Methods of Artificial Intelligence

Constraint Satisfaction Problems (CSPs)

UNIVERSITÄT OSNABRÜCK

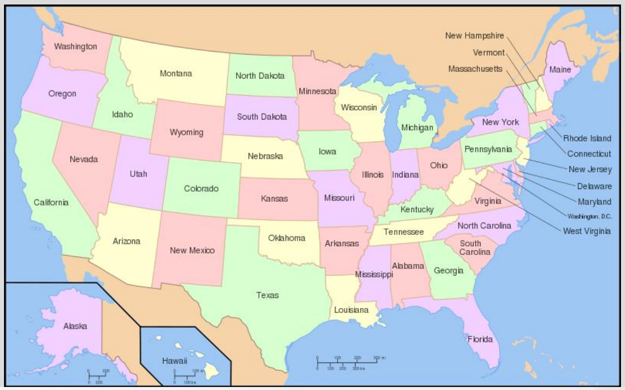
Constraint Satisfaction Problems

Examples and Definition

Constraint Satisfaction Problems (CSPs)

- Some applications (examples partially taken from Sigrid Knust)

  - Color a map with a minimum number of colors



  - Create a timetable for a school

Mo	10a	10b	10c	10d
1.	Müller	Meier	Hess	Schwarz
2.	Müller	Schulz	Hess	Meier
3.	Schulz	Hess	Müller	Weiß
4.	Meier	Hess	Schwarz	Weiß
5.	Hess	Müller	Meier	Schulz
6.	Schwarz	Müller	Weiß	Hess

  - Plan season for a sports league (tournament)

	1.	2.	3.	4.	5.
SVW	FCB	HSV	VFL	BVB	S04
HSV	BVB	SVW	S04	VFL	FCB
VFL	S04	FCB	SVW	HSV	BVB
FCB	SVW	VFL	BVB	S04	HSV
BVB	HSV	S04	FCB	SVW	VFL
S04	VFL	BVB	HSV	FCB	SVW

  - Plan shifts for people (bus drivers, pilots, nurses)

	Mo	Di	Mi	Do	Fr
06-10	Beate	Anna	Beate	Beate	Elke
10-14	Beate	Anna	Beate	Beate	Elke
14-18	Katharina	Elke	Elke	Anna	Anna
18-22	Katharina	Elke	Elke	Anna	Anna
22-02	Elke	Beate	Anna	Katharina	Katharina
02-06	Elke	Beate	Anna	Katharina	Katharina

Constraint Satisfaction Problems (CSPs)

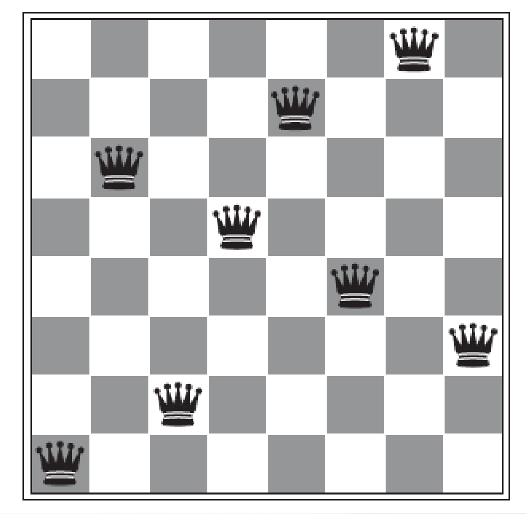
- A constraint satisfaction problem (CSP) consists of:
  - *variables* $X_1, \ldots, X_n$
  - Each variable X_i has an associated *domain* D_i
    - E.g. {true, false}, {red, blue, green}, [2,…,10], *N*, *Z, R*
  - Set of *constraints* R
    - Constraints are defined on variables and restrict the possible values that can be assigned to variables. For example,
    - $X_7 \in$ {red, blue, green} (unary constraint)
    - $X_1 \leq X_2$ (binary constraint)
    - $X_3 + X_4 \geq 4 * X_5 + 2 * X_6$ (4-ary constraint)

Solutions/ Consistent Assignments

- (Complete) variable assignment: assigns to each variable a value from its domain
- Partial variable assignment: assigns values to subset of all variables
- Consistent assignment: does not violate any constraints of CSP
- Solution: consistent complete assignment
- Consistent CSP: there exists a solution

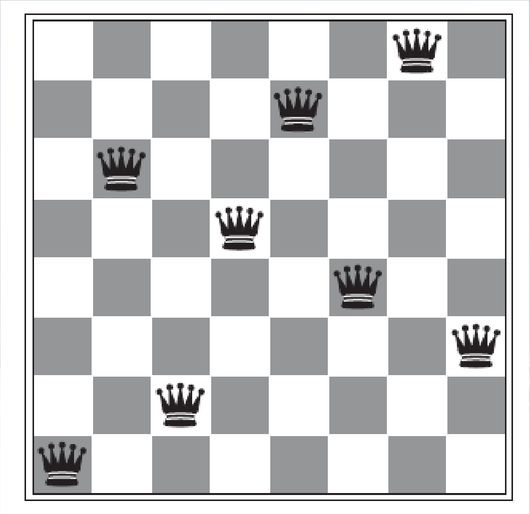
- Main problem: Given CSP,
  - find a solution or
  - report that the CSP is inconsistent

Example: 8-Queens Problem



- **Variables:** $X_1, \ldots, X_8$ (X_i stores row of i-th queen)
- **Domains:** domain of each variable is {1,…,8}
- Board configuration above corresponds to variable assignment (X_1=8, X_2=3, X_3=7, X_4=4, X_5=2, X_6=5, X_7=1, X_8=6) and partial assignments (), (X_1=8), (X_2=3), (X_1=8, X_2=3), …

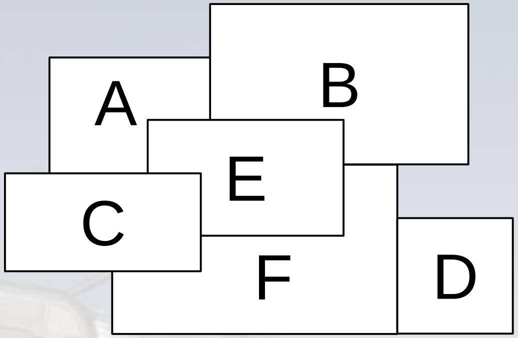
Example: 8-Queens Problem



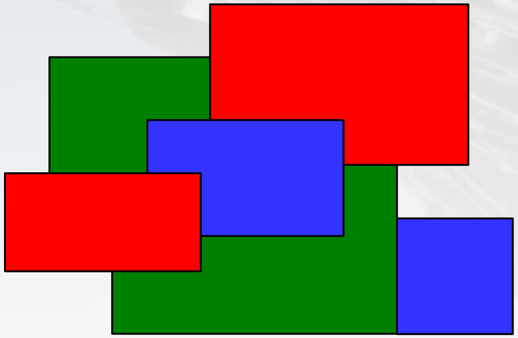
- Constraints: only one queen per vertical, horizontal and diagonal line
- Vertical constraints: are implicitly encoded by choice of variables
- Horizontal Constraints: $X_i \neq X_{i+k}$ for i=1,…,n, k=i+1,…,n
- Diagonal Constraints: $|X_i - X_{i+k}| \neq k$ for i=1,…,n, k=i+1,…,n

Exercise: Map Coloring

- Problem
  - Color the map with three colors such that adjacent regions have different colors

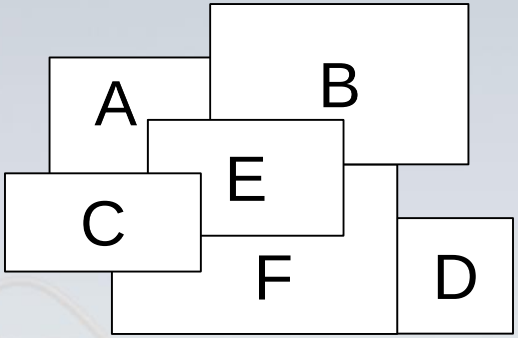


- Define a CSP that corresponds to this Map Coloring problem
  - Define variables
  - Define domains of variables
  - Define constraints



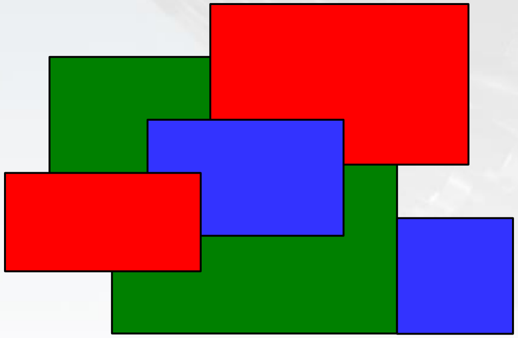
Solution: Map Coloring

- Variables: A, B, C, D, E, F



- Domains: {red, blue, green} for all variables

- Constraints
  - $A \neq B$, $A \neq C$, $A \neq E$,
  - $B \neq E$, $B \neq F$
  - $C \neq E$, $C \neq F$
  - $D \neq F$
  - $E \neq F$



