Constraint Programming

AllDifferent Constraint
(version of 22 November 2022)
AllDifferent

– Scope:
  • An array $x$ of $n$ variables with the same domain, $D$, for each variable $x_i$.
  • The domain $D$ satisfies $|D| \geq n$.

– Requirement:

$$\forall i, j \in \{1..n\} : i \neq j \implies x_i \neq x_j$$

Can be modeled using $n(n-1)/2 = O(n^2)$ NotEqual constraints
public class NotEqual extends AbstractConstraint {
    private final IntVar x, y;

    public void post() {
        if (y.isFixed())
            x.remove(y.min());
        else if (x.isFixed())
            y.remove(x.min());
        else {
            x.propagateOnFix(this);
            y.propagateOnFix(this);
        }
    }

    public void propagate() {
        if (y.isFixed())
            x.remove(y.min());
        else
            y.remove(x.min());
        setActive(false);
    }
}

∀i, j ∈ {1..n} : i ≠ j ⇒ x_i ≠ x_j
What happens when a variable gets fixed?

- Filtering: When a variable gets fixed, its value is removed from the domains of the other variables.

If done by a propagator for AllDifferent, then this filtering is called forward checking (FWC), which achieves value consistency (VC).
What happens when a variable gets fixed?

‣ How much time does it take to filter the binary decomposition?

- Assume $x_2$ gets fixed to 2: we must remove 2 for $x_8$ and $x_9$.
- The decomposition considers all the $n-1$ NotEqual constraints on $x_2$:
  - Although all the constraints with a fixed variable are deactivated, they will still be considered. So $\Theta(n)$ time is caused when a new variable gets fixed.

‣ Aim: Spend time proportional to the number of unfixed variables.

‣ Why is this important?
Where do we spend the most time?

- Experiment on 15-queens: on average, 10.86 of 15 variables are fixed in the scope of AllDifferent.
- So it is important to **avoid considering constraints on fixed variables**.
How to do that?

1. Use a global constraint, and
2. Separate the fixed & unfixed variables in a sparse set (like for Sum):
   - When a variable $x_i$ gets fixed to a value $v$, then apply FWC: iterate over the unfixed variables in order to remove $v$ from their domains.
   - Use a StateInt $nFixed$ in order to implement the sparse set.

```java
int nF = nFixed.value();
for (int i = nF; i < x.length; i++) {
    int idx = fixed[i];
    IntVar y = x[idx];
    if (x[idx].isFixed()) {
        // filter the unfixed variables
        // swap the variables:
        fixed[i] = fixed[nF];
        fixed[nF] = idx;
        nF++;
    }
}

nFixed.setValue(nF);
```
Does it help?

- 15-queens problem: 37,086,270 nodes and 2,279,184 solutions:
  - One global constraint, filtered by FWC, using a sparse set: 43 seconds.
  - Binary decomposition: 63 seconds (+46%).

- What if more variables: same speedup?
Does it help?

☆ 30-queens problem: looking for first $10^6$ solutions:
  – One global constraint, filtered by FWC, using a sparse set: 36 seconds.
  – Binary decomposition: 69 seconds (+91%).

☆ 60-queens problem: looking for first $10^5$ solutions:
  – One global constraint, filtered by FWC, using a sparse set: 18 seconds.
  – Binary decomposition: 51 seconds (+180%).
How good is the decomposition? Weak!

\[ \forall i, j \in \{1..n\} : i \neq j \implies x_i \neq x_j \]

Assume this is a Sudoku row: are only the red values to be filtered? One can filter much more …

The decomposition and a global constraint with FWC cannot detect inconsistency for:
AllDifferent: Feasibility Check (Régin 1994)

- \( x_0 \in \{3,4\} \)
- \( x_1 \in \{1\} \)
- \( x_2 \in \{3,4\} \)
- \( x_3 \in \{0\} \)
- \( x_4 \in \{3,4,5\} \)
- \( x_5 \in \{5,6,7\} \)
- \( x_6 \in \{2,9,10\} \)
- \( x_7 \in \{5,6,7,8\} \)
- \( x_8 \in \{5,6,7\} \)

Step 1: Build the Variable-Value (Bipartite) Graph
There is a solution to \( \text{AllDifferent}([x_0, \ldots, x_8]) \) iff there exists a maximum matching \( M \) of size \( n=9 \) in the variable-value graph.

Definition: \textit{matching} = set of edges without common vertices.
How to find a maximum matching?

- Start with a possibly sub-optimal matching and try & augment it.
- Greedy: Iterate over the variables and fix to the \textit{first} still possible value.

![Graph diagram showing the greedy algorithm failing to match a vertex](image.png)
How to augment a matching?

