MiniCP

Constraint Programming

The AllDifferent Constraint

AllDifferent: Binary Decomposition

AllDifferent

– Scope:
  - An array x of n variables with the same domain, D, for each variable x_i.
  - D satisfies |D| ≥ 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

X != Y

```
public class NotEqual extends AbstractConstraint {
    private final IntVar x, y;


    @Override
    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);
        }
    }

    @Override
    public void propagate() {
        if (y.isFixed())
            x.remove(y.min());
        else y.remove(x.min());
    }
}
```

AllDifferent by Binary Decomposition

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

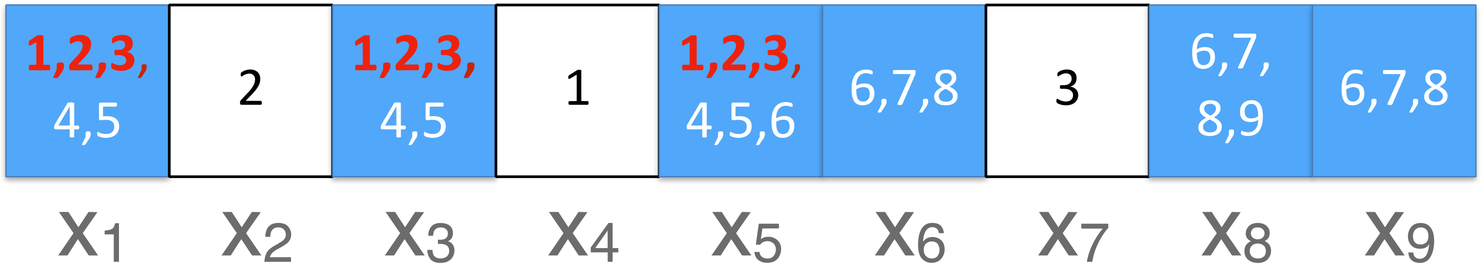
NotEqual

```
public class AllDifferentBinary extends AbstractConstraint {

    private IntVar[] x;

    public AllDifferentBinary(IntVar... x) {
        super(x[0].getSolver());
        this.x = x;
    }

    @Override
    public void post() {
        Solver cp = x[0].getSolver();
        for (int i = 0; i < x.length; i++) {
            for (int j = i + 1; j < x.length; j++) {
                cp.post(new NotEqual(x[i], x[j]));
            }
        }
    }
}
```

AllDifferent: Forward Checking

What happens when a variable gets fixed?

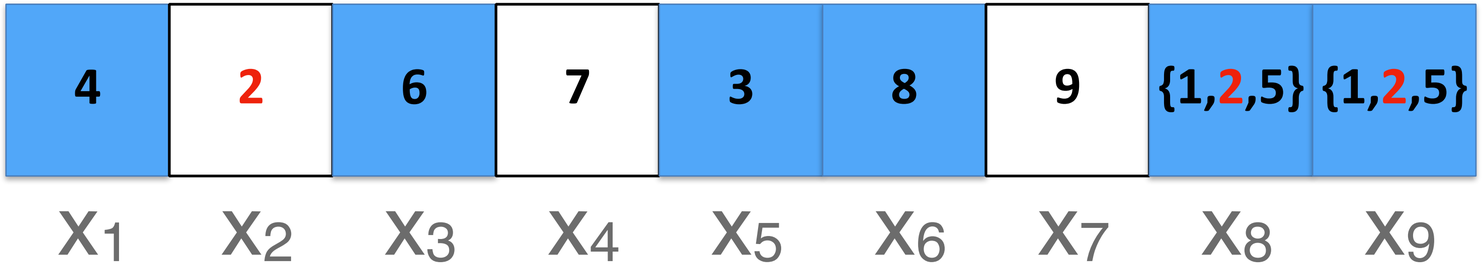
Filtering: When a variable gets fixed,
its value is removed from the domains of the other variables.



This filtering is called forward checking (FWC).

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: so Θ(n) time is caused per newly fixed variable.
- **Aim:** Spend time proportional to the number of *unfixed* variables, and achieve the *same* propagation, but with only *one* filtering algorithm, working on *all* n variables.

Forward Checking

- Separate the indices of fixed & unfixed variables in a sparse set (as for Sum)
- For each index i (of a variable X[i]) in the set NotFixed:
  - If X[i] was fixed to some value v, then:
    - Remove v from the domain of all other variables in NotFixed
    - Move index i to the set Fixed

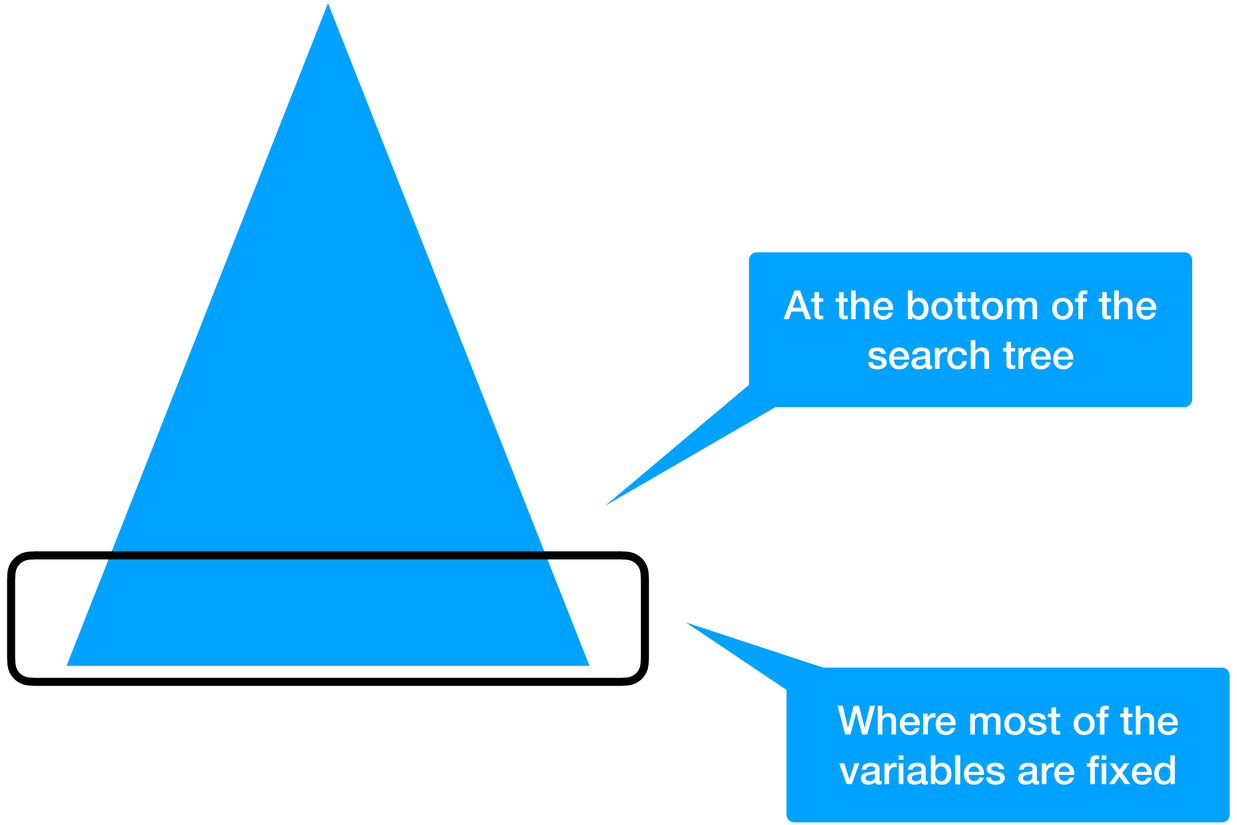
- $\Theta(\#\text{NotFixed}^2)$ time instead of $\Theta(n^2)$ time

How to do that?

