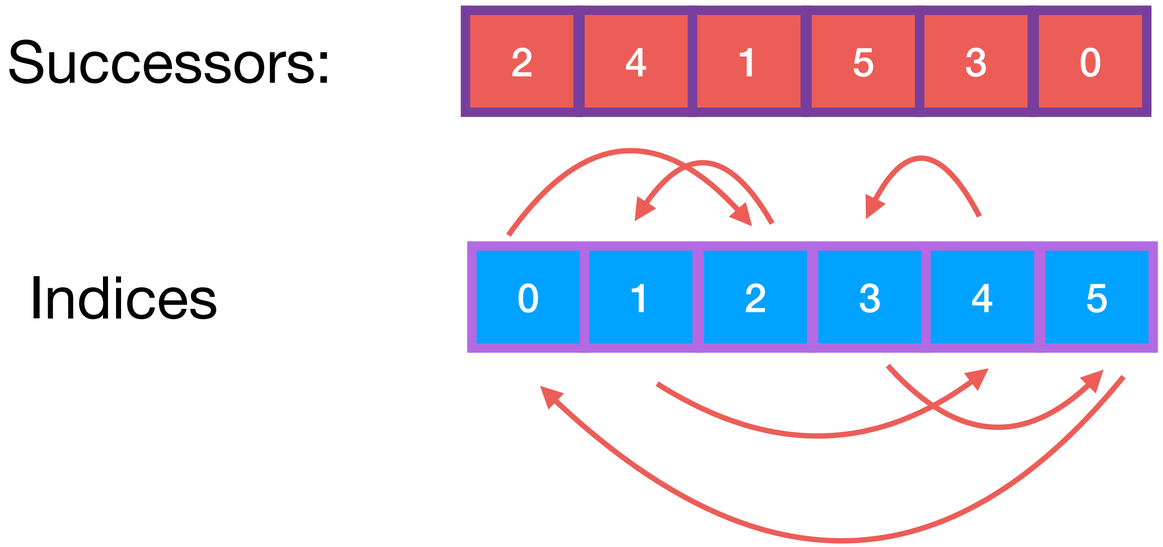
Constraint Programming

Circuit Constraint + Optimization + Large-Neighborhood Search

The Circuit constraint

The Circuit constraint enforces a Hamiltonian cycle on an array of successor variables.



The successors must clearly all be different, but this is not enough!

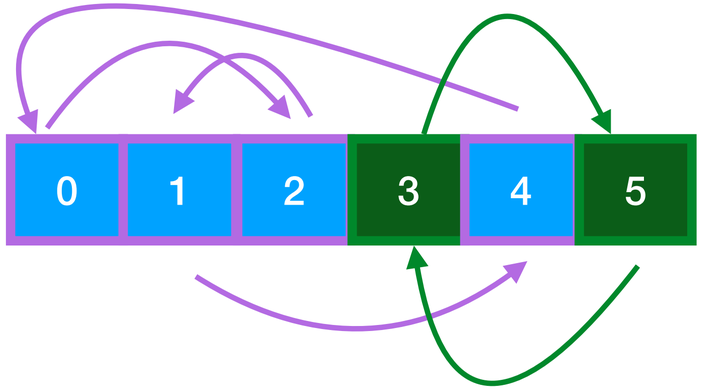
We must also guarantee that the array forms a proper cycle, without sub-cycles.

The Circuit constraint

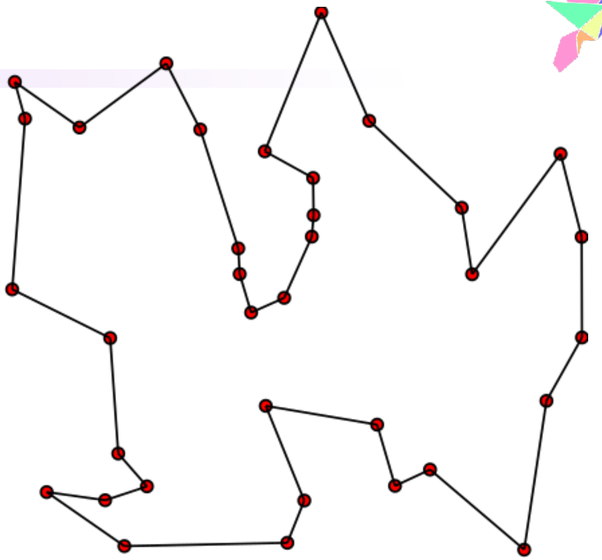
Example of violation of a Circuit constraint: there are two sub-cycles!

Successors:

Indices



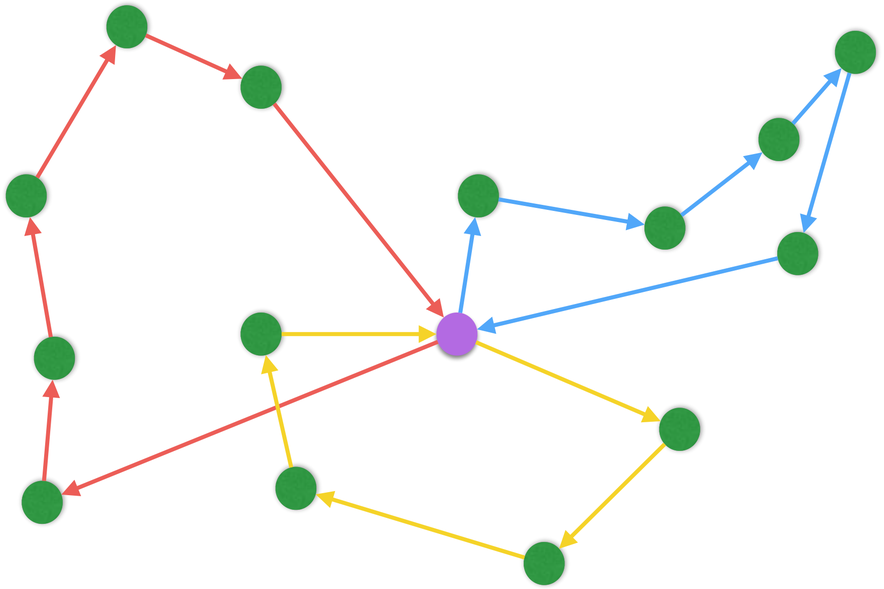
Application: TSP



```
int n;
int[][] distanceMatrix = reader.getMatrix(n, n);

Solver cp = makeSolver(false);
IntVar[] succ = makeIntVarArray(cp, n, n);
IntVar[] distSucc = makeIntVarArray(cp, n, 1000);

cp.post(new Circuit(succ));

for (int i = 0; i < n; i++) {
    cp.post(new Element1D(distanceMatrix[i], succ[i], distSucc[i]));
}

IntVar totalDist = sum(distSucc);

Objective obj = cp.minimize(totalDist);

DFSearch dfs = makeDfs(cp, firstFail(succ));
```

