Chapter 3
Constraint Programming

Paragraph 2
Constraint Programs and Consistency

Search and Inference

- As in Integer Programming, the general outline of the Constraint Programming methodology that we saw last week combines two fundamentally different approaches:
  - An inference component: the shrinking of domain values by constraint filtering until no one constraint alone is able to remove any more values from the variables' domains.
  - A search component: backtracking is used if inference alone is not enough to reduce all domains to singletons or to prove insolvability.

Constraint Satisfaction Problem

- Given a finite set of variables $X = \{X_1,\ldots,X_n\}$.
- Given a set of values $V = \{v_1,\ldots,v_m\}$.
- Given a set of domains $D = \{D_1,\ldots,D_n\}$ such that $D_i \subseteq V$.
- Given a set of constraints $C = \{C_1,\ldots,C_k\}$ with C_i: $\Pi_{r\in Ri}\, D_r \rightarrow$ {true, false}. R_i is called the scope of constraint i.
- We call a tuple $A := (v^1,\ldots,v^n)$ such that $v^i \in D_i$ an assignment or solution. A is called feasible iff $C_i\,(A|_{Ri})$ = true for all i.
- The Constraint Satisfaction Problem (CSP) is given as (X,V,D,C) and asks for computing a feasible assignment or prove that none exists.

Constraint Satisfaction Problem

- The way how the constraints are given is not specified in the previous definition.
- If the cardinality of the scope of all constraints is lower or equal two, we also speak of a binary CSP.
- When the scopes are limited, one can afford to give the constraints as truth tables:

D_r	D_s	C_i
blue	1	true
blue	2	true
blue	3	false
red	1	true
red	2	false
red	3	false

Implicit Enumeration

- We can solve a CSP by brute force enumeration, of course. In this case, we generate a solution and check feasibility, i.e. we only use constraints passively. This method is also known as Generate and Test.
- Again, in order to speed up our search, we need to enumerate the overwhelming part of the search space implicitly.
- How can we use constraints to accomplish this?

Implicit Enumeration

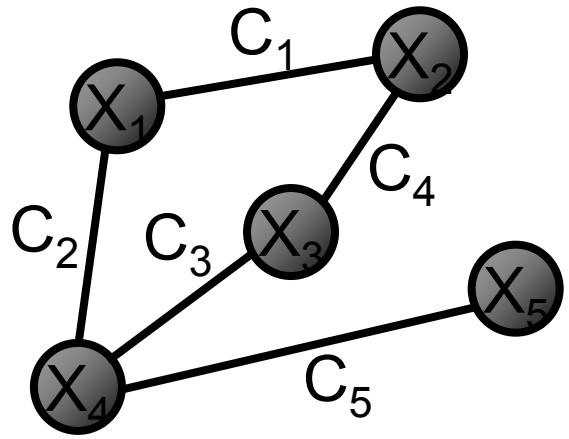
- Generate and test methods are highly affected by a phenomenon that we call thrashing:
  - Say the domain of X_1 was $D_1 = \{1,\ldots,m\}$. Say the scope of C_1 was $\{X_1\}$, and that it took value true iff $X_1 \geq m-3$.
  - Generate and test may try to set $X_1 = 1$, $X_1 = 2$, … and always only recognize a failure when a full solution has been generated.
  - The effect is called thrashing because all failures in a given subtree can be attributed to the same simple mis-assignment.

Node Consistency

- The problem in the previous example can easily be rectified by using the unary constraints of a CSP actively.
- The effects of a unary constraint can simply be used to shrink the domain of the corresponding variable. If we do this for all unary constraints, we say that we achieved node consistency.
- When a domain runs empty, we know that no feasible solution can be found anymore, and we backtrack right away.

Constraint Graph

- Given a binary CSP, we can visualize dependencies of variables by a constraint graph:



Arc Consistency

- Node consistency only avoids very simple forms of thrashing:
  - Assume a constraint over two variables was true iff $X_1 \leq X_2$. Of course, we should never try assignments where this constraint is violated. In general: We can check constraints already when the variables in their scope have been assigned values.
  - Thrashing goes further though: Assume $D_1 = \{2,3\}$, $D_2 = \{1,2\}$, $D_3 = \{3,4\}$, and we have $X_1 \leq X_2$, $X_3 \leq X_1$. How can we detect these inconsistencies before assigning two or even all three variables?

Arc Consistency

- A binary CSP is called arc-consistent iff for all constraints over X_i, X_j for all domain values in D_i (D_j) there exists a domain value in D_j (D_i) that is consistent wrt the constraint.
- Efficient algorithms for achieving arc-consistency are the first step to an efficient CSP solver.