1. Use a global constraint, and
2. Separate the indices of fixed & unfixed variables in a sparse set (as for Sum):
   - When a variable was fixed to a value v:
     iterate over the unfixed variables in order to remove v from their domains.
   - Use a StateInt *nFixed*, denoting the number of fixed variables,
     in order to implement a stateful sparse set.

```
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);
```

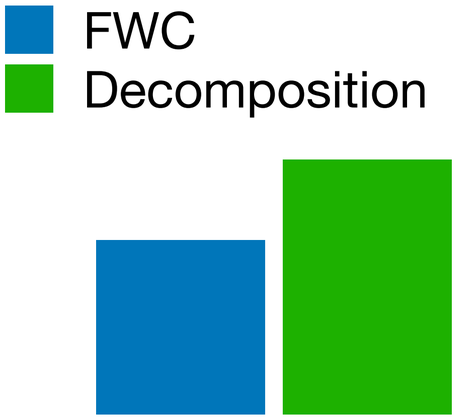
Where do we spend the most time?



- 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

Does it help?

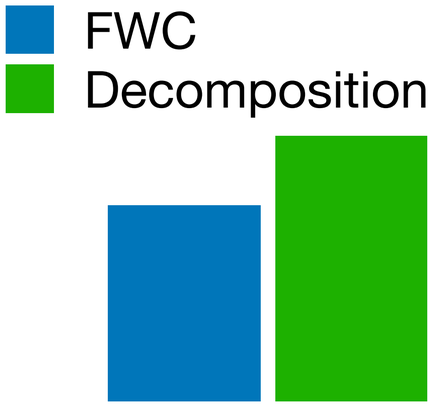
- 15-queens problem: 37,086,270 nodes and 2,279,184 solutions:
  - A global constraint, filtered by FWC, using a sparse set: 43 seconds.
  - Binary decomposition: 63 seconds (+46%).



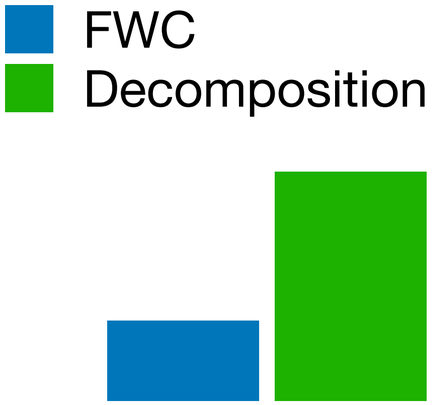
- What if more variables: same speedup?

Does it help?

- 30-queens problem: 36 seconds vs 69 seconds (+91%).



- 60-queens problem (looking for first 10^5 solutions): 18s vs 51s (+180%)

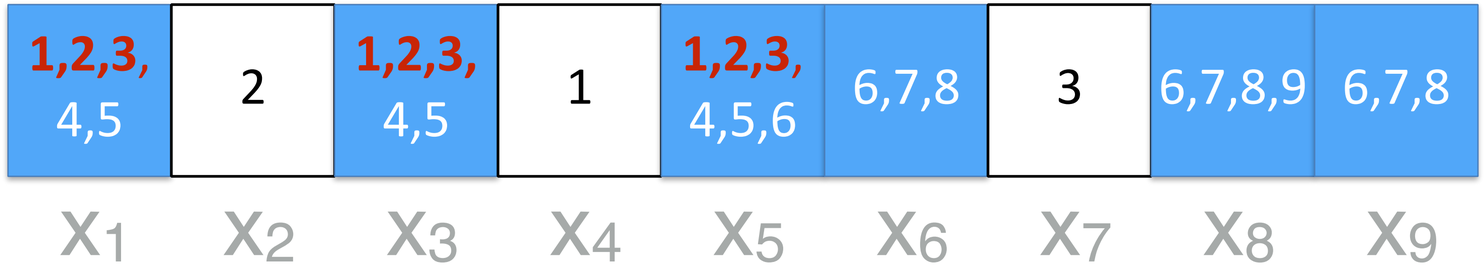


AllDifferent: Feasibility Check

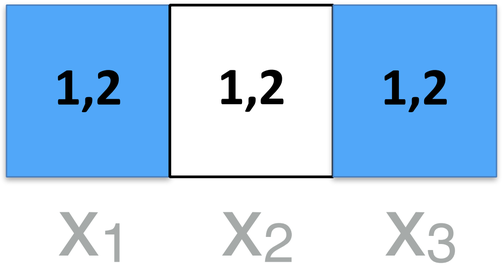
How good is the filtering of the decomposition? Weak!

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

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



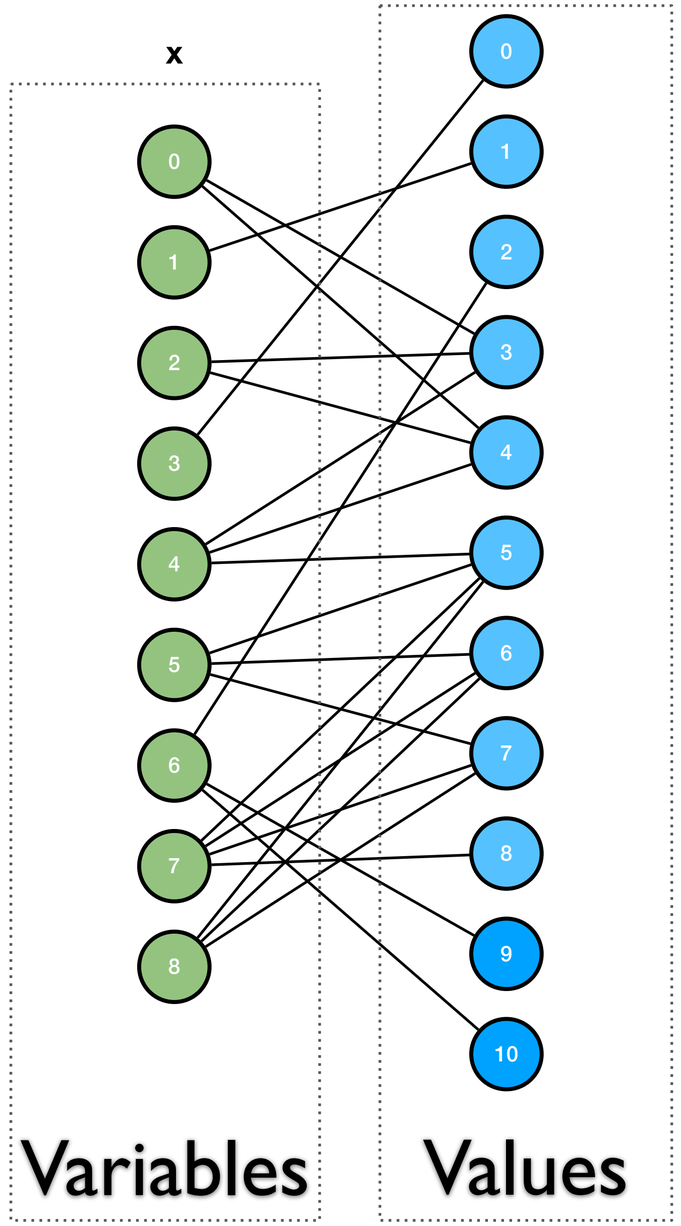
- The decomposition and a global constraint with FWC cannot detect inconsistency for:



AllDifferent: Feasibility Check (Régin 1994)