Representation of Constraints

- CSPs from a logical point of view:
  - Constraints correspond to first-order predicates $P(x_1, x_2,..., x_n)$
  - All constraints must evaluate to true for a solution
    - $X_1 \leq X_2$
      P contains all pairs (v_1, v_2) over domains of X_1 and X_2 such that $v_1 \leq v_2$
    - $X_3 + X_4 \geq 4 * X_5 + 2 * X_6$
      P contains all 4-tuples (v_3, v_4, v_5, v_6) over domains of variables such that $v_3 + v_4 \geq 4 * v_5 + 2 * v_6$
- Explicit representation: store all n-tuples in P
- Implicit representation: implement P as a function that takes n-tuples as arguments and returns true or false

Example

- Consider variables X_1, X_2 with domains {1,2,3}
- Consider constraint $X_1 < X_2$
- Logically, $X_1<X_2$ is the binary predicate

  $P(X_1,X_2) = \{ (1,2), (1,3), (2,3)\}$

- { (1,2), (1,3), (2,3)} is the explicit representation of P.

  Assignment $(X_1=v_1, X_2=v_2)$ satisfies P iff $(v_1,v_2) \in P$

- the implicit representation of P takes a pair (v_1,v_2) as argument and returns true iff $v_1 < v_2$.

  Assignment $(X_1=v_1, X_2=v_2)$ satisfies P iff $P(v_1,v_2) = true$

Exercise

- Suppose X_1 and X_2 both have domain {1,2,..,20},
- $P(X_1, X_2)$ is the constraint $X_1 < X_2$,
- v is a variable assignment for X_1 and X_2.
- How many operations do we have to perform in the worst-case to test whether v satisfies P if
  - P is represented explicitly
  - P is represented implicitly
- What representation is more efficient?

Solution

- Logically, v satisfies P iff $P(v(X_1),v(X_2))$ is true
- If P is represented explicitly, we may have to enumerate all tuples in P to test whether $(v(X_1),v(X_2))$ is in P
- In our example the explicit representation of P contains 19+18+…+1 = 190 tuples that we may have to enumerate
- If P is represented implicitly, we only have to perform a single comparison
- Test $v(X_1) < v(X_2)$
- So the implicit representation is much more efficient
- Remember: if P is defined by a criterion that can be computed easily, choose implicit representation

Constraint Satisfaction Problems

Search Strategies

CSPs vs Classical Search

- Classical search problem
  - Search strategies treat states as black boxes.
  - Goal test is binary (passed or not passed).

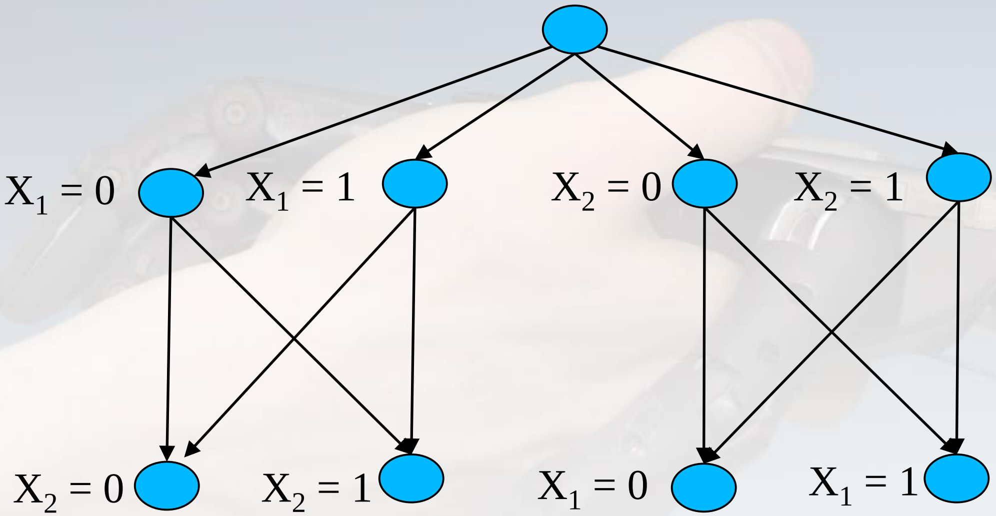
- CSP as a Classical Search Problem
  - States: (partial) variable assignments
    (initial state corresponds to empty assignment)
  - Actions: assign value to unassigned variable
  - Goal states: complete assignments that satisfy all constraints

- Special features of CSPs: Goal test can be
  - Passed (assignment is complete and all constraints are satisfied),
  - Partially passed (no constraint is violated),
  - Not passed (at least one constraint is violated).

- Search strategies for CSPs can exploit this feature.

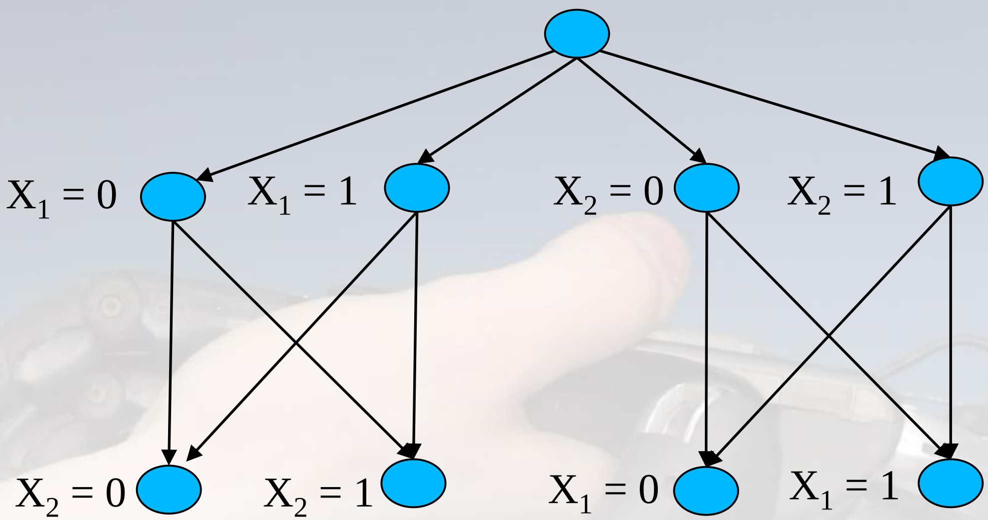
Naive Search Space

- Consider CSP with two Boolean variables



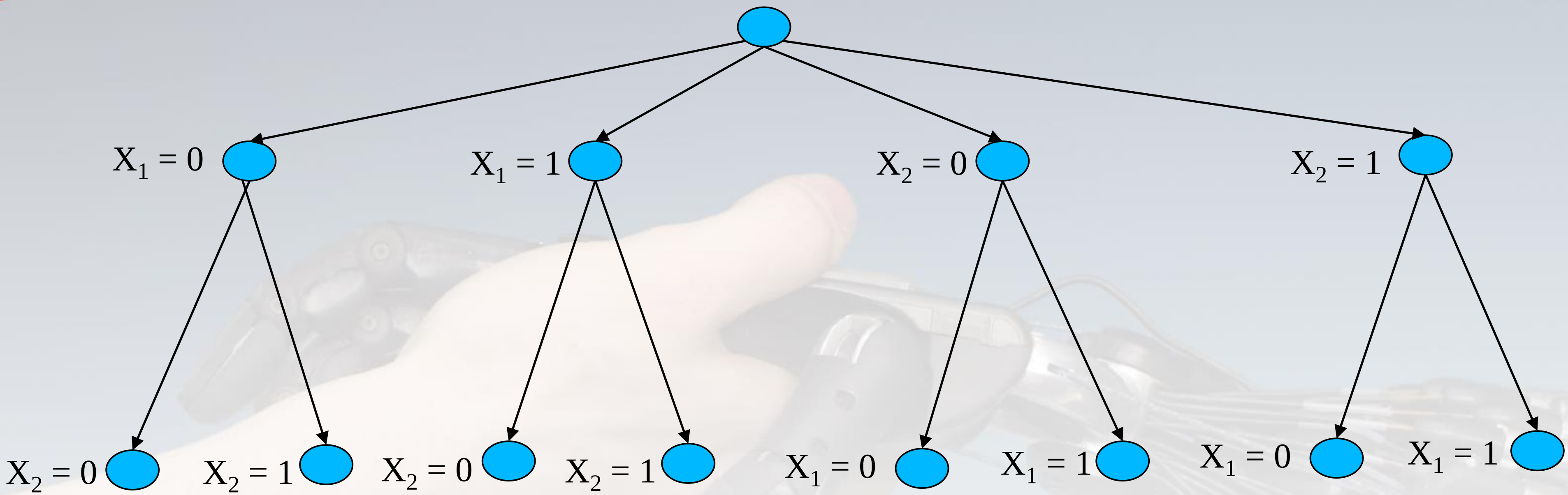
- corresponding search tree contains 1 + 4 + 8 = 13 nodes
- For three Boolean variables 1 + 8 + 32 + 64 = 105 nodes
- For four Boolean variables 1 + 16 + 128 + 512 + 1024 = 1,681 nodes

