Constraint Systems

About the Problem
from the Last Lab Session

Production Line Scheduling

A small company can produce a number of product types

- Every time unit, only a single product unit can be manufactured

The company has received a number of orders

- Each order refers to a single product type
- Each order requires a certain number of product units
- Each order has a deadline, which cannot be exceeded

Some pairs of products ⟨**`p1, p2`**⟩ are associated to a setup time:

- After manufacturing a unit of **`p1`**, before switching to **`p2`**
- We need to wait 1 time unit, or to manufacture another product type

Production Line Scheduling

Goal:

- Model & solve the problem using CP
- Satisfy all constraints
- Minimize the makespan

Let's see our two models, formally presented

A First Possible Model (Model 1)

Model #1

Main idea: variables = manufacturing times of all product units

Parameters:

- n = number of product units
- d_i = deadline for product unit i
- p_i = product type for unit i
- eoh = safe time horizon (largest deadline)
- $S = \{(p_a, p_b), \ldots\}$ = ordered pairs with setups times

A First Possible Model (Model 1)

The full model:

$$\min z = \max_{i=0..n-1} s_i$$

subject to:

$$s_i \neq s_j \quad \forall i, j = 0..n-1, i < j$$

$$s_i \leq d_i \quad \forall i = 0..n-1$$

$$s_j \neq s_i + 1 \quad \forall i, j : (p_i, p_j) \in S$$

$$s_i \in \{0..eoh\} \quad \forall i = 0..n-1$$

The $s_j \neq s_i + 1$ constraints correspond to the setup times:

- If a setup is needed between unit `i` and `j`...
- ...then the two cannot be consecutive

A Second Possible Model (Model 2)

Model #2

Main idea: variables = products to be manufactured at each time point

Parameters:

- `n` = number of products
- `eoh` = safe time horizon (largest deadline)
- `m` = number of orders
- d_k = deadline for order `k`
- p_k = product type for order `k`
- n_k = number of product units for order `k`
- $S = \{(p_a, p_b), \ldots\}$ = ordered pairs with setups times

A Second Possible Model (Model 2)

The full model:

$$\min z = \max_{t=0..eoh} t\,(x_t \neq -1)$$

$$\text{subject to: } \sum_{t=0..d_k} (x_t = p_k) \geq \sum_{\substack{h=0..m-1,\\ d_h \leq d_k}} n_h \qquad \forall k = 0..m-1$$

$$(x_t = p_a) \leq (x_{t+1} \neq p_b) \qquad \forall i, j : (p_a, p_b)$$

$$x_t \in \{-1..n-1\} \qquad \forall t = 0..eoh$$

- `-1` is used for idle production times
- Deadlines:
  - For each order k, the sum of units of p_k produced before d_k…
  - …Must be greater than all units of p_k needed up to d_k
- Makespan: convert time indices to production times (multiplication)

More About this in the Future

This is not the last we see about this problem

Once of the next lecture will be totally dedicated to it!

Constraint Systems

Global Constraints - ALLDIFFERENT

Back to Our PLS Examples

		2	
			3
		3	
	4		

- Remember this?
- It was one of our very first CSP examples

Back to Our PLS Examples

1 2 3 4	1 2 3 4	2	1 2 3 4
1 2 3 4	1 2 3 4	1 2 3 4	3
1 2 3 4	1 2 3 4	3	1 2 3 4
1 2 3 4	4	1 2 3 4	1 2 3 4

- These are the initial domains, with one $\mathtt{x}_{\mathtt{i,j}}$ variable per cell

Back to Our PLS Examples

1 4	1 3	2	1 4
1 2	1 2	4	3
1 2 4	1 2	3	1 4
3	4	1	2

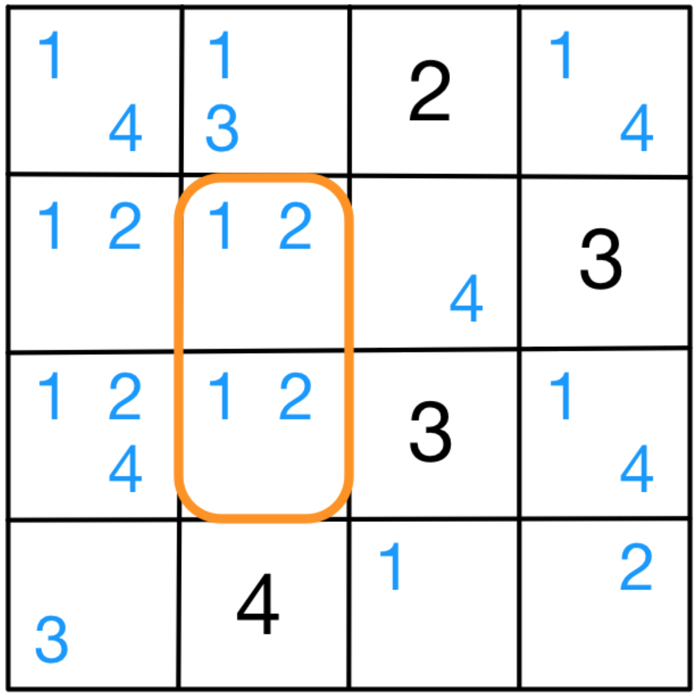
- And these are the domains at the GAC fix point
- ...which was an impressive reduction

Back to Our PLS Examples

1 4	1 3	**2**	1 4
1 2	1 2	4	**3**
1 2 4	1 2	**3**	1 4
3	**4**	1	2

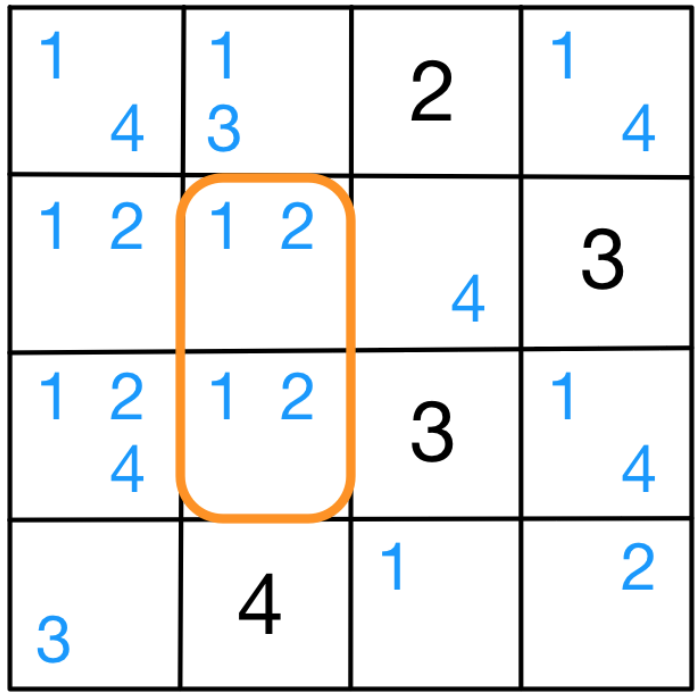
But it could be better!

Back to Our PLS Examples



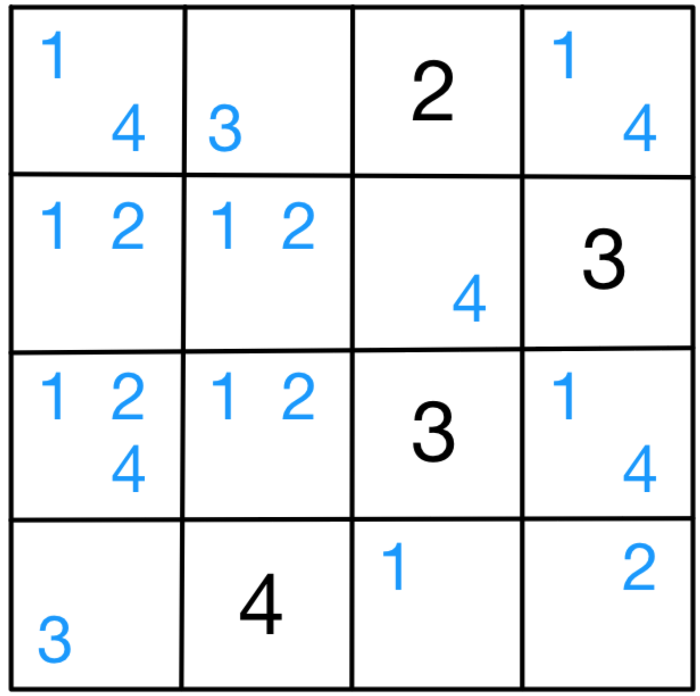
- Consider the two variables $x_{1,1}$ and $x_{2,1}$
- They must take a value in $\{1, 2\}$, which has cardinality 2

Back to Our PLS Examples



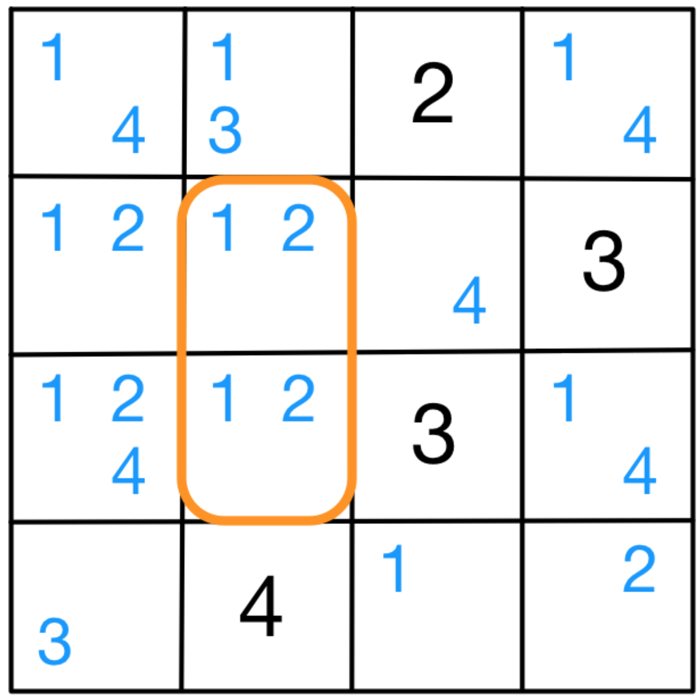
- If we assign 1 or 2 to another variable, the column is infeasible
- Therefore values 1 and 2 must be assigned to them

Back to Our PLS Examples



- And we can remove value 1 from the domain of $x_{0,1}$!

Back to Our PLS Examples



But the constraints were all GAC!

How did we manage to filter more?

Global vs Local Filtering

Main idea: **reasoning on the whole column**

- Let the column variables be **X** and the union of their domains **V**
- The variables on each column must be all different

If we find a a set of values **W** ⊂ **V** and a set of variables **Y** ⊂ **X**, s.t.:

$$|Y| = |W| \quad \text{and} \quad D(x_i) \subseteq W, \quad \forall x_i \in Y$$

Then:

- **W** is called a Hall set for **X**
- The values in **W** will all be taken by the variables in **Y**
- We can prune **W** from the domains of the other variables

Hall Set Filtering for All Different Variables

Formally, we should have $\forall W \subset V, Y \subset X$ with $|Y| = |W|$:

$$D(x_i) \subseteq W, \quad \forall x_i \in Y \quad \Rightarrow \quad D(x_j) = D(x_j) \setminus W, \quad \forall x_j \in X \setminus Y$$

- This "Hall set filtering" enforces GAC on the whole column
- Hence, it is the best we can achieve for a set of all different variables

Now, we could encode the implications as additional constraints

- The would not be necessary...
- ...But they would still allow more filtering

A constraint with this properties is called redundant

Hall Set Filtering for All Different Variables

Still, we have no luck

- Even if we manege to encode this implication as a constraint...

$$D(x_i) \subseteq W, \quad \forall x_i \in Y \quad \Rightarrow \quad D(x_j) = D(x_j) \setminus W, \quad \forall x_j \in X \setminus Y$$

- ...We should still add one such constraint for...

$$\forall W \subset V, Y \subset X \text{ with } |Y| = |W|$$

And this is bad :-(

- The number of subsets of `V` is exponential
- The number of subsets of `X` is exponential

Shall we give up? Let's take a different appraoch instead

Global Alldifferent Constraint

We can introduce a new global constraint:

> **ALLDIFFERENT(X)**, where **X** is a vector of variables

- Semantically, it is equivalent to $x_i \neq x_j, \forall i \neq j$...
- ...But the filtering algorithm can be written ad hoc...
- ...Using virtually any algorithmic technique

In the case of **ALLDIFFERENT** polynomial GAC propagators do exist

- And now we will see one of them...

A Propagator for **ALLDIFFERENT**

We will see an **ALLDIFFERENT** propagator based on network flows:

- The classical propagator is instead based on graph matchings
- For more details, there is a (excellent) paper on the course web site

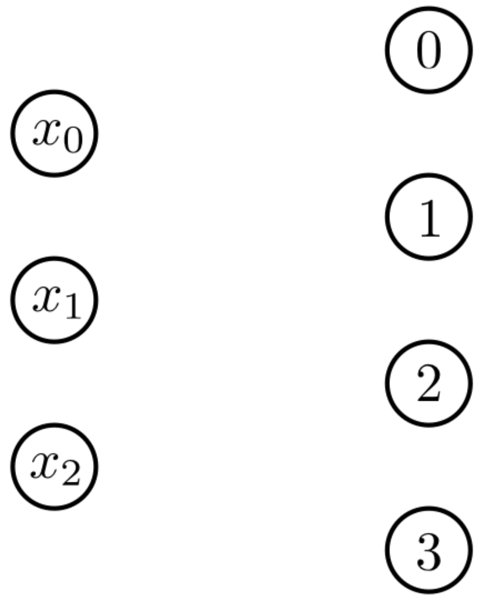
We consider two distinct problems:

- Checking the constraint feasibility
- Performing filtering

We will use this example instance:

ALLDIFFERENT(X), with $x_0 \in \{0, 2\}, x_1 \in \{0, 2\}, x_3 \in \{1, 2, 3\}$

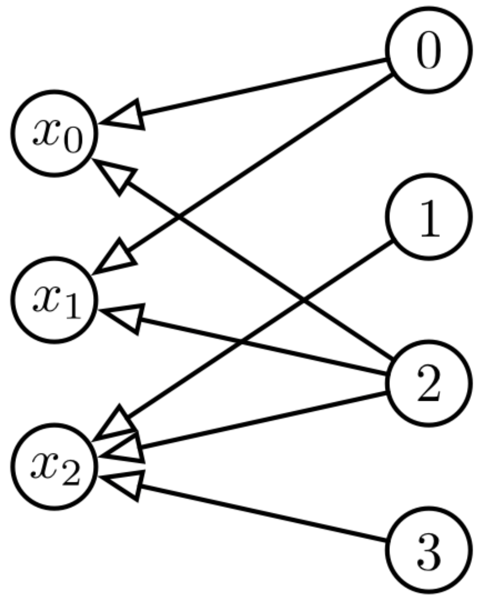
Value Graph



For any constraint, we can define the so-called value graph

- Left-hand nodes = variables
- Right-hand node = values

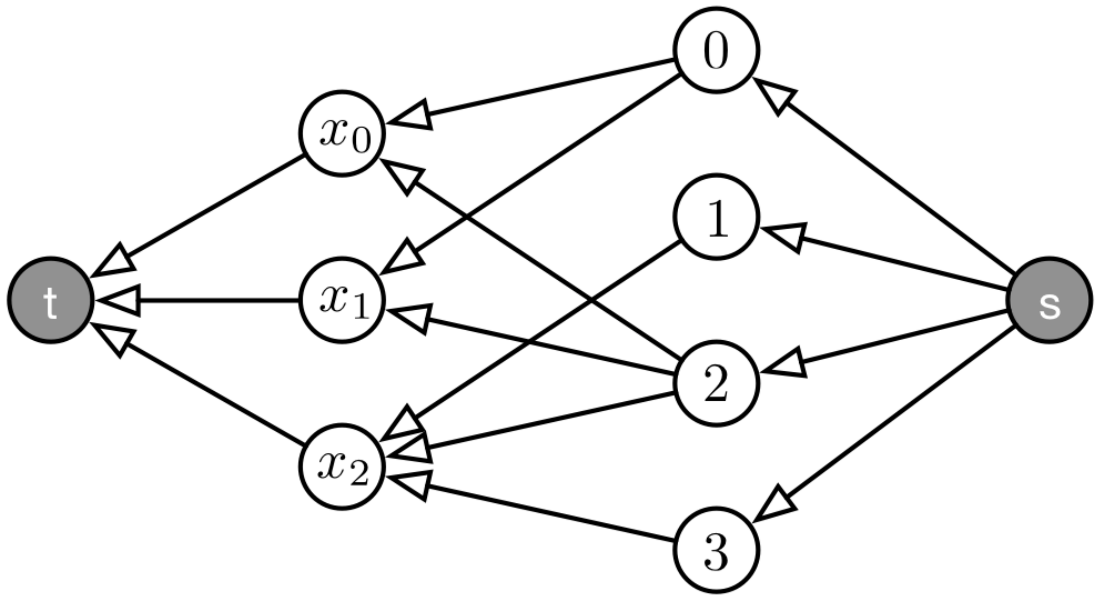
Value Graph



For any constraint, we can define the so-called value graph

- Arcs = possible assignments of values to variables (i.e. the domains)
- The value graph is bipartite (by construction)

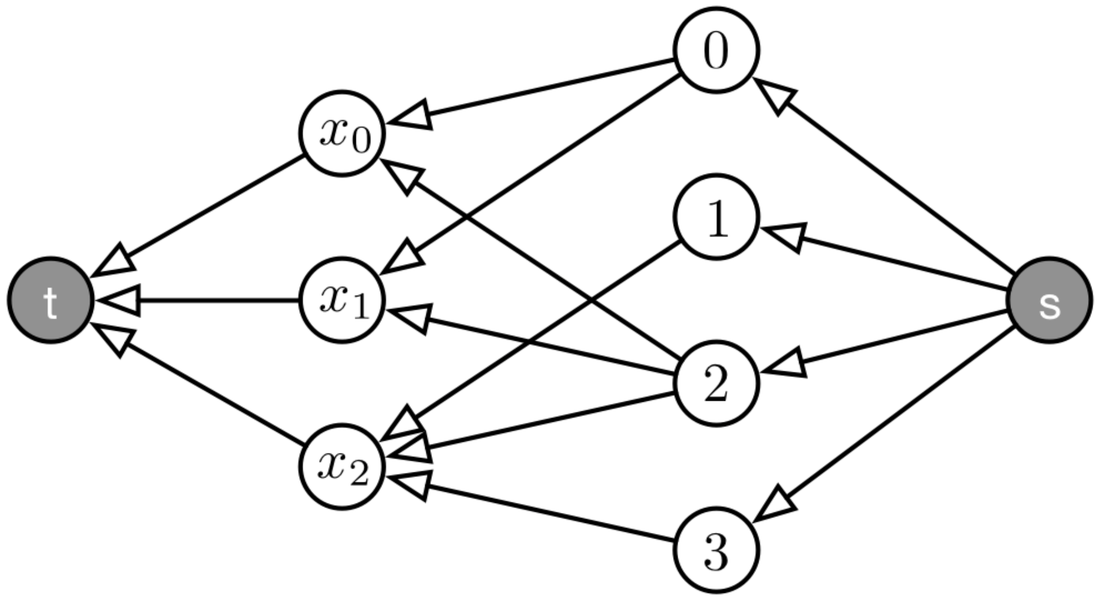
Flow Based Representation



Now, let's add:

- An additional "source" node `s`, connected to all values
- An additional "sink" node `t`, to which all vars are connected

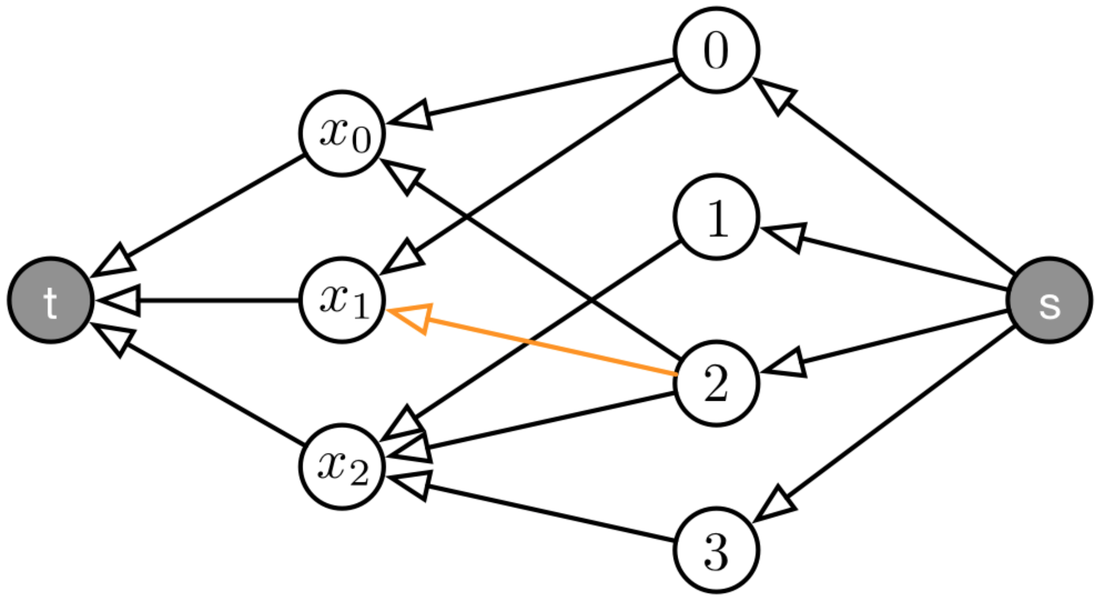
Flow Based Representation



We can view this structure as a "pipe network":

- Each arc is a "tube", with capacity equal to 1
- Flow can originate from `s` and move toward `t`

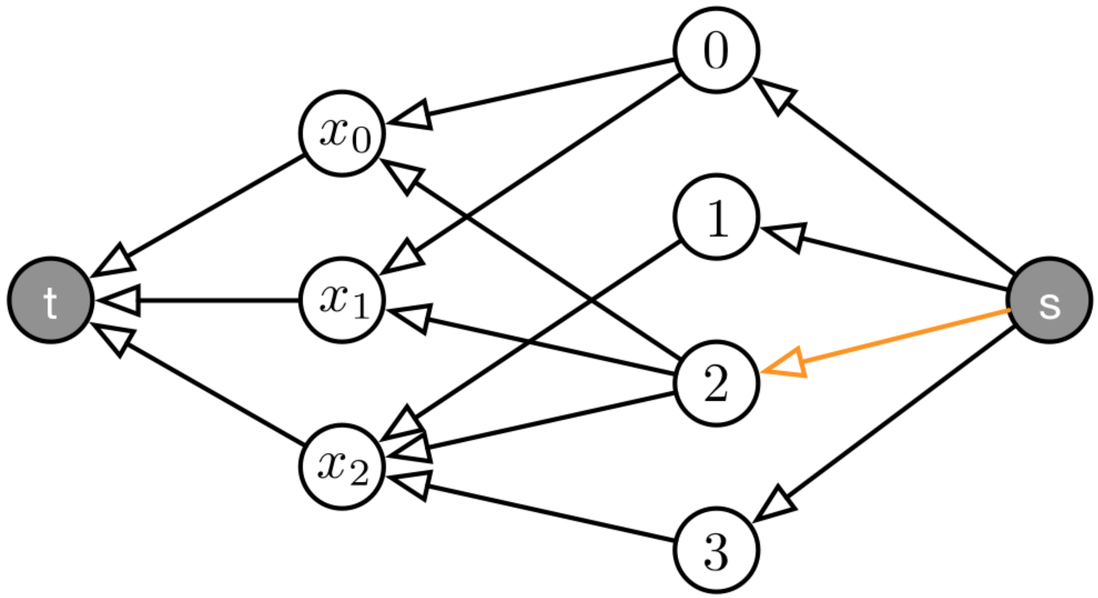
Flow Based Representation



How do we "read" the flow?