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

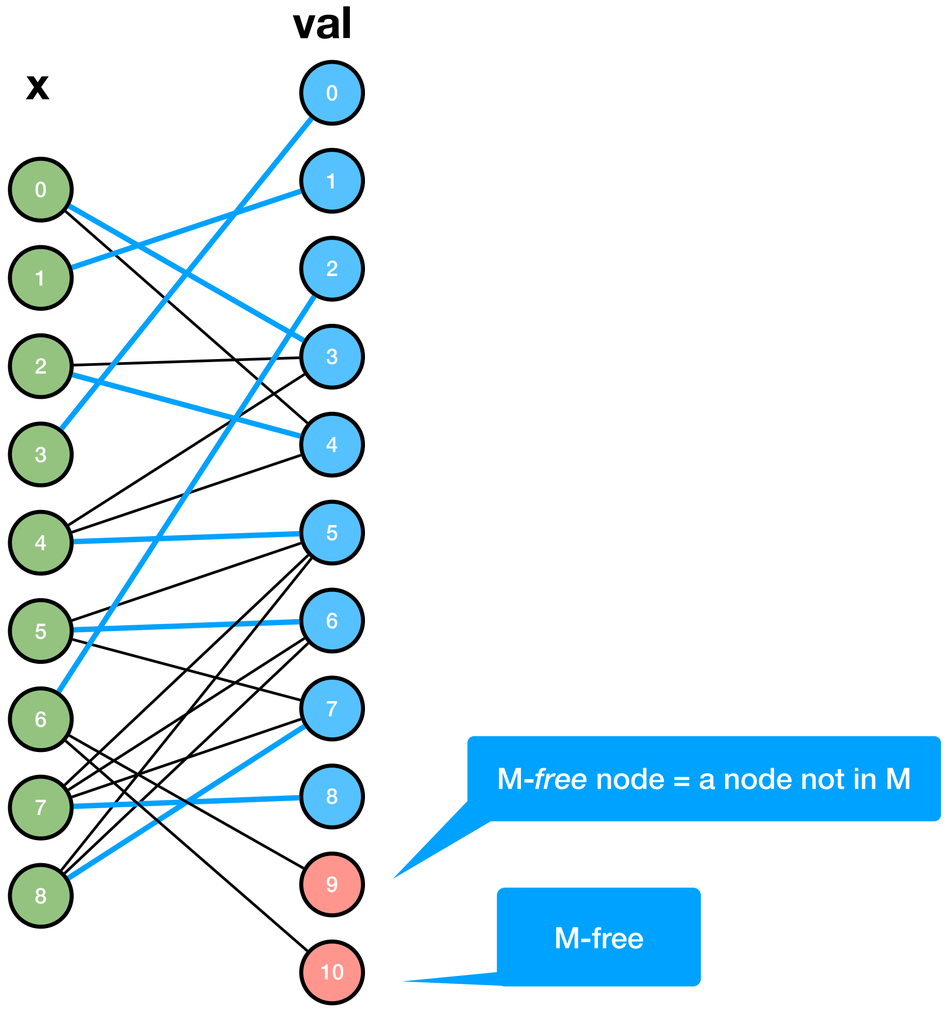
Step 1: Build the Variable-Value (Bipartite) Graph



AllDifferent: Feasibility Check

Our example constraint is thus feasible!

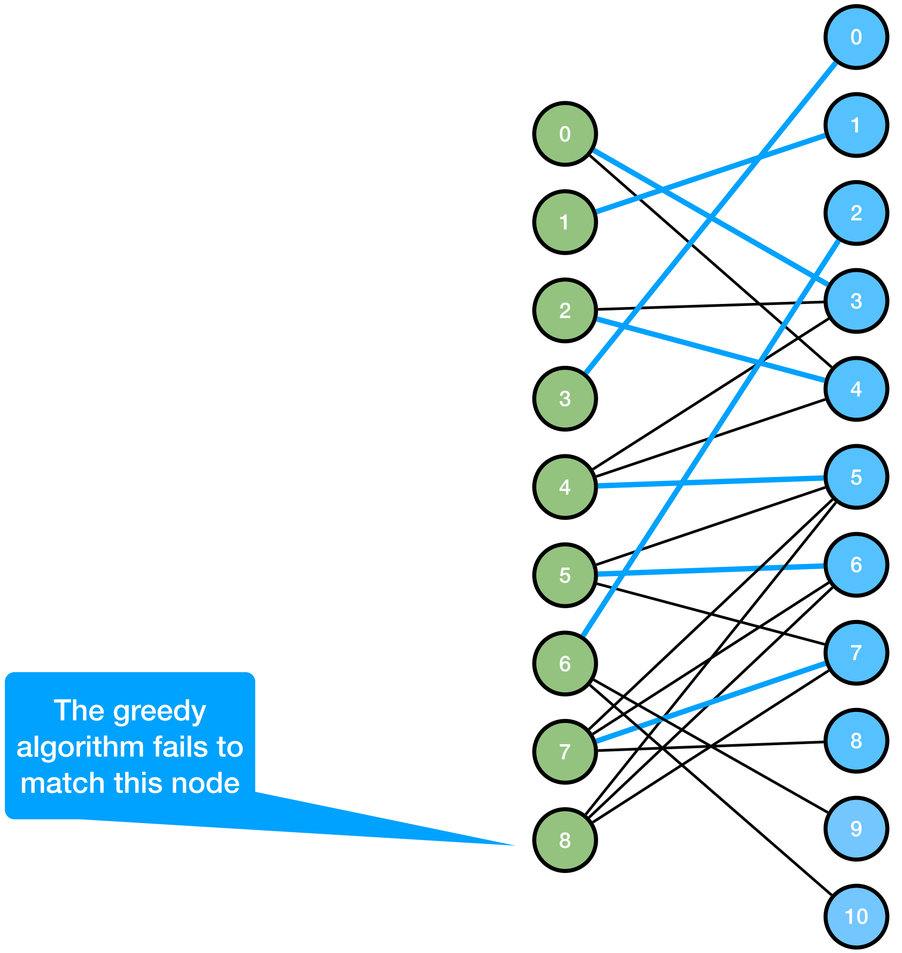
There is a solution to AllDifferent([x_0,…,x_8]) iff there exists a maximum matching **M** of size n=9 in the variable-value graph.



Definition: *matching* = set of edges without common nodes.

How to find a maximum matching?

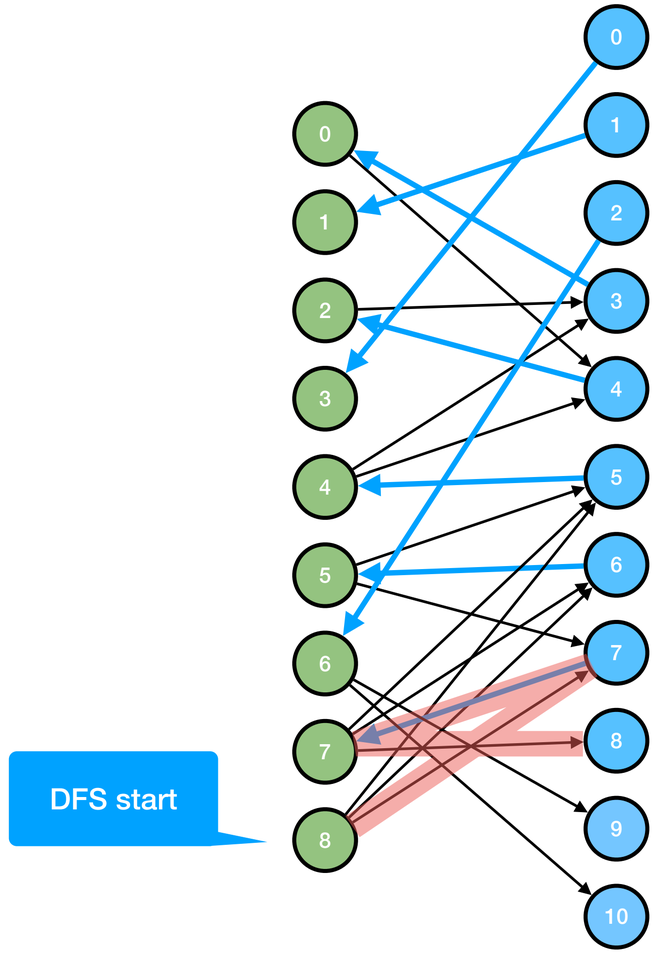
- Start with a possibly sub-optimal matching and try & augment it.
- Greedy: Iterate over the variables and fix to the *first* still possible value.



