Combinations of Local Search and Constraint Programming
Masterclass on Hybrid Methods, Toulouse, 4-5 June 2018

Paul Shaw

IBM Analytics, France
Why?

Local Search
- Provides good solutions quickly
- Good for pure problems with few constraints
- Scales well

Constraint Programming
- Complete, can provide proofs
- Good for problems with lots of constraints
Constraint Programming and Local Search Hybrids in Hybrid Optimization: The Ten Years of CPAIOR

Editors: van Hentenryck, Pascal, Milano, Michela (Eds.)
Outline

- Constraint-based local search
- Local search on the decision path
- Exploring neighborhoods using CP
- Large neighborhood search
- Local search for pruning and propagation
- Local search and dominance
Constraint-based Local Search
By “constraint-based local search”, I mean using local search on CSP-type models. This type of approach was born in the early 1990s. For example:

- Min-conflicts (Minton et al., 1992)
- Breakout (Morris, 1993), GENET (Davenport et al. 1994)
- GSAT (Selman et al., 1992)
Min-conflicts

**Min-Conflicts heuristic:**

*Given*: A set of variables, a set of binary constraints, and an assignment specifying a value for each variable. Two variables *conflict* if their values violate a constraint.

*Procedure*: Select a variable that is in conflict, and assign it a value that minimizes the number of conflicts.² (Break ties randomly.)
Min-conflicts

Min-Conflicts heuristic:

*Given:* A set of variables, a set of binary constraints, and an assignment specifying a value for each variable. Two variables *conflict* if their values violate a constraint.

*Procedure:* Select a variable that is in conflict, and assign it a value that minimizes the number of conflicts.² (Break ties randomly.)

Of course, can fall foul of local minima.

\[
A, B, C \in \{0, 1\}
\]

\[
A + B + C \geq 1 \quad A \neq C \quad B \neq C
\]

Start at \(A = 1, B = 0, C = 0\)

(Only solution is \(A = 0, B = 0, C = 1\))
Min-conflicts

**Min-Conflicts heuristic:**

*Given:* A set of variables, a set of binary constraints, and an assignment specifying a value for each variable. Two variables *conflict* if their values violate a constraint.

*Procedure:* Select a variable that is in conflict, and assign it a value that minimizes the number of conflicts.\(^2\) (Break ties randomly.)

Of course, can fall foul of local minima.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>ABC</td>
</tr>
<tr>
<td>Start</td>
<td>100</td>
</tr>
<tr>
<td>Goal</td>
<td>001</td>
</tr>
<tr>
<td>Goal Nei. 1</td>
<td>000</td>
</tr>
<tr>
<td>Goal Nei. 2</td>
<td>011</td>
</tr>
<tr>
<td>Goal Nei. 3</td>
<td>101</td>
</tr>
</tbody>
</table>
Min-conflicts

**Min-Conflicts heuristic:**
*Given:* A set of variables, a set of binary constraints, and an assignment specifying a value for each variable. Two variables *conflict* if their values violate a constraint.

*Procedure:* Select a variable that is in conflict, and assign it a value that minimizes the number of conflicts.² (Break ties randomly.)

Of course, can fall foul of local minima.

GENET and “breakout” took the same approach:

- At a local minimum, increase the *violation weights* of the constraints in conflict
GSAT

procedure GSAT

Input: a set of clauses $\alpha$, MAX-FLIPS, and MAX-TRIES
Output: a satisfying truth assignment of $\alpha$, if found

begin
  for $i := 1$ to MAX-TRIES
    $T :=$ a randomly generated truth assignment
    for $j := 1$ to MAX-FLIPS
      if $T$ satisfies $\alpha$ then return $T$
      $p :=$ a propositional variable such that a change in its truth assignment gives the largest increase in the total number of clauses of $\alpha$ that are satisfied by $T$
      $T := T$ with the truth assignment of $p$ reversed
    end for
  end for
return “no satisfying assignment found”
end

To fight against local minima:
- Makes a random move among the best
- Restarts the search
Systems

The main CBLS systems are Localizer and COMET

They are toolkits which provide efficient primitives allowing a user to write a model and custom local search procedure.
Systems: Localizer

Localizer uses a declarative language to define *move selection* rules.

Central to this is the idea of *invariants* and *incrementality*.

Invariants are efficiently and incrementally updated when decision variables change their value.

No real concept of constraints or violation
Constraint-based Local Search

Systems: Localizer

Localizer uses a declarative language to define move selection rules.

Variable:
\[ a: \text{array}[1..n] \text{ of boolean}; \]

Invariant:
\[ nbtl[i \text{ in } 1..m] : \text{integer} = \sum(j \text{ in } cl[i].p) a[j] + \sum(j \text{ in } cl[i].n) \neg a[j]; \]
\[ nbClauseSat : \text{integer} = \sum(i \text{ in } 1..m) (nbtl[i] > 0); \]

Satisfiable:
\[ nbClauseSat = m; \]

Objective Function:
\[ \text{maximize } nbClauseSat; \]

Neighborhood:
best move \[ a[i] := \neg a[i] \]
where \( i \) from \( \{1..n\} \)
Constraint-based Local Search

Systems: COMET

COMET uses an essentially imperative language and embeds inside primitives which make it easier to build a local search.