1. Orient the edges for the greedy matching $M$:
   - $\leftarrow$ for edges in $M$
   - $\rightarrow$ for edges not in $M$

2. Search for a path (by DFS) that starts from an $M$-free variable vertex and ends at an $M$-free value vertex. Such a path is called an alternating path.

3. If an alternating path exists, then flip the status/orientation of its edges, else the matching is already maximum.

$M$ is a maximum matching iff no alternating path exists.
How to augment a matching?

1. Orient the edges for the greedy matching $M$:
   • $\leftarrow$ for edges in $M$
   • $\rightarrow$ for edges not in $M$

2. Search for a path (by DFS) that starts from an $M$-free variable vertex and ends at an $M$-free value vertex. Such a path is called an alternating path.

3. If an alternating path exists, then flip the status/orientation of its edges, else the matching is already maximum.

$M$ is a maximum matching iff no alternating path exists.
How to find a maximum matching?

To find a maximum matching:

- Start with any matching (e.g., the empty or greedy one).
- **Iteratively augment** the matching via alternating paths.
- Stop when no more alternating path exists.

Now our example matching is maximum and we know that our example constraint is feasible since that maximum matching has n edges.

What are the edges (domain values) to remove?
Filtering AllDifferent

Remove all the edges that do not belong to some maximum matching that is of size n (i.e., covers all the variables): domain consistency.

- Idea A:
  - Enumerate all the maximum matchings, and collect (by set union) all their edges.
  - Delete any edge that is not in the final collection.

Is this a practical approach? 😐
A important theorem can save us …

- Idea B: Apply a theorem by Berge (1970):

  An edge belongs to some but not all maximum matchings iff, for an arbitrary maximum matching $M$, it belongs to either an even-length alternating path that starts at an $M$-free vertex (case 1), or an even-length alternating cycle (case 2).
A important theorem can save us …

- Idea B: Apply a theorem by Berge (1970):

An edge belongs to some but not all maximum matchings iff, for an arbitrary maximum matching $M$, it belongs to either an even-length alternating path that starts at an $M$-free vertex (case 1), or an even-length alternating cycle (case 2).

Why is this theorem important?
A important theorem can save us …

Idea B: Apply a theorem by Berge (1970):

An edge belongs to some but not all maximum matchings iff, for an arbitrary maximum matching $M$, it belongs to either an even-length alternating path that starts at an $M$-free vertex (case 1), or an even-length alternating cycle (case 2).

Why is this theorem important?
Because we now show that our problem boils down to detecting cycles in a directed graph.

Claude Berge 1926–2002
A important theorem can save us …

› Idea B: Apply a theorem by Berge (1970):

An edge belongs to some but not all maximum matchings iff, for an arbitrary maximum matching $M$, it belongs to either an even-length alternating path that starts at an $M$-free vertex (case 1), or an even-length alternating cycle (case 2).

Why is this theorem important?
Because we now show that our problem boils down to detecting cycles in a directed graph.
What algorithm can be used to detect all cycles?
Berge Theorem: Case 1

An edge belongs to some maximum matching if, for an arbitrary maximum matching \( M \), it belongs to an even-length alternating path that starts at an \( M \)-free vertex.
Berge Theorem: Case 1

An edge belongs to some maximum matching if, for an arbitrary maximum matching $M$, it belongs to an even-length alternating path that starts at an M-free vertex.
Berge Theorem: Case 1

An edge belongs to some maximum matching if, for an arbitrary maximum matching $M$, it belongs to an even-length alternating path that starts at an $M$-free vertex.

You can convince yourself that flipping the status of the edges on this path creates another maximum matching with the edge $(6,10)$ instead of the edge $(6,2)$.

Even-length alternating paths of more than 2 edges can occur in general.
Berge Theorem: Case 2

An edge belongs to some maximum matching if, for an arbitrary maximum matching $M$, it belongs to an even-length alternating cycle.
Berge Theorem: Case 2

An edge belongs to some maximum matching if, for an arbitrary maximum matching $M$, it belongs to an even-length alternating cycle.

Edge (2,3)?
Berge Theorem: Case 2

An edge belongs to some maximum matching if, for an arbitrary maximum matching \( M \), it belongs to an even-length alternating cycle.

You can convince yourself that flipping the status of the edges on this cycle creates another maximum matching with the edges (2,3) and (0,4) instead of (2,4) and (0,3).
Filtering AllDifferent: Domain Consistency

Let $M$ be a maximum matching:

edge $e$ belongs to some maximum matching iff
- $e$ belongs to a particular maximum matching $M$
- $e$ belongs to an even-length alternating path that starts at an $M$-free vertex
- $e$ belongs to an even-length alternating cycle

or

We can treat the two cases (path & cycle) of Berge’s theorem as one, namely by transforming the graph!
Berge Theorem: Merge the Two Cases

Transformation:
Add a dummy node, with an incoming arc from every $M$-free value node and an outgoing arc to every value node in $M$.