Exercise



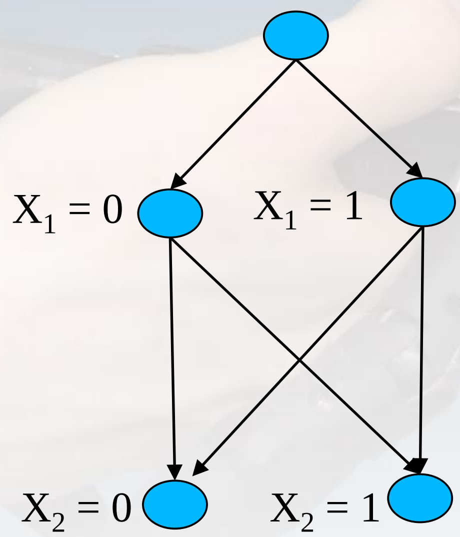
- Draw the search tree corresponding to the search space above

Solution: Search Tree



Simplification

- Path (X_1=0, X_2=0) and (X_2=0, X_1=0) yield the same state
- By fixing the order of variables, we can reduce the search space



  - corresponding search tree contains 1 + 2 + 4 = 7 nodes
  - For three Boolean variables 1 + 2 + 4 + 8 = 15 nodes
  - For four Boolean variables 1 + 2 + 4 + 8 + 16 = 31 nodes

Exercise

- Let the CSP have *n* variables. What is
  - the depth of the search tree?
  - the depth of an arbitrary solution state?
  - the branching factor of a node in the search tree?
- Is the variable ordering relevant?

Solution

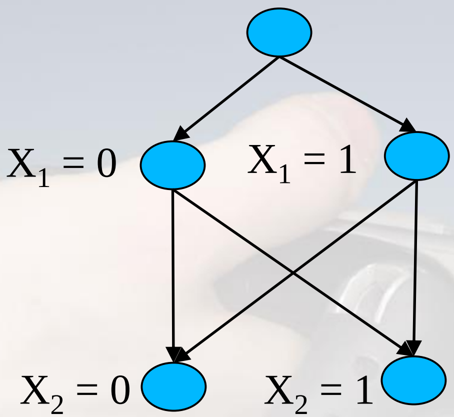
- Let the CSP have *n* variables. What is
  - the depth of the search tree? Answer: *n*
  - the depth of any solution state? Answer: *n*
  - the branching factor of a node in the search tree

  Answer: domain size of the corresponding variable.

- Is the variable ordering relevant?
  - Answers:
    - For finding all solutions: no
    - For efficiency: maybe

Goal Test

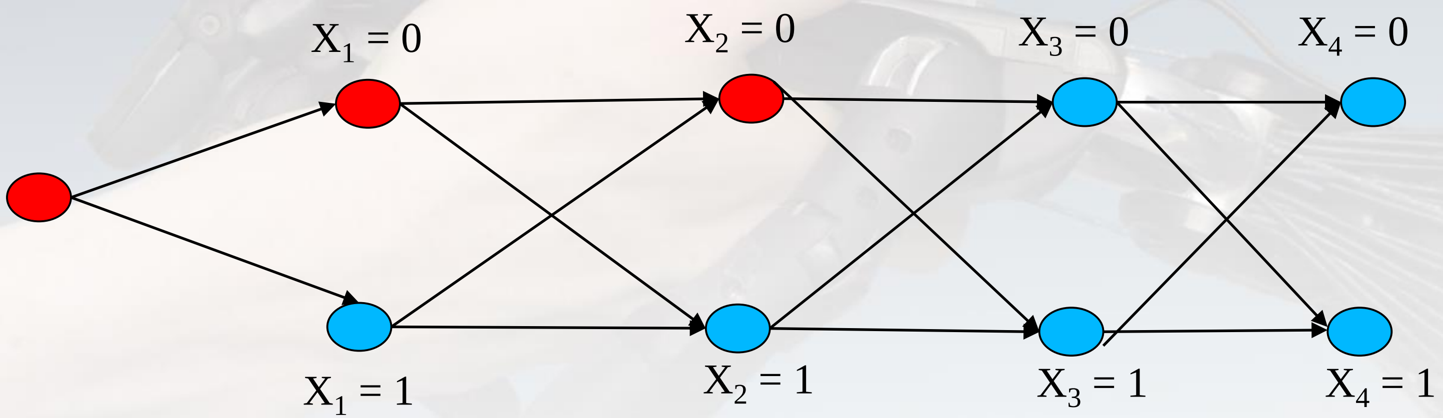
- Simplification reduces search tree significantly



- However, number of leaves (complete assignments) in search tree is still exponential in the number of variables
- Hence, we will have to check an exponential number of candidates whenever CSP is inconsistent
- If we use a bad variable ordering, finding a solution may also take a lot of time if the CSP is consistent

Testing Partial Assignments

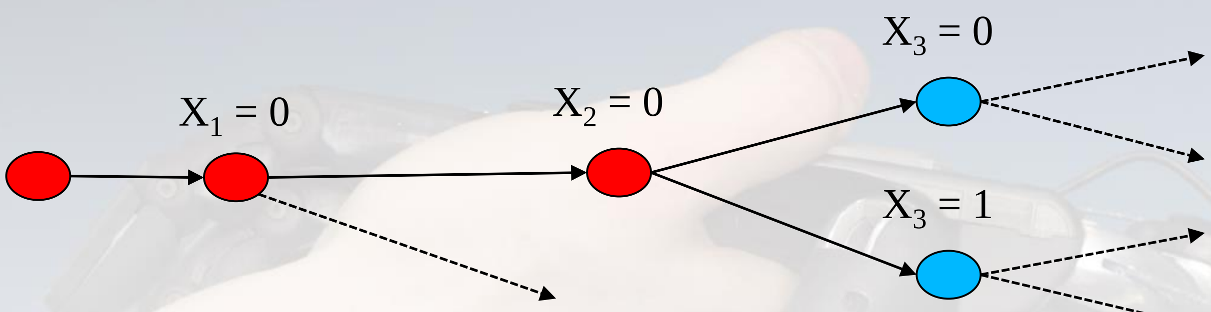
- We may improve runtime by checking constraints at inner nodes
- Inner nodes correspond only to partial assignments but may violate constraints already



- Consider constraint $X_1 + X_2 = 1$
- At state (X_1=0, X_2=0), we can already see that constraint cannot be satisfied along this path
- Therefore, we do not have to look at successor states

Exercise

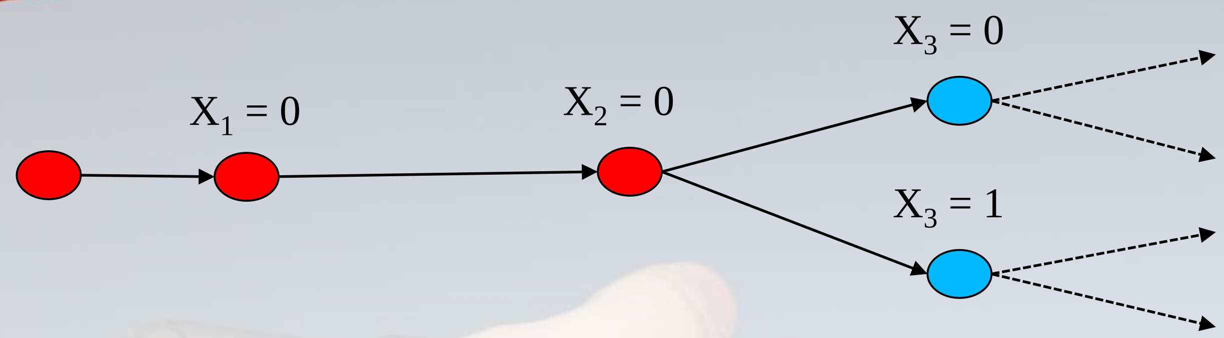
- Suppose that we find that the partial assignment (X_1=0, X_2=0) violates a constraint P(X_1, X_2)



- Can any successor of (X_1=0, X_2=0) satisfy P(X_1, X_2)?
- Assume that there exist *k* other variables X_3, X_4, … X_{2+k} in our CSP:

  How many steps do we save when ignoring all successors of (X_1=0, X_2=0)?