Concept of constraint violation

```
ConstraintSystem S(m);
    S.post(allDifferent(queen));
    S.post(allDifferent(all(i in Size) queen[i] + i));
    S.post(allDifferent(all(i in Size) queen[i] - i));
    m.close();

while (S.violations() > 0)
    selectMax(q in Size)(S.getViolations(queen[q]))
    selectMin(v in Size)(S.getAssignDelta(queen[q],v))
    queen[q] := v;
```
Recently some solvers based on CBLS have been developed. In the Minizinc 2017 challenge:


- **Yuck** [https://github.com/informarte/yuck](https://github.com/informarte/yuck)
Local Search on the Decision Path
Although local search on constraint models can give good results, it seems a shame not to benefit from the principles of locality and constraint propagation.

Some attempts to do this use an essentially local search technique to influence the value heuristic in a constructive search approach.

I’ll look at three examples:

- Decision Repair (Jussien and Lhomme)
- Incomplete Dynamic Backtracking (Prestwich)
- Disco-Novò-Gogo (Sellmann and Ansótegui)
Local Search on the Decision Path

Decision Repair

1. Extend a partial assignment $A = \{d_i\}$ until a dead-end
2. Identify a conflict (set of inconsistent decisions $K \subseteq A$)
3. Choose a decision $d_r \in K$
4. Move to a new assignment $A' = A - \{d_r\} + \{\neg d_r\}$
5. $A \leftarrow A'$ and search goes back to step 1

(There are some tricks to choosing $d_r$ from $K$, by counting how often decisions have recently been involved in conflicts)

Crucially, this algorithm can be considered *local search* because the decision path only changes slightly with each movement.
Local Search on the Decision Path

Depth-First Search compared to Decision Repair

In tree search, to get to the “goal path”, we must move away from the heuristic search path
- In DFS, this is done bottom-up
- In DR, this can be done everywhere on the path

![Depth-First Search](image1)

![Decision Repair](image2)
Incomplete Dynamic Backtracking

1. Extend a partial assignment $A = \{d_i\}$ until a dead-end
2. Choose a subset of decisions $D \subseteq A$, $|D| = b$
3. Move to a new smaller assignment $A' = A - D$
4. $A \leftarrow A'$ and search goes back to step 1

Very much like decision repair, but does it act like local search? (The extension in step 1 is done by a value heuristic, so shouldn’t IDS stick closely to the heuristic?)
Local Search on the Decision Path

Incomplete Dynamic Backtracking

1. Extend a partial assignment \( A = \{d_i\} \) until a dead-end
2. Choose a subset of decisions \( D \subseteq A, |D| = b \)
3. Move to a new smaller assignment \( A' = A - D \)

“We have found that a different type of value ordering heuristic denoted by VH often enhances performance. Instead of finding the best value, it assigns each variable to its last assigned value where possible, with random initial values. This speeds the rediscovery of consistent assignments to subsets of variables. However, IDB attempts to use a different (randomly-chosen) value for one variable each time a dead-end occurs; this appears to help by introducing a little variety.” — S. Prestwich
Local Search on the Decision Path

Disco-Novo-Gogo (restarts with value learning)

1. \[ H = \{ h_i : i \in \{1, \ldots, n\} \land h_i = \text{rnd}(d_i) \} \]

2. Explore \( l \) nodes of DFS tree setting:
   - \( x_i = h_i \) on the left branch
   - \( x_i \neq h_i \) on the right branch

3. Look at the domain of all \( x_i \) in node \( l \)

4. For each \( i \) where \( h_i \) is no longer in the domain of \( x_i \), set \( h_i \) to \( \text{rnd}(d_i - \{h_i\}) \)

5. Update \( l \). Continue search from step 2
Disco-Novo-Gogo (restarts with value learning)

1. \( H = \{ h_i : i \in \{1, \ldots, n\} \land h_i = \text{rnd}(d_i) \} \)

2. Explore \( l \) nodes of DFS tree setting:
   - \( x_i = h_i \) on the left branch
   - \( x_i \neq h_i \) on the right branch

3. Look at the domain of all \( x_i \) in node \( l \)

4. For each \( i \) where \( h_i \) is no longer in the domain of \( x_i \), set \( h_i \) to \( \text{rnd}(d_i - \{h_i\}) \)

5. Update \( l \). Continue search from step 2

Here, a “move” in D-N-G is the change made to \( H \) after every restart completes. The algorithm is *local* in nature as the heuristic tries to stick with the values in \( H \) as in IDS.
These methods share the ability to allow the current “preferred path” to creep towards some ideal where conflicts, backtracks, or right branches are avoided as much as possible.

Contrary to chronological backtracking-type methods different parts of the “preferred path” can be moved towards the ideal roughly fairly throughout the search.