How to augment a matching?

1. Direct the edges for the greedy matching M:
   - ← for edges in **M**
   - → for edges not in **M**
2. Search for a path (by DFS) that starts from an **M**-free variable node and ends at an **M**-free value node. Such a path is called an ***alternating path***.
3. If an alternating path exists, then flip the direction of its arcs, else the matching is already maximum.

M is a maximum matching iff no alternating path exists.

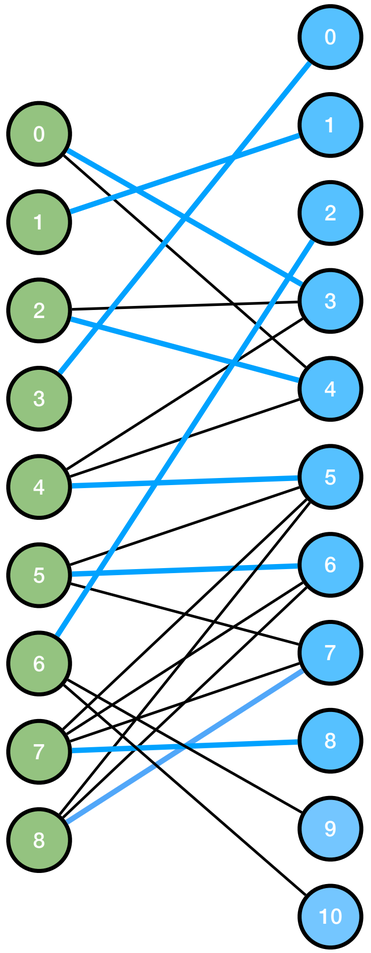


How to find a maximum matching?

To find a maximum matching:

- Start with any matching (e.g., the empty matching).
- 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=9 edges.
- What are the edges (domain values) to remove?



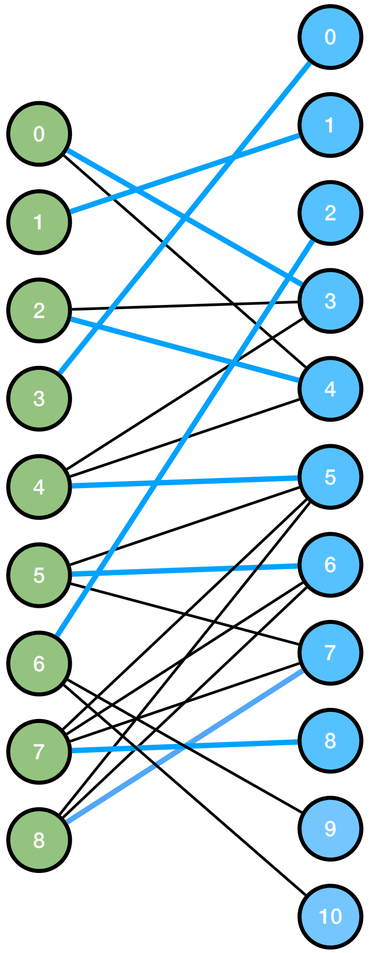
AllDifferent:
Domain Consistency =
Filtering all Inconsistent Values

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? 🤔



An important theorem can save us …

Claude Berge 1926–2002

- 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 node (case 1),
> or an even-length alternating cycle (case 2).

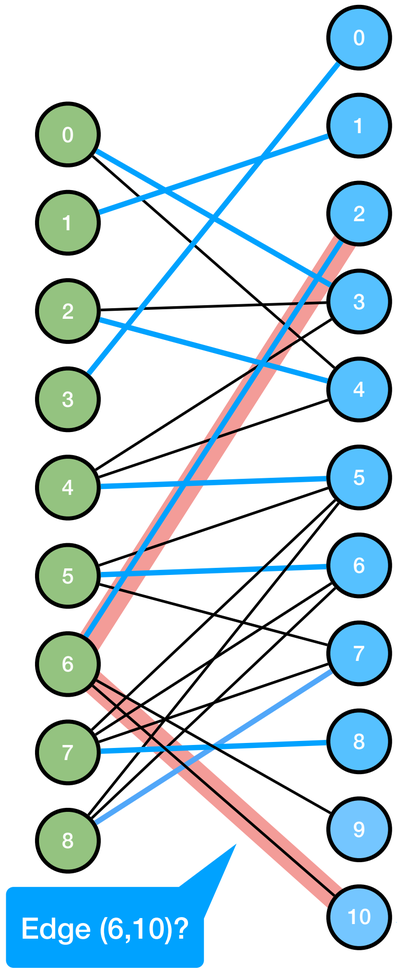
Why is this theorem important?

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 node.

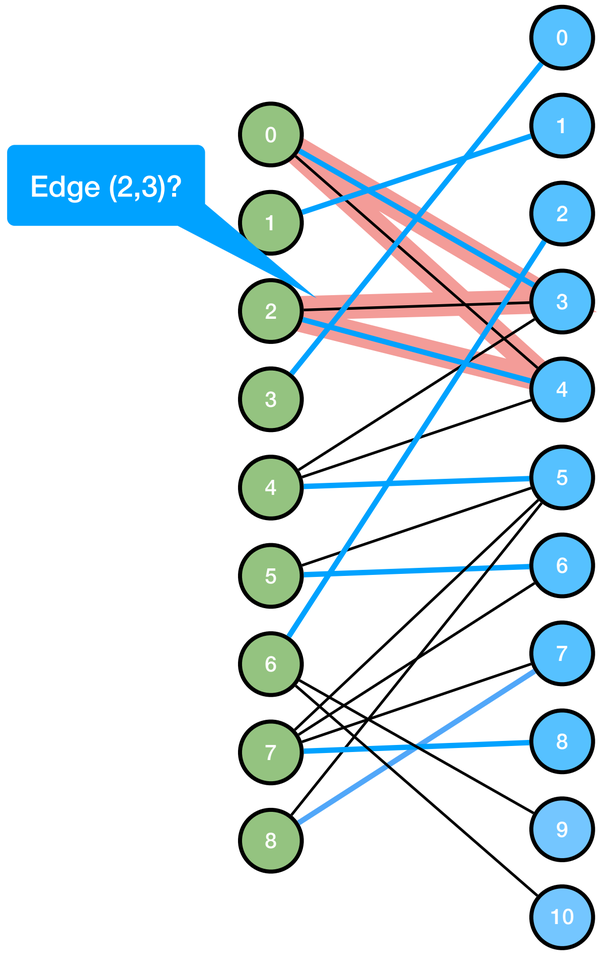


You can convince yourself that flipping the status of the edges on this path creates *another* maximum matching, with the edge (6,10) instead of the edge (6,2).

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.



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

or

- e belongs to an even-length alternating path that starts at an M-free node
- e belongs to an even-length alternating cycle