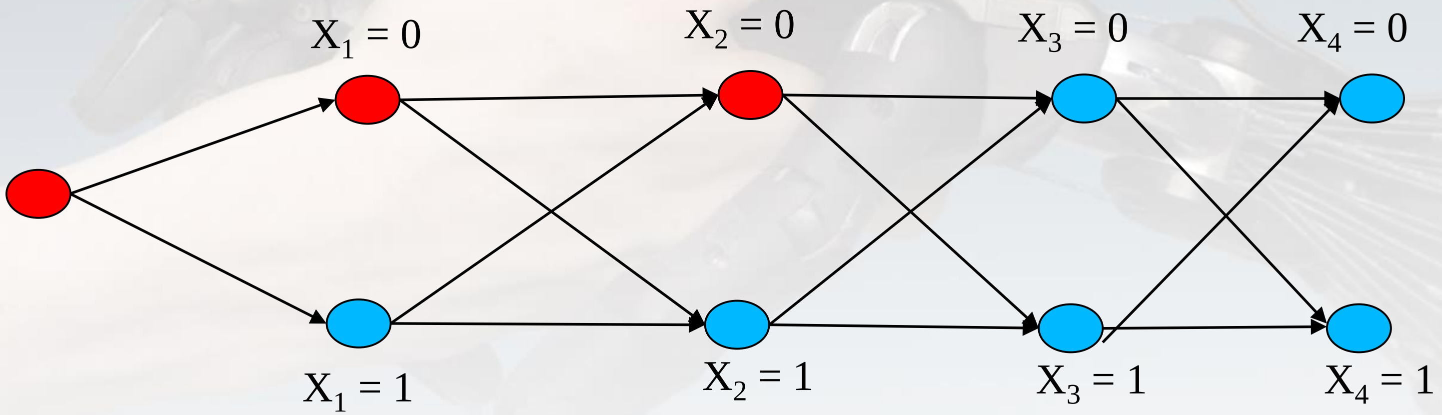
Solution



- Since $P(X_1,X_2)$ depends only on X_1, X_2, no successor of (X_1=0, X_2=0) can satisfy P - hence, there is no point in visiting any of them
- What about the number of successors?
  - X_2=0 has two successors X_3=0 and X_3=1
  - Each of X_3=0 and X_3=1 has two successors X_4=0 and X_4=1
  - Each of X_4=0 and X_4=1 has two successors X_5=0 and X_5=1
  - …
- Overall, we save $2^1 + 2^2 + 2^3 + \ldots + 2^k = 2^{k+1} - 2$ visits
- Hence, detecting inconsistent partial assignments early can save a lot of work

Variable Ordering

- What we can gain from checking partial assignments depends on variable ordering
- For instance, if the only constraint is $X_1 + X_4 = 1$, we can check constraint only at the leaves for the variable ordering below



- Therefore, we should always choose a variable ordering that allows us to detect inconsistencies early
- We can consider different heuristics for this purpose

Backtracking Search

Backtracking Search

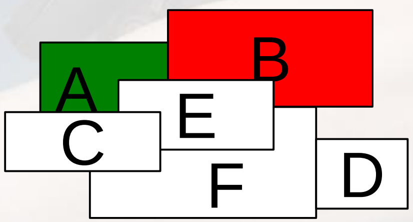
- Backtracking Search takes our observations into account
- variable ordering is not fixed
- variables are chosen dynamically while algorithm runs

- In each iteration, a primitive Backtracking Search
  - selects a variable according to some heuristic
  - selects a value for this variable according to some heuristic
  - tests partial assignment for consistency
    - if consistent, continue until variable assignment is complete
    - if inconsistent, try another value/ variable recursively (backtracking)

- At termination, Backtracking search either found a solution or can report that there exists no solution

Forward Checking

- Basic idea
  - after assigning a value to a variable X
    - check domains of all variables Y that appear in a constraint with X
    - if Y is the only unassigned variable in constraint, then delete all values from Y's domain that cannot make constraint true



	red	green	blue
A	x	✓	
B	✓	x	
C		x	
D			
E	x	x	
F	x		

- When assigning green to A, we can delete green from the domain of neighbors B, C, E
- When assigning red to B, we can delete red from the domain of neighbors A, E, F

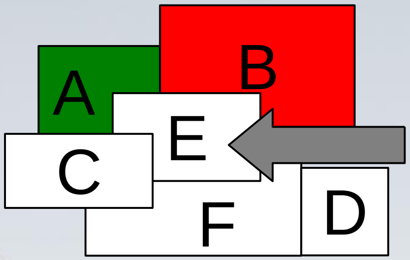
A more complex example

- Suppose we already assigned $X_1 = 2$ in a previous step
- We now assign $X_2 = 1$

- Suppose we have a constraint X1 + X2 + X3 > 5
- Since X_3 is the only unassigned variable in this constraint, we can restrict its domain
- Putting in the assignments of X_1 and X_2, our constraint becomes $2 + 1 + X_3 > 5$, that is we must have $X_3 > 2$
- Hence, we can delete all elements from X_3's domain that are less-than or equal to 2

Most constrained variable heuristic

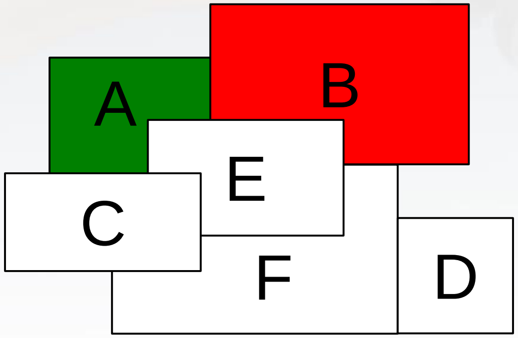
- Basic idea
  - choose unassigned variable with smallest domain
- Intuition
  - By assigning values to most constrained variables first, we may be able to detect inconsistent partial assignments early



	red	green	blue
A	x	✔	
B	✔	x	
C		x	
D			
E	x	x	
F	x		

Most constraining variable heuristic

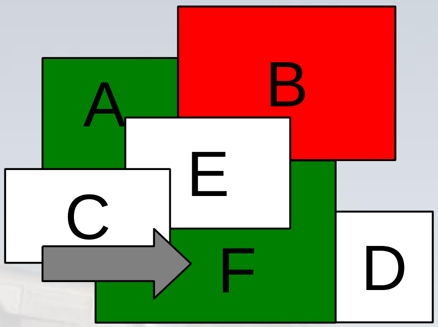
- Basic idea
  - choose variable that constrains highest number of unassigned variables
  - X constrains Y iff there is a constraint involving both X and Y
- Intuition
  - selecting most constraining variables first, may allow us to rule out many search paths early (forward checking will reduce domains)



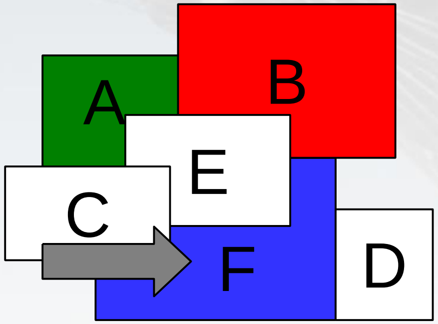
- C constrains E, F
- D constrains F
- E constrains C, F
- F constrains C, D, E

Least constraining value heuristic

- Basic Idea
  - choose *value* that rules out the smallest number of values in remaining variables
  - Can be expensive to compute
- Intuition
  - Value should restrict remaining variables as little as possible



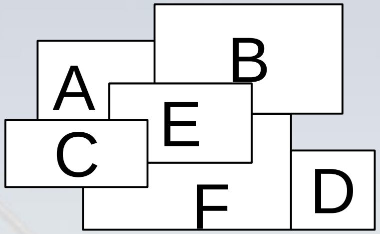
less constraining value
(does not change anything for C and E)



more constraining value
(further restricts domains of C and E)