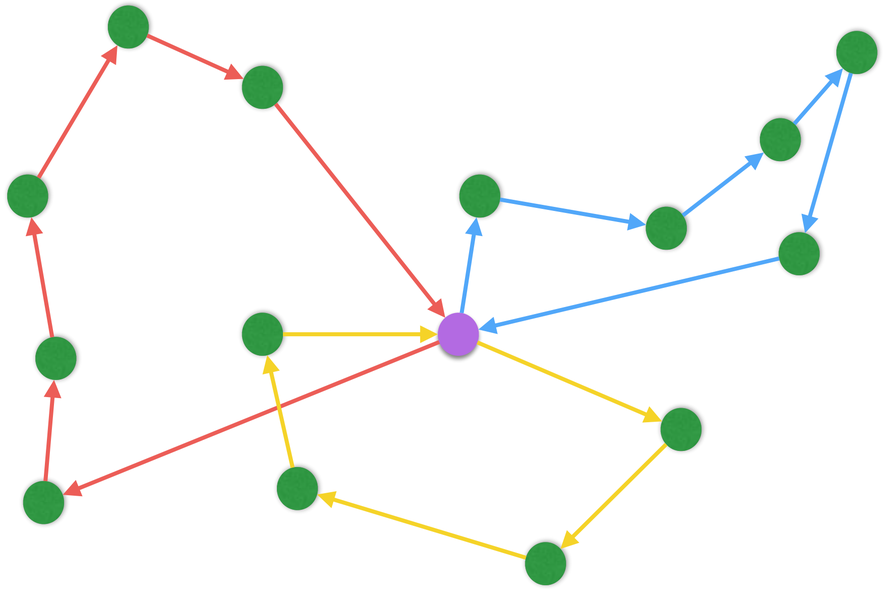
Application: Vehicle routing

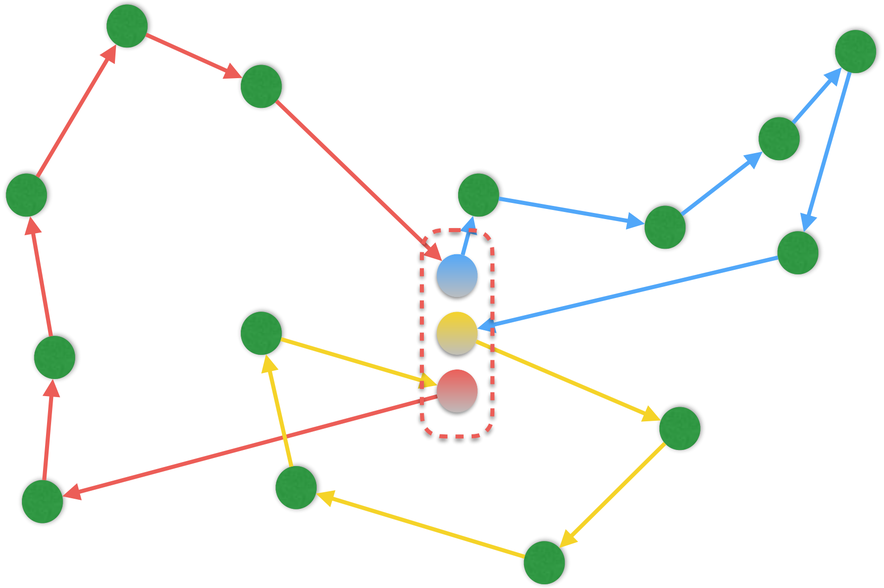
- 1 depot, 3 vehicles, 1 distance matrix.
- Visit all the customers and minimize the total distance.
- How to model this with a Circuit constraint?



Application: Vehicle routing

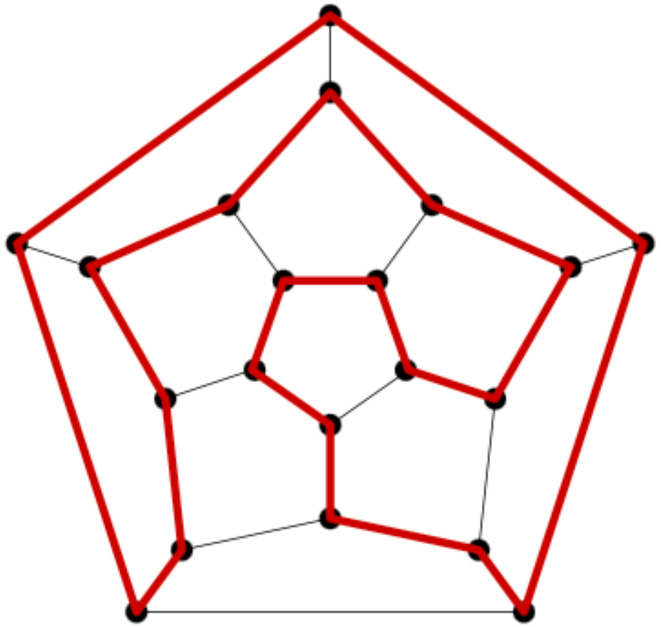
- Duplicate the depot for every vehicle.
- Now we can state a Circuit constraint by threading the tours of the vehicles through the depots into a giant tour:





Hamiltonian-cycle problem

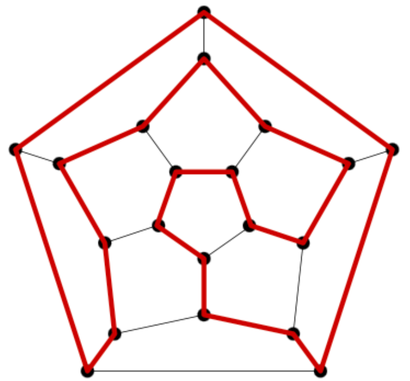
- Find a cycle that visits each node in an undirected graph exactly once



- Determining whether such a cycle exists in a graph is NP-complete

Achieving domain consistency for Circuit is NP-hard

- Achieving domain consistency for a Circuit constraint is NP-hard
- Reduction from Hamiltonian cycle in an undirected graph G=(V,E):
  - Introduce a variable $succ_i$ for every node i in V,
    denoting the successor of node i:
    $D(succ_i) = \{ j : (i,j) \text{ in } E \}$
  - Apply domain-consistent filtering for Circuit($succ_1$,…,$succ_n$):
    - If no failure: YES! there exists a Hamiltonian cycle in G.
    - Otherwise: NO! there exists no Hamiltonian cycle in G.



Constraint Programming

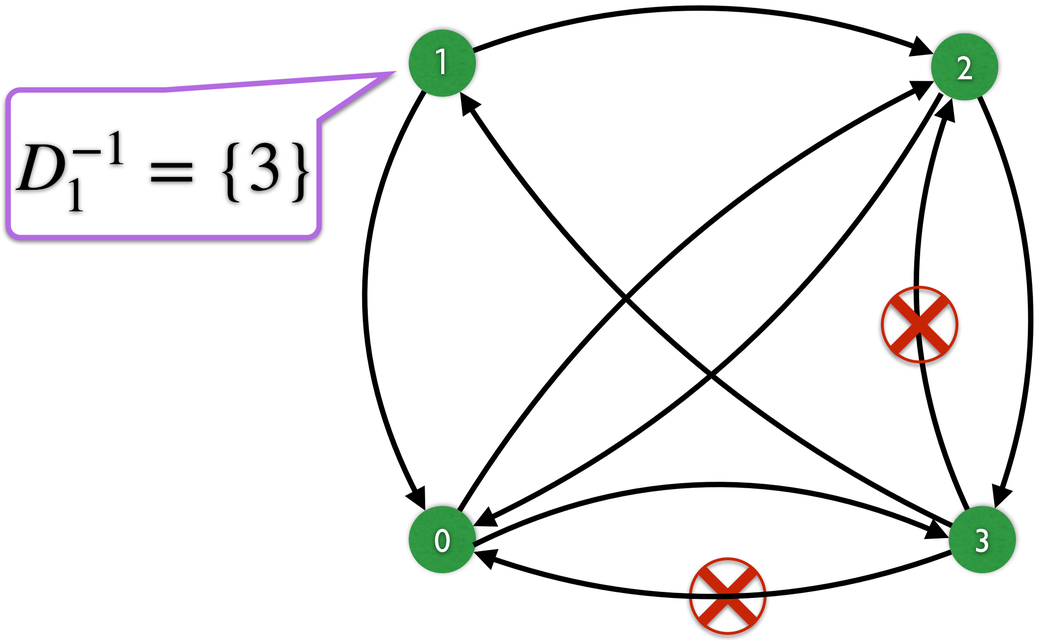
Circuit Filtering: Degree Reasoning

NP-hard, so we want to relax the filtering

- Degree-based filtering (weaker than AllDifferent and insufficient)
- Partial-path-based filtering

Degree-based filtering

- In a Hamiltonian cycle, the in-degree and out-degree of every node are 1.
- Notation: $D_j^{-1} = \{i \mid j \in D(\text{succ}[i])\}$
  (set of indices of the variables with *j* as a possible successor)
- If D(succ[i]) = { j }, then remove j from D(succ[k]) for all k ≠ i (forward checking)
- If D_j^{-1} = { i }, then reduce D(succ[i]) to { j }



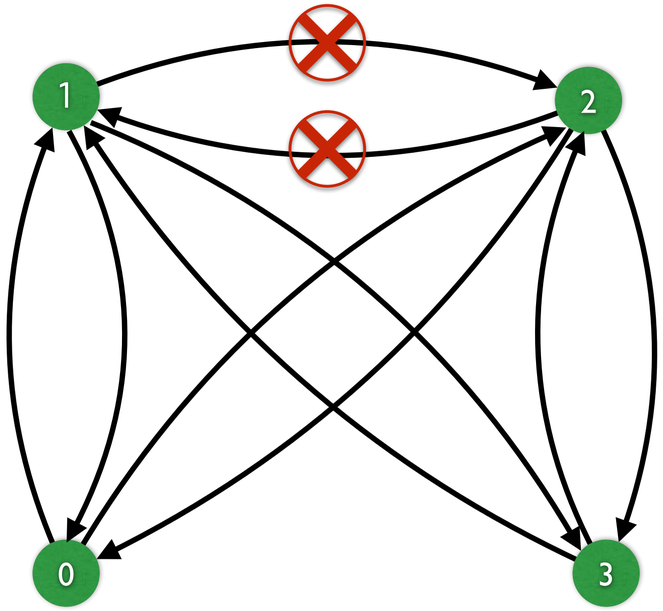
Hooker, J. N. (2012). *Integrated methods for optimization*

Degree-based filtering: Implementation

- If D(succ[i]) = { j }, then remove j from D(succ[k]) for all k ≠ i
  - Same as forward-checking filtering for AllDifferent.
    Make it efficient with a sparse set to split fixed and unfixed variables.
- If D_j^{-1} = { i }, then reduce D(succ[i]) to { *j* }
  - Requires counting for each value j, the number of D(succ[i]) with element j
  - Can be done incrementally during the search:
    - Split values into: fixed/unfixed ones (sparse set)
    - Split variables into fixed/unfixed ones (sparse set)
    - Consider values j and variables i not yet in the fixed partition for the counting