Limited Discrepancy Search, although fair in which branches can be taken to the right, doesn’t have the same ability to “creep” towards an ideal path as it has no notion of a current preference.
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Local Search on the Decision Path

Depth-first search
Local Search on the Decision Path

Depth-first search

Limited discrepancy search
Local Search on the Decision Path

Depth-first search

Limited discrepancy search

“Local” approach
Exploring Neighborhoods using CP
Some work has been done on exploring traditional local search neighborhoods in a CP context.

The motivation was to accelerate the convergence of CP on classical optimization problems, for which good local search neighborhoods are already known.

Additionally, the hope was that when side-constraints or hard cost constraints are present, many neighbors will be illegal, and CP would prune these efficiently.

Addresses *optimization* problems and most work has been on vehicle routing problems.
Exploring Neighborhoods using CP

Simple Example - a “change one variable” neighborhood

Given a constraint programming (minimization) problem $P$ with
- variables $x = \langle x_0, \ldots, x_n \rangle$
- Assume w.l.o.g. that $x_0$ is the objective variable
- Current solution $s = \langle s_0, \ldots, s_n \rangle$

Find best change

Minimize $x_0$ subject to

$$P \land \left( \sum_{i=1}^{n} [x_i \neq s_i] = 1 \right)$$

Find any improving change

$$P \land (x_0 < s_0) \land \left( \sum_{i=1}^{n} [x_i \neq s_i] = 1 \right)$$
Exploring Neighborhoods using CP

Performing a “swap” (method of Pesant et al.)

Eased by adding new “neighborhood” variables and constraints
Exploring Neighborhoods using CP

Performing a “swap” (method of Pesant et al.)

Eased by adding new “neighborhood” variables and constraints

\( a \) and \( b \) are two new variables with domains \( \{1, \ldots, n\} \). They represent the indices of the two \( x \) variables which swap values.

**Neighborhood model**

\[
\begin{align*}
    a, b & \in \{1, \ldots, n\} \\
    a & < b
\end{align*}
\]

**Activity constraints**

\[
\begin{align*}
    x[a] & = s[x[b]] \\
    x[b] & = s[x[a]]
\end{align*}
\]

**Quiescence constraints**

\[
(a = i) \lor (b = i) \lor (x_i = s_i) \quad \forall \ i \in \{1, \ldots, n\}
\]
Exploring Neighborhoods using CP

Another example: 3-rotation

**Neighborhood model**

\[
\begin{align*}
a, b, c & \in \{1, \ldots, n\} \\
s[a] & < s[b], s[a] < s[c], s[b] \neq s[c]
\end{align*}
\]

**Activity constraints**

\[
\begin{align*}
x[a] & = s[x[b]] \\
x[b] & = s[x[c]] \\
x[c] & = s[x[a]]
\end{align*}
\]

**Quiescence constraints**

\[
(a = i) \lor (b = i) \lor (c = i) \lor (x_i = s_i) \quad \forall \, i \in \{1, \ldots, n\}
\]
Exploring Neighborhoods using CP

Generalized TSP insertions (GENIUS-CP)

GENIUS is a powerful “generalized” insertion heuristic for the TSP family of problems.

An empty routing plan can be completed by inserting customers one at a time at the least cost.

The insertion heuristic can also be used for improvement by removing a customer and reinserting it at the best place.
Exploring Neighborhoods using CP

Generalized TSP insertions (GENIUS-CP)

The CP-based neighborhood model allows LS to be carried out on the CP model, boosting convergence speed.

Allows great flexibility in the model compared to hand coding (e.g. MTW, PDP), so great for trying out new ideas.

Can be 10x slower than a non-CP hand coded search (GENIUS-TW)
What causes the inefficiency?

1. CP is quite a heavy deductive technique to use especially when most variables will take a previously known value.

2. The search trees created for most neighborhoods don’t have many branches (and many variables are fixed on the last branch), so the it is rare to be able to discount a large number of illegal neighbors with a single prune.

3. Most “standard” depth-first search explorations of the neighborhood will produce more variable fixings than is strictly necessary.
What happens when exploring a neighborhood? Let's take the simplest one: changing the value of one of $n$ 0-1 variables. Current solution is $A \ldots H = 0$
What happens when exploring a neighborhood? Let’s take the simplest one: changing the value of one of \( n \) 0-1 variables. Current solution is \( A \ldots H = 0 \)

FLIP-8 needs 43 variable fixings
FLIP-\( n \) needs \( n(n + 3)/2 - 1 \) fixings
By using a standard divide-and-conquer approach, we can reduce the number of variable fixings in the worst case.
By using a standard divide-and-conquer approach, we can reduce the number of variable fixings in the worst case.

FLIP-8 needs 32 variable fixings
FLIP-\(n\) needs \(n(\log_2 n + 1)\) fixings
By using a standard divide-and-conquer approach, we can reduce the number of variable fixings in the worst case.

In general, when changing a constant number of variables, complexity can be improved from $O(n^k)$ to $O(n^{k-1} \log n)$.
Exploring Neighborhoods using CP

The main advantage to exploring traditional neighborhoods using CP is that it is easy to study the behavior of different neighborhoods on different models, since both models and neighborhoods are simple to create in a CP framework.