Exercise: Map Coloring

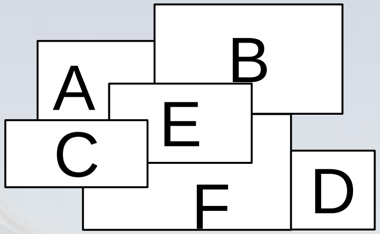
- Perform Backtracking search for the given map coloring problem
  - Use *most-constrained variable heuristic* for variable selection. If choice is not unique, choose *most-constraining* variable from the most-constrained ones (tie-breaking). If several choices remain, choose *lexicographically* (A < B < C < D < E) from the remaining ones
  - Select values *lexicographically* (blue < green < red)
  - Use *forward checking*



	red	green	blue
A			
B			
C			
D			
E			
F			

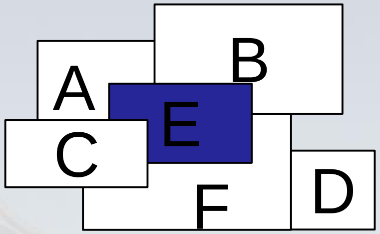
Solution: Map Coloring Iteration 1

- All variables can take 3 values. Hence, all variables are ‘most constrained’
- E and F are most constraining because they restrict 4 variables (B,C,D,E)
- We choose E because it comes lexicographically before F
- We assign blue to E because it comes lexicographically first



	red	green	blue
A			
B			
C			
D			
E			
F			

Solution: Map Coloring Iteration 1

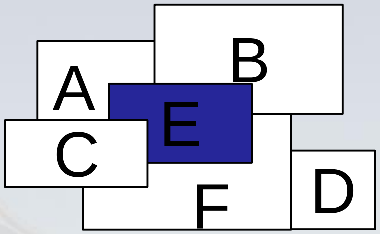


- In the forward-checking step, we eliminate blue from all neighbors of E
- We find that our partial assignment is still consistent and continue

	red	green	blue
A			x
B			x
C			x
D			
E			✓
F			x

Solution: Map Coloring Iteration 2

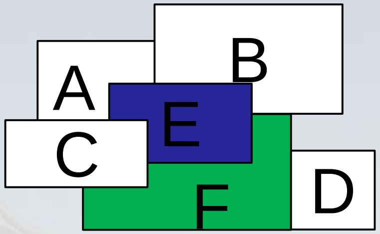
- A, B, C and F can take only 2 values and are most constrained
- F is most constraining because it restricts 3 remaining variables (B,C,D)
- We assign green to F because it comes lexicographically before red



	red	green	blue
A			x
B			x
C			x
D			
E			✔
F			x

Solution: Map Coloring Iteration 2

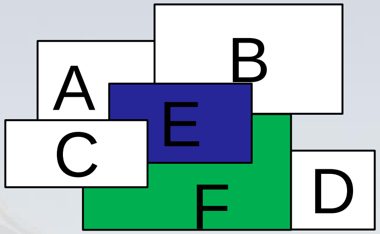
- In the forward-checking step, we eliminate green from all unassigned neighbors of F
- We find that our partial assignment is still consistent and continue



	red	green	blue
A			x
B		x	x
C		x	x
D		x	
E			✓
F		✓	x

Solution: Map Coloring Iteration 3

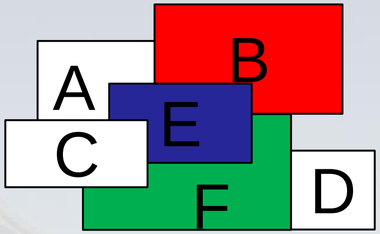
- B and C can take only 1 value and are most constrained
- They are both most constraining since they both constrain only A
- We choose B because it comes lexicographically before C
- We assign red to B because it is the only remaining value



	red	green	blue
A			x
B		x	x
C		x	x
D		x	
E			✔
F		✔	x

Solution: Map Coloring Iteration 3

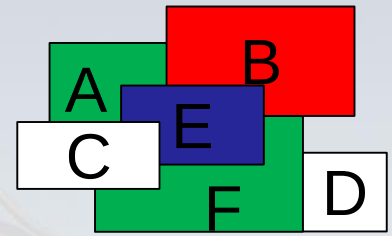
- In the forward-checking step, we eliminate red from all unassigned neighbors of B
- We find that our partial assignment is still consistent and continue



	red	green	blue
A	x		x
B	✓	x	x
C		x	x
D		x	
E			✓
F		✓	x

Solution: Map Coloring Iteration 4

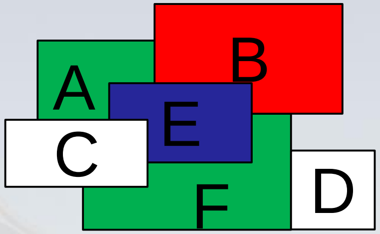
- A and C can take only 1 value and are most constrained
- They are both most constraining since they both constrain 0 unassigned variables
- We choose A because it comes lexicographically before C
- We assign green to A because it is the only remaining value



	red	green	blue
A	x	✔	x
B	✔	x	x
C		x	x
D		x	
E			✔
F		✔	x

Solution: Map Coloring Iteration 4

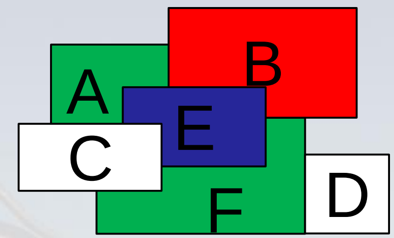
- In the forward-checking step, we eliminate green from all unassigned neighbors of A
- We find that our partial assignment is still consistent and continue



	red	green	blue
A	x	✓	x
B	✓	x	x
C		x	x
D		x	
E			✓
F		✓	x

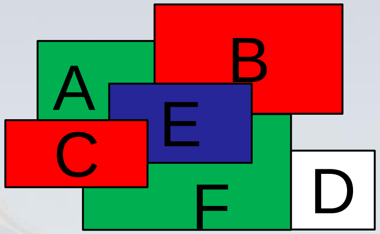
Solution: Map Coloring Iteration 5

- C can take only 1 value and is most constrained
- We assign red to C because it is the only remaining value



	red	green	blue
A	x	✓	x
B	✓	x	x
C		x	x
D		x	
E			✓
F		✓	x

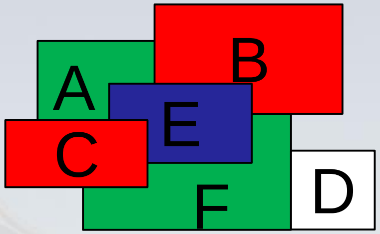
Solution: Map Coloring Iteration 5



- In the forward-checking step, we eliminate red from all unassigned neighbors of B
- We find that our partial assignment is still consistent and continue

	red	green	blue
A	x	✓	x
B	✓	x	x
C	✓	x	x
D		x	
E			✓
F		✓	x

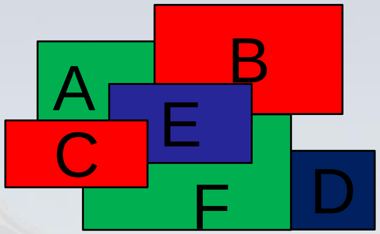
Solution: Map Coloring Iteration 6



- D can take only 2 values and is most constrained
- We assign blue to D because it comes lexicographically before red

	red	green	blue
A	x	✓	x
B	✓	x	x
C	✓	x	x
D		x	
E			✓
F		✓	x

Solution: Map Coloring Iteration 6



	red	green	blue
A	x	✓	x
B	✓	x	x
C	✓	x	x
D		x	✓
E			✓
F		✓	x

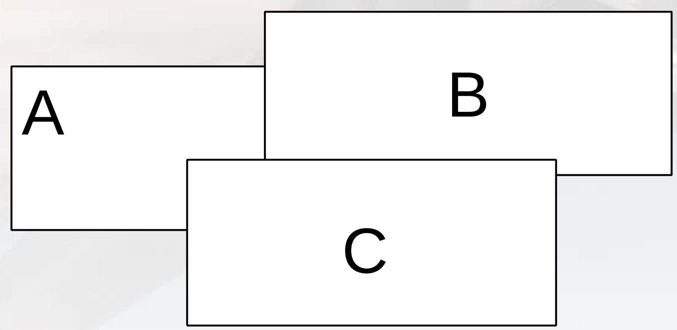
- In the forward-checking step, we eliminate blue from all unassigned neighbors of D
- We find that our partial assignment is still consistent.
- Since our assignment is complete and consistent, we found a solution

Backtracking

- If we assigned a value to a variable such that the partial assignment becomes inconsistent, we perform backtracking
  - for last assigned variable, undo domain restrictions
  - try the next value for this variable
  - if no more value exists, go back to previous variable and perform the same steps
  - if no more variables exist, report that the CSP is inconsistent