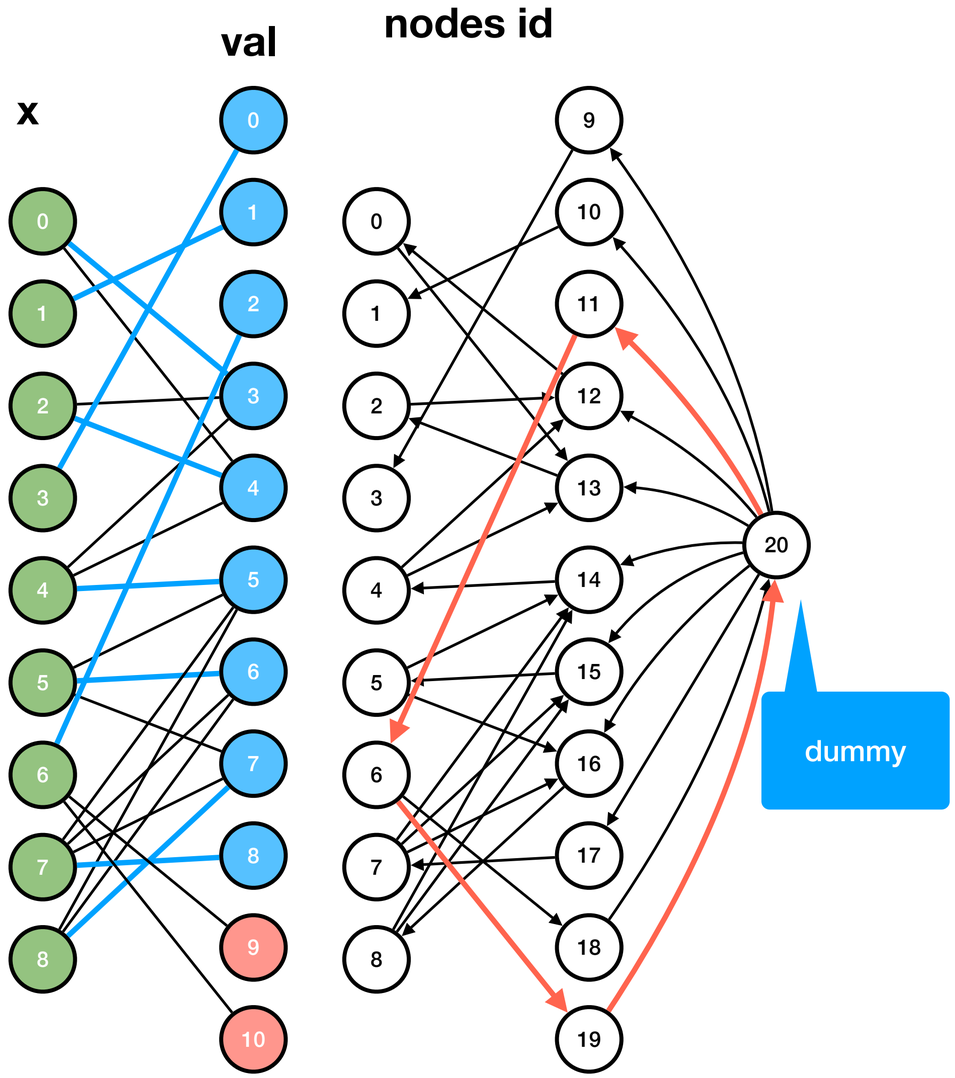
We can treat the two cases (path & cycle) of Berge's theorem as one, namely by transforming the graph!

Berge Theorem: Merge the Two Cases

Transformation:
Direct the graph (like above) and 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 contains an arc from a node in M to an M-free node corresponds to an *even-length alternating path* w.r.t. M. (Using dummy node = case1)

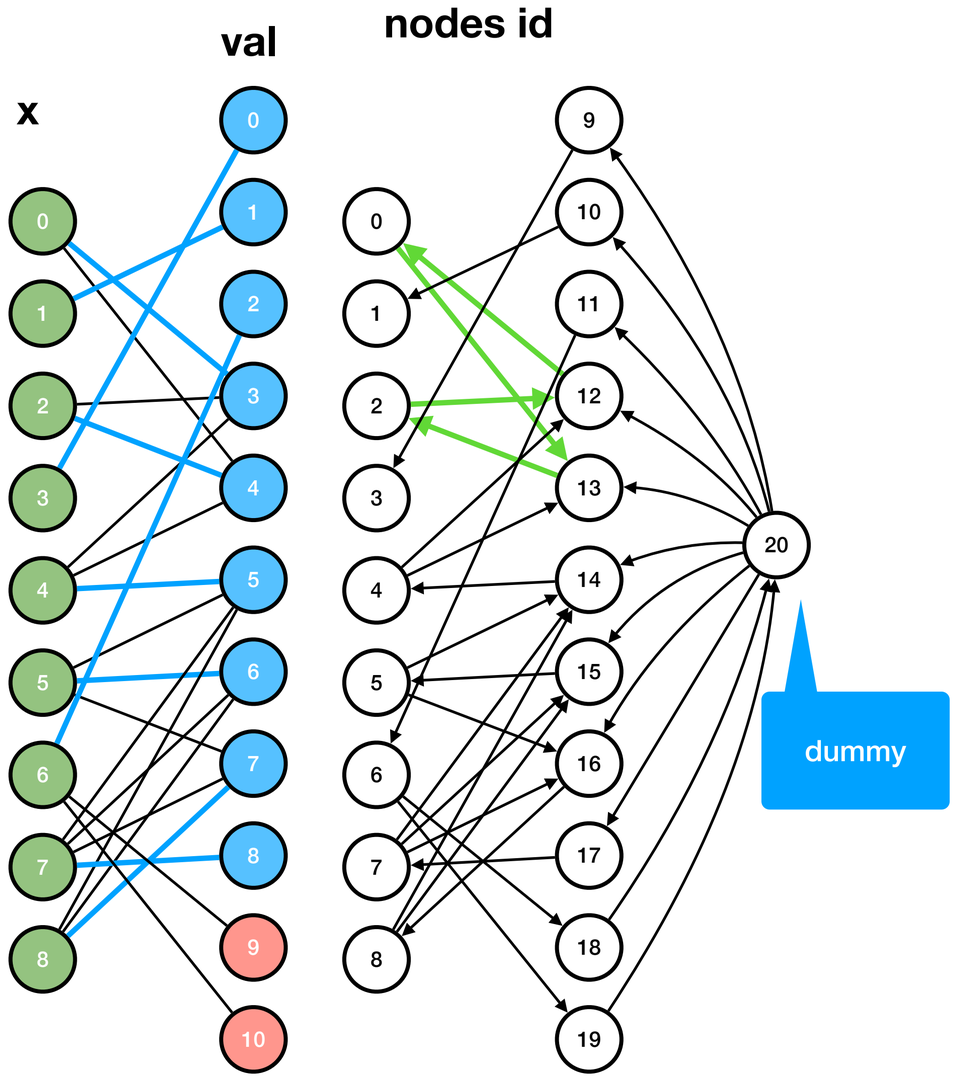


Berge Theorem: Merge the Two Cases

Transformation:
Direct the graph (like above) and 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 contains an arc from a node in M to an M-free node corresponds to an *even-length alternating path* w.r.t. M. (Using dummy node = case 1)
- Every directed cycle that does not contain an arc from a node in M to an M-free node corresponds to *an even-length alternating cycle* w.r.t. M. (Not using dummy node = case2)