But there are practical issues when using it for *local search*:

- When there are *few* side constraints, a non-CP approach is generally less expensive as constraints play a minor role.
- When there are *many* side constraints, most traditional neighborhoods get easily stuck in local minima.
- CP is often too heavy to be competitive without specific optimizations (even with the above exploration tree).

So...
Large Neighborhood Search
Large Neighborhood Search

What’s the idea?

(a) Solution
(b) Choose
(c) Relax
(d) Reoptimize
Large Neighborhood Search

What’s the idea?

(a) Solution
(b) Choose
(c) Relax
(d) Reoptimize
Large Neighborhood Search

The Basic Algorithm

1. Choose a fragment (mutable elements of the current solution)
2. Decide on the “acceptance” criterion
3. Use a resource-limited search to try find new values of the mutable elements under the acceptance criterion
4. If a new solution was found, it becomes the current solution
5. Go to step 1
LNS is a primarily a local search method
As such it shares with local search:

- **Advantages**
  - Tends to find good solutions quickly
  - Has good scaling properties
  - Can exploit meta-heuristic methods

- **Disadvantages**
  - Not guaranteed to find the optimal (local minima)
  - Needs an initial solution to start
  - Does not (directly) apply to decision problems
Large Neighborhood Search

Some particular properties of LNS

- With respect to standard local search methods
  - Is less affected by the local minima problem
  - When the problem is constrained, can more easily stay in the feasible region
  - If you have a complete solver already, easy to implement

- With respect to complete (CP) methods
  - Very large problems can be tackled (memory)
  - Less dependent on a good branching heuristic
  - Reduces the “horizon” problem of propagation
  - Needs a neighborhood definition
Large Neighborhood Search

Things you can play with

- Choice of the fragment
- Acceptance criterion
- Subproblem resolution
Large Neighborhood Search

Things you can play with

- Choice of the fragment
- Acceptance criterion
- Subproblem resolution
Acceptance criterion

The simplest method is greedy search

\[ x_0 \leq s_0 - \epsilon \]

However, it is easy to implement other methods. Record-to-record travel:

\[ x_0 \leq s^*_0 + \delta \]

Simulated annealing:

\[ x_0 \leq s_0 - T \log(\nu) \]

where \( \nu \) is drawn uniformly in \( (0, 1] \)
Acceptance criterion

The simplest method is greedy search

\[ x_0 \leq s_0 - \epsilon \]

However, it is easy to implement other methods. Record-to-record travel:

\[ x_0 \leq s^*_0 + \delta \]

Simulated annealing: \[ P(\text{accept}) = \exp((s_0 - x_0)/T) \]

\[ x_0 \leq s_0 - T \log(\nu) \]

where \( \nu \) is drawn uniformly in \((0, 1]\)
Large Neighborhood Search

Things you can play with

- Choice of the fragment
- Acceptance criterion
- Subproblem resolution
Large Neighborhood Search

Fragment size

\[ \text{Probability} \]

\[ \text{Fragment Size} \]

\[ P(\text{find improving} \mid \text{exists}) \]

\[ P(\text{exists improving}) \]

Paul Shaw

Combinations of Local Search and Constraint Programming
NK-style problem (Stuart Kauffman)

$n$ 0-1 variables $\langle x_i \rangle$. Each is connected to $k$ other variables (here randomly chosen)

A constraint for each $x_i$ ensures only $p2^{n+1}$ combinations of the values of $x_i$ and its neighbors are legal. Each legal combination is associated with a randomly ascribed fitness value (here 0-10). Maintained by a table constraint.

The goal is to choose values for the $x_i$ variables so as to maximize the total fitness.

I used $N = 200$, $K = 4$ and $p = 0.7$ or 0.8 or 1.0 to look at different problem tightness.
Unconstrained ($p = 1$). Works well from 12-40 variables.
Moderately constrained ($p = 0.8$) Best value looks to be around 60 variables
Tightly constrained ($\rho = 0.7$) Best value looks to be 100-120 variables
Choosing the fragment shape

Simplest method:
- Choose the fragment randomly
- Can actually work quite well in practice
- Easy!

Use problem knowledge (especially when problem structure is well-known and/or problem is very large compared to the fragment size):
- Try to create “coherent” fragments
- Requires a bit more thought
- But, beware of bias. Randomize!