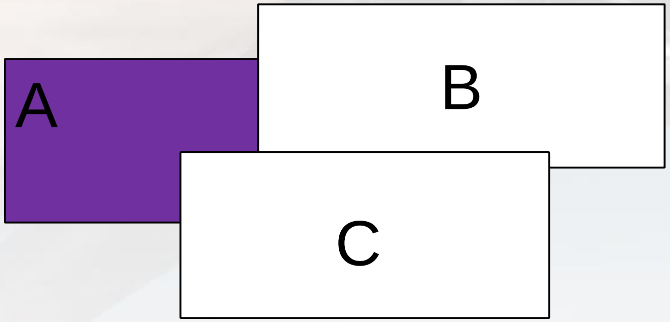
Example: Inconsistent CSP

- assign color from {violet, yellow} to variables in lexicographic order (we do not apply any selection heuristics)
- we use forward checking



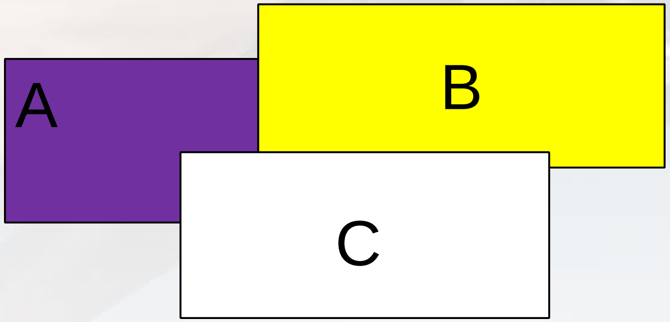
Example: Inconsistent CSP

- assign violet to A
- delete violet from domain of B and C



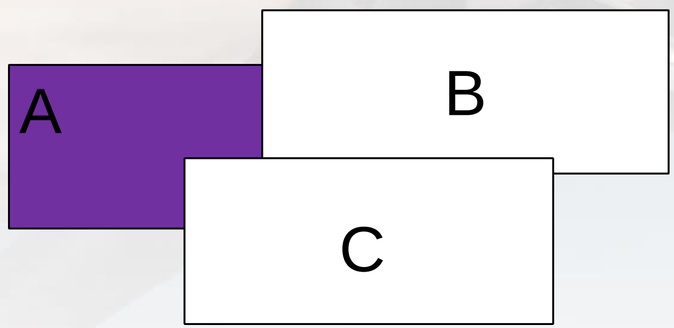
Example: Inconsistent CSP

- assign yellow to B
- delete yellow from domain of C



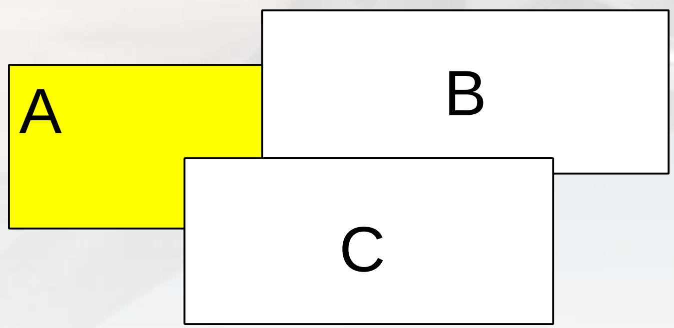
Example: Inconsistent CSP

- domain of C is now empty
- we add yellow to domain of C again (undo domain restriction)
- we cannot assign another color to B



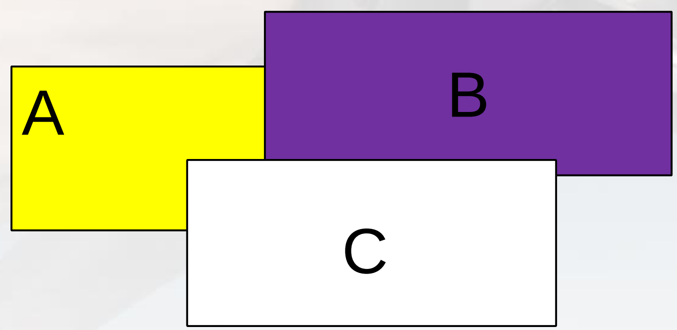
Example: Inconsistent CSP

- Since B has no more values, we go back to A
- we add violet to domain of B and C again (undo domain restriction)
- we assign yellow to A
- We delete yellow from domains of B and C



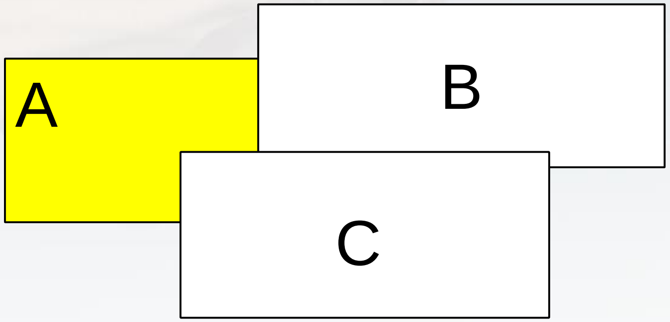
Example: Inconsistent CSP

- assign violet to B
- delete violet from domain of C



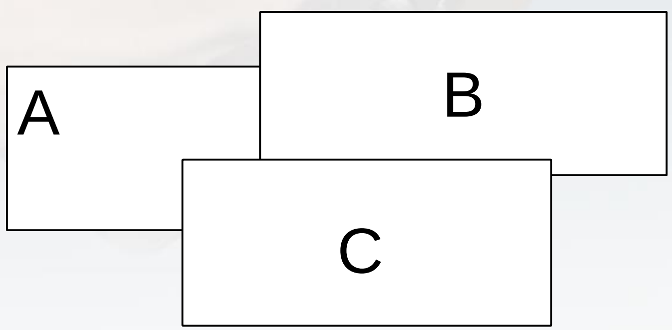
Example: Inconsistent CSP

- domain of C is empty again
- we add yellow to domain of C again (undo)
- we cannot assign another color to B



Example: Inconsistent CSP

- we add violet to domain of B and C again (undo)
- we cannot assign another color to A
- we report that CSP is inconsistent



Consistency Concepts for CSPs

Node Consistency

Arc Consistency

Path Consistency

k-Consistency

Consistency Concepts

- Basic idea: delete useless elements from domains of variables
- This works similar to forward checking but can even be applied before any value is assigned (pre-processing)

- The most important consistency concepts are
  - Node Consistency
  - Arc Consistency
- More involved concepts exist, but are difficult to compute

- Tradeoff: If preprocessing time exceeds time for solving CSP directly, we do not gain anything

Normalized CSPs

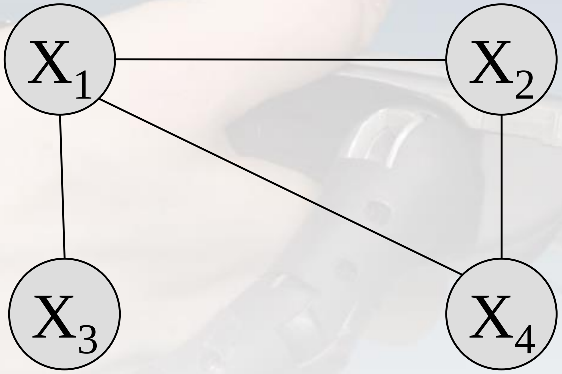
- We restrict our discussion to normalized CSPs that contain only unary and binary constraints

Domains:	$D_1, D_2, \ldots, D_n,$
Unary constraints:	$P_1\ (X_1), P_2\ (X_2), \ldots, P_n\ (X_\mathrm{n}),$
Binary constraints:	$P_{1,2}(X_1, X_2), P_{1,3}(X_1, X_3), \ldots, P_{n\text{-}1,n}(X_{\mathrm{n}\text{-}1}, X_\mathrm{n}).$

- This does not mean any loss of generality (we can transform each finite CSP into an ‘equivalent’ normalized CSP in polynomial time)

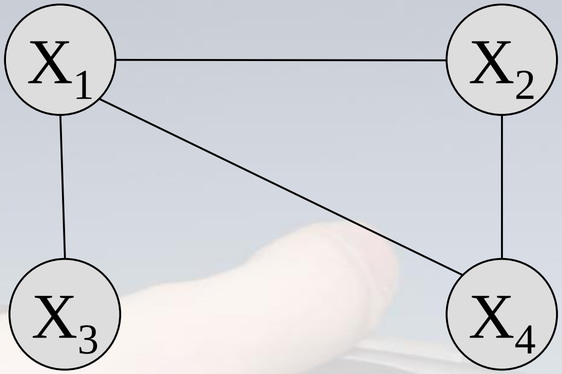
Constraint Graph for Normalized CSPs

- Normalized CSPs can be easily represented in constraint graph



- Node X_i can be identified with constraint P_i
- Edge between X_i and X_j can be identified with constraint P_{ij}

Node Consistency



- We call X_i node consistent iff all $x \in D_i$ are in P_i
- We call CSP node consistent iff all nodes are node consistent
- We can easily make CSP node consistent by letting $D_i = P_i$
- For example, consider frequency assignment problem: If P_i allows only particular frequencies for X_i, we just modify D_i accordingly

Exercise: Node Consistency

Consider frequency assignment problem with

- Variables: A, B, C, D, E, F
- Domains: $\{f_1, f_2, f_3\}$

- Suppose, we have a constraint $A \neq f_1$
  - Is problem node consistent?
  - How can you make CSP node consistent?