Arc Consistency

procedure REVISE (i,j)
  DELETED := false
  for each v in D_i do
    if there is no such w in D_j such that (v,w) is consistent,
    i.e., no (v,w) satisfies all the constraints on X_i, X_j then
      delete v from D_i
      DELETED := true
    end_if
  end_for
  return DELETED
end REVISE

Arc Consistency

```
procedure AC-3 (Constraint Graph G)
  Q := {(i,j) | (i,j) is a directed arc in G, i ≠ j}
  while Q ≠ ∅ do
    select and delete (i,j) from Q
    if REVISE (i,j) then
      Q := Q ∪ {(p,i) | (p,i) is a directed arc in G, p≠i, p≠j}.
    end_if
  end_while
end AC-3
```

Arc Consistency

- What is the computational complexity of the proposed method AC-3?
- For a binary constraint network, if there are n nodes, domain size is d and there are a arcs, the complexity of AC-3 is $O(ad^3)$.
- Can this complexity be improved upon? Note that a lot of work is done many times by checking all domain values even if their supporting counterpart has not been removed!

Arc Consistency

```
procedure AC-4 (G)
  Q := INITIALIZE(G)
  while Q ≠ ∅ do
    select and delete any variable/value pair <j,w> from Q
    for each <i,v> from the set of supported values S_{j,w} do
      counter[(i,j),v] := counter[(i,j),v] - 1
      if counter[(i,j),v] = 0 & v is still in D_i then
        delete v from D_i
        Q := Q ∪ {<i,v>}
      end_if
    end_for
  end while
end AC-4
```

Arc Consistency

- It can be shown: The time-complexity of AC-4 is in $O(ad^2)$. This is optimal!
- However, AC-4's memory requirements are prohibitively large in practice.
- Improvements like Bessiere et al.'s AC-6 reduce those memory requirements as well.

Search Management

- As we investigated it for Integer Programming, we need to make decisions on how we want to organize our backtrack search.
- Branching Variable Selection
  - Smallest Domain - First Fail Strategy
- Branching Direction (or Value) Selection
  - First Succeed Strategy
- Assignment Selection
- Search Strategy
  - Limited Discrepancy Search
  - Depth-bounded Discrepancy Search

Backtrack-Free Search

- Arc-consistency improves the efficiency of CSP search algorithms tremendously.
- However, in general search is still needed, and the insoluability of CSPs may only be proven after extensive search.
- Like we investigated total unimodularity for IPs, there are also conditions, under which we can show that our inference technique allows us to solve CSPs in polynomial time: One such condition is that the constraint graph is cycle-free!