Portfolios can be used (see ALNS)
Large Neighborhood Search

Random selection vs “informed” selection

Random Selection

Informed Selection
Some examples of fragment selection ideas:

- Think about how cost is to be reduced:
  - In scheduling, to reduce the makespan, the critical path must be changed.
  - In bin packing, to reduce the number of bins, one bin must be emptied of its contents.
  - In routing, long arcs will increase cost, so place customers which are close together (not far apart) in the fragment.
  - In max clique, any node with degree $d$ in a solution of value $d + 1$ must be removed from the clique.
Some examples of fragment selection ideas:

- Think about constraints and structure:
  - Models often have “helper” variables whose values depend on the basic decision variables (e.g. $\text{area} = \text{wid} \times \text{height}$). We must let the values of these variables change!
  - In a constraint $\sum_i b_i = 1$, one must include the variable at value 1 in the fragment.
  - On a timeline, place all elements in a particular time window in the fragment.
  - In timetabling or matrix-like problems, relax variables related to one dimension: rooms, resources, days.
Some examples of fragment selection ideas:

- Think about adapting the fragment over time:
  - Make the fragment potentially bigger as search goes on.
  - In a 0-1 problem, if you know that good solutions have many 1s, favor adding variables with value 0 to the fragment. Reduce this tendency over time.
Large Neighborhood Search

Selection rule used for vehicle routing in [Shaw, 98]

1. Let $E = \{1, \ldots, n\}$ be the solution elements
2. The initial fragment $F$ is a single random element of $E$
3. Let $r$ be a random element of $F$
4. Form a sequence $\langle z_i \rangle$ from the set $E - F$ such that $R(s, r, z_i) \geq R(s, r, z_{i+1})$
5. Let $k = 1 + \lfloor |E - F| y^d \rfloor$ ($y$ random in $[0, 1)$)
6. Add $z_k$ to $F$; go to step 3 if the fragment is too small

For the VRP, close customers on the same route are more related. $R(s, a, b) = 1/( \text{dist}(a, b) + \text{differentVehicles}(s, a, b) )$
Large Neighborhood Search

Things you can play with

- Choice of the fragment
- Acceptance criterion
- Subproblem resolution
Large Neighborhood Search

Subproblem resolution

In a CP context, this is a tree search with propagation.

Typically, depth-first search is used, although other methods like limited discrepancy search are possible.

The resource limit is normally based on a number of backtracks.
  - Normally “not too big” values are fine

Variable and value selection rules tend to be a bit less important than in complete search, but if you have good rules for the entire problem, you should re-use them here!
  - Again, beware of bias. Randomize!

Again, portfolios can be used (see ALNS)
Subproblem resolution

Additional things to consider:

- Should you take the first improving solution, or find the best solution in the fragment?
- To create the fragment, should you build a sub-model corresponding to the fragment from scratch? Or have a complete model, and fix all variables not in the fragment.
- Impose additional constraints? Say $cost = mkspan + late$. Then insist $cost$ reduced but $mkspan$ not degraded. Can increase $P(find$ $improving | exist)$
- In scheduling, although start/end times should be relaxed, we wish to maintain relative order for many tasks: impose this by adding precedence constraints in the subproblem
What’s the effect of propagation strength on LNS?

I changed the basic NK constraint computing the fitness function at each node from a global *table* constraint establishing *domain consistency* to one expressed in disjunctive form (much weaker)
Different fragment sizes with a lower level of consistency.
Comparing strong and weak consistency
Comparing strong and weak consistency
Comparing strong and weak consistency
Large Neighborhood Search

Meta-strategies: are they useful for LNS? In particular, can restarting help?
Meta-strategies: are they useful for LNS? In particular, can restarting help?

(Maxima are the best of 10 runs)

<table>
<thead>
<tr>
<th>p</th>
<th>fragment size</th>
<th>$F_{med}$ @ 1000s</th>
<th>$F_{max}$ @ 100s</th>
<th>$F_{max}$ @ 10s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>30</td>
<td>1422</td>
<td>1431</td>
<td>1429</td>
</tr>
<tr>
<td>0.8</td>
<td>60</td>
<td>1315.5</td>
<td>1337</td>
<td>1314</td>
</tr>
<tr>
<td>0.7</td>
<td>100</td>
<td>1231</td>
<td>1229</td>
<td>1219</td>
</tr>
</tbody>
</table>
Meta-strategies: are they useful for LNS? In particular, can restarting help?

(Maxima are the best of 10 runs)

<table>
<thead>
<tr>
<th>$p$</th>
<th>fragment size</th>
<th>$F_{med}$ @ 1000s</th>
<th>$F_{max}$ @ 100s</th>
<th>$F_{max}$ @ 10s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>30</td>
<td>1422</td>
<td>1431</td>
<td>1429</td>
</tr>
<tr>
<td>0.8</td>
<td>60</td>
<td>1315.5</td>
<td>1337</td>
<td>1314</td>
</tr>
<tr>
<td>0.7</td>
<td>100</td>
<td>1231</td>
<td>1229</td>
<td>1219</td>
</tr>
</tbody>
</table>

Yes! Clearly even a simple restart strategy would help LNS on the less constrained instances.
Important things about using LNS

- The most important thing is the fragment selection. Try to find the sweet spot. Use domain knowledge if you have it. Use random selection as a baseline. Consider portfolios.
- The subproblem resolution tends to be less important, but use a good branching heuristic if you have it. Keep the resource limit reasonable e.g. 100 backtracks.
- You may be able to apply avoid creating “trivial” fragments.
- Look out for search stagnation and apply meta-strategies. Easiest is multi-restart.
- Have some level of randomness everywhere. The benefit of randomness tends increase as search goes on.
Large Neighborhood Search

Introductory material:


Large Neighborhood Search

Future directions?