Solution: Node Consistency

Consider frequency assignment problem with

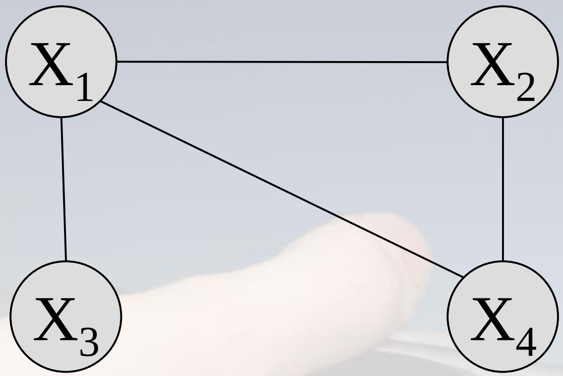
- Variables: A, B, C, D, E, F
- Domains: $\{f_1, f_2, f_3\}$

- Suppose, we have a constraint $A \neq f_1$
  - Is problem node consistent?

    Answer: No because the domain of A contains f_1.
  - How can you make CSP node consistent?

    Answer: We set the domain of A to $\{f_2, f_3\}$ (delete f_1)

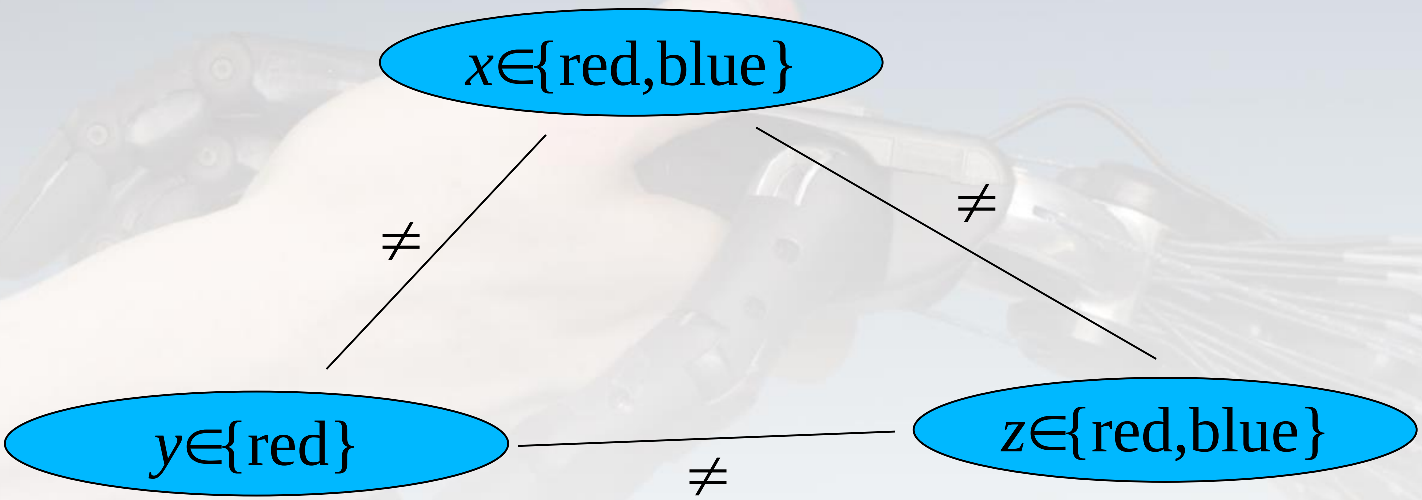
Arc Consistency



- We call X_i arc consistent with X_j iff for all $x \in D_i$ there is a $y \in D_j$ *such that* (x,y) is in P_{ij}
- Intuitively, for each value that X_i can take, X_j can take a value such that the binary constraint between X_i and X_j is satisfied
- We call CSP arc consistent iff all pairs of nodes are arc consistent
- Maintaining arc-consistency is more sophisticated

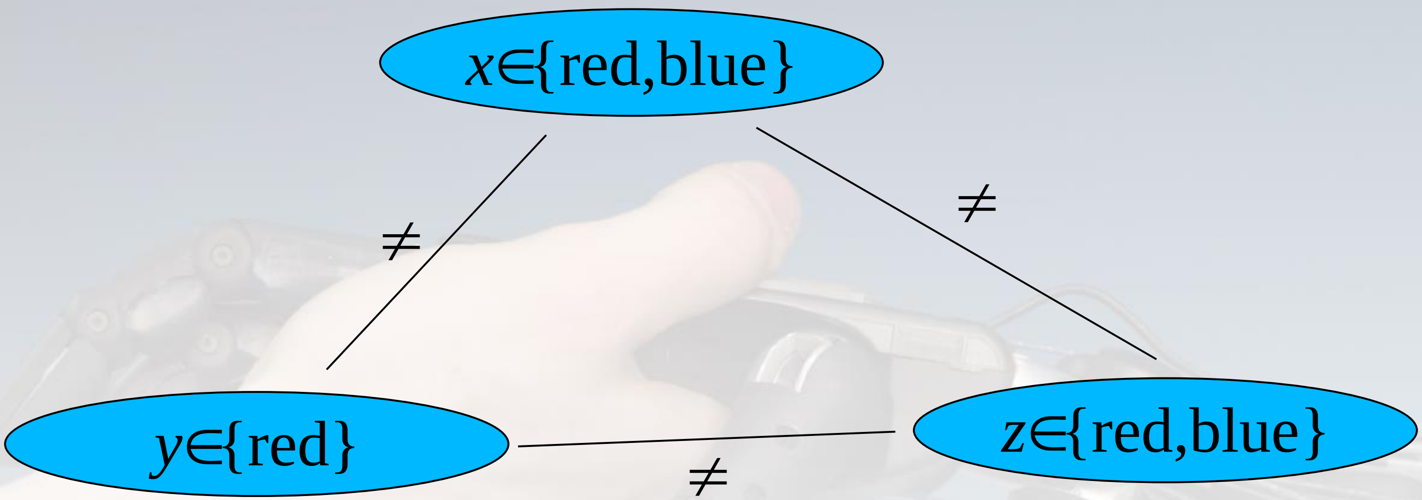
Exercise: Arc Consistency

- Consider the following Graph Coloring Problem



- What variables are arc consistent with each other?
- Is the CSP arc consistent?

Solution: Arc Consistency



- x and z are arc consistent
- x and y are not arc consistent
- y and z are not arc consistent
- Hence, the CSP is not arc consistent

AC-1 for establishing Arc Consistency

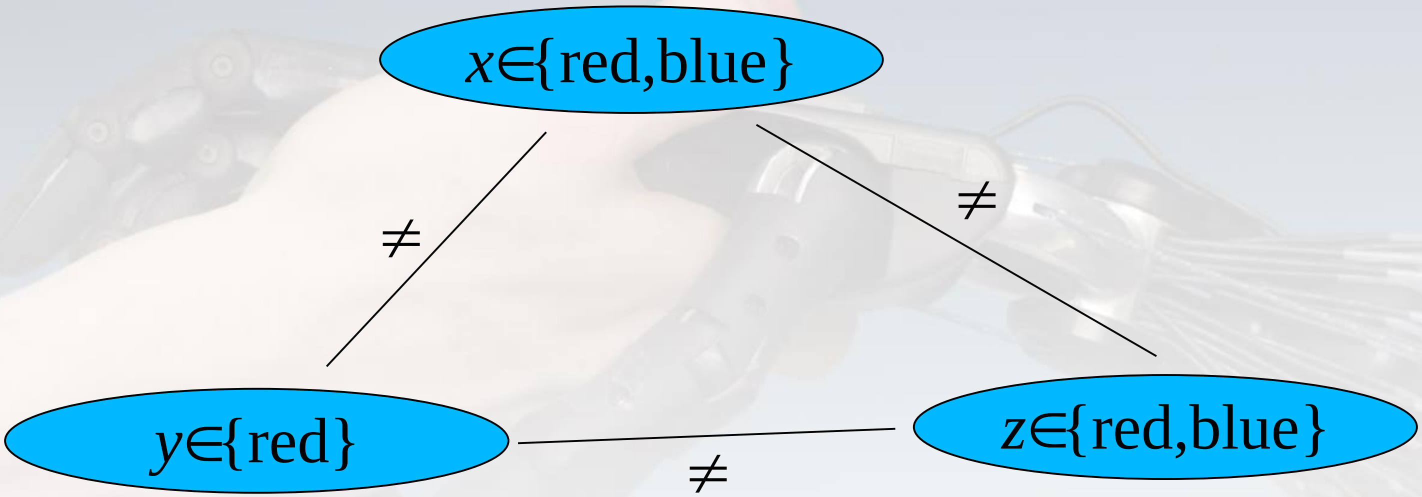
- AC-1 is the easiest (but not the most efficient) algorithm for making CSPs arc consistent