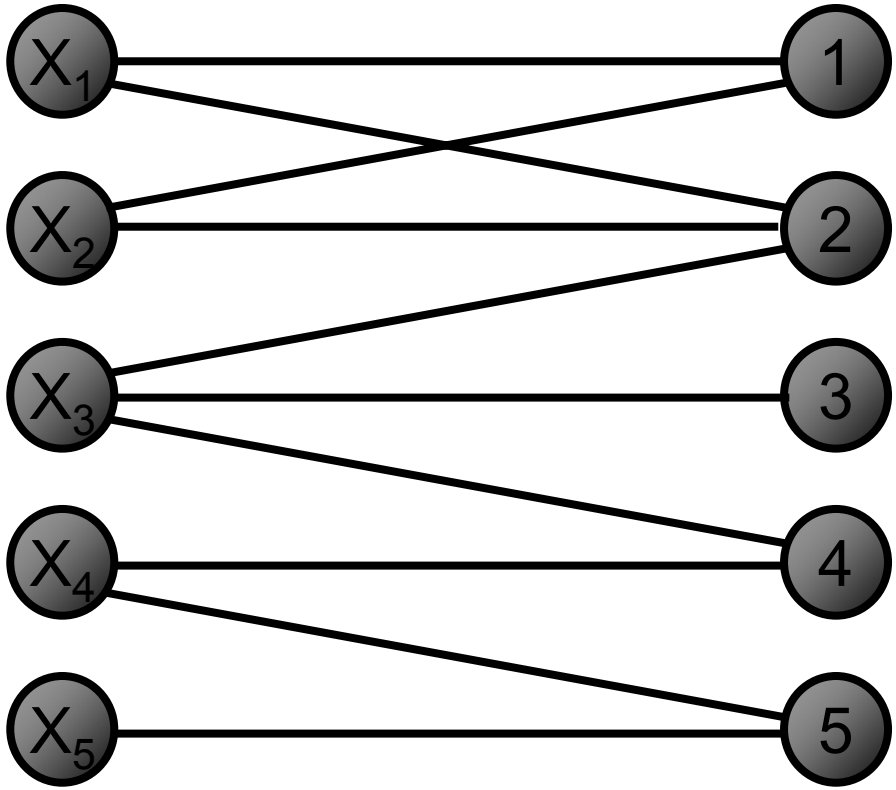
Generalized Arc-Consistency

- Many important constraints involve more than just two variables.
- The best we can hope for with respect to a constraint involving n variables is that we can guarantee that:
  - For every domain value of variable i
  - There exists an assignment to the remaining variables
  - Such that the constraint is fulfilled.

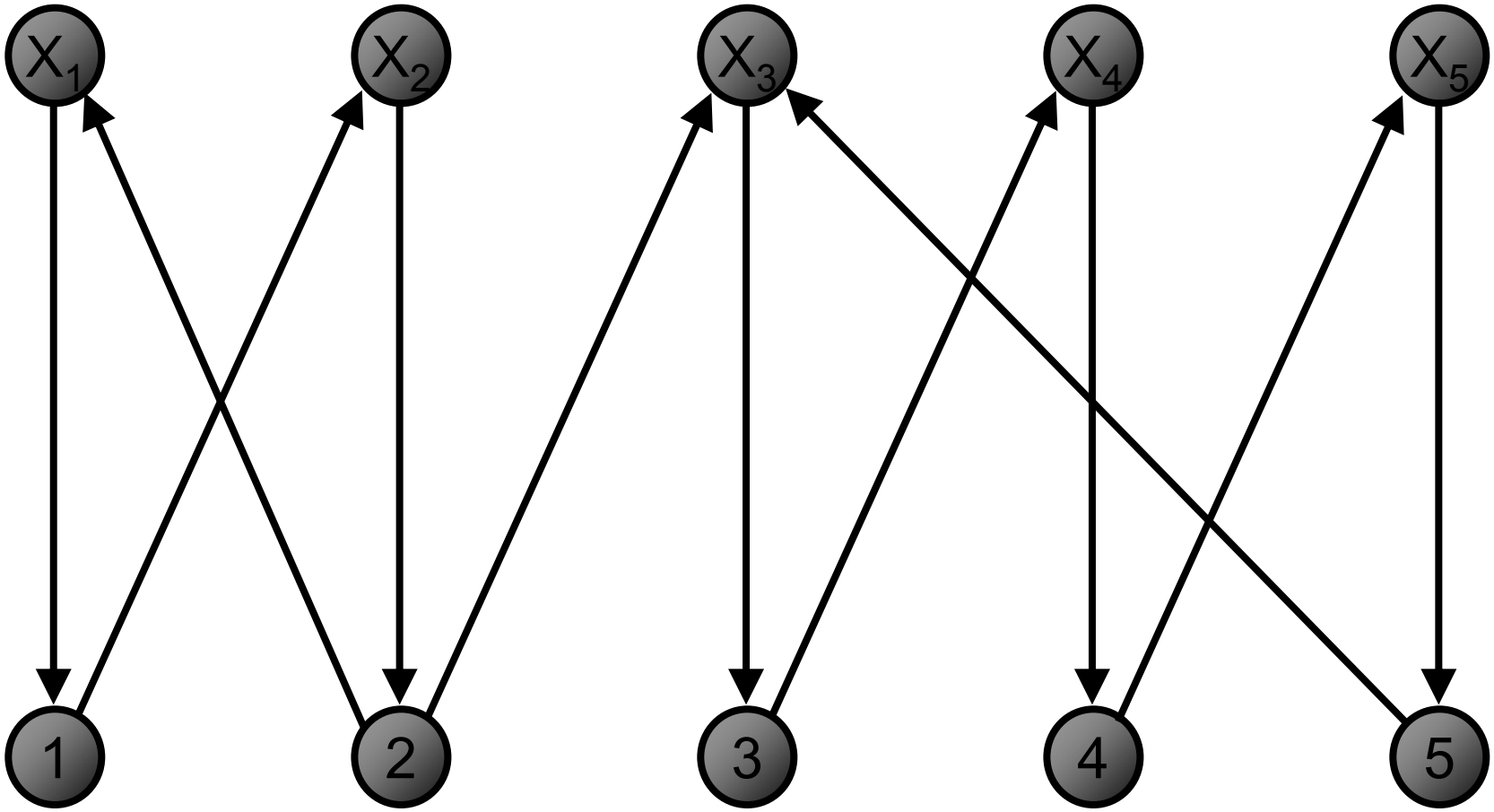
Generalized Arc-Consistency

- Consider the AllDifferent constraint:
  - $X_1,\ldots,X_n$ must take pairwise different values
- Note that arc-consistency on constraints $X_i \neq X_k$ cannot detect inconsistency of the following situation:
  - $D_1 = \{1,2\}$, $D_2 = \{1,2\}$, $D_3 = \{1,2\}$
- How can we achieve generalized arc-consistency for the (global) AllDifferent Constraint?