- LNS for decision problems
- Constraining the sub-problem
- Meta-heuristics adapted to LNS
- Problem reformulation for LNS suitability
- Machine learning
Local Search for Pruning and Propagation
Basic idea here is to find a *witness* which serves as an existence proof of a result which can be used in pruning or propagation.

I’ll look at three examples:

- Prove that certain values *cannot* be domain-filtered (Galinier *et al.*)
- Prove that the current branch *cannot* lead to a solution (Harvey and Sellmann)
- Prove that the current branch is *dominated* by another and can be pruned (Focacci and Shaw)
Support Discovery in Some-Different

All-different and some-different

- An “all-different” constraint enforces difference between all pairs of variables
- A “some-different” constraint enforces difference between a specifiable subset of pairs of variables
- Domain consistency of “some-different” is NP-complete (equivalent to graph coloring)
Support Discovery in Some-Different

Let $N = \{1, 2, \ldots, n\}$, $C \subseteq \{\langle i, j \rangle : i, j \in N \land i < j\}$, and $D = \{d_i : i \in N \land d_i \subseteq K\}$ $K$ is a color set.

Some-different can be considered as the domain consistency of the coloring problem $P$:

Find $x = \langle x_1, \ldots, x_n \rangle$ such that

$$\forall i \in N \ x_i \in d_i$$

$$\forall \langle i, j \rangle \in C \ x_i \neq x_j$$

The domain consistency problem $DC(P)$ is then to either prove that $P$ has no solution or to find reduced domains $d' = \langle d'_1, \ldots, d'_n \rangle$ such that $d'_i \subseteq d_i$ and $k \in d'_i$ if and only if there is a solution $x$ to $P$ with $x_i = k$. 
Each complete coloring $x$ gives a point coloring $\langle i, x_i \rangle$ for each node $i$.

By finding as many complete colorings as possible, we hope to cheaply identify many unique point colorings and thus required values $r_i$ for each node $i$.

The reduced domain will be contained between these required values and the initial domain: $r_i \subseteq d'_i \subseteq d_i$.

Complete colorings are found using tabu search which stops when either we know that $\forall i \in N \ d'_i = d_i$ or when a number of iterations pass without finding any new required member.
Support Discovery in Some-Different

The tabu search process tries to minimize two different objectives which are blended together.

Let \( L \subseteq \{ \langle i, k \rangle : i \in N \land k \in d'_j \} \) be the set of point colorings found up to the current point in the algorithm. The cost function used is:

\[
f(x) = \left( \alpha \sum_{\langle i, j \rangle \in C} [x_i = x_j] \right) + |L \cap \{ \langle i, x_i \rangle : i \in N \}|
\]

\( \alpha \) is adjusted according to the difficulty of finding complete assignments.
Support Discovery in Some-Different

After tabu search, if \( \exists_{i \in N} r_i \neq d_i \), then for each \( k \in (d_i - r_i) \), \( x_i \) is fixed to \( k \) and a complete search determines if the remaining problem can be colored.

If the subproblem cannot be colored, \( k \notin d'_i \).

If the subproblem can be colored, \( k \in d'_i \) and the coloring is used to enrich \( \langle r_i \rangle \).
How can \( nm \) golfers play in \( m \) groups of \( n \) golfers each week over \( p \) weeks such that no golfer plays with another more than once?

Solution to a 5-3-7 problem.

<table>
<thead>
<tr>
<th>week 1</th>
<th>group 1</th>
<th>group 2</th>
<th>group 3</th>
<th>group 4</th>
<th>group 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td>13</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>7</td>
<td>12</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>10</td>
<td>14</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>11</td>
<td>15</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>group 1</td>
<td>group 2</td>
<td>group 3</td>
<td>group 4</td>
<td>group 5</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>week 1</td>
<td>1 2 3</td>
<td>4 5 6</td>
<td>7 8 9</td>
<td>10 11 12</td>
<td>13 14 15</td>
</tr>
<tr>
<td>week 2</td>
<td>1 4 7</td>
<td>2 5 8</td>
<td>3 * *</td>
<td>* * *</td>
<td>* * *</td>
</tr>
</tbody>
</table>
Social Golfers: Pruning and propagation

<table>
<thead>
<tr>
<th>week 1</th>
<th>group 1</th>
<th>group 2</th>
<th>group 3</th>
<th>group 4</th>
<th>group 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>week 1</td>
<td>1 2 3</td>
<td>4 5 6</td>
<td>7 8 9</td>
<td>10 11 12</td>
<td>13 14 15</td>
</tr>
<tr>
<td>week 2</td>
<td>1 4 7</td>
<td>2 5 8</td>
<td>3 * *</td>
<td>* * *</td>
<td>* * *</td>
</tr>
</tbody>
</table>

6

9

12 11

10

15 14

13
Social Golfers: Pruning and propagation

<table>
<thead>
<tr>
<th>week 1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>group 1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>group 2</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>group 3</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>group 4</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>group 5</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Diagrams:
- Group 1: 9
- Group 2: 12 - 11
- Group 3: 15 - 14
- Group 4: 10
- Group 5: 13
### Social Golfers: Pruning and propagation

<table>
<thead>
<tr>
<th></th>
<th>group 1</th>
<th>group 2</th>
<th>group 3</th>
<th>group 4</th>
<th>group 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>week 1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>week 2</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

![Graph for Social Golfers](image-url)
The clique 10-11-12 needs three open groups.
No extension to a complete solution. We can prune.

