The Hamiltonian-Circuit Constraint

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

Example: [2,4,1,5,3,0]

The successors must all be different, but this is not enough! From now on, we assume that an AllDifferent constraint is also filtering.

We must also guarantee that the array forms a proper circuit, without sub-circuits.
The Hamiltonian-Circuit Constraint

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

Example: [2,4,1,5,3,0]

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

From now on, we assume that an AllDifferent constraint is also filtering.

We must also guarantee that the array forms a proper circuit, without sub-circuits.
The Hamiltonian-Circuit Constraint

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

Example: [2,4,1,5,3,0]

The successors must all be different, but this is not enough!
From now on, we assume that an AllDifferent constraint is also filtering.

We must also guarantee that the array forms a proper circuit, without sub-circuits.
The Hamiltonian-Circuit Constraint

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

Example: [2,4,1,5,3,0]

The successors must all be different, but this is not enough! From now on, we assume that an AllDifferent constraint is also filtering.

We must also guarantee that the array forms a proper circuit, without sub-circuits.
The Hamiltonian-Circuit Constraint

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

Example: [2,4,1,5,3,0]

The successors must all be different, but this is not enough! From now on, we assume that an AllDifferent constraint is also filtering.

We must also guarantee that the array forms a proper circuit, without sub-circuits.
The Hamiltonian-Circuit Constraint

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

Example: [2,4,1,5,3,0]

The successors must all be different, but this is not enough!
From now on, we assume that an AllDifferent constraint is also filtering.

We must also guarantee that the array forms a proper circuit, without sub-circuits.
The Circuit constraint enforces a Hamiltonian circuit on an array of successor variables.

Example: [2,4,1,5,3,0]

The successors must all be different, but this is not enough! From now on, we assume that an AllDifferent constraint is also filtering.

We must also guarantee that the array forms a proper circuit, without sub-circuits.
The Hamiltonian-Circuit Constraint

Example of violation of a Circuit constraint: there are two sub-circuits (and AllDifferent holds)!

Example: $[2, 4, 1, 5, 0, 3]$
The Hamiltonian-Circuit Constraint

Example of violation of a Circuit constraint: there are two sub-circuits (and AllDifferent holds)!

Example: \([2,4,1,5,0,3]\)
The Hamiltonian-Circuit Constraint

Example of violation of a Circuit constraint: there are two sub-circuits (and AllDifferent holds)!

Example: \([2, 4, 1, 5, 0, 3]\)
The Hamiltonian-Circuit Constraint

Example of violation of a Circuit constraint: there are two sub-circuits (and AllDifferent holds)! Example: $[2, 4, 1, 5, 0, 3]$
Example of violation of a Circuit constraint: there are two sub-circuits (and AllDifferent holds)!

Example: \([2, 4, 1, 5, 0, 3]\)
The Hamiltonian-Circuit Constraint

Example of violation of a Circuit constraint: there are two sub-circuits (and AllDifferent holds)!

Example: [2, 4, 1, 5, 0, 3]
The Hamiltonian-Circuit Constraint

Example of violation of a Circuit constraint: there are two sub-circuits (and AllDifferent holds)!

Example: \([2,4,1,5,0,3]\)
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?
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:
Achieving domain consistency for a Circuit constraint is NP-hard.

Detecting a Hamiltonian cycle in an undirected graph $G(V,E)$ is NP-hard.

Reduction:

- Introduce a variable $x_i$ for every node $i$ in $V$, denoting the successor of node $i$:
  \[ D(x_i) = \{u : (i,u) \text{ in } E}\]
- Apply domain-consistent filtering for Circuit($x_1,\ldots,x_n$):
  - If no failure, then there is a Hamiltonian cycle.
  - Otherwise there is no Hamiltonian cycle.
We thus want to relax the 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?
We thus need to relax the filtering

- If a partial path has fewer than \((n-1)\) edges, where \(n = \#\text{nodes}\), 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?

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

- If a partial path has fewer than \((n-1)\) edges, where \(n = \#\text{nodes}\), 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 \textit{AllDifferent} detects \(\text{succ}[4] \neq 12\):
We store three pieces of information for each node $i$:
- $\text{dest}[i] = \text{destination of the partial path starting at } i$ (i.e., if $\text{succ}[i]$ is not fixed, $\text{dest}[i] = i$)
- $\text{orig}[i] = \text{origin of the partial path going through } i$ (i.e., if no $\text{succ}[v]$ is fixed to $i$, $\text{orig}[i] = i$)
- $\text{lengthToDest}[i] = \text{number of edges from } i \text{ to } \text{dest}[i]$

Examples:
- $\text{dest}[6] = 6$, $\text{orig}[6] = 0$, $\text{lengthToDest}[\text{orig}[6]] = 4$
- $\text{dest}[8] = 12$, $\text{orig}[12] = 8$, $\text{lengthToDest}[8] = 4$
- $\text{dest}[0] = 6$, $\text{orig}[3] = 0$, $\text{lengthToDest}[3] = 2$
Circuit: Updating the values in $O(1)$ time

- Assuming the branching decision $\text{succ}[6] = 8$ (on backtrack: $\text{succ}[6] = 7$), the updates that join the purple and red partial paths are as follows:
  - $\text{dest}[\text{orig}[6]] := \text{dest}[8]$, hence: $\text{dest}[0] = 12$
  - $\text{orig}[\text{dest}[8]] := \text{orig}[6]$, hence: $\text{orig}[12] = 0$
  - $\text{lengthToDest}[\text{orig}[6]] :+ = \text{lengthToDest}[8] + 1$, hence: $\text{lengthToDest}[0] = 9$
  - since $9 < 15 - 1$, we infer: $\text{succ}[12] \neq 0$

- Hence $\text{succ}[12] = 7$, joining the purple & blue partial paths;
  - **AllDifferent**: $\text{succ}[4] \neq 8$

- Hence $\text{succ}[4] = 0$, which completes the circuit
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
Can we make this algorithm incremental?

1: **procedure** PROPOGATECIRCUIT
2: \( dest[i] \leftarrow i, \forall i \)
3: \( orig[i] \leftarrow i, \forall i \)
4: \( lengthToDest[i] \leftarrow 0, \forall i \)
5: for \( i = 0 \) to \( n - 1 \) do
6: if \( |D(x_i)| = 1 \) then
7: \( j \leftarrow \min(D(x_i)) \)
8: \( dest[orig[i]] \leftarrow dest[j] \)
9: \( orig[dest[j]] \leftarrow orig[i] \)
10: \( lengthToDest[orig[i]] \leftarrow lengthToDest[orig[i]] + lengthToDest[j] + 1 \)
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 = \( O(\text{number of new fixed variables}) < O(n) \)
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.