- If there is flow on arc $v_j \rightarrow x_i$, then v_j is assigned to x_i
- The capacity is 1 = v_j cannot be assigned twice to the same var

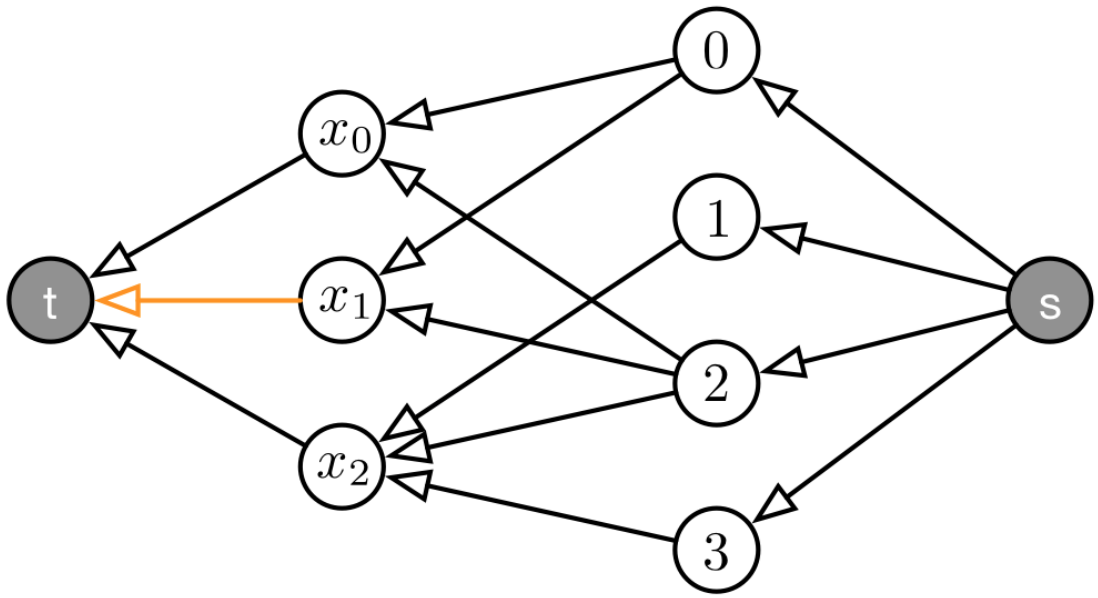
Flow Based Representation



How do we "read" the flow?

- If there is flow on arc `s → vj`, then `vj` is used
- The capacity is 1 ⇒ each `vj` can be used at most once

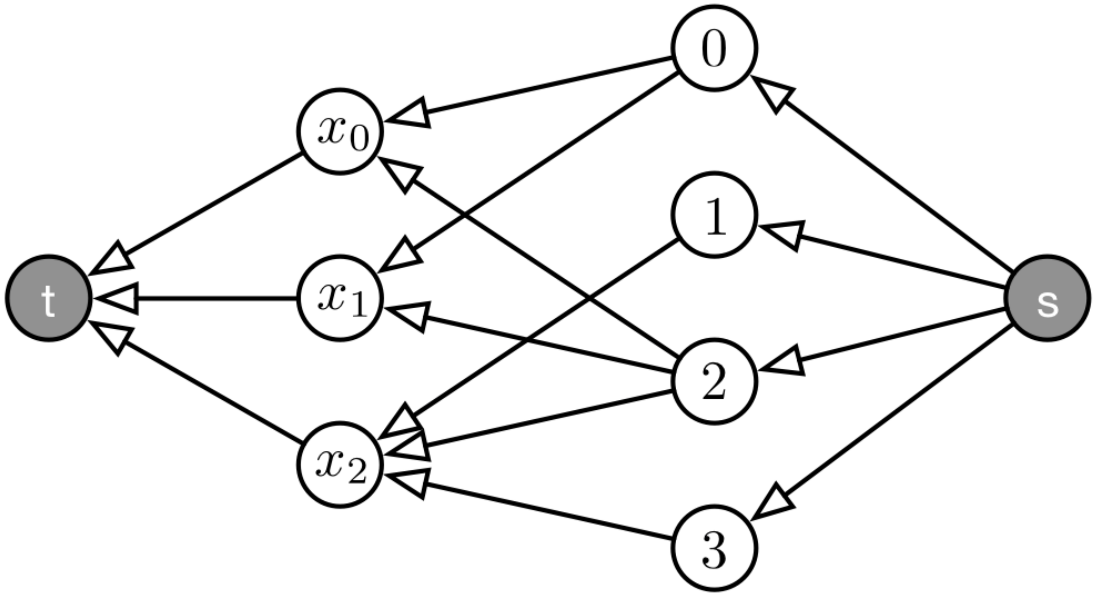
Flow Based Representation



How do we "read" the flow?

- If there is flow on arc $x_i \rightarrow t$, then x_i is assigned
- The capacity is 1 $\Rightarrow$ each x_i can be assigned at most once

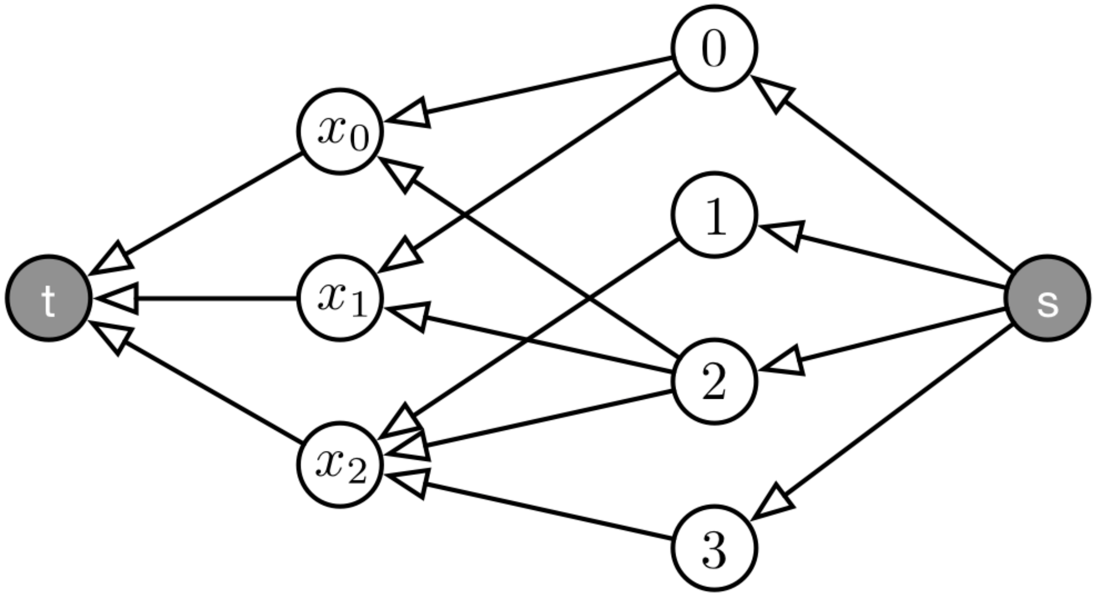
Flow Based Representation



Important consequence:

- A solution exists iff all variables are assigned
- I.e. if there exist a flow with total value equal to $|\mathbf{x}|$

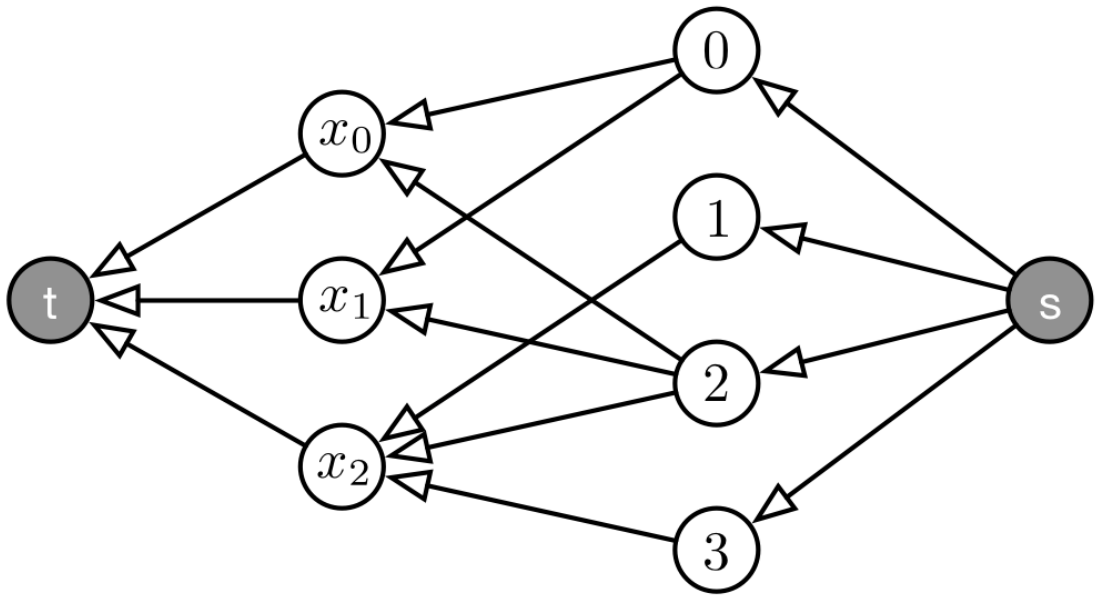
Flow Based Representation



Hence, we have our feasibility check:

- Route the maximum possible flow from `s` to `t`
- Check if the total flow value is equal to `X`

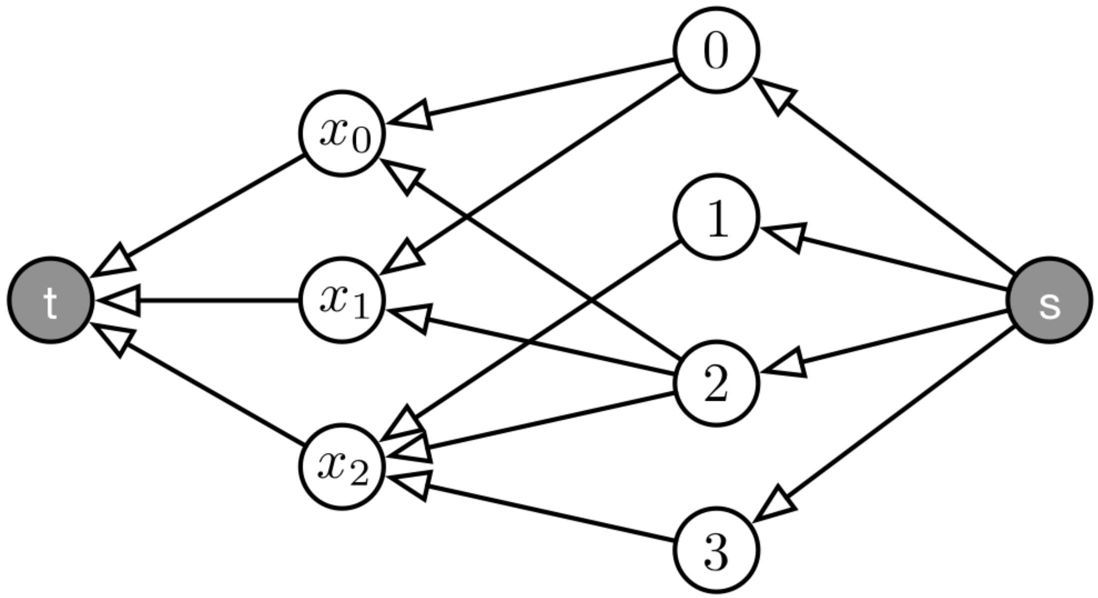
Maximum Flow for ALLDIFFERENT



How do we solve this maximum flow problem?

- Several algorithms are available
- We will use a specialization of the Edmonds-Karp algorithm
- In turn, it is specialization of the Ford-Fulkerson method

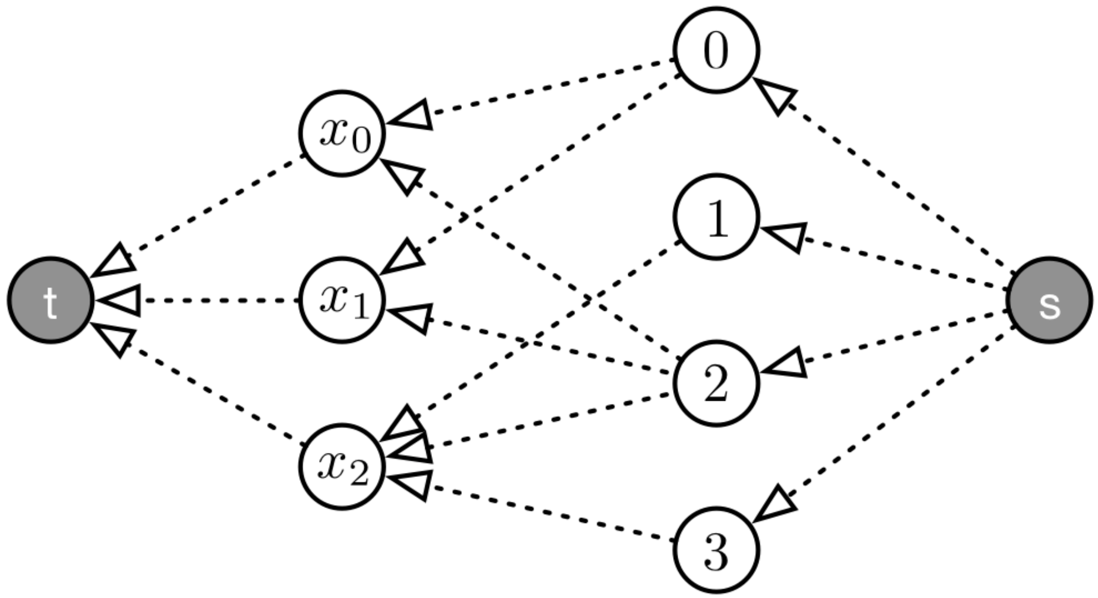
Maximum Flow for ALLDIFFERENT



General idea:

- Start from a feasible flow (i.e. all capacities respected)
- Iteratively augment the flow value

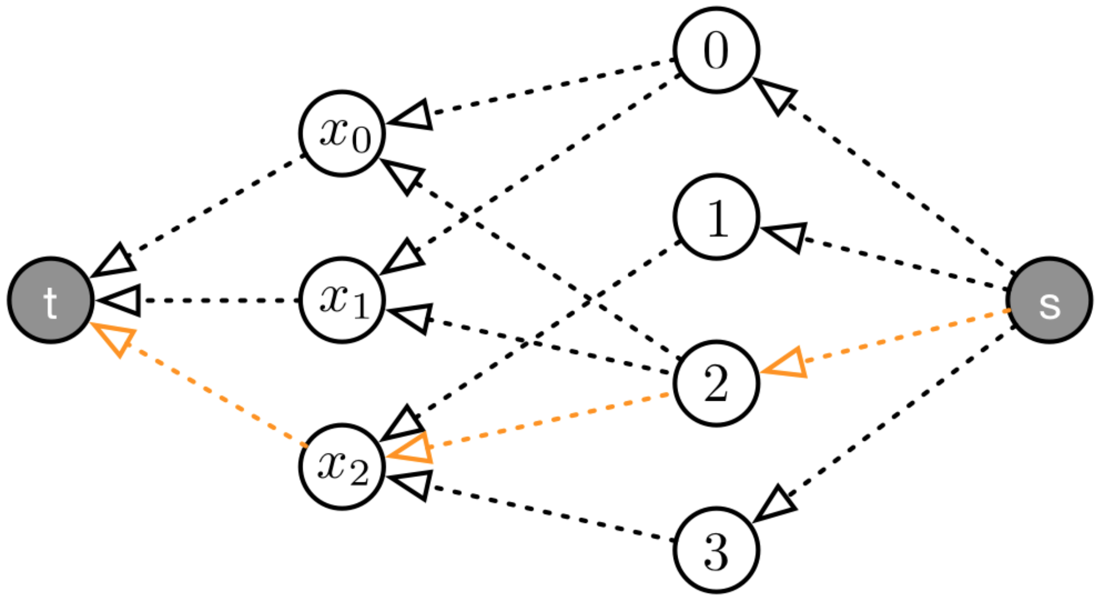
Maximum Flow for ALLDIFFERENT



Our (trivial) initial flow:

- The flow `f(a → b)` for all arcs is 0
- **Notation:** dotted arcs: no flow, solid arcs: `f(a → b) = 1`

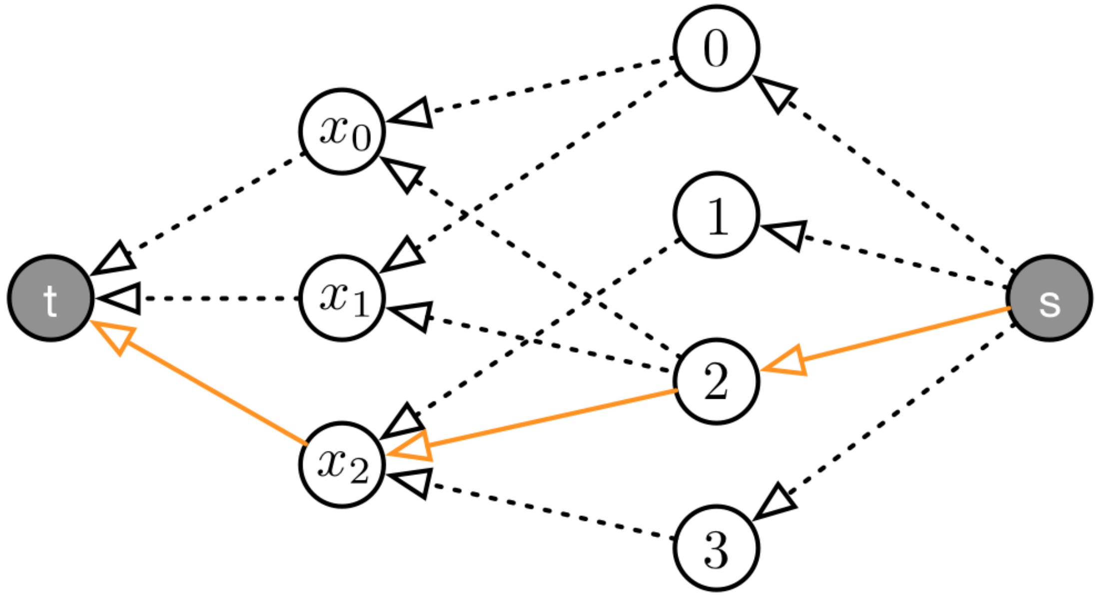
Maximum Flow for ALLDIFFERENT



Augmenting the flow value

- Find the shortest **s – t** path...
- ...made of non-saturated arcs (i.e. `f(a → b) < 1`)