<table>
<thead>
<tr>
<th>week 1</th>
<th>group 1</th>
<th>group 2</th>
<th>group 3</th>
<th>group 4</th>
<th>group 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>week 1</td>
<td>1 2 3</td>
<td>4 5 6</td>
<td>7 8 9</td>
<td>10 11 12</td>
<td>13 14 15</td>
</tr>
<tr>
<td>week 2</td>
<td>1 4 7</td>
<td>2 5 8</td>
<td>3 6 9</td>
<td>10 * *</td>
<td>11 * *</td>
</tr>
</tbody>
</table>

12?
Here, search can also be pruned.
Social Golfers: Pruning and propagation

Here, search can also be pruned. Group 3 needs to have two empty places, one for each 3-clique.
# Social Golfers: Pruning and propagation

Even here we can do something. Propagate!

<table>
<thead>
<tr>
<th>week 1</th>
<th>group 1</th>
<th>group 2</th>
<th>group 3</th>
<th>group 4</th>
<th>group 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>week 1</td>
<td>1 2 3</td>
<td>4 5 6</td>
<td>7 8 9</td>
<td>10 11 12</td>
<td>13 14 15</td>
</tr>
<tr>
<td>week 2</td>
<td>1 4 7</td>
<td>2 5 8</td>
<td>3 * *</td>
<td>* * *</td>
<td>* * *</td>
</tr>
</tbody>
</table>
Members of the two 3-cliques take up the two remaining slots in the third group. Therefore, neither golfer 6 nor golfer 9 can be assigned to the third group.
Cliques are discovered by local search through an oscillation-based process of expanding and contracting a potential clique.
Pruning branches in TSPs using local search

Exploration of a partial tour during tree search

A
B
C
D
E
F
G
We would like to avoid partial tours like this!

**Reasoning:** Tours with crossed arcs are not locally optimal
And locally optimal solutions cannot be globally optimal
Pruning branches in TSPs using local search

Exploration of a partial tour during tree search

Can be done by adding \( O(|N|^2) \) constraints

\[
d_{i, \text{next}(i)} + d_{j, \text{next}(j)} \leq d_{i, \text{next}(j)} + d_{j, \text{next}(i)} \quad \forall i, j \in N, i < j
\]
Exploration of a partial tour during tree search

Problem: Such constraints can make finding any solution hard
- Not great in an anytime context
- Can even slow down proof because of lack of upper bounds
Pruning branches in TSPs using local search

Exploration of a partial tour during tree search

Basic idea: Disallow the current partial tour if it is not locally optimal and the improved tour has already been explored earlier in the search tree.
Suppose all extensions of $A \rightarrow B \rightarrow C$ already explored.

![Diagram showing the graph of branches in TSPs using local search.](image)
Suppose all extensions of $A \rightarrow B \rightarrow C$ already explored

Add cities in the current tour not in $ABC$ to the green tour
- In any order, ensuring that $G$ (last city) remains last
- Now, the green tour has at least all the cities of the current partial tour
Suppose all extensions of $A \rightarrow B \rightarrow C$ already explored

Perform local search on the green partial tour, but don’t move $ABC$ or $G$
Suppose all extensions of $A \rightarrow B \rightarrow C$ already explored.

Perform local search on the green partial tour, but don’t move $ABC$ or $G$.

The goal is to find a partial tour shorter than the current one.
Suppose all extensions of $A \rightarrow B \rightarrow C$ already explored

- The green partial tour $g$ is no longer than the current partial tour $t$
- $g$ visits the same set (or a superset) of the cities that $t$ visits
- $g$ ends at the same city as $t$ (here, city $G$)
- Thus, $g$ can be extended to a full tour $g^*$ no worse than any full extension of $t$
- But, we already explored $g^*$ since it is an extension of $ABC$
- **We can abandon extensions to $t$ and prune the search**
Suppose all extensions of $A \rightarrow B \rightarrow C$ already explored.

When *time windows are present*, the goal is to find a (green) partial tour which:
- is no longer than the current one (quality, as before)
- departs from $G$ no later than the current one (feasibility)
Local Search and Dominance
Let’s look at this again

We can add constraints to avoid crossed arcs (although it might be a bad idea in that particular case).

Is there a generalization?

This idea has been used for a long time. A formalism was published by Chu and Stuckey at CP 2012.
The idea itself is really simple: we don’t want to explore assignments which can be bettered via a local search move, so let’s add constraints which disallow these assignments.

The move considered must preserve feasibility too. We need to ensure that if the assignment before the move was feasible, then so will the assignment after the move.

