Constraint Programming

Optimization and Large-Neighborhood Search
(version of 15 November 2022)
A CSP is a constraint satisfaction problem:
- A triplet $\langle X, D, C \rangle$ with constraints $C$ on variables $X$ of domain $D$.

A COP is a constrained optimization problem:
- A quadruplet $\langle X, D, C, f \rangle$, where the objective function $f$ is defined over a subset of $X$.
- Without loss of generality, we assume $f$ is to be minimized.

What we want for a COP:
Find among the feasible solutions of $\langle X, D, C \rangle$, that is in $\mathcal{S}(\langle X, D, C \rangle)$, a solution $\sigma^*$ that minimizes $f$. 

How shall we do this?

▷ Idea:
  – Step 1: Find a feasible solution $\sigma_0$ (i.e., establish that the CSP part is feasible).
  – Step 2: Add the constraint $c_0 \not= f(\sigma) < f(\sigma_0)$.
  – Step 3: Continue solving.
  – Step 4: At iteration $i$:
    • If we find a feasible solution $\sigma_i$, then tighten by adding the “betterness” constraint $c_i \not= f(\sigma) < f(\sigma_i)$.
    • If we do not find a feasible solution, then the previous solution, $\sigma_{i-1}$, is a global minimum.
  – The process ends when some iteration does not find a feasible solution.

▷ Caveats:
  – Solutions are found “deep” in the search tree.
  – The “betterness” constraints must not disappear when backtracking!
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to prove its optimality.
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to prove its optimality.
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to prove its optimality.
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to prove its optimality.
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to prove its optimality.
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to prove its optimality.
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to \textit{prove} its optimality.
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to *prove* its optimality.
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to prove its optimality.
Example: Minimization Problem

Each time a solution is found, the next one is strictly better. Here, 8 solutions were discovered before the last, optimal one. Note that, after a best solution was found, we continued exploring in order to prove its optimality.
Minimization in MiniCP

IntVar totCost = sum(weightedDist);

Objective obj = cp.minimize(totCost);

DFSearch dfs = makeDfs(cp, firstFail(x));

dfs.onSolution(() -> System.out.println("objective:" + totCost.min()));

SearchStatistics stats = dfs.optimize(obj);
In practice

```java
public interface Objective {
    void tighten();
}
Objective ADT
```

```java
public class Minimize implements Objective {
    private int bound = Integer.MAX_VALUE;
    private final IntVar x;

    public Minimize(IntVar x) {
        this.x = x;
        x.getSolver().onFixPoint(() -> x.removeAbove(bound));
    }

    public void tighten() {
        this.bound = x.max() - 1;
        throw InconsistencyException.INCONSISTENCY;
    }
}
```

Called each time a solution is found during the search in order to let the tightening of the bound occur such that the next-found solution is better.

Pruning w.r.t. the bound is done at (the start of) every fixpoint computation.

Called when finding $\alpha_i$ to update the bound and make sure we backtrack.
public class DFSearch {
    private Supplier<Procedure[]> branching;
    private StateManager sm;
    private List<Procedure> solutionListeners = new LinkedList<Procedure>();
    private List<Procedure> failureListeners = new LinkedList<Procedure>();

    public DFSearch(StateManager sm, Supplier<Procedure[]> branching) {
        this.sm = sm;
        this.branching = branching;
    }

    public void onSolution(Procedure listener) { solutionListeners.add(listener); }
    public void onFailure(Procedure listener) { failureListeners.add(listener); }
    private void notifySolution() { solutionListeners.forEach(s -> s.call());}
    private void notifyFailure() { failureListeners.forEach(s -> s.call());}

    public SearchStatistics optimize(Objective obj) {
        onSolution(() -> obj.tighten());
        return solve(new SearchStatistics());
    }

    private void dfs(SearchStatistics statistics) { ... }
}
Hookup of the Objective into the Solver

```java
public class DFSearch {
    private Supplier<Procedure[]> branching;
    private StateManager sm;
    private List<Procedure> solutionListeners = new LinkedList<Procedure>();
    private List<Procedure> failureListeners  = new LinkedList<Procedure>();

    public DFSearch(StateManager sm, Supplier<Procedure[]> branching) {
        this.sm = sm;
        this.branching = branching;
    }

    public void onSolution(Procedure listener){ solutionListeners.add(listener);}
    public void onFailure(Procedure listener) {  failureListeners.add(listener);}

    private void notifySolution() { solutionListeners.forEach(s -> s.call());}
    private void notifyFailure()  {  failureListeners.forEach(s -> s.call());}

    public SearchStatistics optimize(Objective obj) {
        onSolution(() -> obj.tighten());
        return solve(new SearchStatistics());
    }

    private void dfs(SearchStatistics statistics) { ... }
}
```