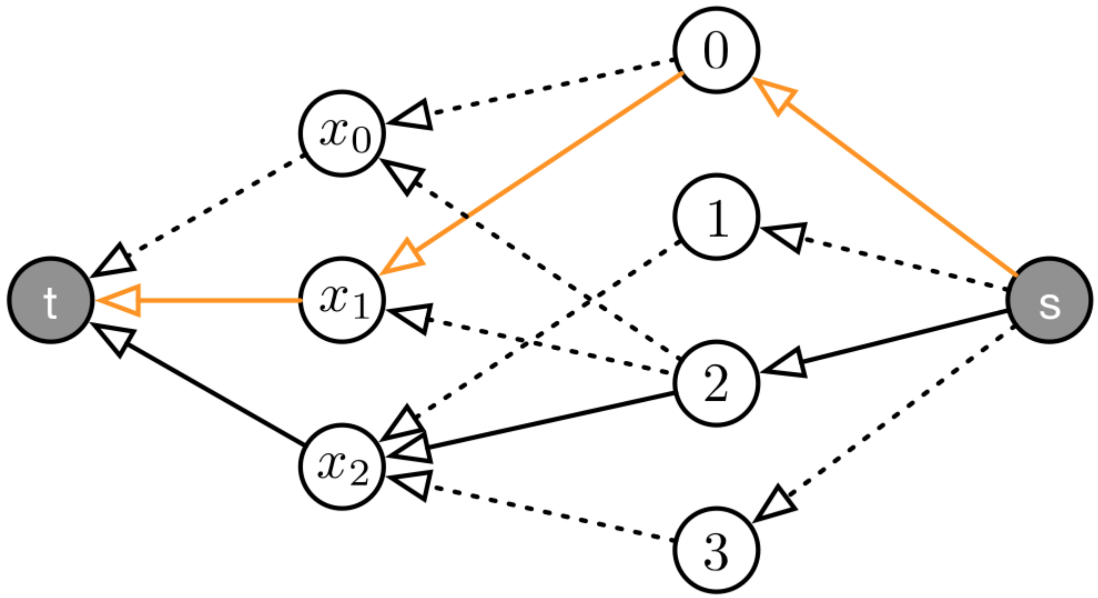
Maximum Flow for ALLDIFFERENT



Augmenting the flow value

- Route 1 unit of flow along the path

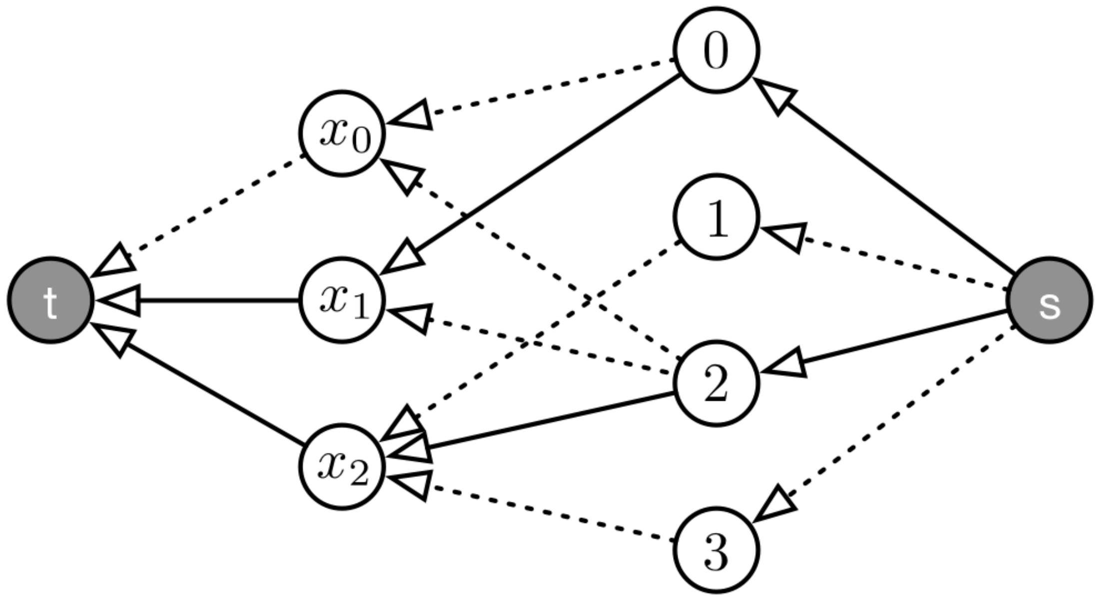
Maximum Flow for ALLDIFFERENT



Augmenting the flow value

- Repeat the process

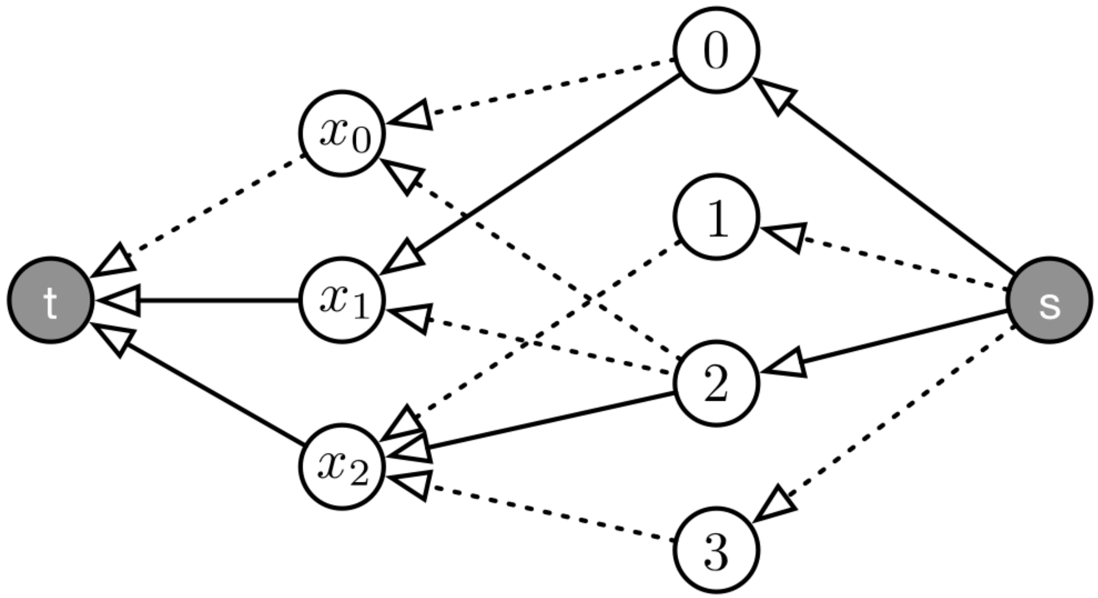
Maximum Flow for ALLDIFFERENT



Augmenting the flow value

- Until no more paths can be found

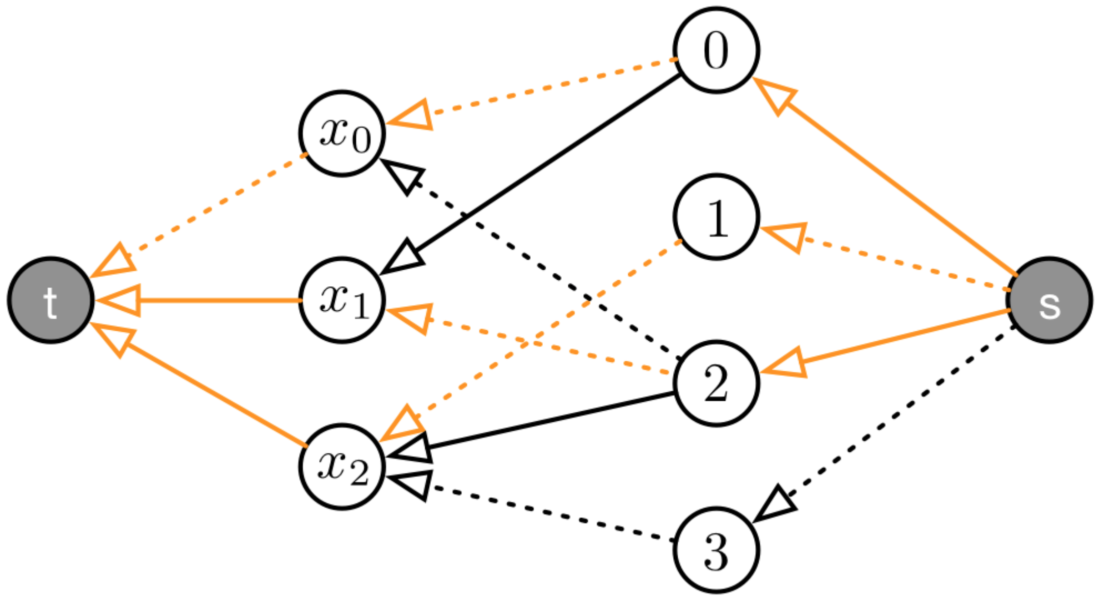
Maximum Flow for ALLDIFFERENT



Is that all? No, actually.

- Here it looks like we have reached a dead-end
- The total flow value is 2, which is less than $|X|$
- Hence, the constraint should be infeasible

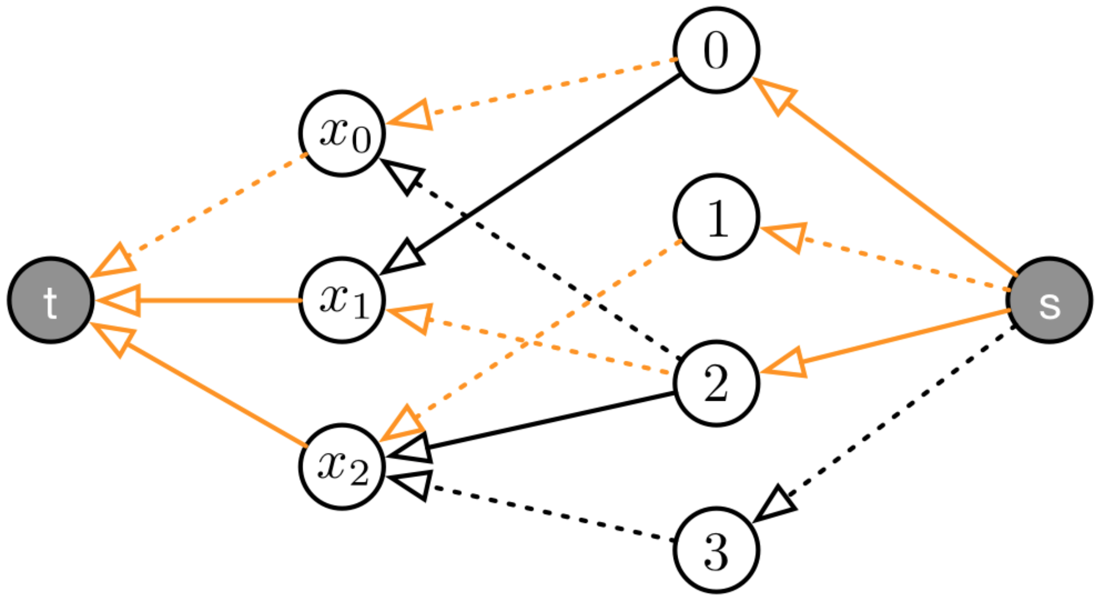
Maximum Flow for ALLDIFFERENT



But solutions do actually exist!

- For example $x_0 = 0, x_1 = 2, x_2 = 1$
- Which corresponds to routing flow along the orange arcs

Maximum Flow for ALLDIFFERENT



We are missing the ability to "undo" past choices

- We could use backtracking, but that is expensive
- Luckily, for flow problems there is a cheaper alternative...

Residual Graph for ALLDIFFERENT

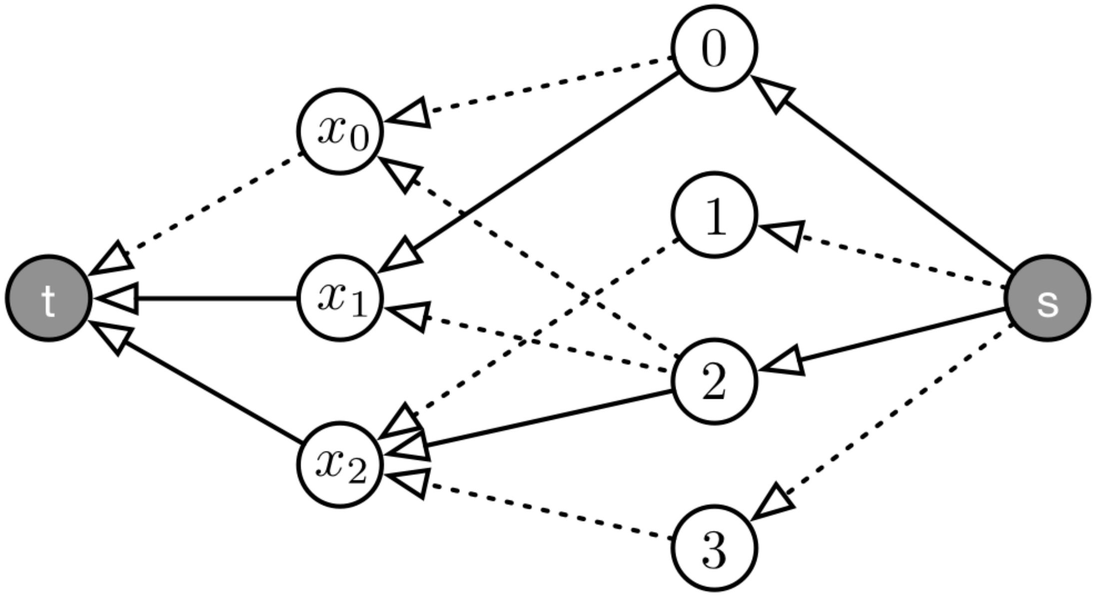
Main idea: we search for paths on a Residual Graph

- The residual graph has the same nodes as the original graph
- There is an arc **a → b** in the residual graph iff:
  - The is an arc **a → b** in the original graph and **f(a → b) = 0**
  - There is an arc **b → a** in the original graph and **f(b → a) = 1**

Intuitively, the residual graph:

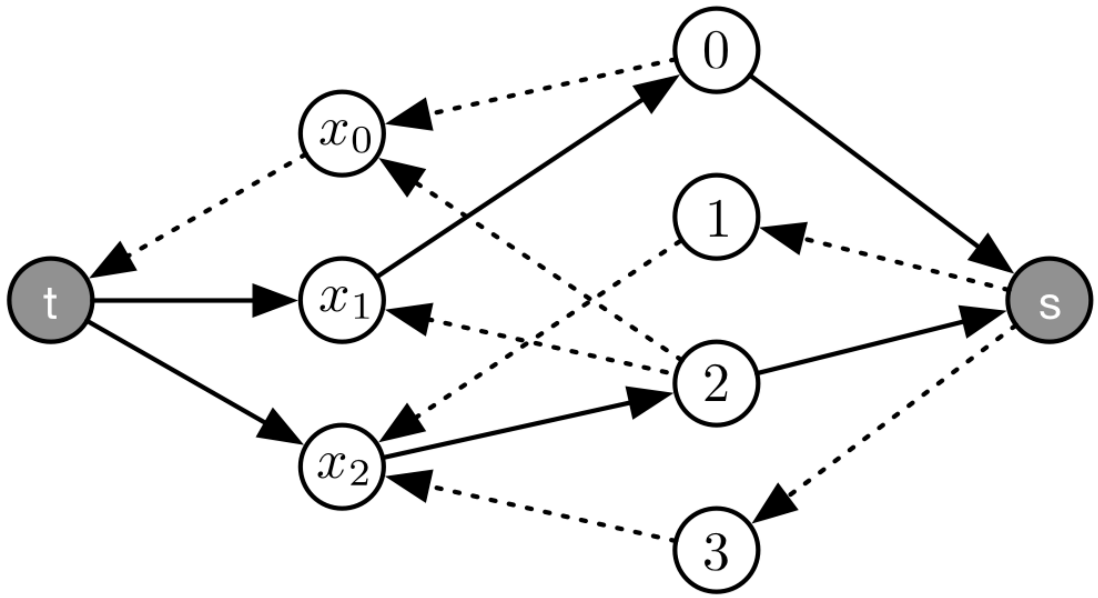
- Has an arc **a → b** if we can increase the flow along **a → b**
  - I.e. the flow is lower than the capacity (always 1)
- Has an arc **b → a** if we can decrease the flow along **a → b**
  - I.e. the flow is non-zero

Residual Graph for **ALLDIFFERENT**



- So, given our "dead-end" graph and flow

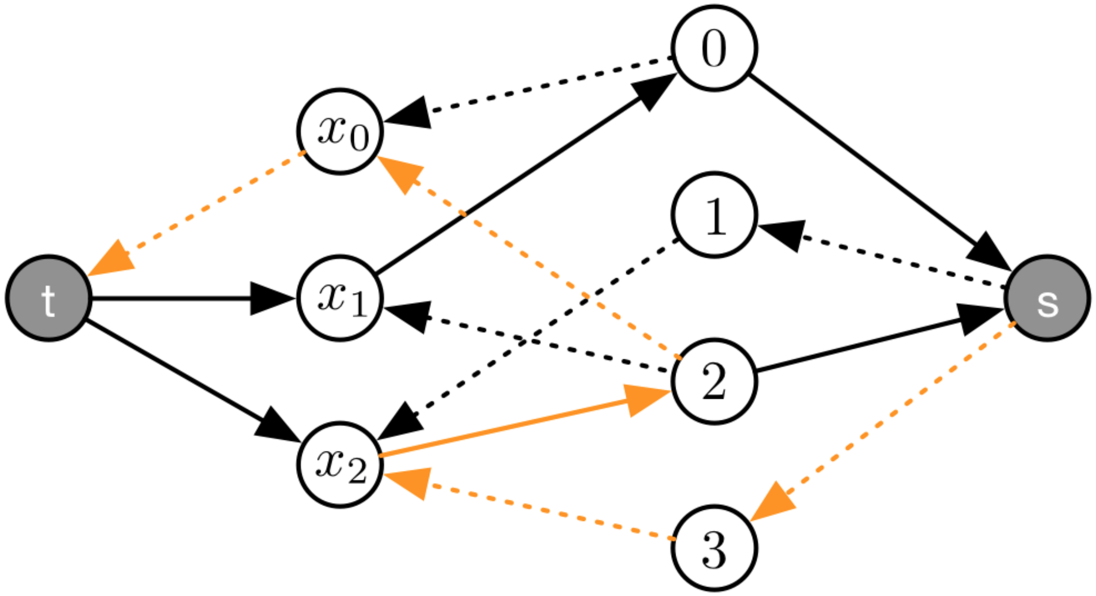
Residual Graph for **ALLDIFFERENT**



- So, given our "dead-end" graph and flow
- We obtain the following residual graph

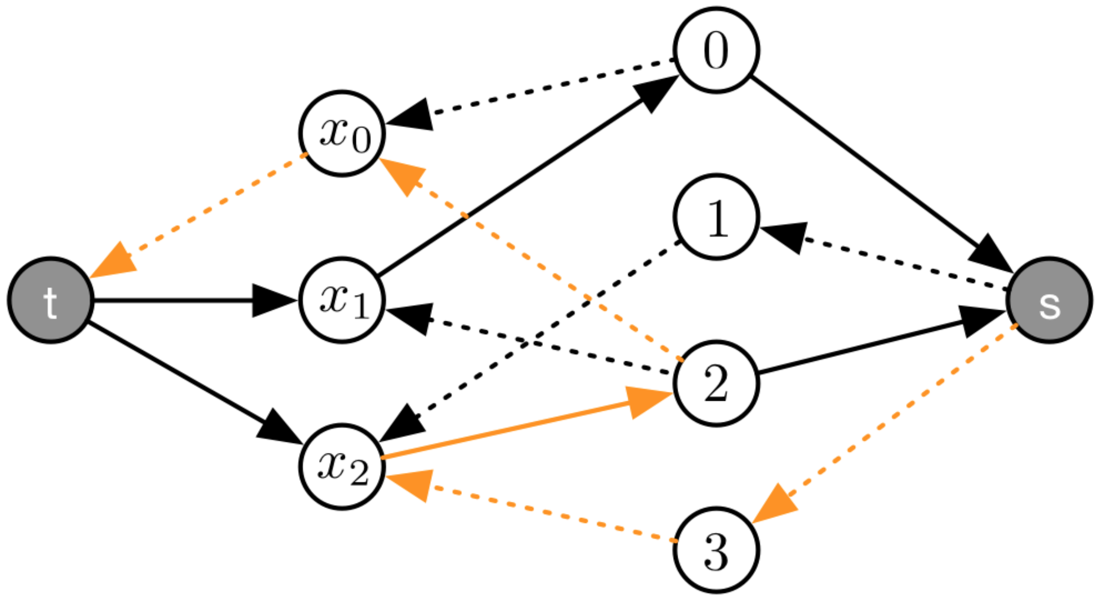
Notation: black arrow heads for the residual graph