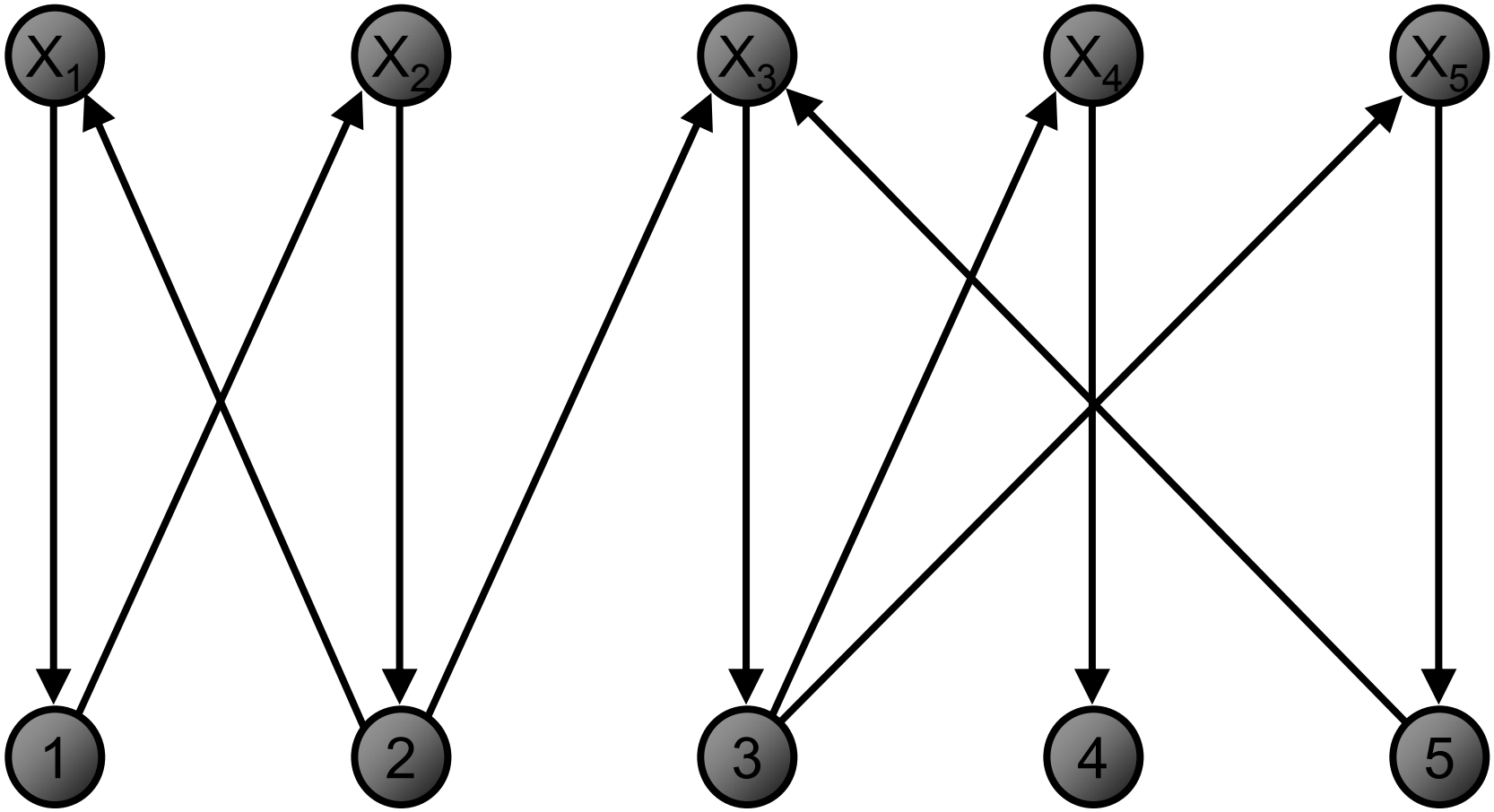
AllDifferent



AllDifferent



AllDifferent



Thank you!