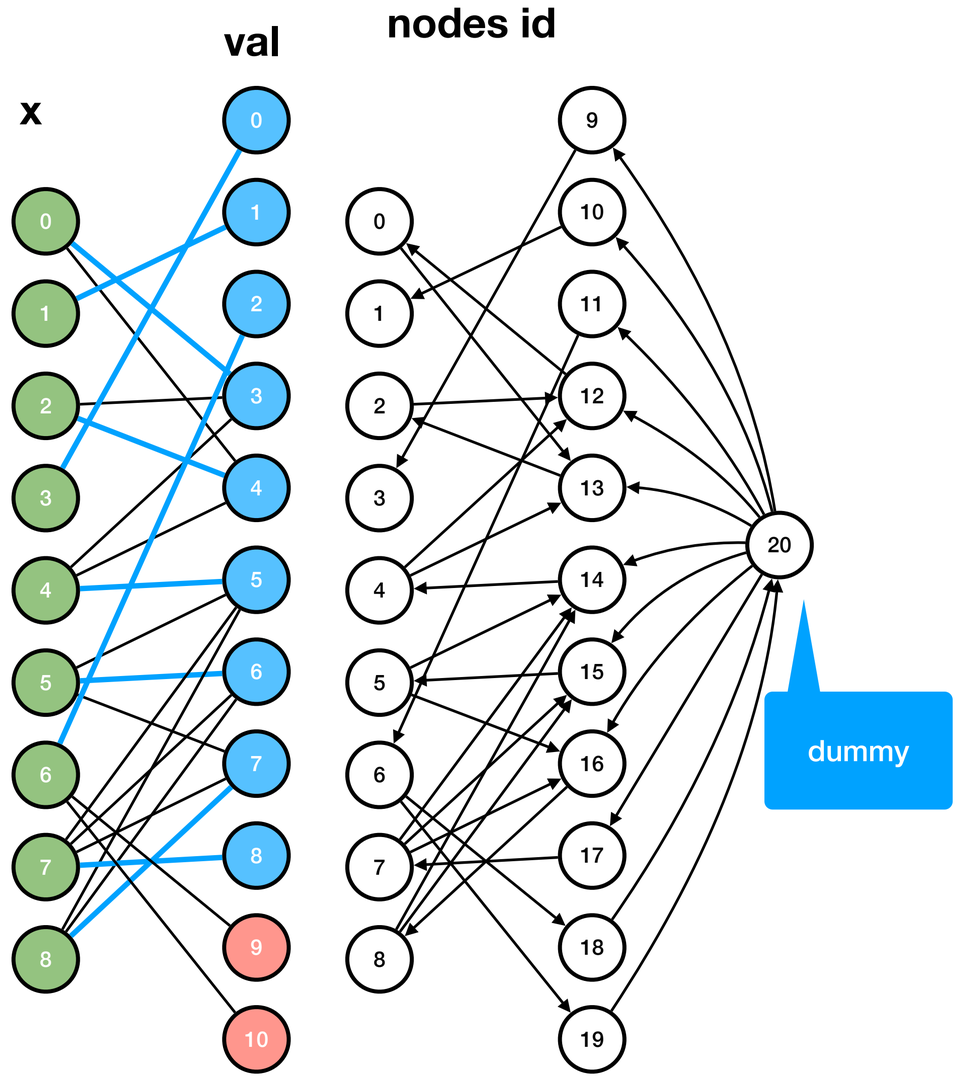


Berge Theorem: Merge the Two Cases

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.

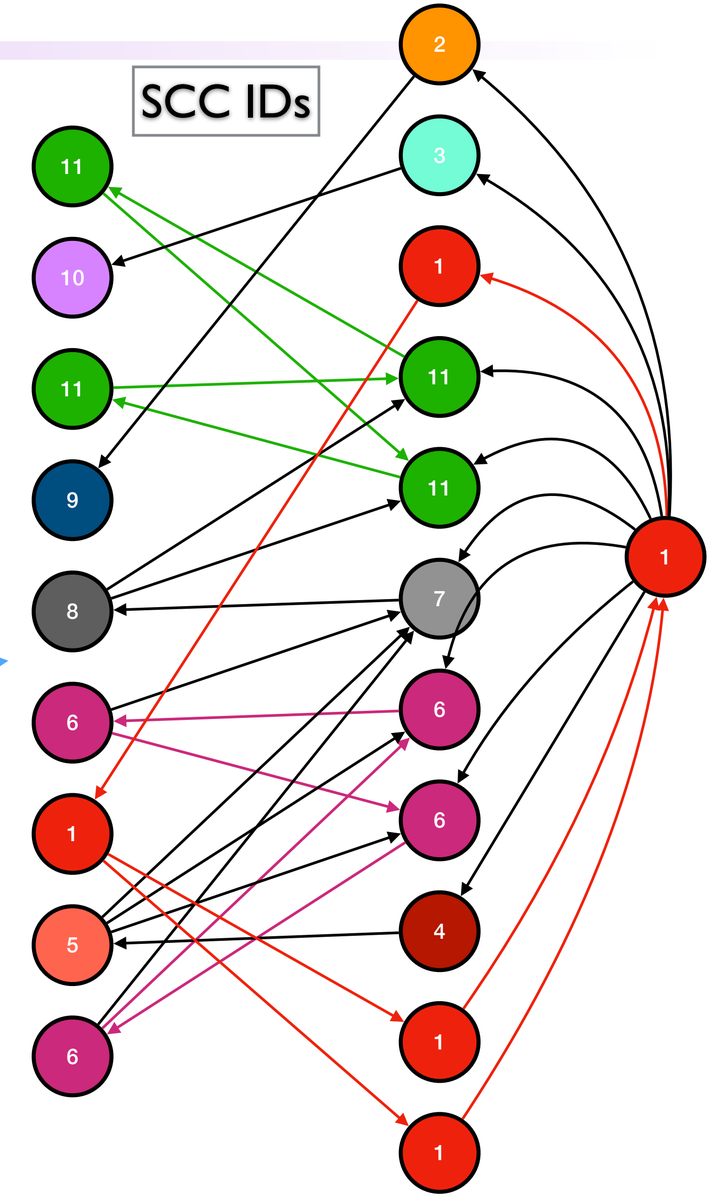


Domain Consistency (Régin 1994)

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

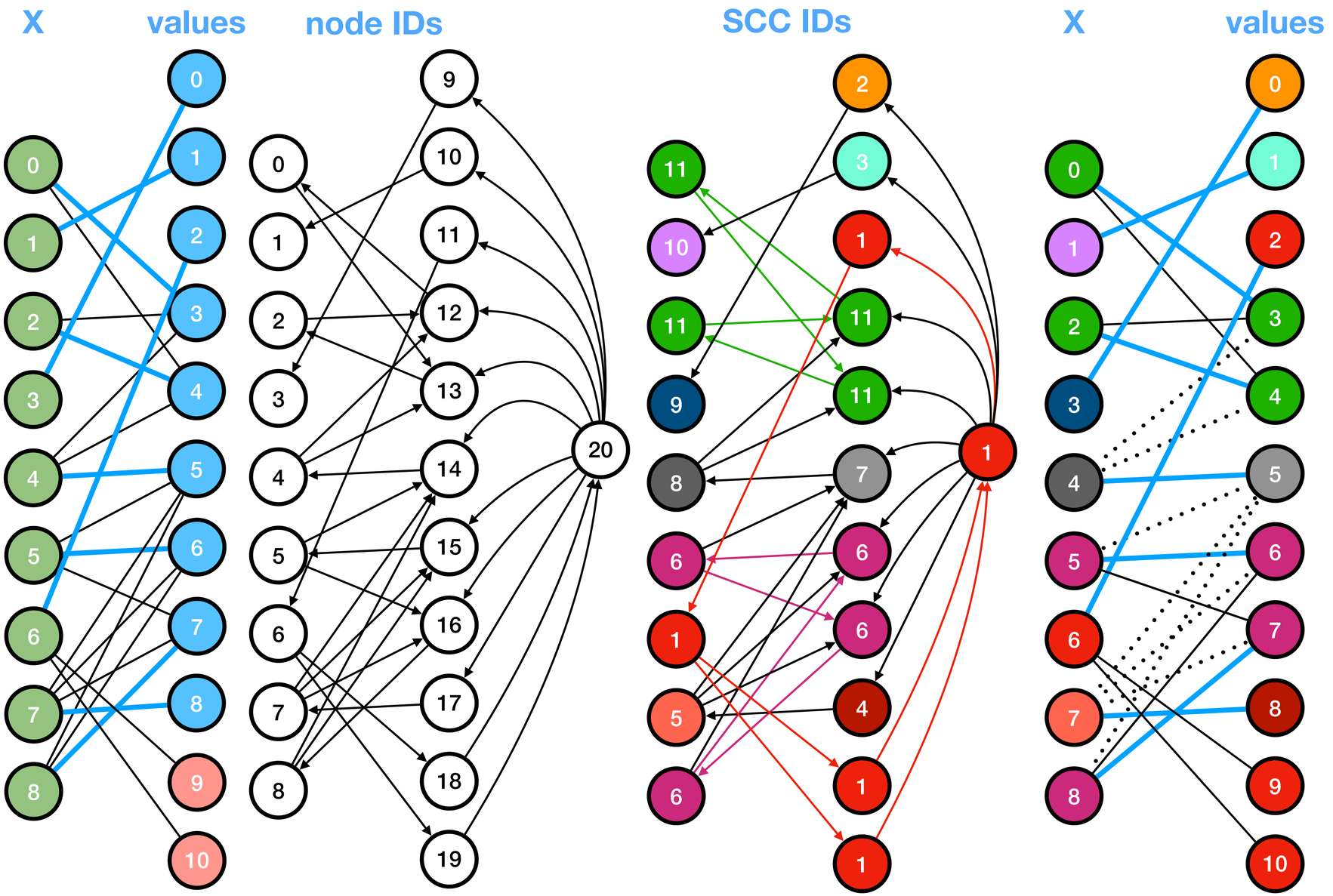
Filtering algorithm for AllDifferent(X):

- 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 and a & x belong to two different SCCs of the transformed graph for M.