Residual Graph for ALLDIFFERENT



- Our max-flow algorithm stays the same as before
- Except that we look for shortest **s – t** paths on the residual graph

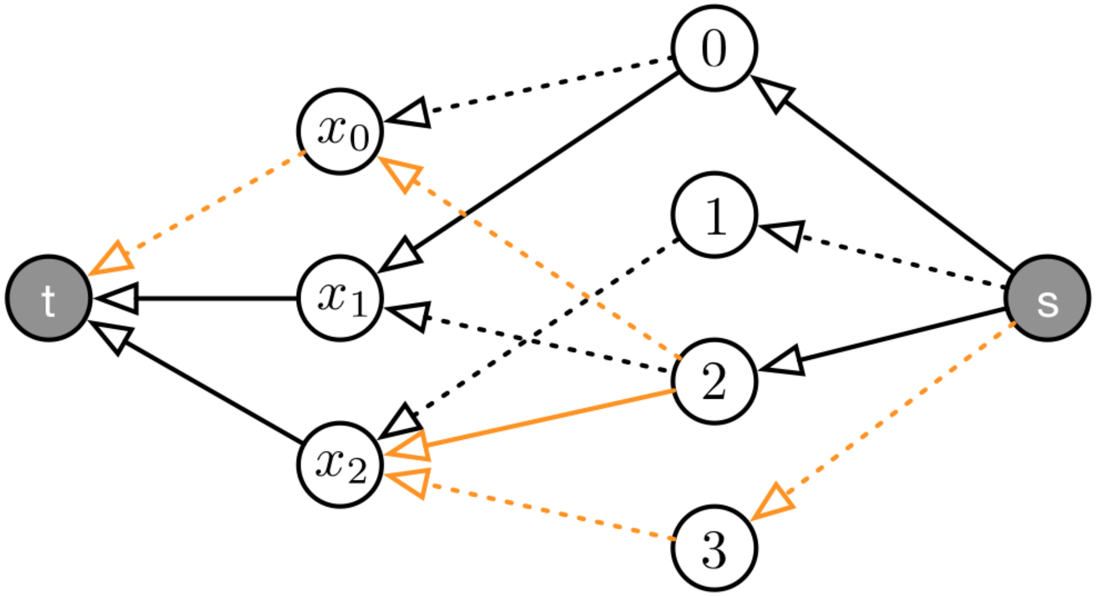
Residual Graph for ALLDIFFERENT



When we route flow along the path we need to:

- Increase flow on forward arcs (`f(a → b) = 0` in the orig. graph)
- Decrease flow on backward arcs (`f(b → a) = 1` in the orig. graph)

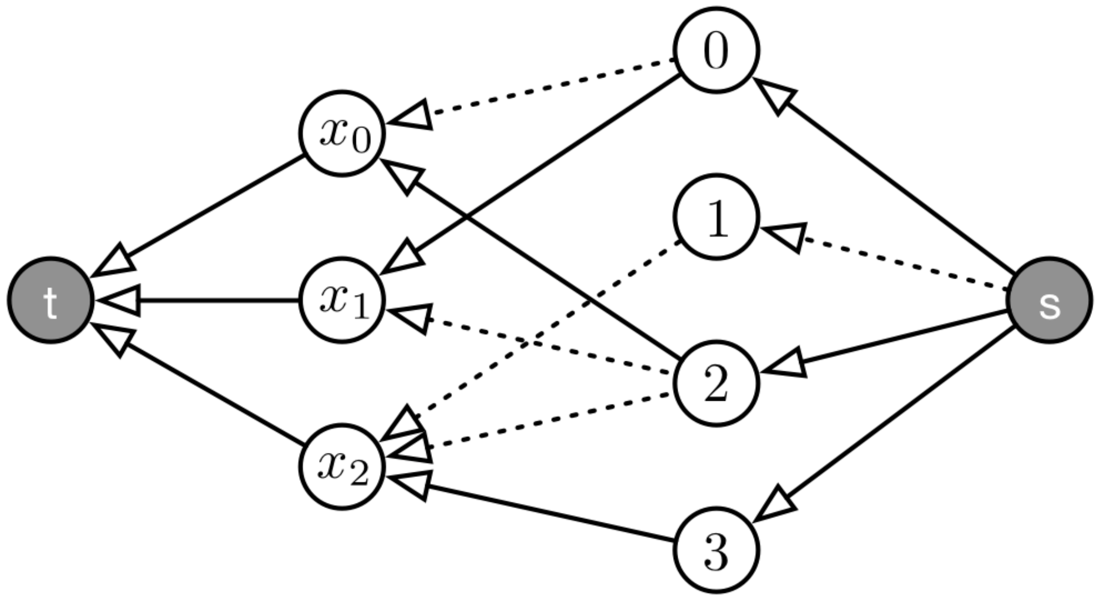
Residual Graph for ALLDIFFERENT



For example:

- This is our shortest path on the original graph

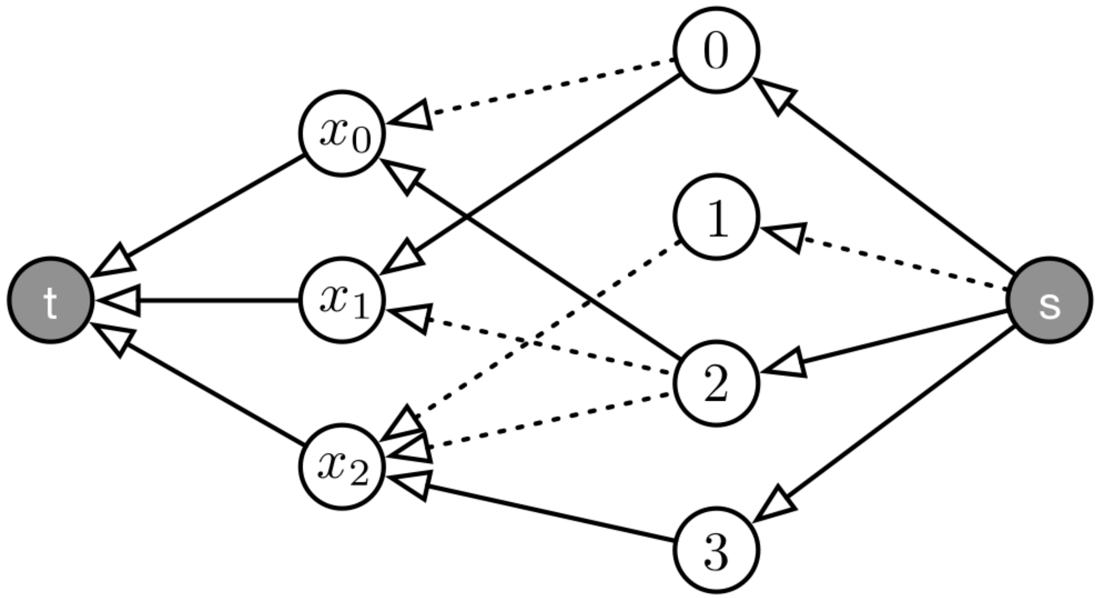
Residual Graph for **ALLDIFFERENT**



For example:

- This is our shortest path on the original graph
- And this is what we obtain after re-routing flow

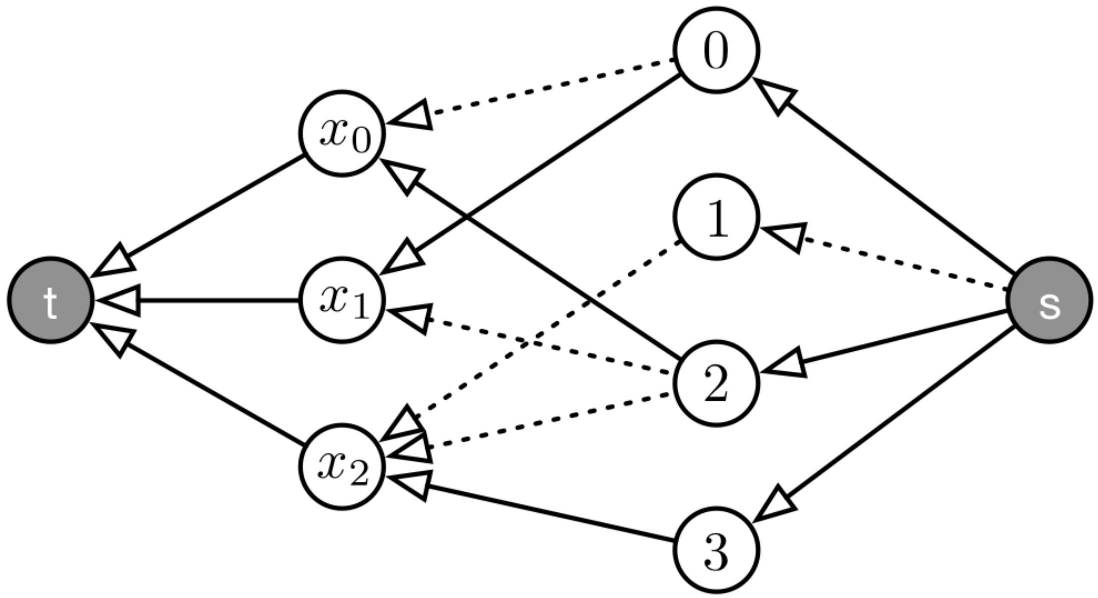
Residual Graph for `ALLDIFFERENT`



The total flow value is |`x`|, hence <u>the constraint is feasible</u>

- This flow corresponds to a feasible solution!
- I.e. $x_0 = 2$, $x_1 = 0$, $x_2 = 3$ (look at the solid arcs)

Residual Graph for ALLDIFFERENT



- It can be proved that the algorithm finds the maximum possible flow
- Complexity `O(#edges)` for finding paths via Dijkstra
- `#edges` = $O\left(\sum_{x_i \in X} |D(x_i)|\right)$

- At most $|X|$ iterations, hence total complexity

$$O\left(|X| \sum_{x_i \in X} |D(x_i)|\right)$$

Consistency Checking for `ALLDIFFERENT`

Ok, we have our consistency checker!

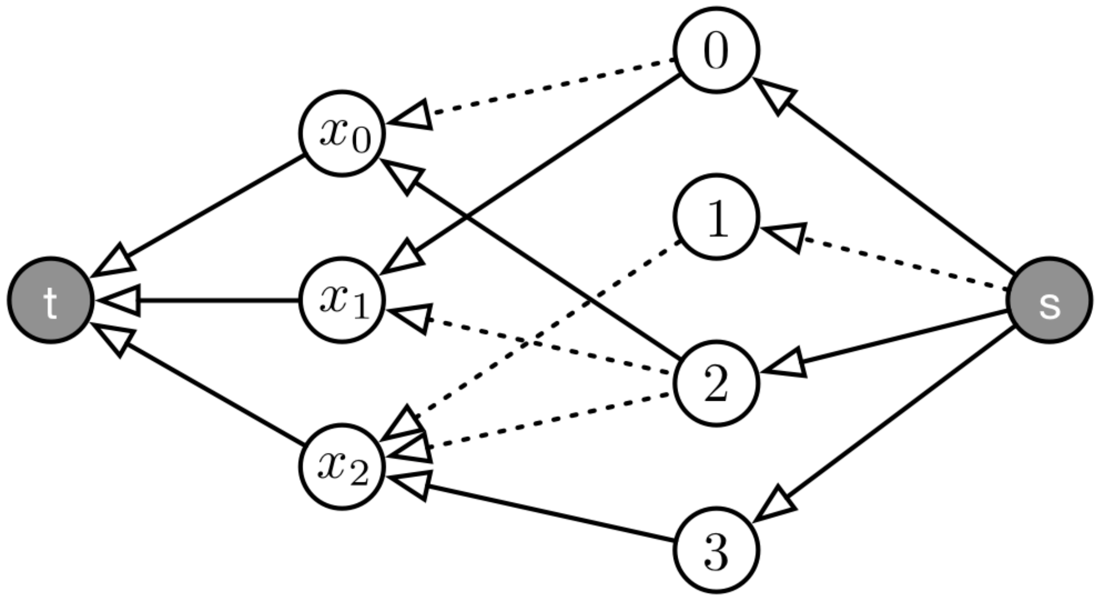
- We just need to build the flow graph
- Solve the max flow problem
- Check if the final flow value is $|\mathtt{X}|$

Side note:

- There is no need to actually build the residual graph
- We can work with the original graph by adjusting the formulas
- Showing residual graph is makes the presentation easier

And what about the filtering?

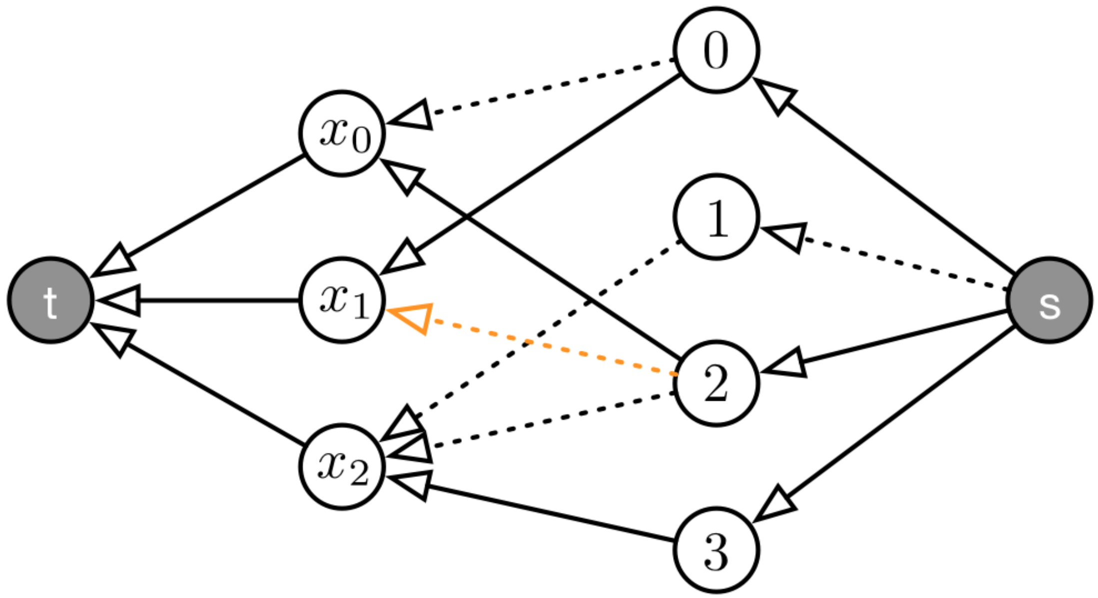
Filtering for ALLDIFFERENT



- Value-variable arcs = assignments
- Solid arcs in our final flow = feasible assignments
- Obviously, we cannot prune them

What about the value-variable arcs with no flow?

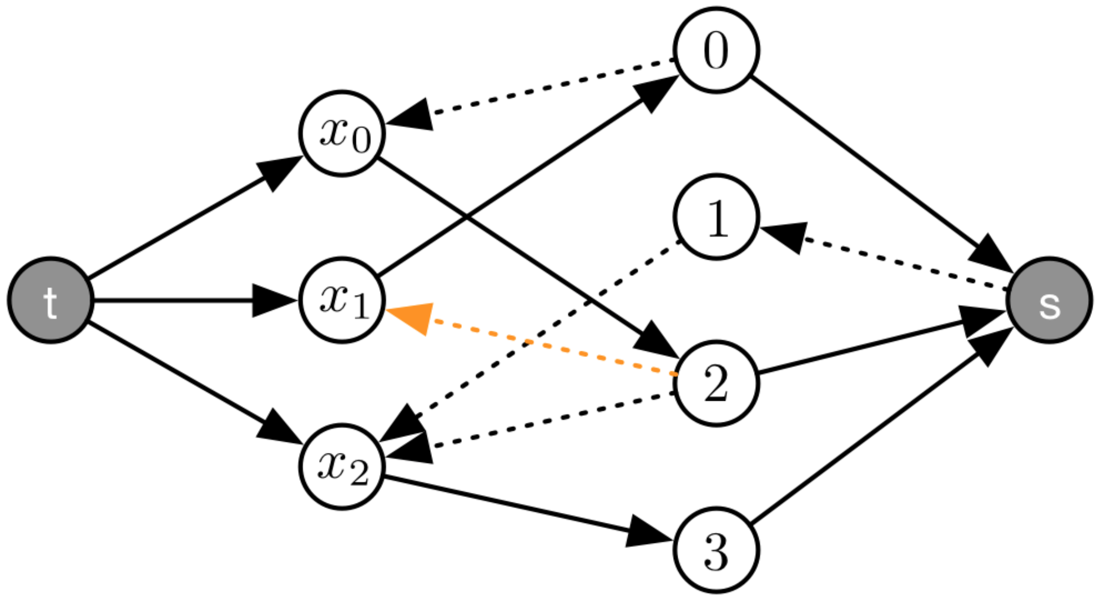
Filtering for ALLDIFFERENT



Consider for example arc 2 → x_1

- The corresponding value-variable value is feasible...
- ...iff we manage to route some flow through the arc

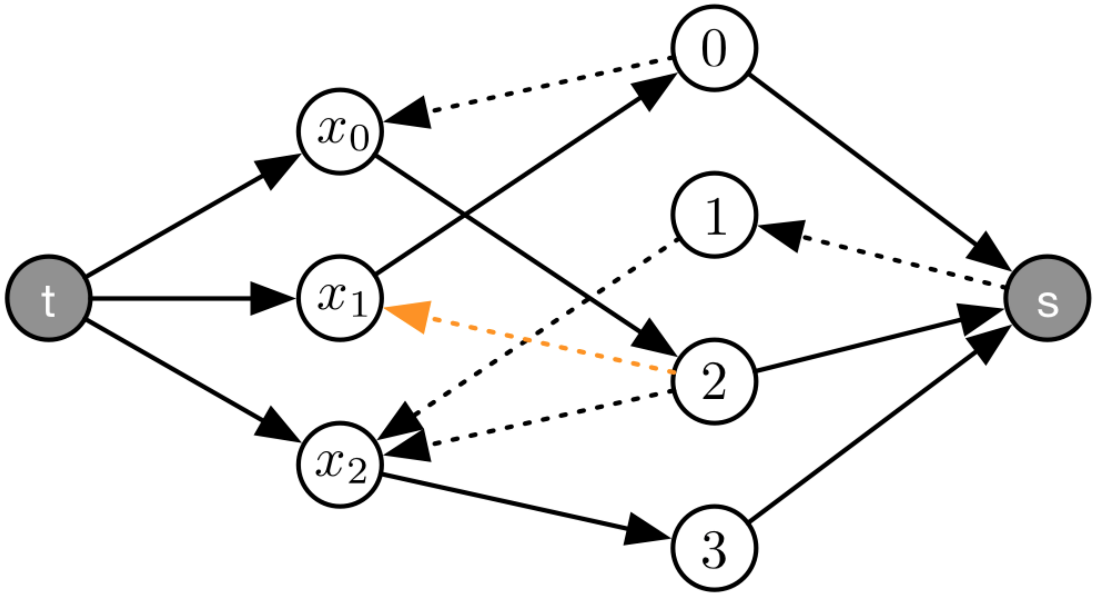
Filtering for ALLDIFFERENT



Flow routing takes place on the residual graph

- We need to route flow through 2 → x_1
- But <u>the total flow value must stay the same</u>

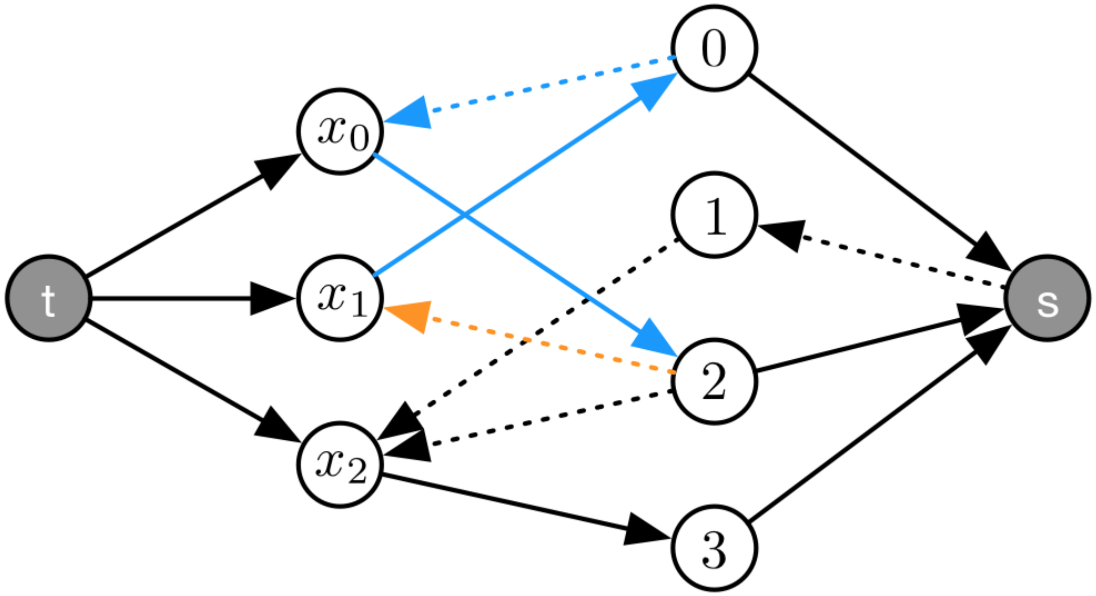
Filtering for ALLDIFFERENT



Hence, we are looking for a cycle on the residual graph!