Degree-based filtering is weaker than AllDifferent-DC

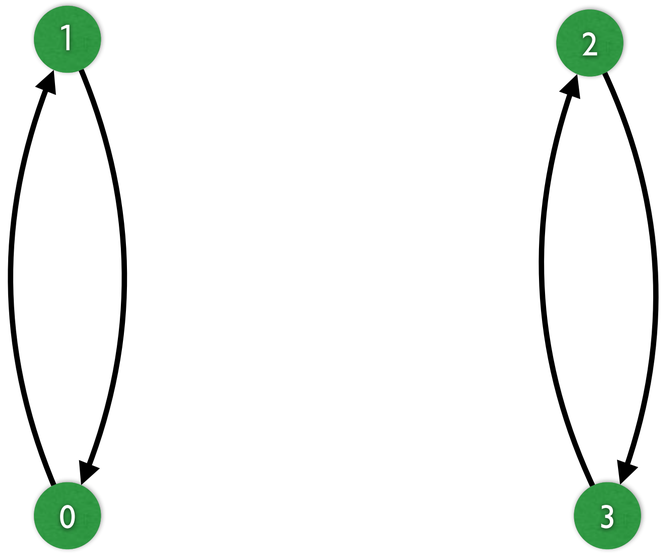
- On the example below, degree-based filtering is not able to detect the possible filtering since all in & out degrees are at least 2
- D(succ[0])={1,2}, D(succ[1])={0,~~2~~,3}, D(succ[2])={0,~~1~~,3}, D(succ[3])={1,2}



- But degree-based filtering is fast to execute

Again, AllDifferent-based filtering is not enough

- Since it does not prevent the creation of sub-cycles
- The degree-based filtering is also OK with this:



Constraint Programming

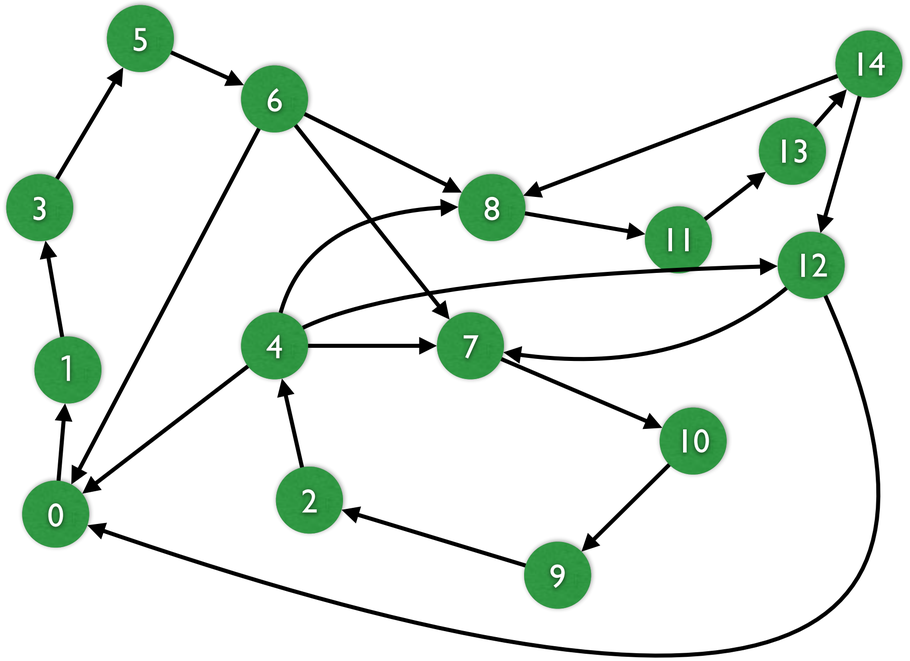
Circuit Filtering: Partial-Path Reasoning

NP-hard, so we want to relax the filtering

- Degree-based filtering (weaker than AllDifferent and insufficient)
- Partial-path-based filtering, together with AllDifferent-DC filtering

Partial-path-based filtering

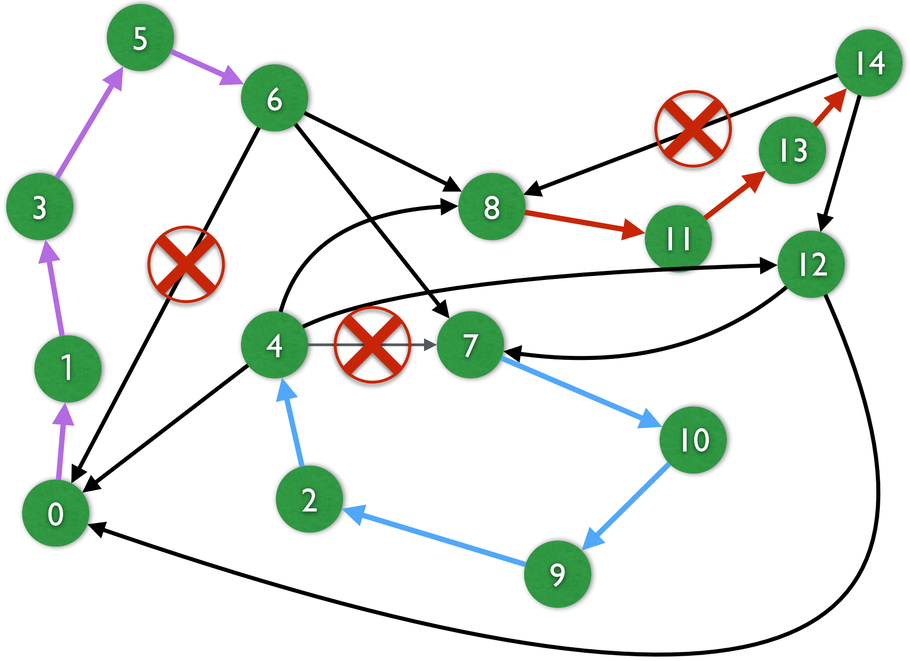
- Filtering idea: Detect partial paths and prevent them from closing.
- A *partial path* is a maximal consecutive sequence of nodes (successor variables) with a unique successor (current singleton domains).
- For example, what are the partial paths in the following graph?



