local Optimization
Incremental Local Optimization

- The greedy algorithm makes a mistake at step $n$
- The mistake is discovered at step $n+k$
- Try to repair the steps $n..n+k$
- Resume the greedy construction at step $n+k+1$
ILO for general CSPs

A simple incomplete method:

- For a variable ordering $v_1 \ldots v_n$
- Compute a lower bound $lb$
- Start assigning variables
- Choose the value $a_{ik}$ such that $v_i = a_{ik}$ yields the least increase in $lb$
- Whenever $lb$ strictly increases,
  - keep $v_i = a_{ik}$,
  - un-assign all variables linked to $v_i$ and
  - try to re-assign them to find the least increase for $lb$
ILO illustrated on dTP

- Enriched greedy construction scheme:
  - Place clients on a stack
  - Insert them one by one minimizing the insertion cost. For client \( i \), instantiate
    - \( visitedBy_i \)
    - \( succ_i \)
  - Apply local optimization on the truck assigned to \( i \)
    - Change the order of visits \( j \) (forall \( j \mid visitedBy_j = visitedBy_i \))
  - If an improving sub-route is found, change it
**GENeralized Insertion in CP**

- Allows insertion between non-adjacent customers
- Performs a local optimization simultaneously with the insertion

- $c_i$, $c_j$, and $c_k$ are defined as finite domain variable and their value are identified thru the solution of specific constraint programming model link to the original routing model.
Illustration on a search space
Ejection chains during a greedy process

- Recursive version of the ILO approach
  - For a variable ordering $v_1\ldots v_n$
  - Choose the value $a_{ik}$ such that $v_i = a_{ik}$ yields the least increase $\Delta lb$ in $lb$
  - When $\Delta lb > 0$, un-assign some variable $v_i$ so that $lb$ decreases
  - Reassign $v_i$ to some other value
  - Go-on un-assigning / re-assigning past variables until the least increase in $lb$ is found
Ejection chains on dTP

visitedBy_{i_1}

truck k_1
truck k_2
truck k_3
truck k_4
truck k_5

Reschedule k_4
ok without i_2

Reschedule k_1
ok without i_3

Reschedule k_3
ok with all clients
Finding a good ejection chain

- Search for the smallest ejection chain in breadth first search
- Similar to the search for augmenting paths (flows)
Outline

1. Introduction
2. A didactic optimization problem (dTP)
   - Motivations for cooperation
3. A zoo of CP / LS hybrids
   - Sequential combination
   - Master / sub-problem decomposition
   - Improved neighborhood exploration
   - CP Neighborhood search
   - Large neighborhood search
   - Local moves during construction
   - Local moves over a heuristic
Local moves over a heuristic

- LS is defined as variations over a solution.
- LS can also be applied over an encoding of a solution
  - For a greedy CP method, the search heuristic itself is an encoding
  - Idea: instead of exploring the whole tree, explore variations of the constructive heuristic.
Local search over a heuristic

- Two families of methods
  - Local moves over a value ordering heuristic
    - Restricted candidate lists
    - GRASP
    - Discrepancy based search
  - Local moves over a variable ordering heuristic
    - List scheduling heuristics
    - Preference-based programming
Local search over the value selection heuristic

Constructive heuristic
Restricted candidate list

- At each choice point a function $h$ is evaluated for all possible choices:
  - the $k$ “worst” choices (with high value for $h$) are discarded
  - the choice that minimizes $h$ is considered as preferred decision
  - the preferred decision is taken, the remaining choices are taken upon backtracking
Restricted candidate list
Restricted candidate list
GRASP

“Greedy Randomized Adaptative Search Procedure”

At each choice point a function $h$ is evaluated for all possible choices:
- the preferred decision is chosen by a random function biased towards choices having small value for $h$
- the preferred decision is taken
- the process is iterated until a stopping condition is met
GRASP
Discrepancy based search

- Idea: good solutions are more likely to be constructed by following always but a few times the heuristic
  - during search, count the number of times the heuristic is not followed (number of discrepancies)
  - a maximal number of discrepancies is allowed when generating solutions in the tree.
Discrepancy based search

$max\text{Discr} = 1$
Example:

Procedure solve(P)
    while (not stopping condition)
        Solution S = ∅
        int failLimit = 50
        bool result = solveGRASP(P,S,failLimit)
        if (result)
            P = (P ∧ (Cost(P) < Cost(S)))
**Example:**

Procedure `solveGRASP(P,S,failLimit)`

```plaintext
while (unscheduled clients exist and failLimit not reached)
    Client v = selectVariable(P,S)
discardBadValues(P,S)  //Restricted Candidate List
    InsertionPosition position = evaluateRandom(P,S,v)
try (insert(P,S,v, position) OR
    notInsert(P,S,v, position))
```
Local search over the variable selection heuristic

- In some problems, a solution can be described by a variable ordering
  - Natural value ordering heuristics

- Examples:
  - List-scheduling heuristics
  - Configuration problems

- Local moves can be applied on the variable sequence itself
Local moves on a heuristic

- Standard process in Genetic Algorithms:
  - Encode the solution
  - Apply local changes to the encoding
  - Construct the new solution (can be done by a CP-based solver)

- In CP: Preference-based programming
Preference-based programming

Example on Job-Shop scheduling:

- Consider a ordered list of tasks (priority list)

- Choice point: (Schedule asap OR Postpone)
  
  - Take one task at a time from the list and schedule it at its earliest start time
  - otherwise “postpone” the decision on the task for later

- Local moves on the preferred list of tasks generate different schedules
- Use tree search to explore a neighborhood of the preferred list
Conclusion

Real life combinatorial optimization problems often require crafting hybrid optimization methods:
- local search is a technique that can complement CP
- many hybrids are possible

« Is it cookery or alchemy ? » M. Wallace

Recipes and tools are emerging …