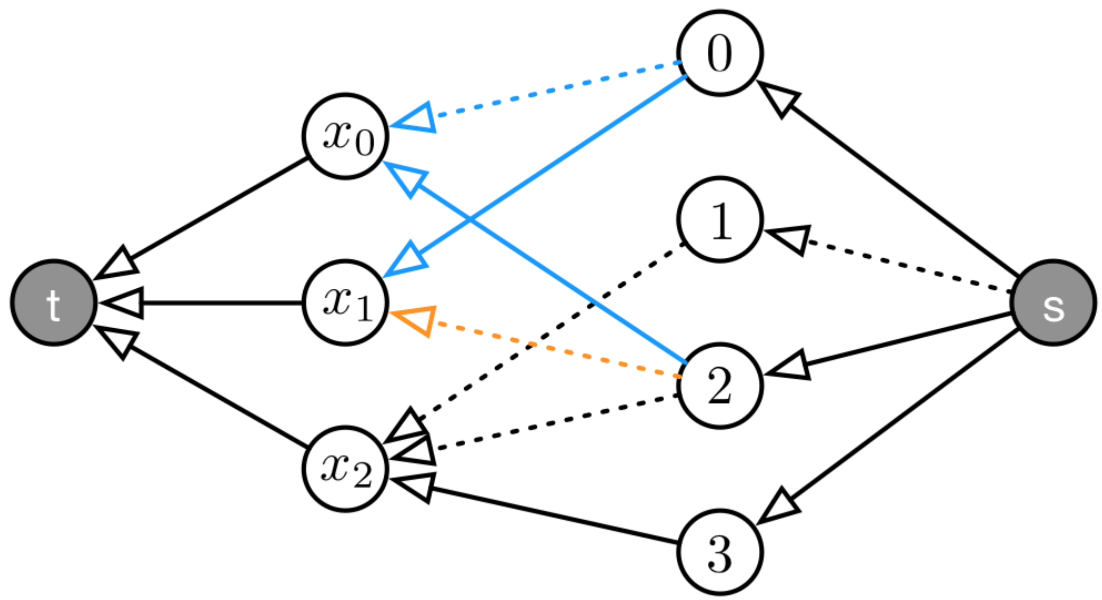
- The cycle should contain arc `2 → x1`
- Therefore, we just need to look for a path from `x1` to `2`

Filtering for ALLDIFFERENT



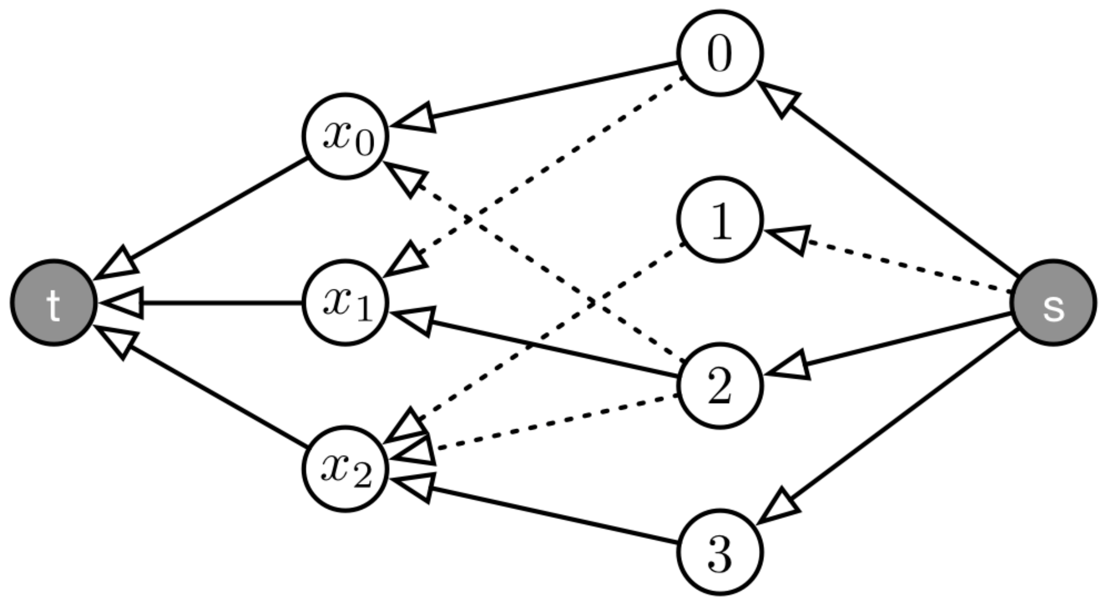
For example this one

Filtering for ALLDIFFERENT



This is how it looks on the original graph

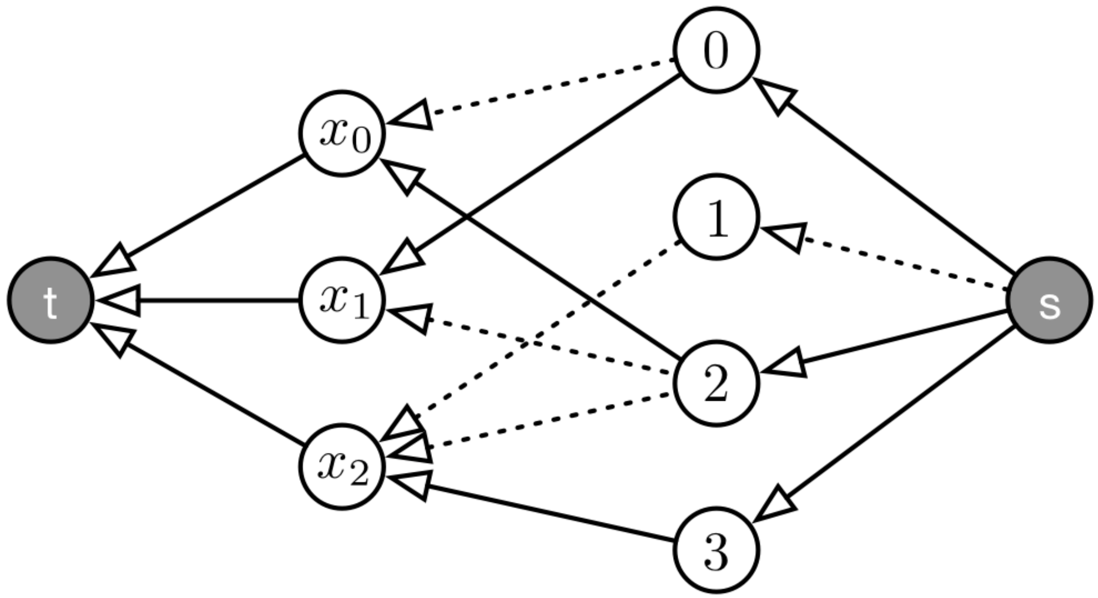
Filtering for ALLDIFFERENT



And this is what would happen by routing flow along the cycle

- We get another feasible flow with value equal to | x |
- Hence, we get another feasible solution

Filtering for ALLDIFFERENT

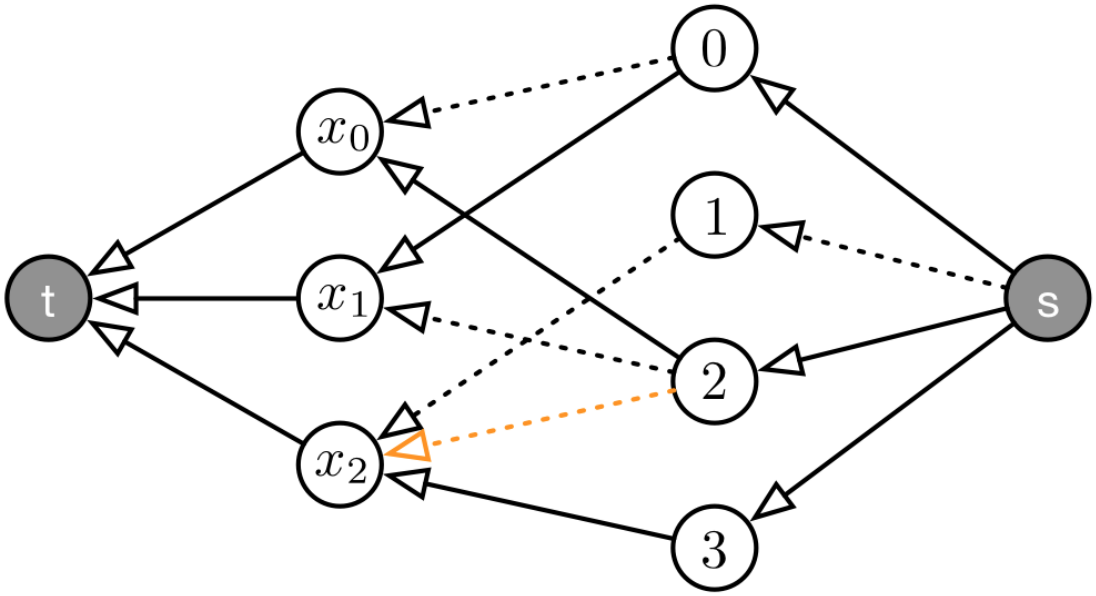


In practice, there is no need to route flow along the cycle

- It is sufficient to check if a cycle exists

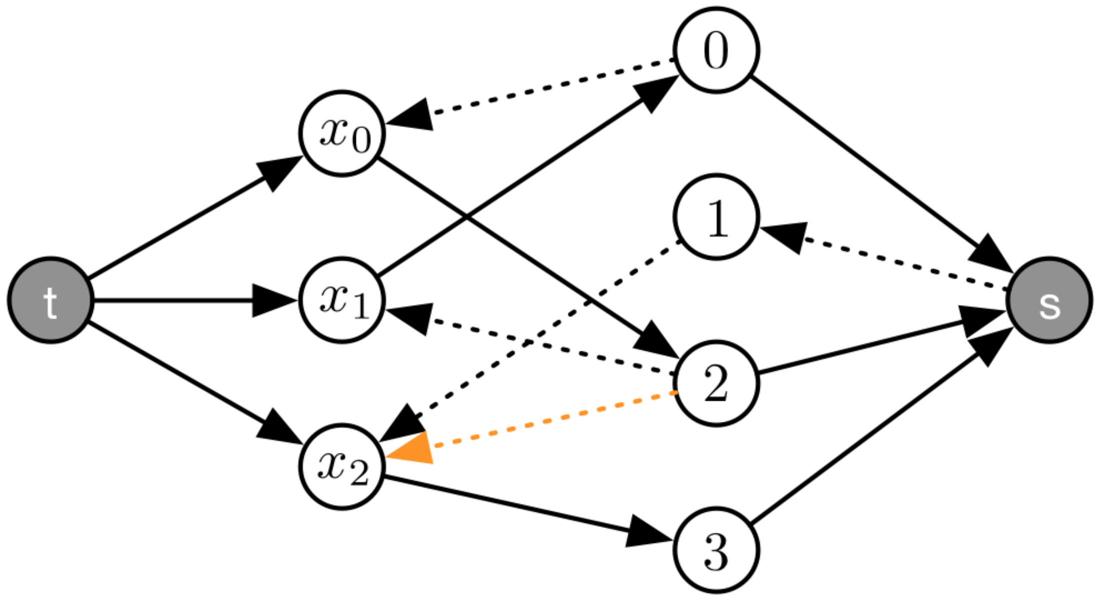
Let's check another value-variable pair...

Filtering for ALLDIFFERENT



For example, let us check arc $2 \rightarrow x_2$

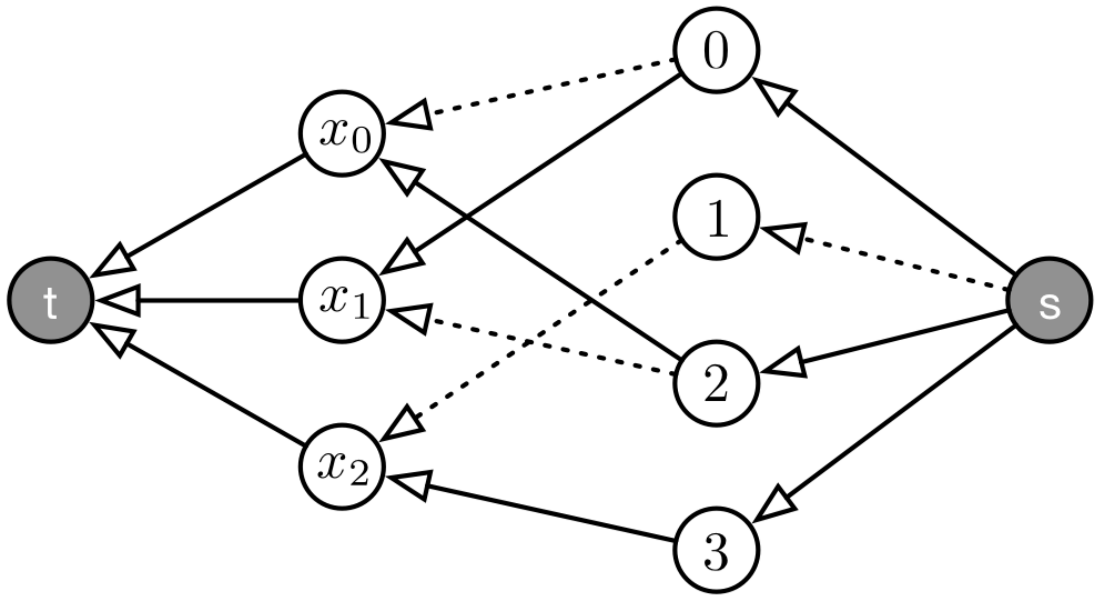
Filtering for ALLDIFFERENT



For example, let us check arc 2 → x_2

- We look for a cycle containing 2 → x_2 in the residual graph
- And none can be found

Filtering for ALLDIFFERENT



Therefore, there is no way we can route flow through 2 → x_2

- We can remove arc 2 → x_2 from the original graph
- And prune value 2 from the domain of x_2

Filtering for ALLDIFFERENT

We can prune a value v from the domain of x_i iff:

- We have $f(v \rightarrow x_i) = 0$
- and there is no cycle containing $v \rightarrow x_i$ in the residual graph

An important note:

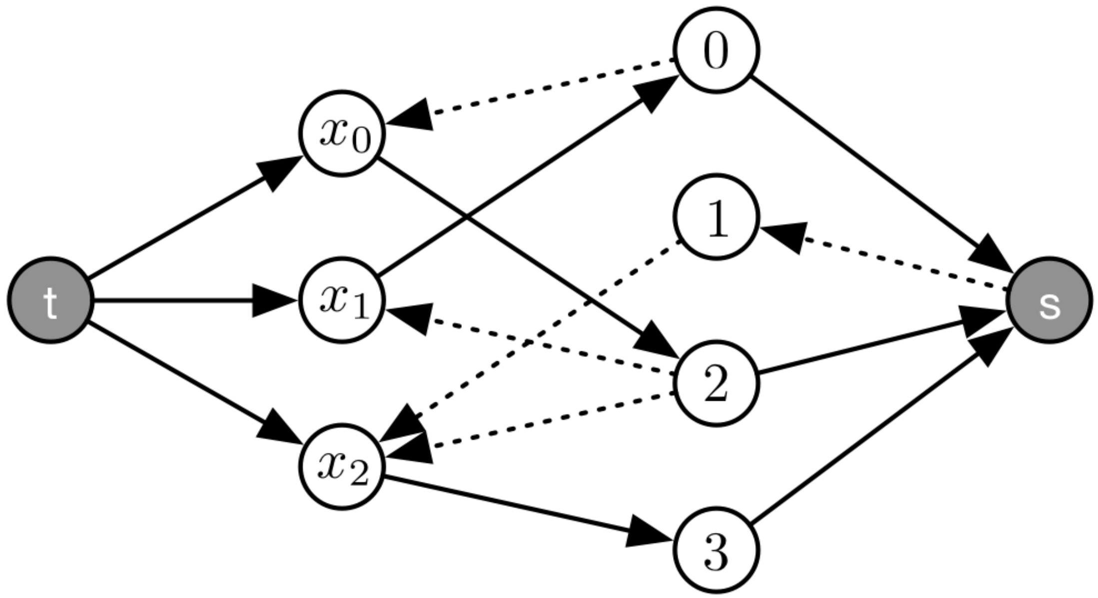
- The residual graph contains either $v \rightarrow x_i$ or $x_i \rightarrow v$
- Never both

Hence, we can simplify the second condition:

- "and there is no cycle containing v and x_i in the residual graph"

I.e. iff v and x_i are in different strongly connected components

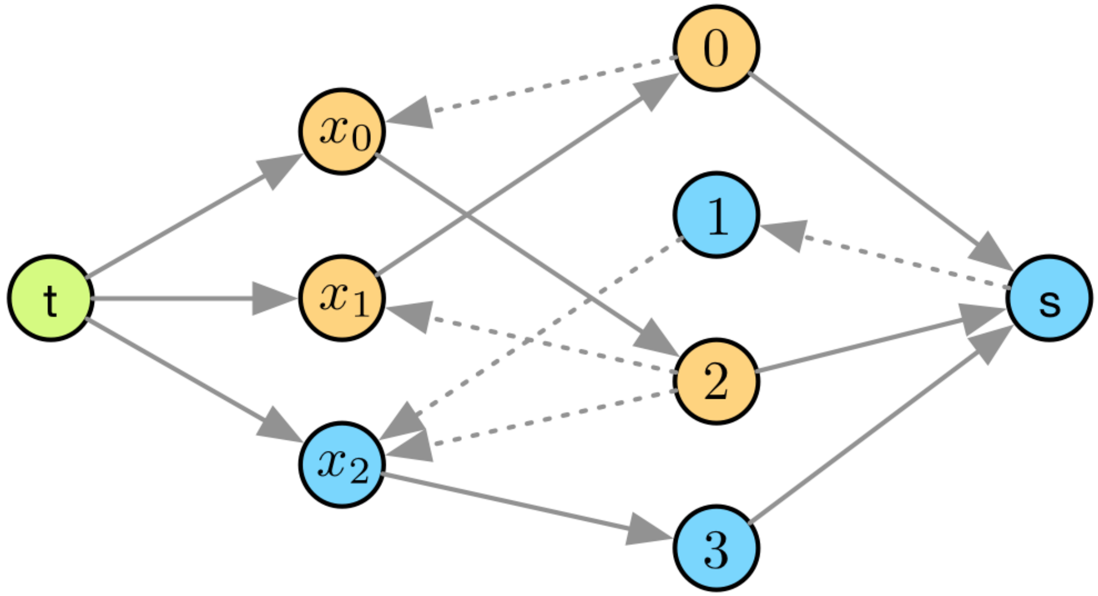
Filtering for **ALLDIFFERENT**



Strongly Connected Component:

- A set of nodes
- Each node can be reached from any other node of the component
- True iff all pairs of nodes appear in a cycle

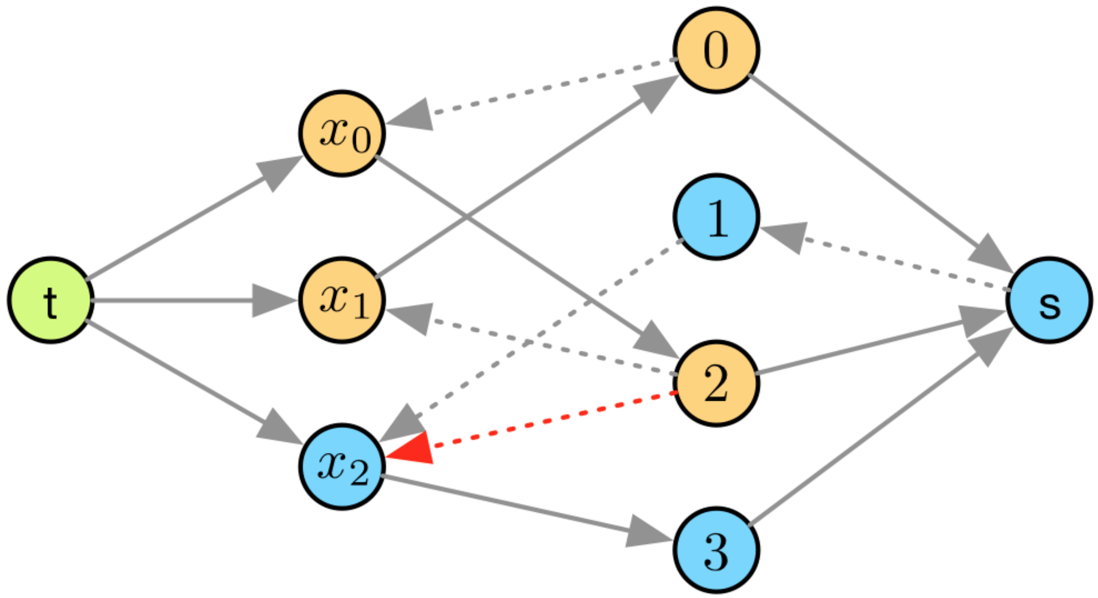
Filtering for ALLDIFFERENT



Here are the SCC of our residual graph (before filtering)

- They can be found efficiently with an algorithm by Tarjan
- The complexity is $O\left(|X| + \sum_{x_i \in X} |D(x_i)|\right)$

Filtering for ALLDIFFERENT



We can speed-up filtering by using SCC

- We can prune every arc `v → x_i` with `f(v → x_i) = 0`
- Provider that `v` and `x_i` are in different SCC

Global Constraints: a First Wrap-Up

`ALLDIFFERENT` is our first example of global constraint:

> **A global constraint is a constraint that corresponds to a set of constraints**

Global constraints are very important in CP

- They are more expressive (easier to state)
- They may allow for more powerful or more efficient propagation

Global constraints will be the focus of the next few blocks of slides

Constraint Systems

Global Constraints - GCC

A Simple Shift Scheduling Problem

A small shop makes use of shift work

- There are three types of shifts: full-day, half-day, night
- The work week consists of 4 days

Each employee should perform:

- A single night shift, at least a half-day shift, at most a full-day shift

Moreover:

- Night shifts cannot be performed on the first day
- Full-day shifts cannot be performed on the last day
- Half-day shifts can be performed only on the first and the last day

A Simple Shift Scheduling Problem

Let's try to model the shift assignment for a single employee:

- We identify shifts using numbers (full-day = 0, half-day=1, night=2)
- We use a variable for each day (i.e. $X = \{x_0, x_1, x_2, x_3\}$)

We can encode the allowed types of shift in the domains:

$$x_0 \in \{0, 1\}, x_1 \in \{0, 2\}, x_2 \in \{0, 2\}, x_3 \in \{1, 2\}$$

And the other restrictions?