The black arcs connect different SCCs:
those not corresponding to the matching M must be removed.

AllDifferent Filtering: Summary

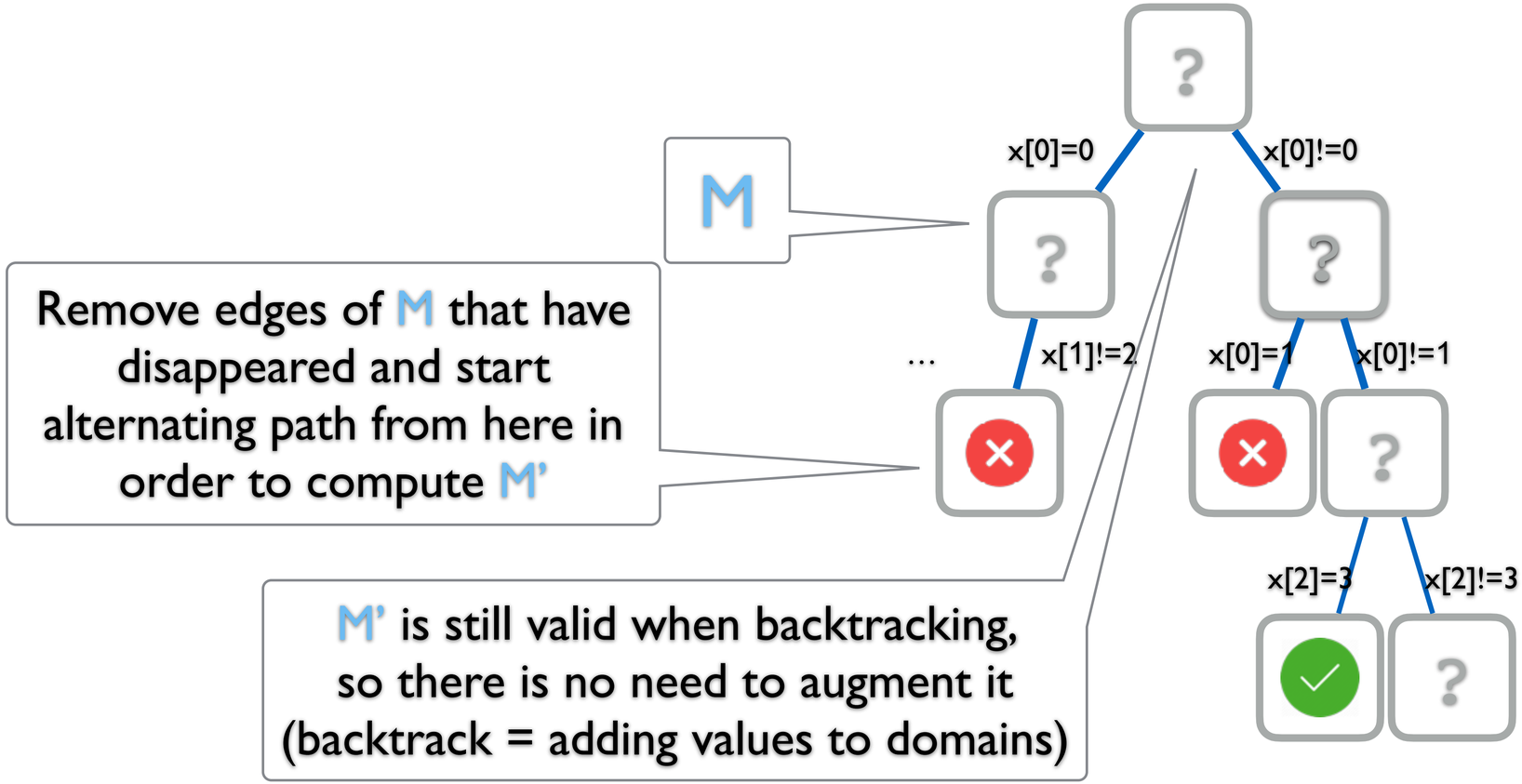


Implementation Considerations:
Backtracking and Incremental Updates
+
Explanations on the programming assignment

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 above).
- Re-filter.

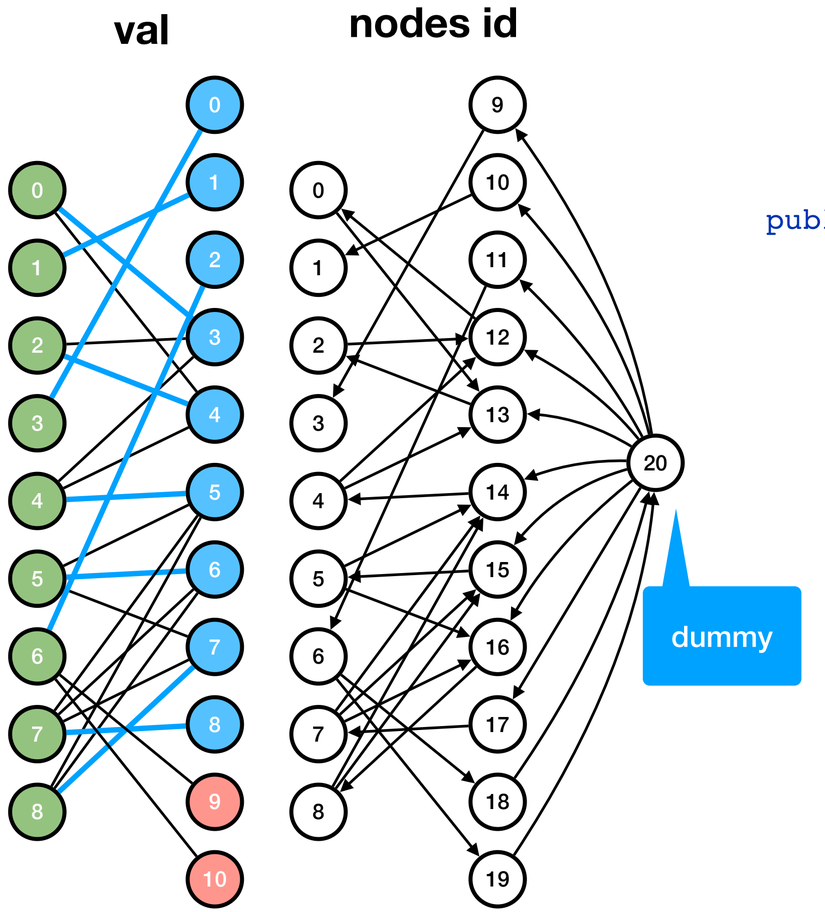


Assigment

```
public class MaximumMatching {

    public MaximumMatching(IntVar... x);

    public int compute(int[] result);
}
```

Graph API (👩‍💻 assigment)

```
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]; }
    };
```



🎁

```
public static int[] stronglyConnectedComponents(Graph graph);
```

AllDifferent Implementation (👨‍💻 assignment)

```
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;
}
```

in[i] = in adjacent nodes

Residual Graph

AllDifferent Implementation (👩‍💻 assignment)

```
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
    }
```