Partial-path-based filtering

- If a partial path has fewer than (n–1) edges, where n = #nodes, then it must not be closed, as otherwise we would have a sub-cycle.
- The question is now: How to do this efficiently, in O(n) time?

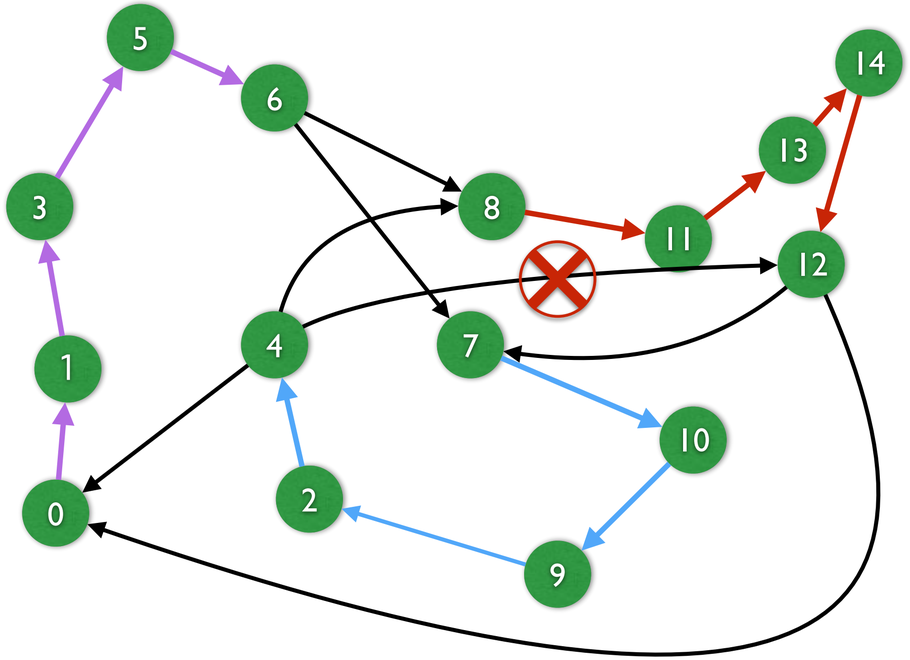
For example, the crossed edges below were *already* filtered when the nodes 4, 6, and 14 became the endpoints of their partial paths:



Partial-path-based filtering

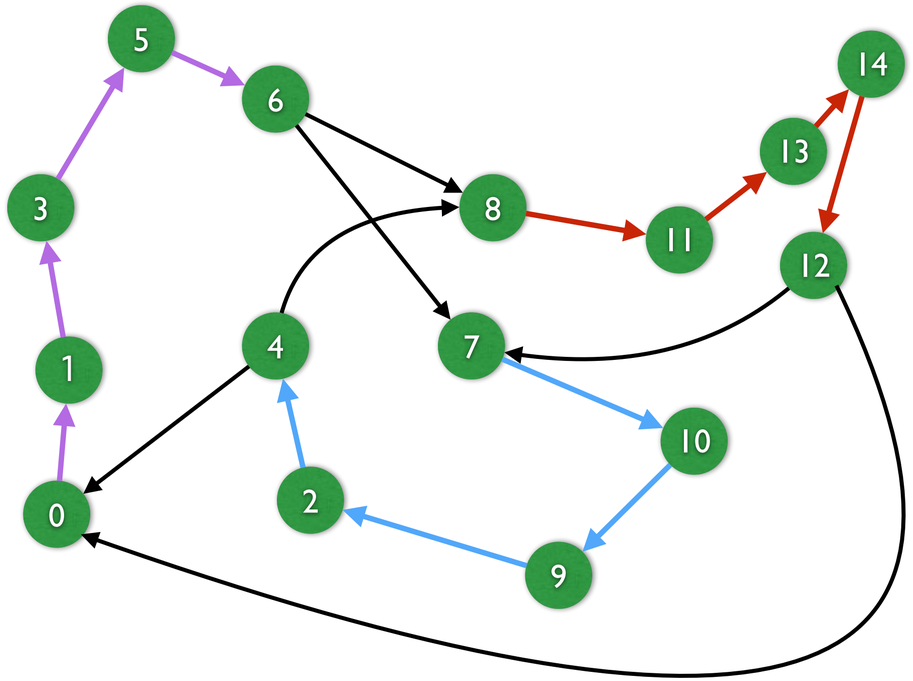
- If a partial path has fewer than (n–1) edges, where n = #nodes, then it must not be closed, as otherwise we would have a sub-circuit.
- The question is now: How to do this efficiently, in O(n) time?

Continuing our example, the successor of 14 then became 12 and the red partial path became longer, and AllDifferent detects succ[4] ≠ 12:



Data structures

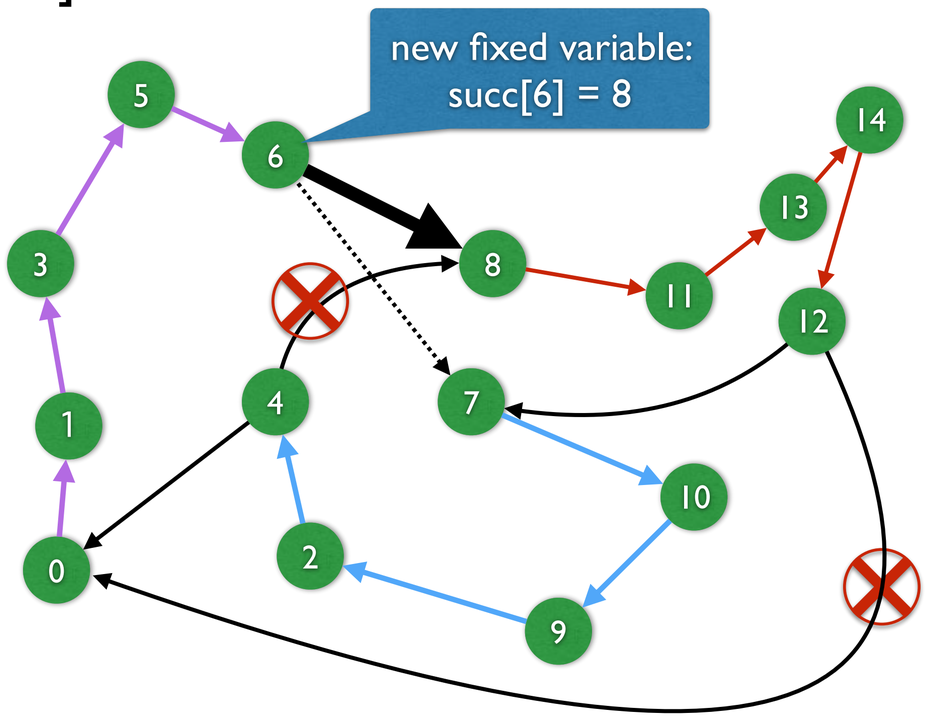
- We store three pieces of information for each node i:
  - dest[i] = destination of the partial path starting at i (dest[i]=i if succ[i] is not fixed)
  - orig[i] = origin of the partial path going through i (orig[i]=i if no succ[v] is fixed to i)
  - lengthToDest[i] = number of edges from i to dest[i]
- Examples:
  - dest[6]=6, orig[6]=0, lengthToDest[orig[6]]=4
  - dest[8]=12, orig[12]=8, lengthToDest[8]=4
  - dest[0]=6, orig[3]=0, lengthToDest[3]=2



Circuit: Updating the values in O(1) time

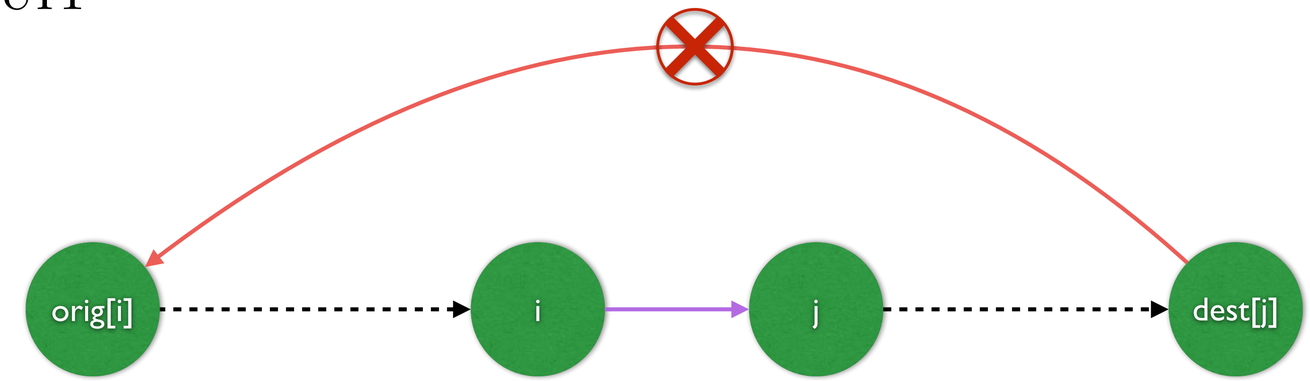
- Assuming the branching decision succ[6] = 8 (on backtrack: succ[6] = 7), the 3 updates that join the purple and red partial paths are as follows:
  - dest[orig[6]] := dest[8], hence: dest[0] = 12 (but, for example, dest[3] remains 6)
  - orig[dest[8]] := orig[6], hence: orig[12] = 0
  - lengthToDest[orig[6]] :+= lengthToDest[8] + 1, hence: lengthToDest[0] = 9
  - since 9 < 15 – 1, we infer: succ[12] ≠ 0
- Hence succ[12] = 7, joining the purplered & blue partial paths;

  AllDifferent: succ[4] ≠ 8
- Hence succ[4] = 0, which completes the cycle



Filtering algorithm (not idempotent!)

```
1:  procedure PropagateCircuit
2:      dest[i] ← i, ∀i
3:      orig[i] ← i, ∀i
4:      lengthToDest[i] ← 0, ∀i
5:      for i = 0 to n − 1 do
6:          if |D(x_i)| = 1 then
7:              j ← min(D(x_i))
8:              dest[orig[i]] ← dest[j]
9:              orig[dest[j]] ← orig[i]
10:             lengthToDest[orig[i]] ← lengthToDest[orig[i]] + lengthToDest[j] + 1
11:             if lengthToDest[orig[i]] < n − 1 then
12:                 x[dest[j]].remove(orig[i])
13:             end if
14:         end if
15:     end for
16: end procedure
```



new partial path = concatenation

Can we make this algorithm incremental?

```
1:  procedure PropagateCircuit
2:      dest[i] ← i, ∀i
3:      orig[i] ← i, ∀i
4:      lengthToDest[i] ← 0, ∀i
5:      for i = 0 to n − 1 do
6:          if |D(x_i)| = 1 then
7:              j ← min(D(x_i))
8:              dest[orig[i]] ← dest[j]
9:              orig[dest[j]] ← orig[i]
10:             lengthToDest[orig[i]] ← lengthToDest[orig[i]] + lengthToDest[j] + 1
11:             if lengthToDest[orig[i]] < n − 1 then
12:                 x[dest[j]].remove(orig[i])
13:             end if
14:         end if
15:     end for
16: end procedure
```

Store each of these values in a StateInt, so that the partial paths are restored at backtrack