- Not an `ALLDIFFERENT`
- We can model them with meta-constraints...

A Simple Shift Scheduling Problem

- *"Each employee should work a single night shift"*

$$\sum_{x_i \in X} (x_i = 2) = 1$$

- *"Each employee should work at least a half-day shift"*

$$\sum_{x_i \in X} (x_i = 1) \geq 1$$

- *"Each employee should work at most a full-day shift"*

$$\sum_{x_i \in X} (x_i = 0) \leq 1$$

A Simple Shift Scheduling Problem

This approach is correct, but has poor propagation

$$(x_0 = 2) + (x_1 = 2) + (x_2 = 2) + (x_3 = 2) = 1$$

$$(x_0 = 1) + (x_1 = 1) + (x_2 = 1) + (x_3 = 1) \geq 1$$

$$(x_0 = 0) + (x_1 = 0) + (x_2 = 0) + (x_3 = 0) \leq 1$$

$$x_0 \in \{0, 1\}, x_1 \in \{0, 2\}, x_2 \in \{0, 2\}, x_3 \in \{1, 2\}$$

A Simple Shift Scheduling Problem

This approach is correct, but has poor propagation

$$\{0\} + (x_1 = 2) + (x_2 = 2) + (x_3 = 2) = 1$$

$$(x_0 = 1) + \{0\} + \{0\} + (x_3 = 1) \geq 1$$

$$(x_0 = 0) + (x_1 = 0) + (x_2 = 0) + \{0\} \leq 1$$

$$x_0 \in \{0, 1\}, x_1 \in \{0, 2\}, x_2 \in \{0, 2\}, x_3 \in \{1, 2\}$$

- By filtering on the $(x_i = v)$ constraints we get this
- At this point, we are stuck

No filtering can be done based on the sums

A Simple Shift Scheduling Problem

This approach is correct, but has poor propagation

$$\{0\} + (x_1 = 2) + (x_2 = 2) + (x_3 = 2) = 1$$

$$(x_0 = 1) + \{0\} + \{0\} + (x_3 = 1) \geq 1$$

$$(x_0 = 0) + (x_1 = 0) + (x_2 = 0) + \{0\} \leq 1$$

$$x_0 \in \{0, 1\}, x_1 \in \{0, 2\}, x_2 \in \{0, 2\}, x_3 \in \{1, 2\}$$

By reasoning globally, however, we can deduce that:

- Values `0` and `2` cannot be assigned more than once
- Therefore value `1` must be assigned twice
- Hence $x_0 = 1$, $x_3 = 1$

We can try embed this reasoning inside a global constraint

The Global Cardinality Constraint

How shall we define it? In our example, we are interested in:

- Counting the occurrences (i.e. cardinality) of specific values
- Restricting the maximum/minimum cardinality

So, we could define a **Global Cardinality Constraint**:

`GCC(X, V, L, U)`, where:

- `X` is a vector of variables x_i
- `V` is a vector of values v_j
- `L` is a vector of cardinality lower bounds l_j for v_j
- `U` is a vector of cardinality upper bounds u_j for v_j

The Global Cardinality Constraint

The Global Cardinality Constraint is very important in practice:

- Restriction on cardinalities appear in many models
  - Shift-scheduling
  - Timetabling problems
  - Sport and tournament scheduling
  - Capacity constraints (for identical demands)
  - …
- Even `ALLDIFFERENT(X)` could be encoded as

`GCC(X, V, [0..0], [1..1])`

  - Although it is more efficient to use `ALLDIFFERENT` when possible

A Propagator for GCC

How do we perform filtering for GCC?

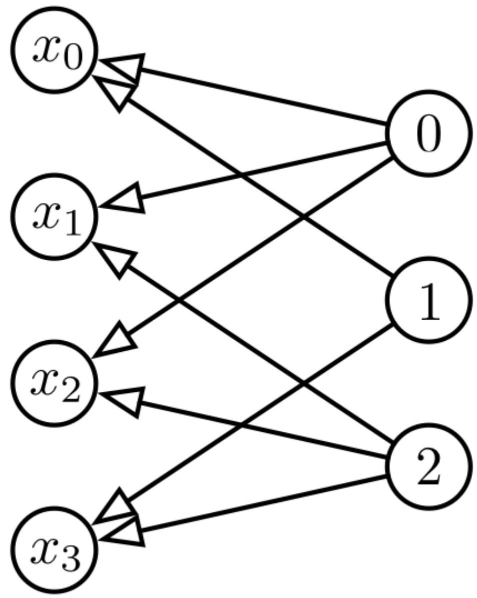
Main idea: exploit the similarity with the **ALLDIFFERENT**

- We will write a consistency checker based on network flows
  - Same flow interpretation as in the **ALLDIFFERENT**
- And then we will define flow-based filtering rules

Actually, our **ALLDIFFERENT** propagator was originally designed for **GCC** !

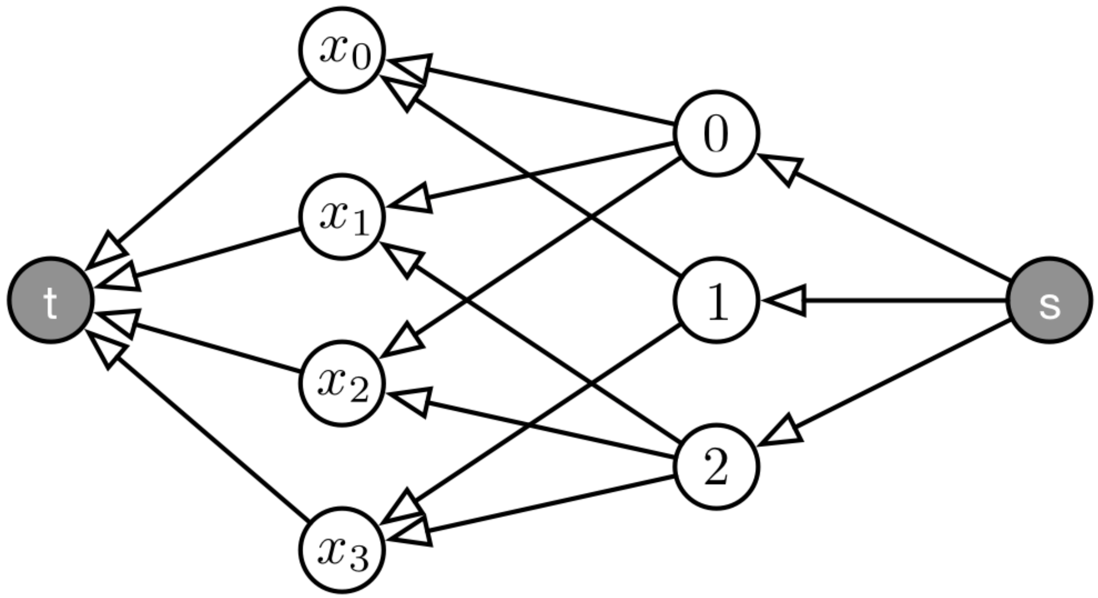
Once again we start from the value graph...

A Propagator for GCC: Consistency Checking



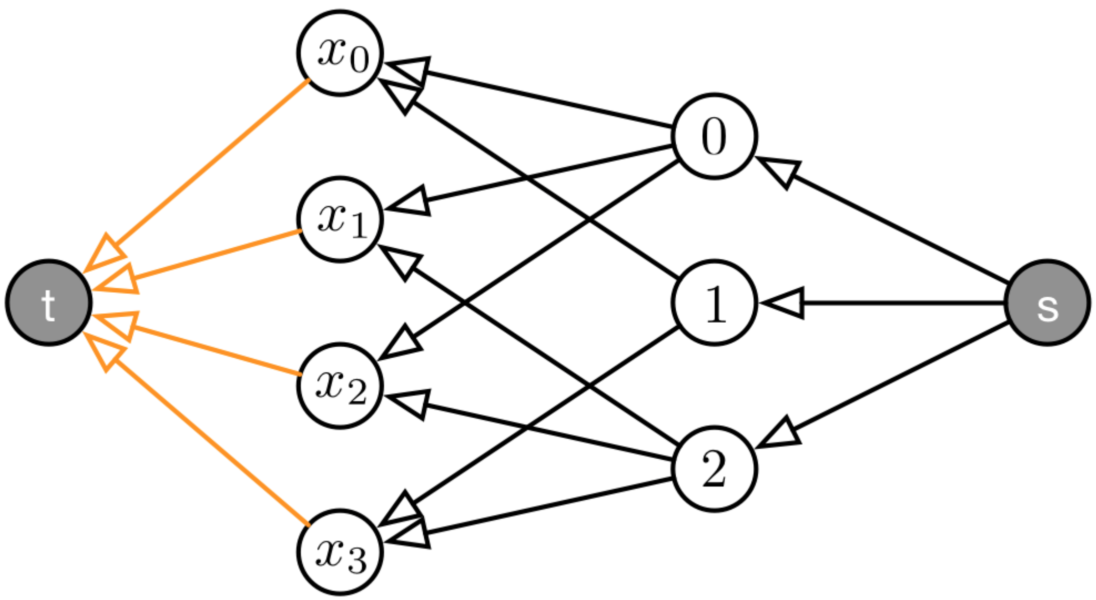
- This is the value graph for our example GCC instance

A Propagator for GCC: Consistency Checking



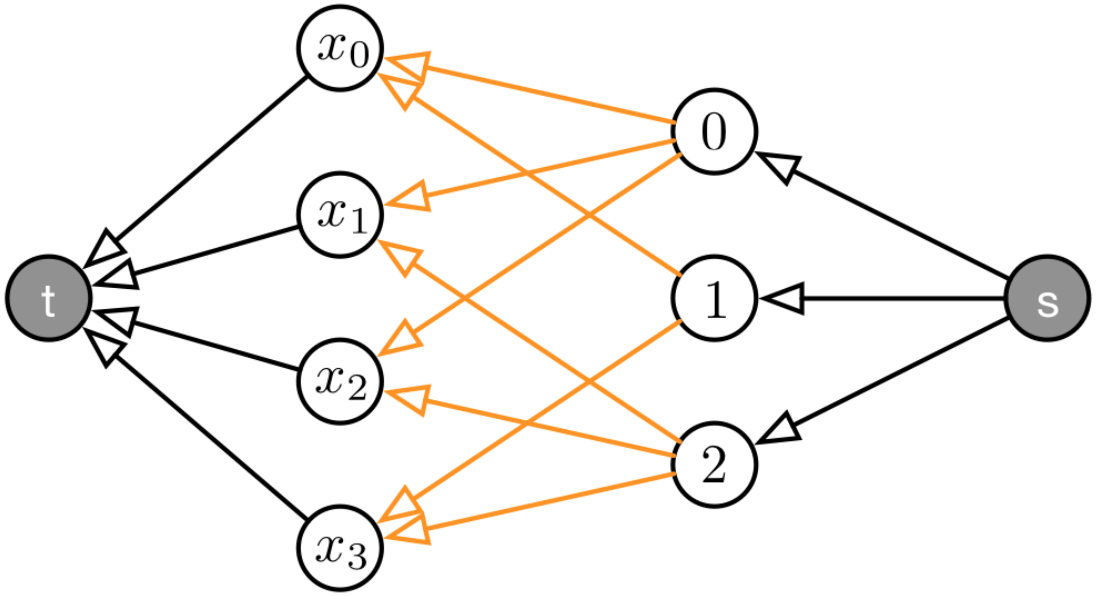
- This is the value graph for our example `GCC` instance
- We add a source `s` and sink `t` node, as for `ALLDIFFERENT`

A Propagator for GCC: Consistency Checking



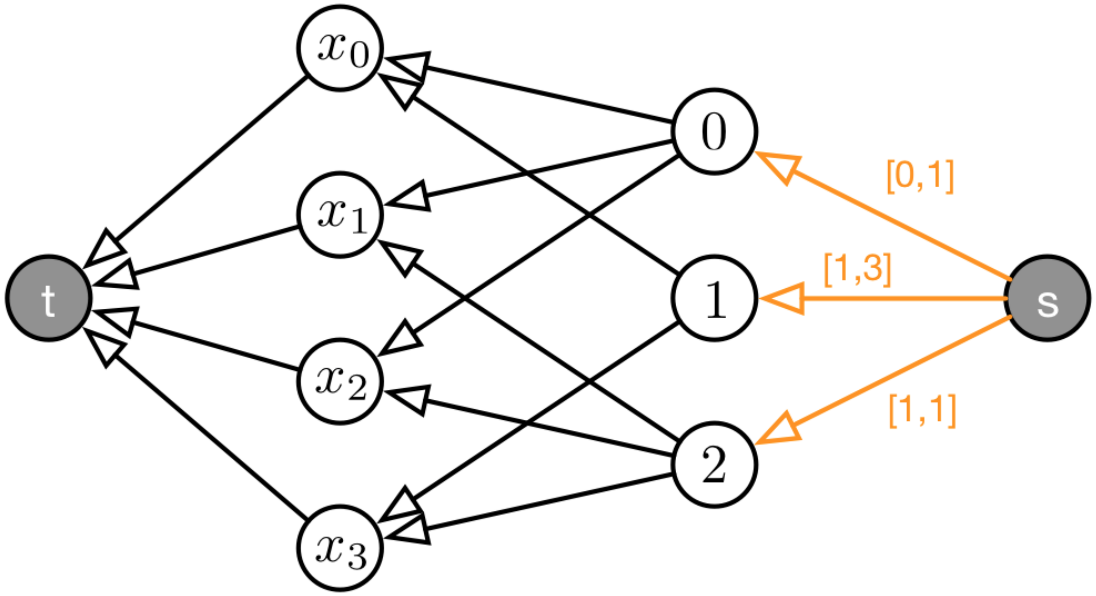
- These arcs have capacity `1`, as for the `ALLDIFFERENT`
  - I.e. each variable cannot be assigned more than once

A Propagator for GCC: Consistency Checking



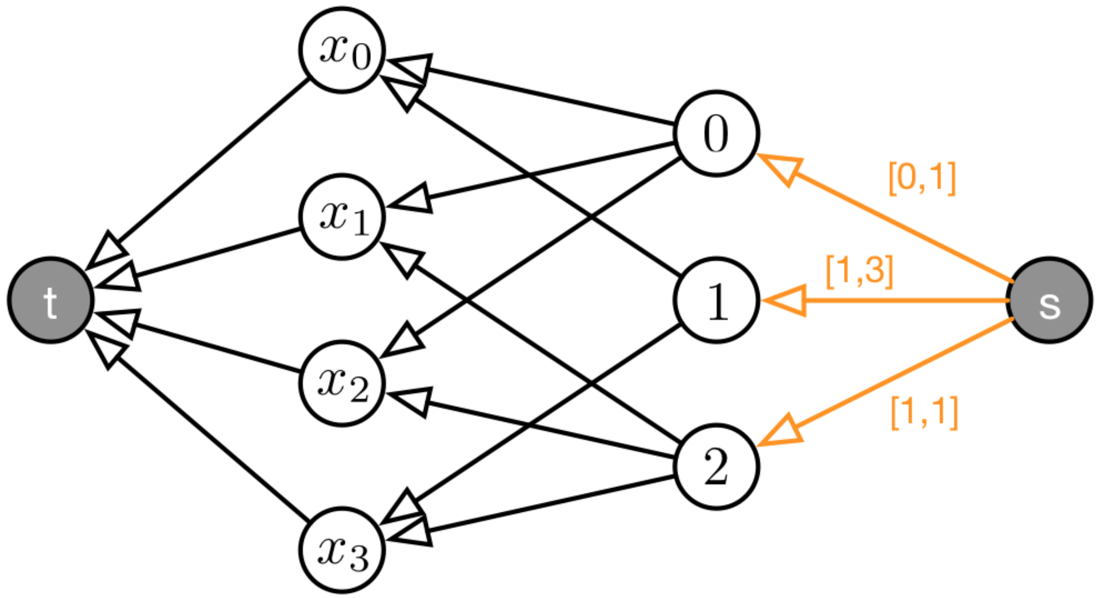
- These arcs have capacity `1`, as for the **`ALLDIFFERENT`**
  - I.e. each variable cannot be assigned more than once
- There arcs have capacity `1`, as for the **`ALLDIFFERENT`**
  - I.e. each value cannot be assigned twice to the same variable

A Propagator for GCC: Consistency Checking



- But these arcs have a capacity equal to $\mathtt{U_i}$…
  - I.e. they cannot be used more than $\mathtt{U_i}$ times
- …and they have a demand equal to $\mathtt{L_i}$
  - I.e. they must be used at least $\mathtt{L_i}$ times

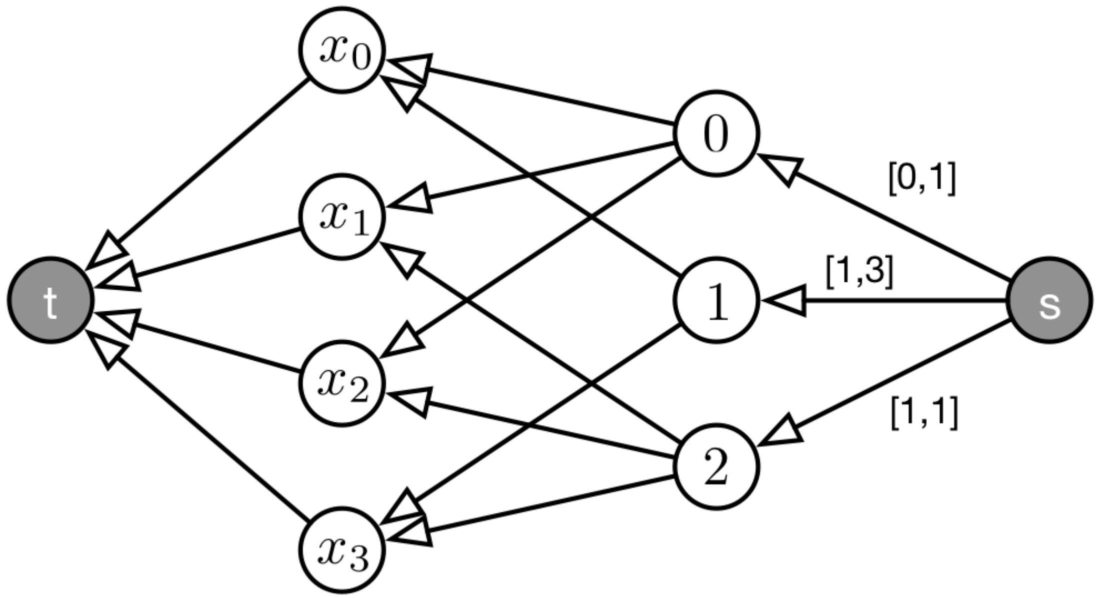
A Propagator for GCC: Consistency Checking



Notation:

- Label `[Li,Ui]` to show the demand and capacity
- Demand = 0 and capacity = 1 for unlabeled arcs