Now:
- Every directed cycle that does not contain an arc from a vertex in $M$ to an $M$-free vertex corresponds to an alternating cycle w.r.t. $M$.
- Every directed cycle that contains an arc from a vertex in $M$ to an $M$-free vertex corresponds to an even-length alternating path w.r.t. $M$. 
Berge Theorem: Merge the Two Cases

Transformation:
Add a dummy node, with an incoming arc from every $M$-free value node and an outgoing arc to every value node in $M$.

Now:
- Every directed cycle that does not contain an arc from a vertex in $M$ to an $M$-free vertex corresponds to an alternating cycle w.r.t. $M$.
- Every directed cycle that contains an arc from a vertex in $M$ to an $M$-free vertex corresponds to an even-length alternating path w.r.t. $M$.

Corollary (Régin’s idea):
- An edge $e$ belongs to some maximum matching iff either $e$ belongs to the initial matching $M$ or $e$ belongs to some cycle in the transformed graph for $M$. 
Transformation:
Add a dummy node, with an incoming arc from every $M$-free value node and an outgoing arc to every value node in $M$.

Now:

- Every directed cycle that does not contain an arc from a vertex in $M$ to an $M$-free vertex corresponds to an alternating cycle w.r.t. $M$.
- Every directed cycle that contains an arc from a vertex in $M$ to an $M$-free vertex corresponds to an even-length alternating path w.r.t. $M$.

Corollary (Régin’s idea):

- An edge $e$ belongs to some maximum matching iff either $e$ belongs to the initial matching $M$ or $e$ belongs to some cycle in the transformed graph for $M$.

All cycles can be computed in time linear in the size of the graph, by finding all strongly connected components with the Kosaraju or Tarjan algorithm.
Domain Consistency (Régin 1994)

All vertices belonging to some directed cycle can be identified by finding all strongly connected components (SCC).

Filtering algorithm for AllDifferent(X):

- Compute a matching \( M \) that covers X in the variable-value graph.
- Remove all the edges \((x,a)\) (i.e., delete all \( a \) from all \( D(x) \)) where \((x,a)\) is not in \( M \) or \( a \) & \( x \) belong to two different SCCs of the transformed graph for \( M \).

The black edges connect different SCCs: those not corresponding to the matching \( M \) must also be removed.
AllDifferent Filtering: Summary

1. **step 1:**
   - Maximum matching

2. **step 2:**
   - Directed graph

3. **step 3:**
   - Strongly connected components

4. **step 4:**
   - Filtering
Incrementality of the Filtering

When called:
- Remove the edges that represent already filtered values.
- Recompute a maximum matching if necessary:
  if edges of the maximum matching were removed,
  then augment this matching (see slide 14).
- Re-filter.

Remove edges of $M$ that have disappeared and start
alternating path from here in
order to compute $M'$

$M'$ is still valid when backtracking,
so there is no need to augment it
(backtrack = restoring values to domains)
public class AllDifferentDC extends AbstractConstraint {
    private IntVar[] x;
    private final MaximumMatching maximumMatching;
    private final int nVar;
    private int nVal;
    // residual graph
    private ArrayList<Integer>[] in;
    private ArrayList<Integer>[] out;
    private int nNodes;
    private Graph g = new Graph() {
        @Override public int n() { return nNodes; }
        @Override public Iterable<Integer> in(int idx) { return in[idx]; }
        @Override public Iterable<Integer> out(int idx) { return out[idx]; }
    };
    private int[] match;
    private boolean[] matched;
    private int minVal;
    private int maxVal;
}
AllDifferent Implementation (assignment)

```java
public class AllDifferentDC extends AbstractConstraint {
    private IntVar[] x;
    private final MaximumMatching maximumMatching;
    private final int nVar;
    private int nVal;
    // residual graph
    private ArrayList<Integer>[] in;
    private ArrayList<Integer>[] out;
    private int nNodes;
    private Graph g = new Graph() {
        @Override public int n() { return nNodes; }
        @Override public Iterable<Integer> in(int idx) { return in[idx]; }
        @Override public Iterable<Integer> out(int idx) { return out[idx]; }
    };
    private void updateGraph() {// TODO}
    public void propagate() {
        // TODO Implement the filtering
        // hint: use maximumMatching.compute(match) to update the maximum matching
        // use updateRange() to update the range of values
        // use updateGraph() to update the residual graph
        // use GraphUtil.stronglyConnectedComponents to compute SCC's
    }
}
```

in[i] = in adjacent nodes
Complexity

- Runtime: $O(m\sqrt{n})$ time, where:
  - $m$ is the number of edges,
  - $n$ is the number of variables
    (use the Hopcroft-Karp algorithm for finding a maximum matching).
- The strongly connected components are computed in $O(m)$ time.