Trigger this code only when x_i is fixed

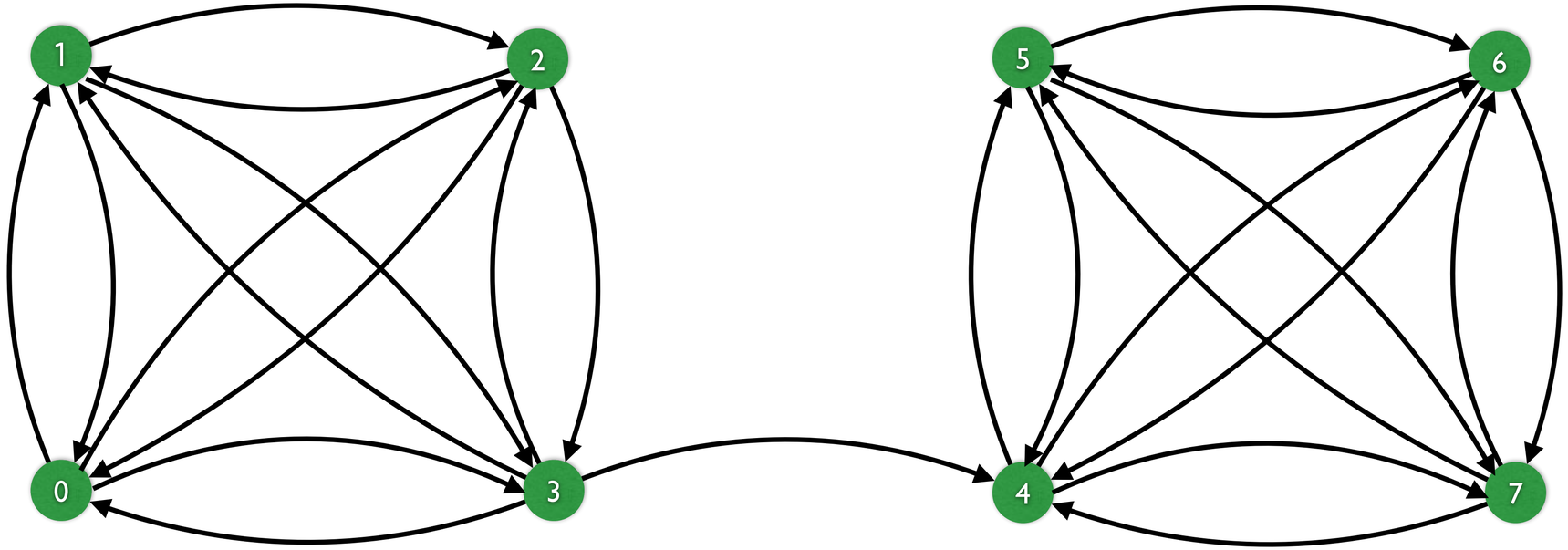
Overall complexity = O(number of new fixed variables) < O(n)

Constraint Programming

Circuit Constraint: SCC feasibility check

Feasibility Check

- Circuit is not feasible if there is more than one strongly connected component (SCC) in the graph induced by the current domains.



- Compute the SCCs with the Tarjan or Kosaraju algorithm:
  if there is more than one SCC, then fail.

Constraint Programming

Optimization of some objective function

Reminder

- A CSP is a constraint satisfaction problem:
  - A triplet ⟨X,D,C⟩ where …
- A COP is a constrained optimization problem:
  - A quadruplet ⟨X,D,C,f⟩
  - The objective function f is defined over a subset of the variables X.
  - Without loss of generality, we assume f is to be minimized.
- What we want for a COP:

  Find among the feasible solutions to ⟨X,D,C⟩,

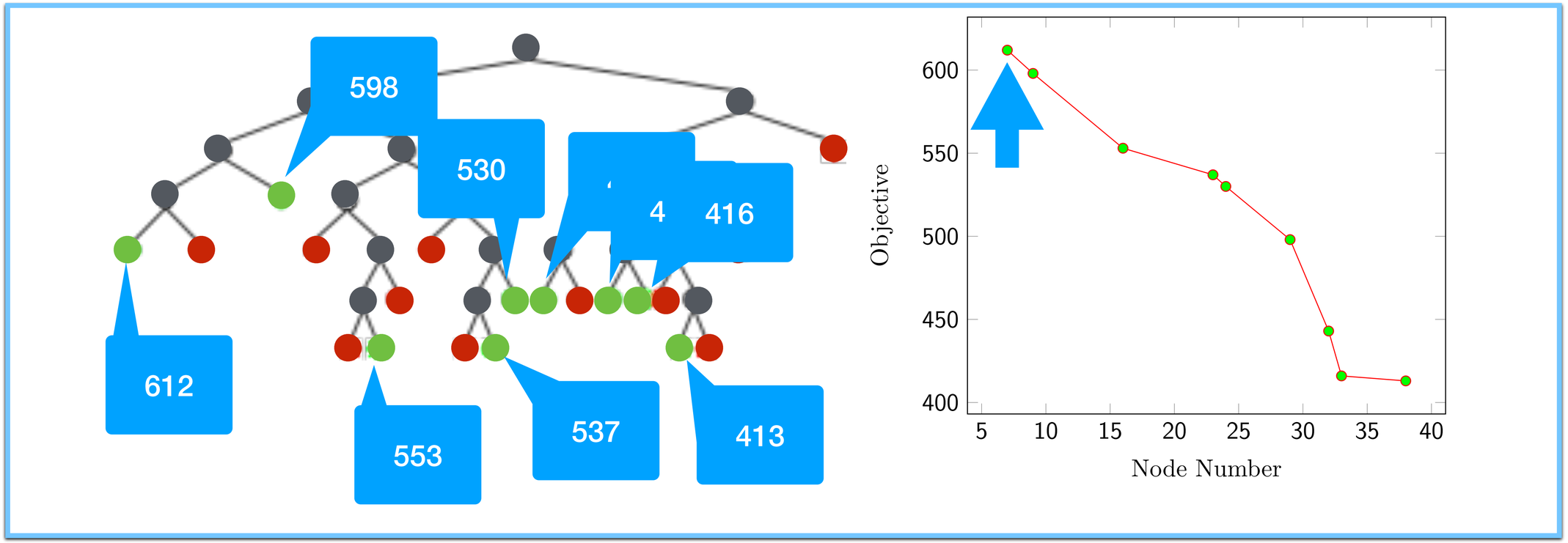
  i.e., in $\mathcal{S}(\langle X, D, C\rangle)$, a solution σ* that minimizes f.