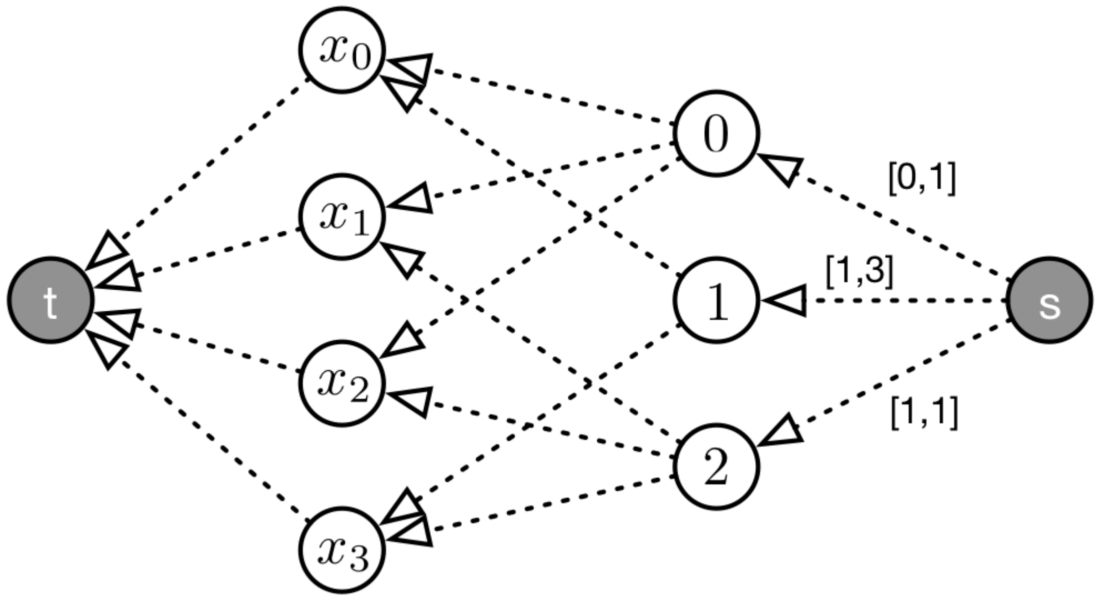
A Propagator for GCC: Consistency Checking



The constraint is feasible iff we can find a flow such that:

- There is flow on all $x_i \rightarrow t$ arcs (i.e the max flow value is |X|)
- The capacity <u>and demand</u> constraints on all arcs are satisfied

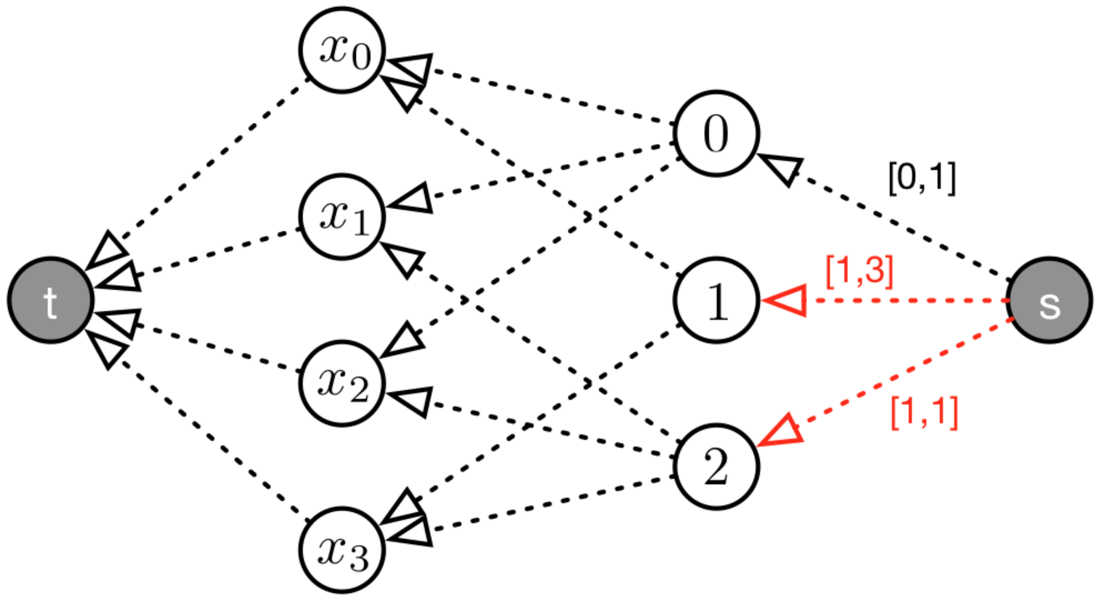
A Propagator for GCC: Consistency Checking



Like in the `ALLDIFFERENT` case we start from a zero flow

- However, with the `GCC` <u>the zero flow may be infeasible</u>

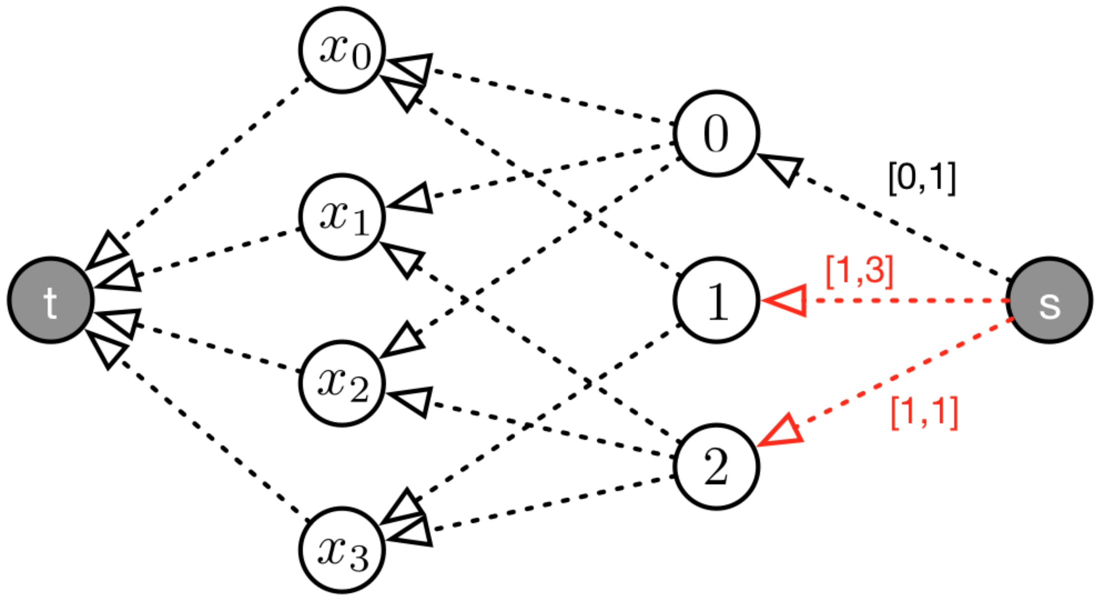
A Propagator for GCC: Consistency Checking



- In our example, the demand on **s → 1** and **s → 2** are not satisfied

Hence, we need to fix the flow before starting to maximize

A Propagator for GCC: Consistency Checking



Main idea for fixing the flow:

- Treat the value nodes (i.e. `0, 1, 2`) as source nodes
- Route `l`$_i$ flow units from these nodes to the sink

Residual Graph for the GCC

Routing flow is done on the residual graph

The residual graph for the GCC flow network:

- Contains a node for each node in the original graph
- Contains an arc `a → b` in two cases

Case 1 (forward arcs):

- The original graph contains an arc `a → b` with capacity `c`
- We have that `c − f(a → b) > 0`

Intuitively: it possible to route more flow along `a → b`

- The residual graph arc has capacity `c − f(a → b)`

Residual Graph for the GCC

Routing flow is done on the residual graph

The residual graph for the GCC flow network:

- Contains a node for each node in the original graph
- Contains an arc `a → b` in two cases

Case 2 (backward arcs):

- The original graph contains an arc `b → a` with demand `d`
- We have that `f(b → a) − d > 0`

Intuitively: it is possible to reduce the flow along `b → a`

- The residual graph arc has capacity `f(b → a) − d`

Residual Graph for the GCC

Routing flow is done on the residual graph

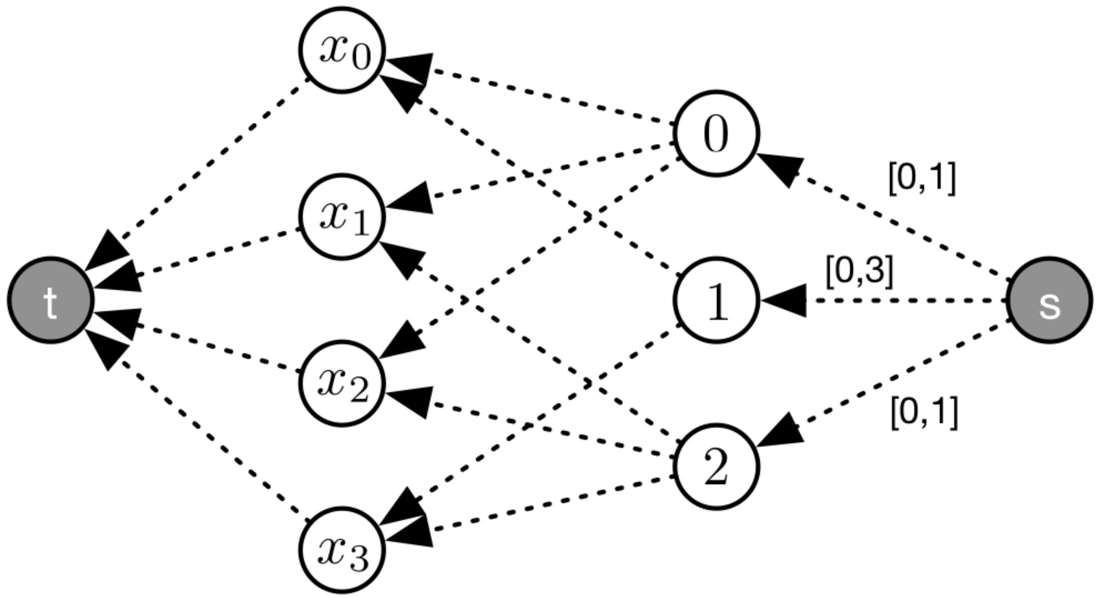
The residual graph for the GCC flow network:

- Contains a node for each node in the original graph
- Contains an arc `a → b` in two cases (see previous slides)

Side effect: there can be no arc with demand in the residual graph

- This is the general definition of residual graph for flow problems
- The `ALLDIFFERENT` was a special case

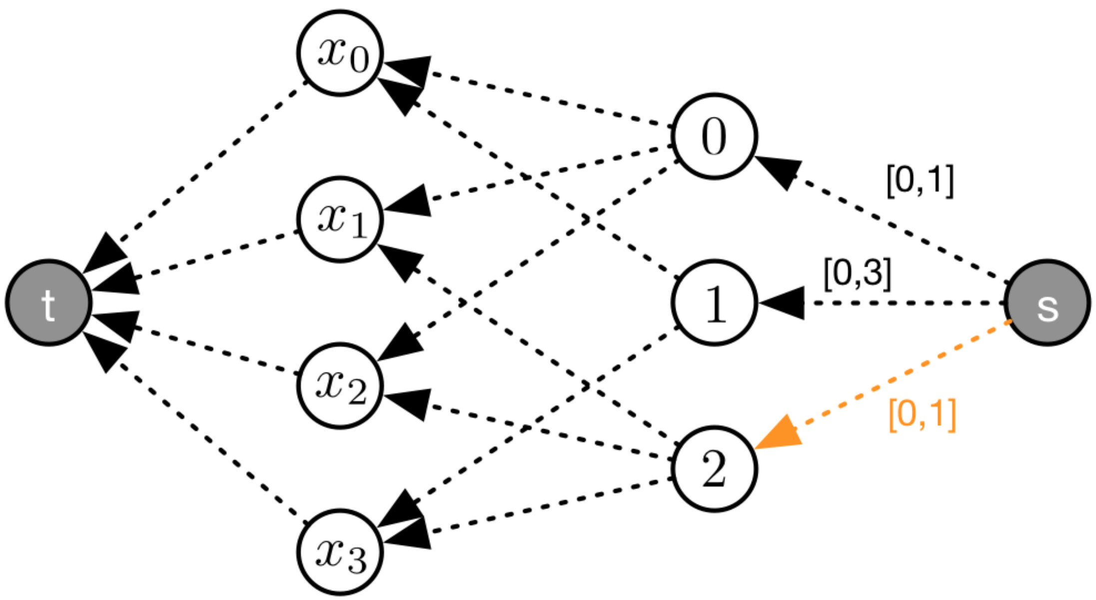
Fixing the Initial Flow for the GCC



This the residual graph for the zero flow

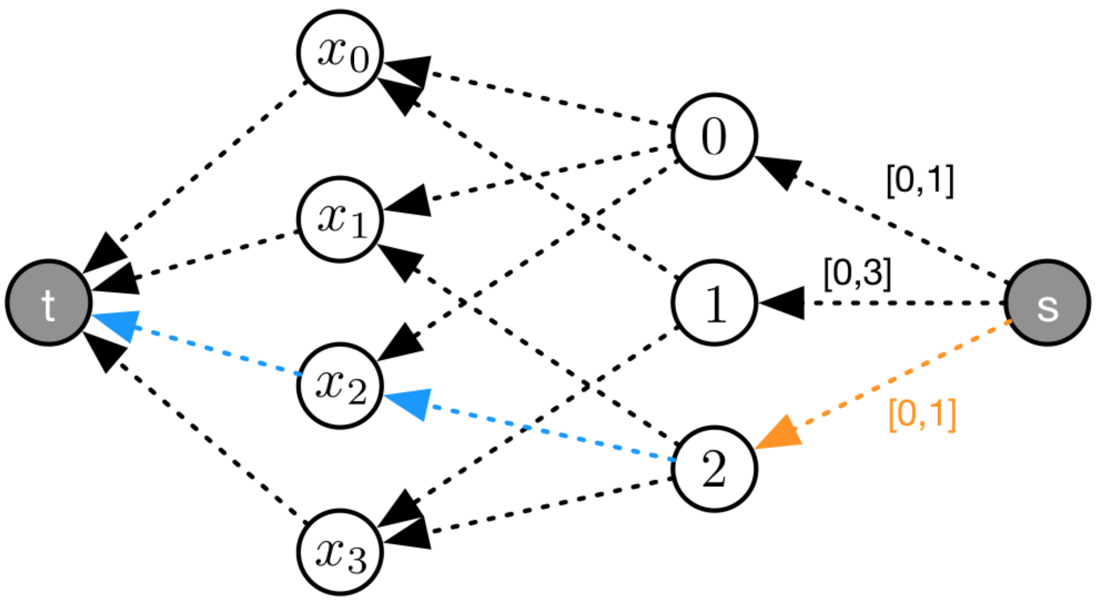
- There are only forward arcs at this stage

Fixing the Initial Flow for the GCC



The demand on arc **s → 2** in the original graph is 1

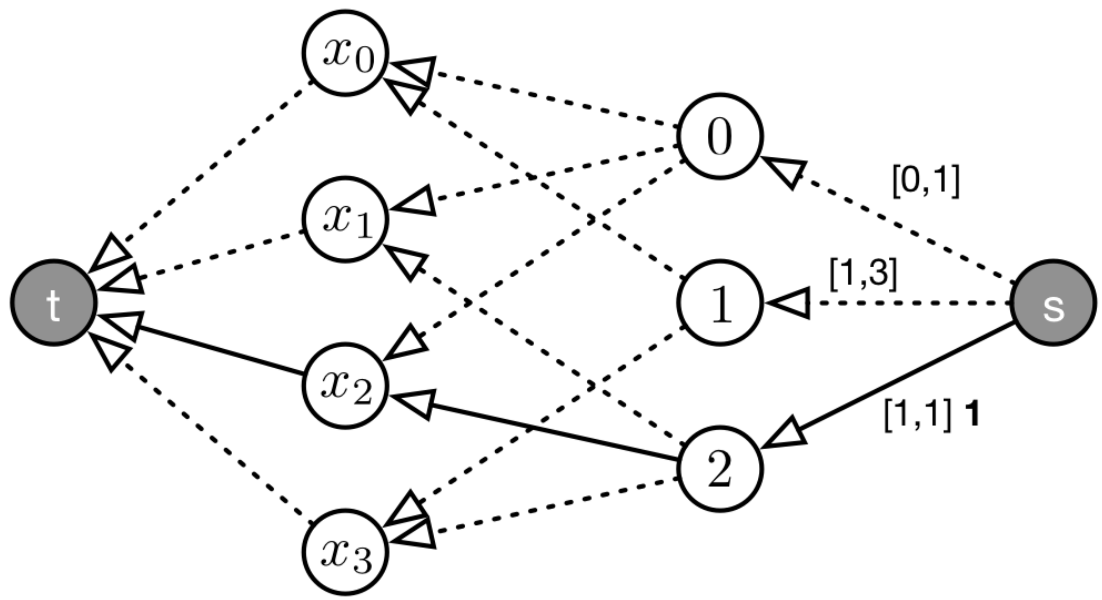
Fixing the Initial Flow for the GCC



The demand on arc `s → 2` in the original graph is 1

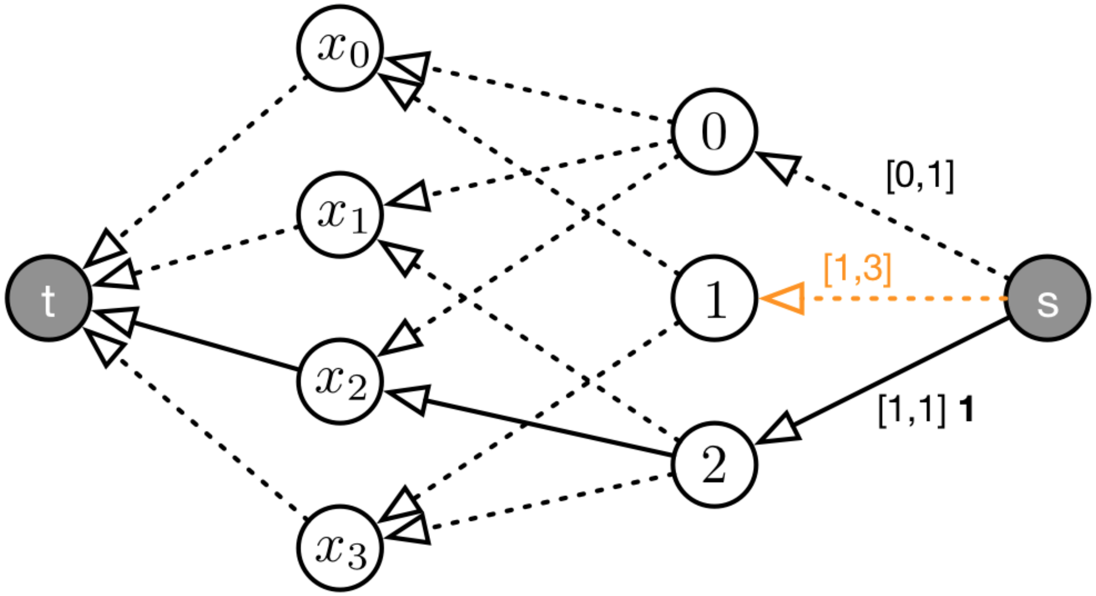
- We search for a shortest path from `2` to `t` (e.g. Dijkstra's algorithm)
- Flow to route = min capacity of all arcs on the path
- For the GCC graph, this value is always 1

Fixing the Initial Flow for the GCC



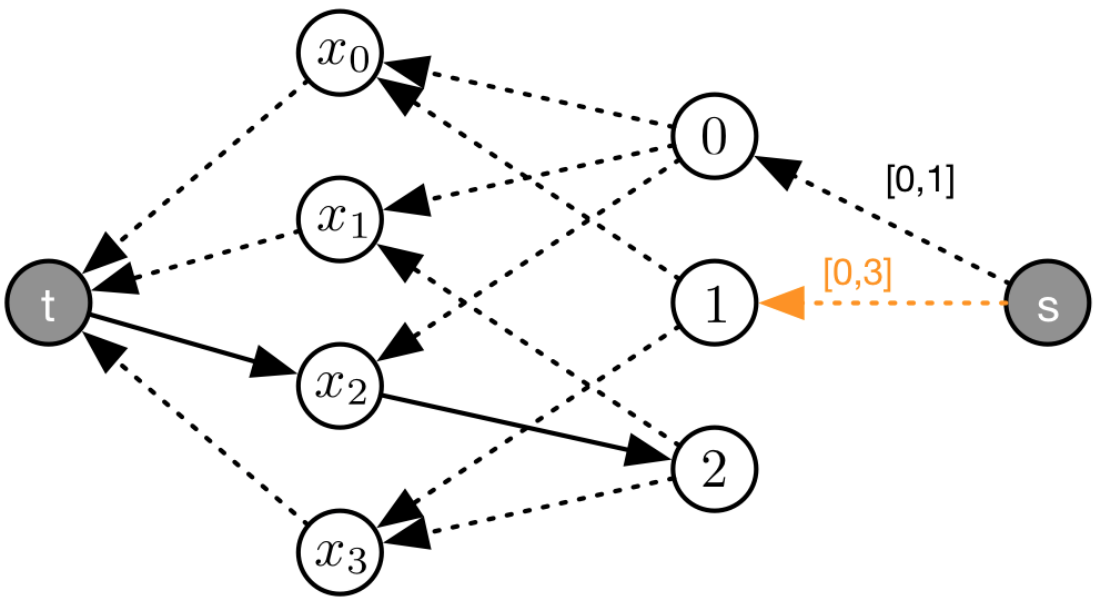
This is the resulting flow status on the original graph