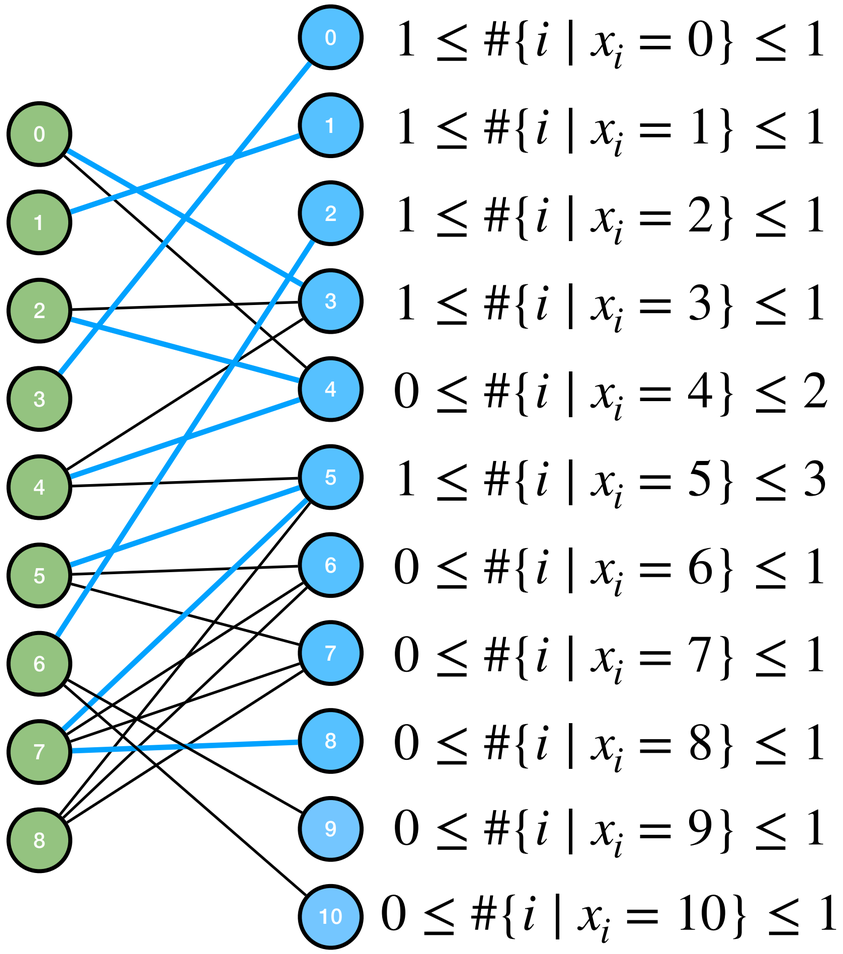
in[i] = in adjacent nodes

Time Complexity

- $O(m\sqrt{n})$, 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.
  - Fun fact: in practice, most of the time is spent there because just a few iterations are needed to retrieve (if needed) the maximum matching.

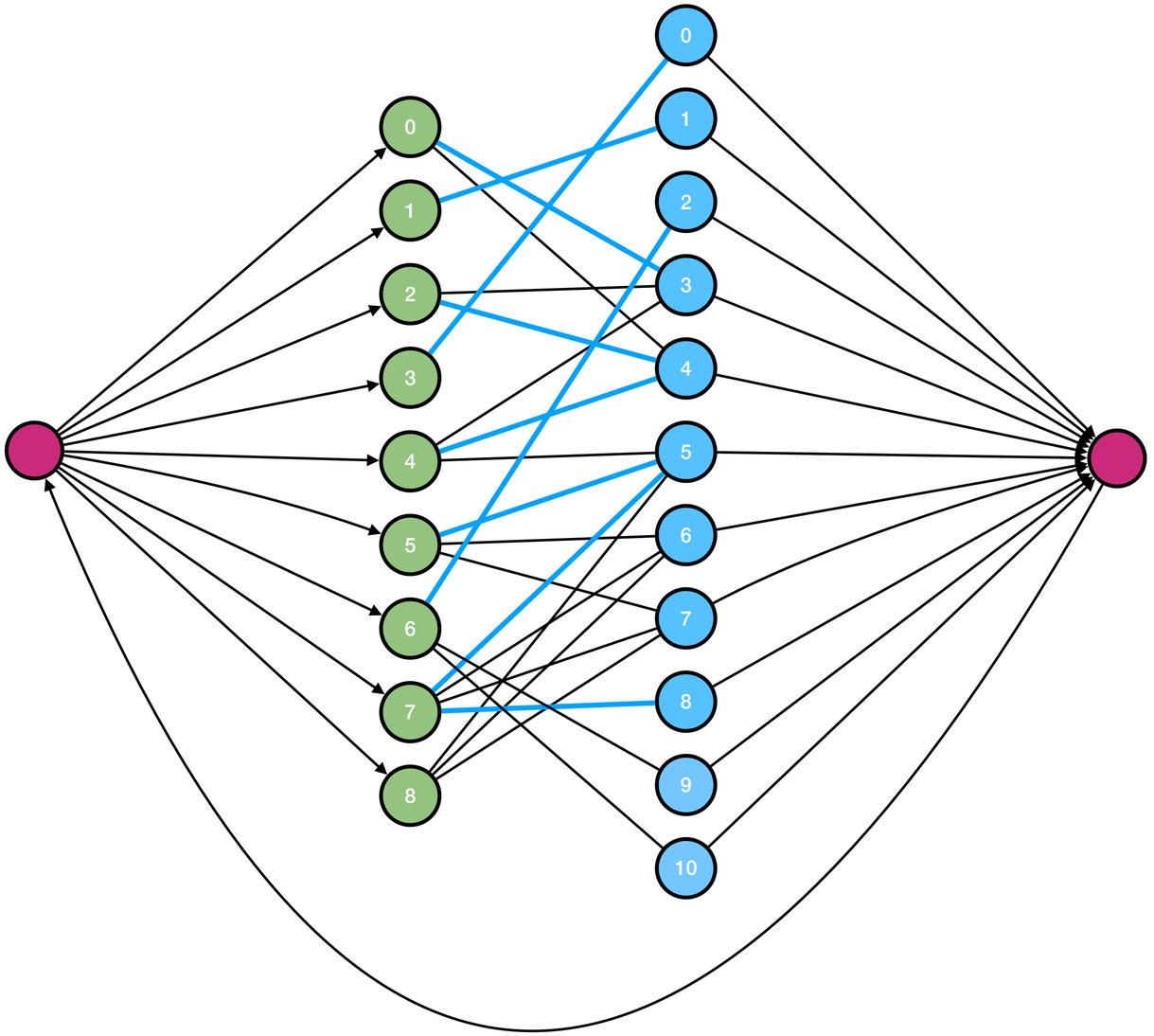
Generalizing the AllDifferent Constraint

Global Cardinality Constraint (GCC)



Régin, Jean-Charles. "Arc consistency for global cardinality constraints with costs." *International Conference on Principles and Practice of Constraint Programming*. 1999.

GCC Feasibility Check = Maximum Flow with Cardinality



$$1 \leq \#\{i \mid x_i = 0\} \leq 1$$
$$1 \leq \#\{i \mid x_i = 1\} \leq 1$$
$$1 \leq \#\{i \mid x_i = 2\} \leq 1$$
$$1 \leq \#\{i \mid x_i = 3\} \leq 1$$
$$0 \leq \#\{i \mid x_i = 4\} \leq 2$$
$$1 \leq \#\{i \mid x_i = 5\} \leq 3$$
$$0 \leq \#\{i \mid x_i = 6\} \leq 1$$
$$0 \leq \#\{i \mid x_i = 7\} \leq 1$$
$$0 \leq \#\{i \mid x_i = 8\} \leq 1$$
$$0 \leq \#\{i \mid x_i = 9\} \leq 1$$
$$0 \leq \#\{i \mid x_i = 10\} \leq 1$$

Régin, Jean-Charles. "Arc consistency for global cardinality constraints with costs." *International Conference on Principles and Practice of Constraint Programming*. 1999.