Tighten objective when finding $\sigma_i$. 
Minimization: Summary

Minimization is implemented as a regular DFS that enumerates feasible solutions under two listeners:

- onSolution: the objective bound is tightened (to the current bound minus 1);
- onFixPoint: the objective variable is restricted to be at most the bound.
The Weakness of CP

- Potentially huge search tree for optimization problems.
- Poor exploration of the search space.

- Some problems are just too hard to solve.
- Solution: Adopt a local search (LS) style to discover good solutions faster.
The Weakness of CP

- Potentially huge search tree for optimization problems.
- Poor exploration of the search space.

- Some problems are just too hard to solve.
- Solution: Adopt a local search (LS) style to discover good solutions faster.
How to fix this? By local search!

› When solving gets stuck for too long without improving: restart at another place.
› Intensify the search where it looks promising.
How to fix this? By local search!

- When solving gets stuck for too long without improving: restart at another place.
- Intensify the search where it looks promising.
Large-Neighborhood Search (LNS)

LNS = Fix + Relax + Restart

1. Find a first feasible solution $S^*$.
2. Randomly relax $S^*$ and re-optimize under a search limit: 
   \[ \text{relax} = \text{fix some variables to their values in } S^* \text{ and unfix the other variables.} \]
3. Replace $S^*$ by the best solution found
Large-Neighborhood Search (LNS)

LNS = Fix + Relax + Restart

1. Find a first feasible solution $S^*$.
2. Randomly relax $S^*$ and re-optimize under a search limit:
   \[ \text{relax} = \text{fix some variables to their values in } S^* \text{ and unfix the other variables.} \]
3. Replace $S^*$ by the best solution found

It can be more problem-specific than that. For example, in scheduling, good practice is:
\[ \text{relax} = \text{keep some of the precedences from the best solution.} \]
Advantages of LNS over classical LS

- The neighborhood is large:
  - No need for a meta-heuristic in order to avoid local optima.

- Modeling power of CP (declarative):
  - No need for designing a complex neighborhood.
  - Ease of implementation.

- Scalability of LS:
  - Very good «any-time» behavior.
Example: How to solve QAP with LNS?

Without LNS:

```java
int[][] w = new int[n][n]; // Weights
int[][] d = new int[n][n]; // Distance

Solver cp = makeSolver();
IntVar[] x = makeIntVarArray(cp, n, n);
cp.post(allDifferent(x));
IntVar[] weightedDist = new IntVar[n * n];
int ind = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        weightedDist[ind++] = mul(element(d, x[i], x[j]), w[i][j]);
IntVar totCost = sum(weightedDist);
Objective obj = cp.minimize(totCost);
DFSSearch dfs = makeDfs(cp, firstFail(x));
```
int[][] w = new int[n][n]; // Weights
int[][] d = new int[n][n]; // Distance  (reading hidden)
Solver cp = makeSolver();
IntVar[] x = makeIntVarArray(cp, n, n);
// Constraints and objective ... (hidden)
DFSearch dfs = makeDfs(cp, firstFail(x));

int[] xBest = IntStream.range(0, n).toArray();
dfs.onSolution(() -> {
    for (int i = 0; i < n; i++)
        xBest[i] = x[i].min();
});

int nRestarts = 1000;
int failLimit = 100;
Random rand = new java.util.Random(0);
for (int i = 0; i < nRestarts; i++) {
    dfs.optimizeSubjectTo(obj, 
    statistics -> statistics.numberOfFailures() >= failLimit, () -> {
      for (int j = 0; j < n; j++)
          if (rand.nextInt(100) < 75)
              cp.post(equal(x[j], xBest[j]));
    });
}