Essentially:

If then new assignment is strictly better and at least as feasible as the original assignment, the original assignment should be forbidden.
Local Search and Dominance

Method:

1. Assume a problem $P$, a function $f$ to minimize and an assignment $s$
2. $\Theta$ is a set of transformations likely to map solutions to better solutions
3. For each $\theta \in \Theta$
   - Find constraint $\kappa_F$ obeying $(P(s) \land \kappa_F(s)) \Rightarrow P(\theta(s))$
   - Find constraint $\kappa_I$ obeying $(P(s) \land \kappa_I(s)) \Rightarrow f(\theta(s)) < f(s)$
   - Add the constraint $\neg \kappa_F \land \neg \kappa_I$
Local Search and Dominance

Examples

- Knapsack
- Bin packing
- Maximum independent set
- Max-cut
Knapsack

Maximize $\sum_{i=1}^{n} v_i x_i$ such that $\sum_{i=1}^{n} w_i x_i \leq C$

Move: introduce a new item to the knapsack: $x_k = 0 \rightarrow x_k = 1$

The new assignment is of course better if it is feasible. It is feasible if $x_k = 0 \land \sum_{i \neq k} w_i x_i \leq C - w_k$.

The dominance is formed from the negation of this constraint:

$$x_k = 1 \lor \sum_{i \neq k} w_i x_i > C - w_k$$

or

$$(C' - w_k + 1)x_k + \sum_{i \neq k} w_i x_i > C - w_k$$
Knapsack II

Maximize $\sum_{i=1}^{n} v_i x_i$ such that $\sum_{i=1}^{n} w_i x_i \leq C$

Move: Swap the positions (in/out) of two objects $j$ and $k$

Better if value goes up:

$$v_j x_k + v_k x_j > v_j x_j + v_k x_k$$

Definitely feasible if weight does not go up:

$$w_j x_k + w_k x_j \leq w_j x_j + w_k x_k$$

Taking the combination and negating (new configuration must be no better or less feasible than previous), we get:

$$w_i \geq w_j \land v_i < v_j \Rightarrow x_i \leq x_j$$
Bin packing

Pack \( n \) objects into containers of capacity \( C \) so that the number of containers is minimized. Here, we will only need the variables \( \langle l_i \rangle \) representing the load of each bin.

Move: combine the contents of two bins \( i \) and \( j \) in one. Clearly the new solution is better. It is feasible if \( l_i + l_j \leq C \)

Dominance constraint:

\[
    l_i + l_j > C
\]

This leads to other simple rules such as the sum of the loads of the two lightest bins must be more than \( C \)

Also, no more than one bin can be half full or less.
Maximum independent set

Maximize $\sum_{i=1}^{n} x_i$ with $x_i \in \{0, 1\}$ and $\forall \langle j, k \rangle \in E, x_i + x_j \leq 1$
Local Search and Dominance

Maximum independent set

Maximize $\sum_{i=1}^{n} x_i$ with $x_i \in \{0, 1\}$ and $\forall \langle j, k \rangle \in E, x_i + x_j \leq 1$

Move: add a new node $k$ to the independent set. The move necessarily leads to a better solution. It is feasible if it does not conflict with another node in the independent set.

$$(x_k = 0) \land \sum_{\{j: \langle j, k \rangle \in E\}} x_j = 0$$

Negating, the dominance constraint becomes:

$$x_k + \sum_{\{j: \langle j, k \rangle \in E\}} x_j \geq 1$$
Max-cut

Maximize \( \sum_{(i,j) \in E} |x_i - x_j| \) with \( x_i \in \{0, 1\} \)

Move: transfer a node \( k \) from one side of the cut to another. The move is necessarily feasible. It leads to a better solution if the number of foreign neighbors is strictly more than the number before the move.
Local Search and Dominance

Max-cut

Maximize $\sum_{\langle i,j \rangle \in E} |x_i - x_j|$ with $x_i \in \{0, 1\}$

Let $\nu_i$ be an integer set containing the neighbors of node $i$ and the number of “right” neighbors is $\sigma_i = \sum_{j \in \nu_i} x_j$

The move is better if:

$((x_i = 0) \land \sigma_j < \frac{|\nu_i|}{2}) \lor ((x_i = 1) \land \sigma_j > \frac{|\nu_i|}{2})$

Negating, the dominance constraint is:

$((x_i = 0) \Rightarrow \sigma_i \geq \frac{|\nu_i|}{2}) \land ((x_i = 1) \Rightarrow \sigma_i \leq \frac{|\nu_i|}{2})$

Which reduces to:

$\frac{|\nu_i|}{2} \leq \frac{|\nu_i|}{2} x_i + \sigma_i \leq |\nu_i|$
Local Search and Dominance

Max-cut

Different runs with and without different dominance rules

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Number of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

Standard model
+ dominance
+ dominance & tiebreak

Paul Shaw
Combinations of Local Search and Constraint Programming
Combinations of Local Search and Constraint Programming
Masterclass on Hybrid Methods, Toulouse, 4-5 June 2018

Paul Shaw
IBM Analytics, France