How shall we do this?

- Idea, called branch-and-bound:
  - Step 1: Find a feasible solution σ_0 (i.e., the COP is feasible).
  - Step 2: Add the constraint $c_0 \stackrel{\text{def}}{=} f(\sigma) < f(\sigma_0)$.
  - Step 3: Continue solving.
  - Step 4: At iteration i:
    - If we find a feasible solution σ_i, then tighten by adding the "betterness" constraint $c_i \stackrel{\text{def}}{=} f(\sigma) < f(\sigma_i)$.
    - If we do not find a feasible solution, then the previous solution, σ_{i-1}, is a global optimum.
  - 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, 9 solutions were discovered before the last, optimal one.
Notice that, after the best solution was found,
we need to continue the search in order to prove its optimality.



Minimization in MiniCP

Total cost of the QAP denoted by an objective variable

```
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);
```

Creation of the objective

Creation of the DFS

Print obj. var. at each solution

Pass the objective to
DFS branch-and-bound

In practice

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

Objective ADT

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

```
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;
    }
}
```

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

Called when finding σ_i to update the bound

Hookup of the Objective into the Solver

```
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 σ_i

Minimization: Summary

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

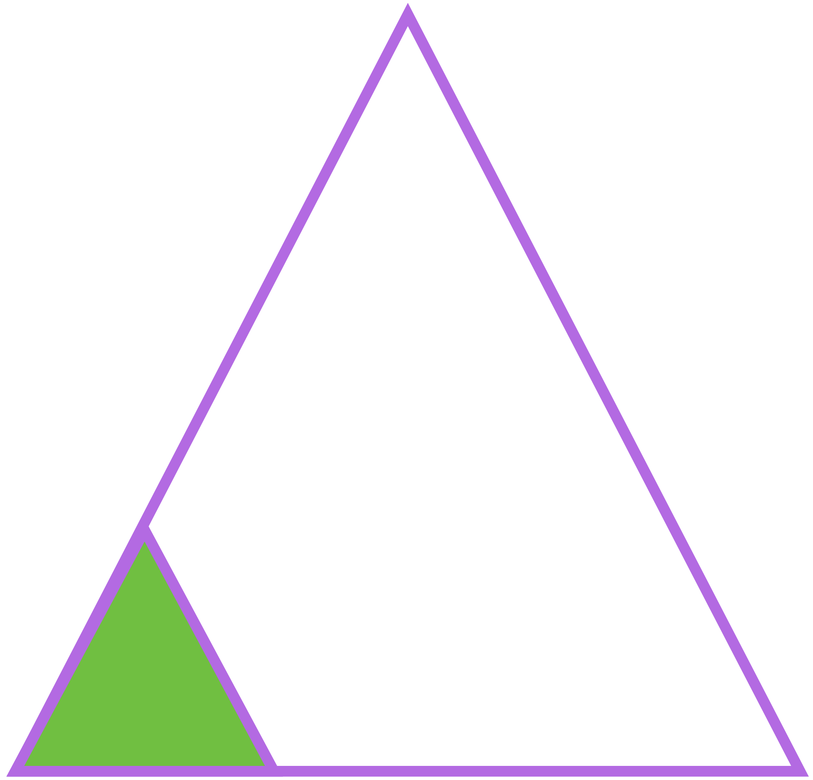
- onSolution: the objective bound is tightened (to the current bound minus 1);
- onFixPoint: the objective variable is restricted to be at most the bound.

Constraint Programming

Large-Neighborhood Search

The Weakness of CP

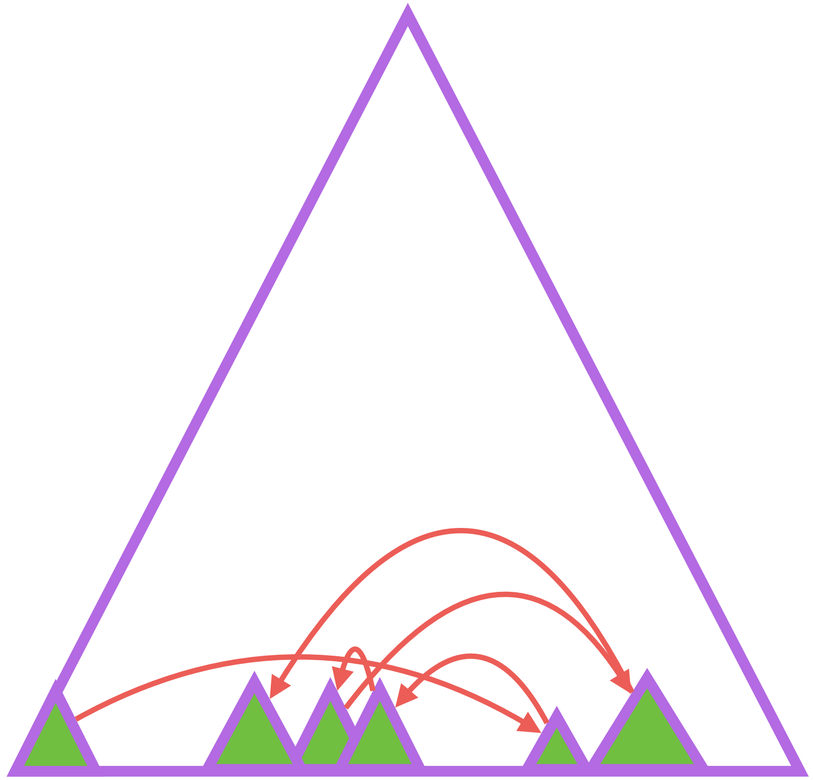
- Potentially huge search tree for optimization problems.
- Poor exploration of the search space.



  - 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.



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:
   relax = fix some variables to their values in S* and unfix the other variables.
3. Replace S* by the best solution found

It can be more general than that.
For example, in scheduling, good practice is:
relax = 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:

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

QAP with LNS

```
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]));
            }
    );
}
```

Store and update current best solution

Fix a random 75% of variables