Fixing the Initial Flow for the GCC



Next, we try to satisfy the demand on arc **s → 1**

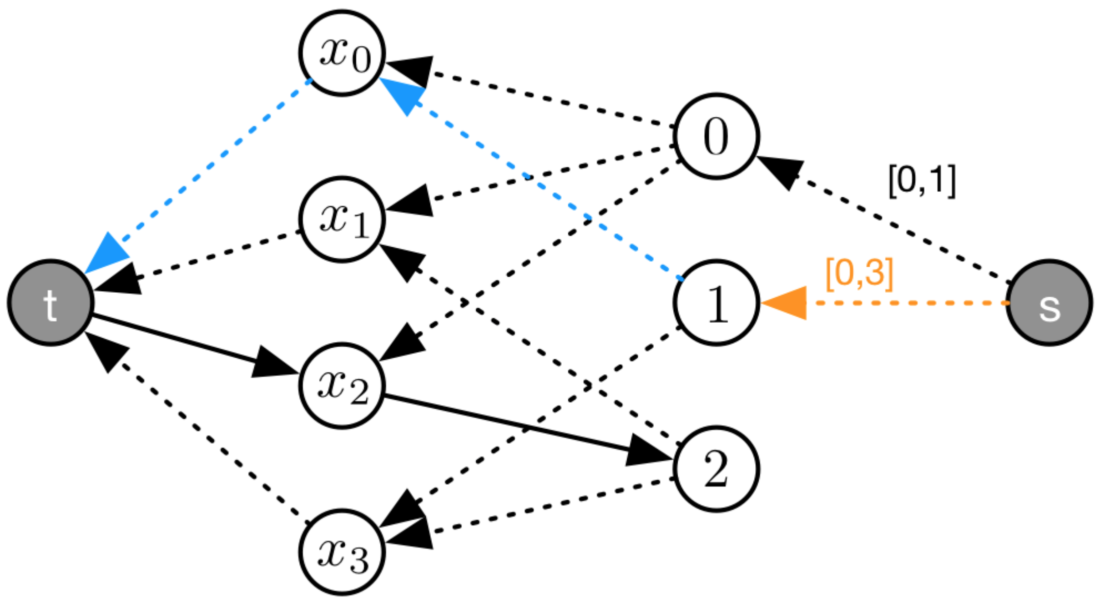
Fixing the Initial Flow for the GCC



Next, we try to satisfy the demand on arc s → 1

- The residual graph contains some backward arcs
- As expected no arc has a demand value

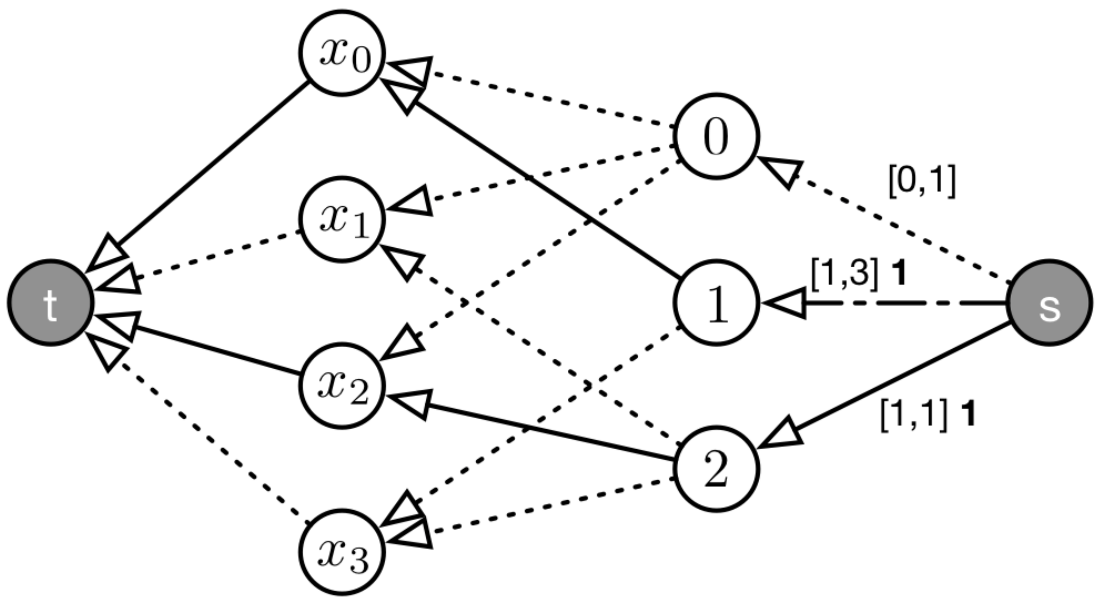
Fixing the Initial Flow for the GCC



Next, we try to satisfy the demand on arc `s → 1`

- We search for a path from `1` to `t`
- We route 1 unit of flow

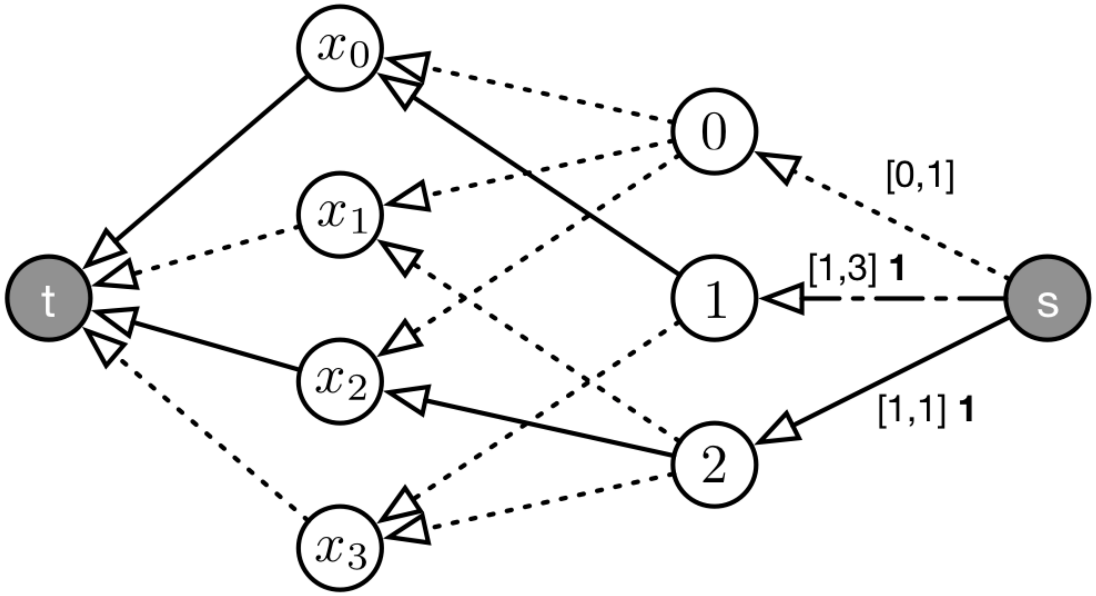
Fixing the Initial Flow for the GCC



And we obtain the following flow on the original graph

- Dotted arcs = zero flow
- Solid arcs = saturated arcs
- Dash-dotted arcs = non-saturated arcs with non-zero flow

Fixing the Initial Flow for the GCC

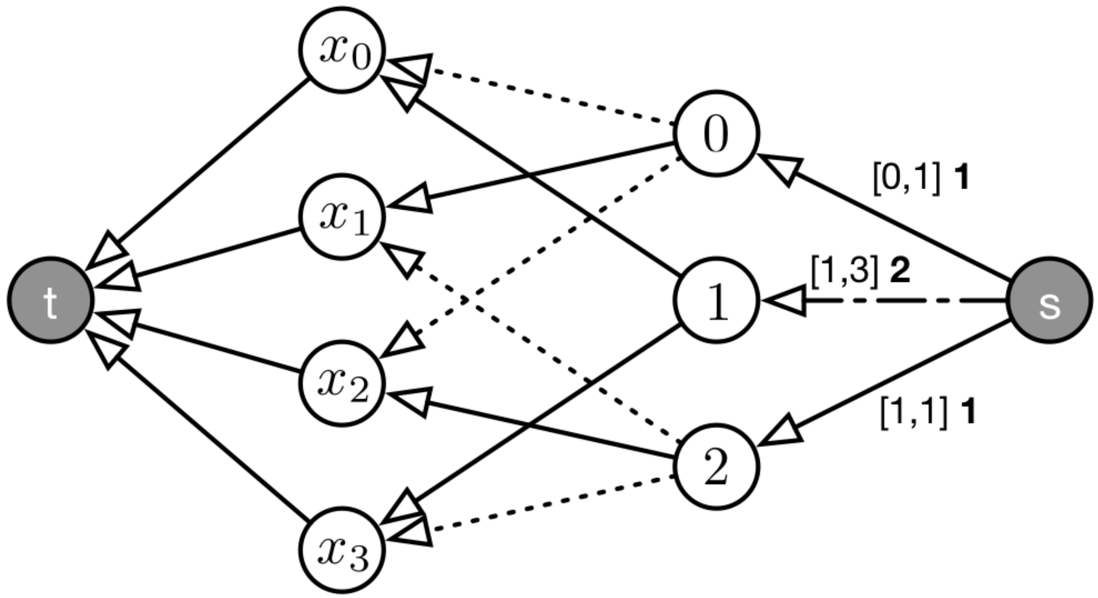


The flow is feasible

- If the demands cannot be satisfied, then no feasible solution exists

We can now maximize the flow by routing flow on `s – t` paths

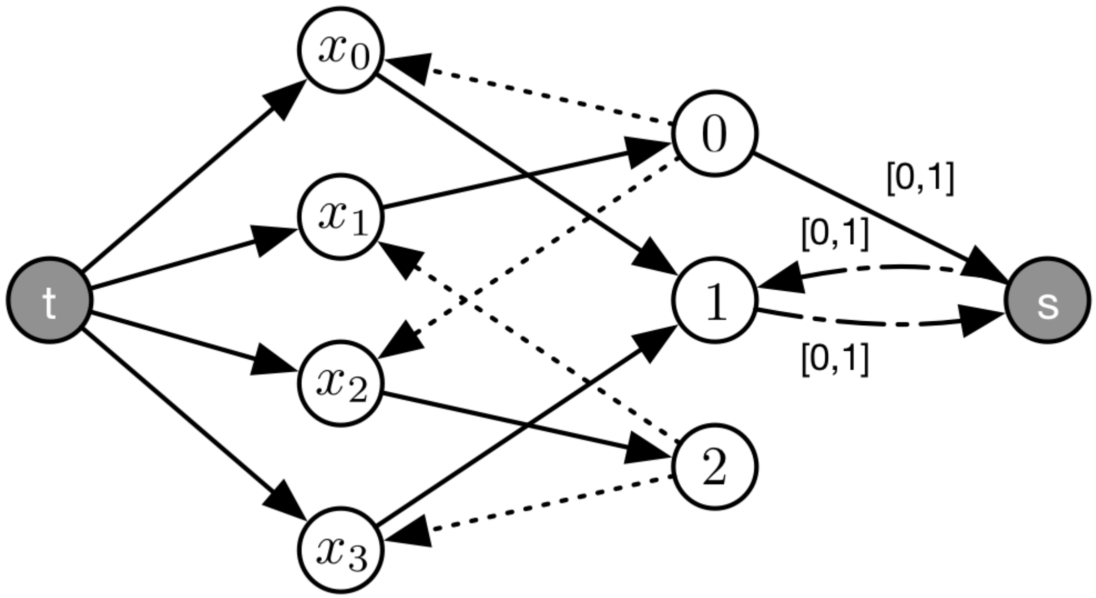
A Propagator for GCC: Consistency Checking



This is the flow at the end of the maximization process

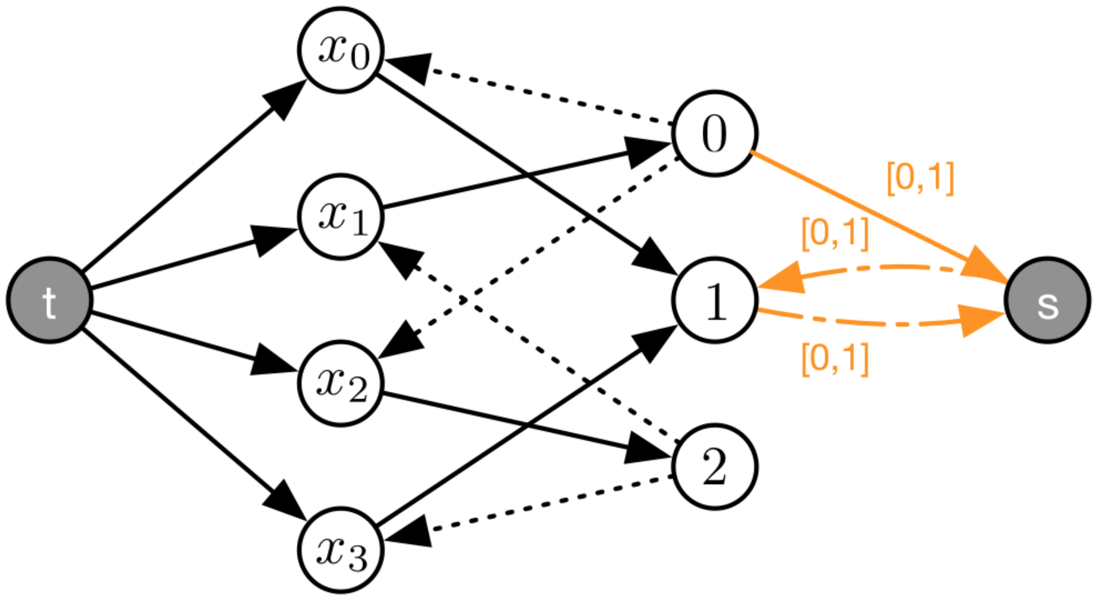
- The corresponding solution is $x_0 = 1$, $x_1 = 0$, $x_2 = 2$, $x_3 = 1$
- If the max flow value is lower than $|x|$, the constraint is infeasible

A Propagator for GCC: Filtering



Filtering can be performed by reasoning on the residual graph

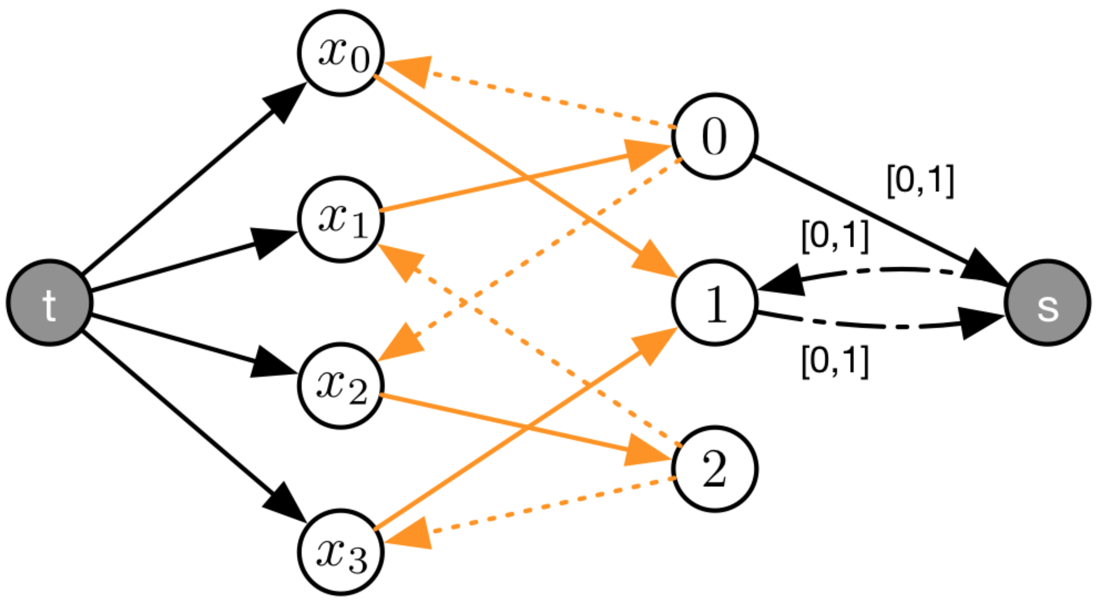
A Propagator for GCC: Filtering



Filtering can be performed by reasoning on the residual graph

- Notice the forward and backward arcs between `1` and `s`

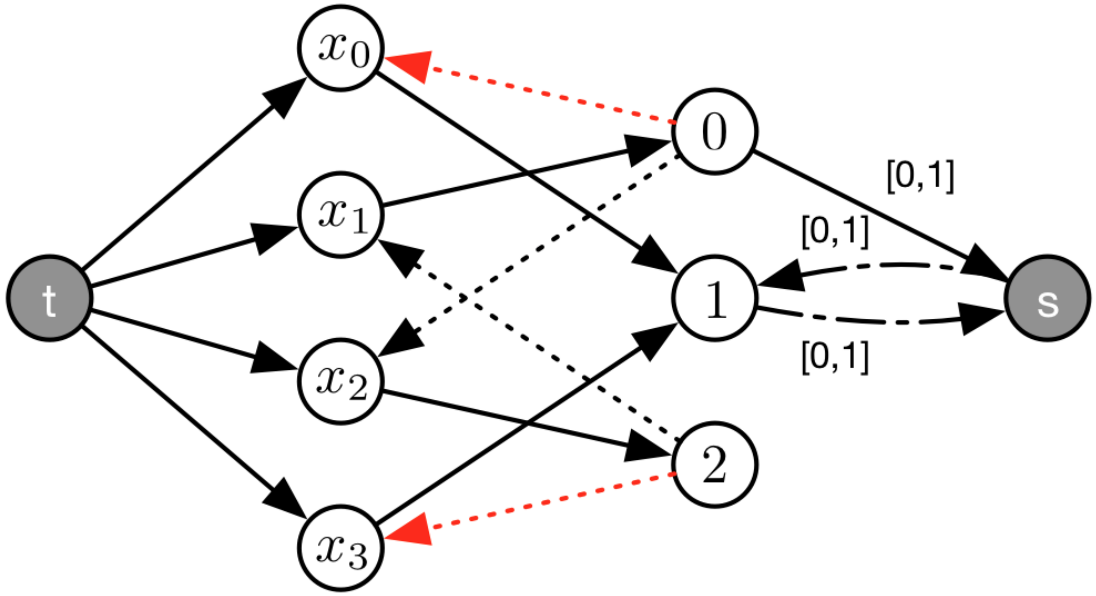
A Propagator for GCC: Filtering



Filtering can be performed by reasoning on the residual graph

- The value-variable arcs are identical to those in the `ALLDIFFERENT`
- Hence, we can filter based on (lack of) cycles
- And use strongly connected components to speed up the process

A Propagator for GCC: Filtering



In our case, no cycle including `0` → x_0 and `2` → x_3 exists

- Hence, we can prune value `0` from `D(x`$_0$`)`, and `2` from `D(x`$_3$`)`

The `DISTRIBUTE` Constraint

In solvers the `GCC` is sometimes called `DISTRIBUTE`

Actually, `DISTRIBUTE` is a more general constraint, with signature:

`DISTRIBUTE(X, V, N)`

- `X` is a vector of variables x_i
- `V` is a vector of values v_j
- `N` is a vector of cardinality variables n_j

The DISTRIBUTE Constraint

In solvers the GCC is sometimes called DISTRIBUTE

Actually, `DISTRIBUTE` is a more general constraint, with signature:

`DISTRIBUTE(X, V, N)`

Two important differences w.r.t. our definition:

- The cardinality bounds are specified via $D(n_j)$
- The employed propagator can filter the n_j variables

Which means that we can use `DISTRIBUTE` for counting!

Other Constraints in the Same Family

Many solver provide other similar constraints

If we simply need to count the occurrences of a value, we have:

```
COUNT(X, v, c)
```

Where:

- **X** is a vector of integer variables
- **v** is an integer value
- **c** is either an integer or a variable

The constraint is satisfied if value **v** is taken **c** times in **X**

Other Constraints in the Same Family

Many solver provide other similar constraints

If we need to limit the occurrences of a single value, we have:

`ATMOST(X, v, c)`

Where:

- **`X`** is a vector of integer variables
- **`v`** is an integer value
- **`c`** is an integer

The constraint is satisfied if value **`v`** is taken less than **`c`** times in **`X`**

- The case with c variable is subsumed by **`COUNT(X, v, c)`**

Other Constraints in the Same Family

Many solver provide other similar constraints

If all values must be different, except for a special one, we have:

```
ALLDIFFERENTEXCEPT(X, v)
```

Where:

- **`X`** is a vector of integer variables
- **`v`** is an integer value

All value can be taken at most once, except for **`v`**

- Useful to model empty bins or empty slots