Revise(X_i, X_j) *(make X_i arc-consistent with X_j)*
  for each v_i in D_i
    if there is no v_j in D_j such that (v_i, v_j) in R_{ij}
      delete v_i from D_i

AC-1(X,D,C) *(make all variables arc-consistent)*
  do
    for each binary constraint P_{ij}
      Revise(X_i, X_j)
      Revise(X_j, X_i)
  until all domains remain unchanged or domain becomes empty

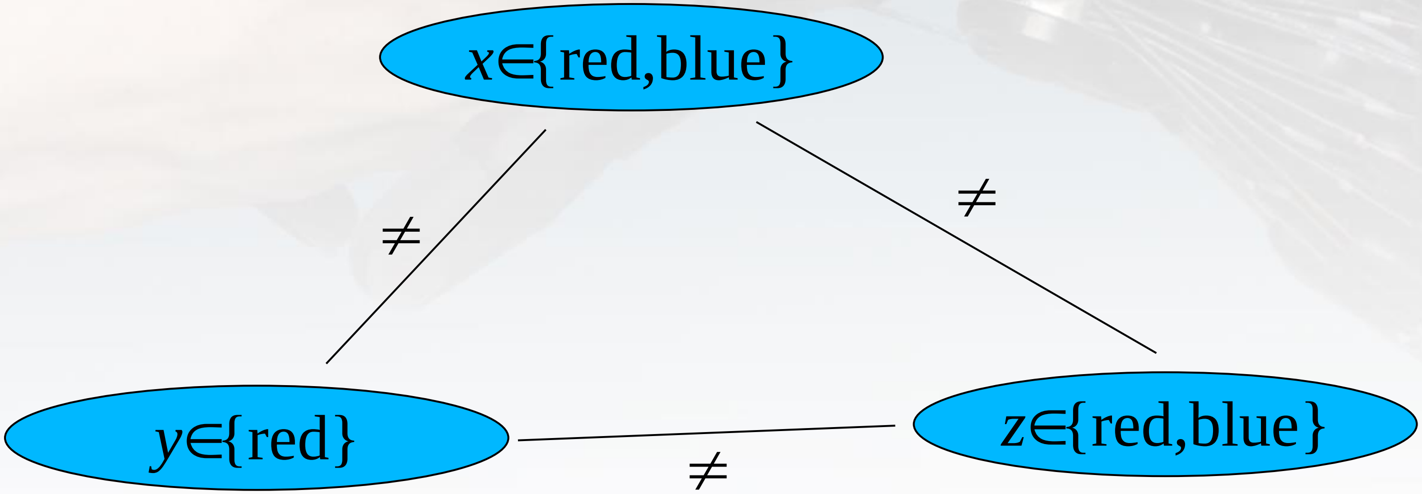
Example

- Consider again the following Graph Coloring Problem



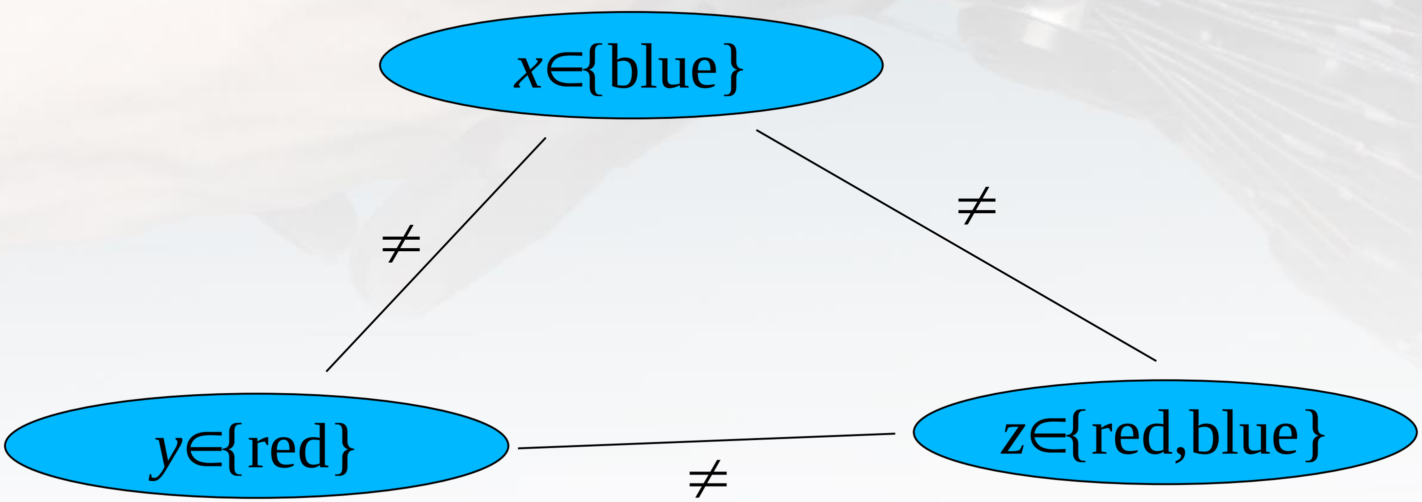
Example

- We start with P_{xy}
- We call Revise(x,y)
- For red in D_x there is no value v in D_y such that red $\neq$ v
- Hence, we delete red from D_x
- For blue in D_x, we can select red in D_y



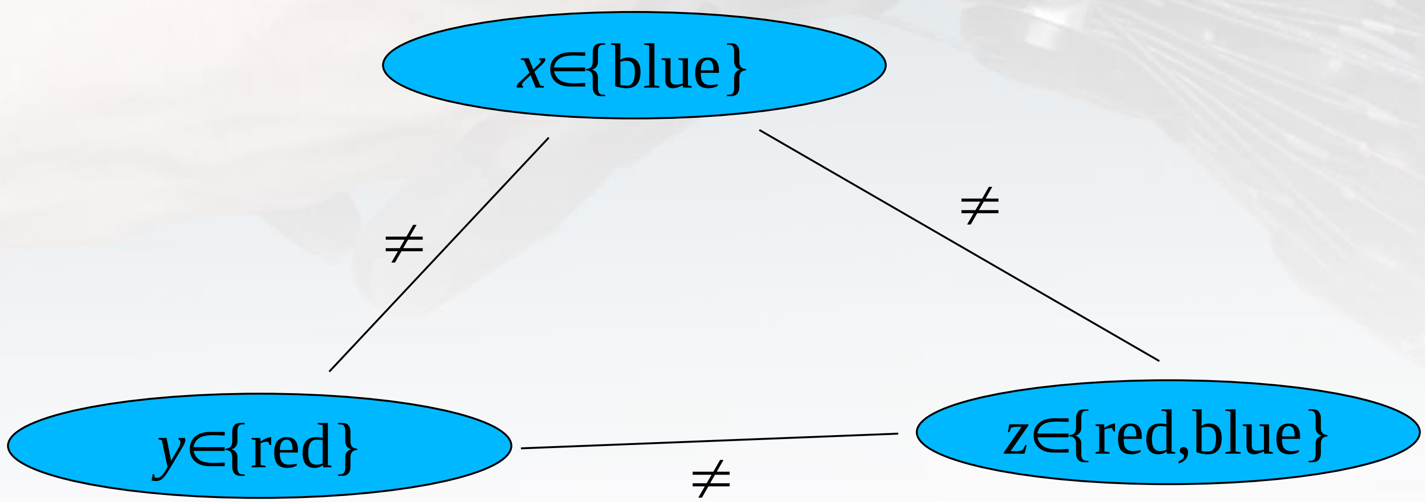
Example

- We now call Revise(y,x)
- For red in D_y, we can select blue in D_x



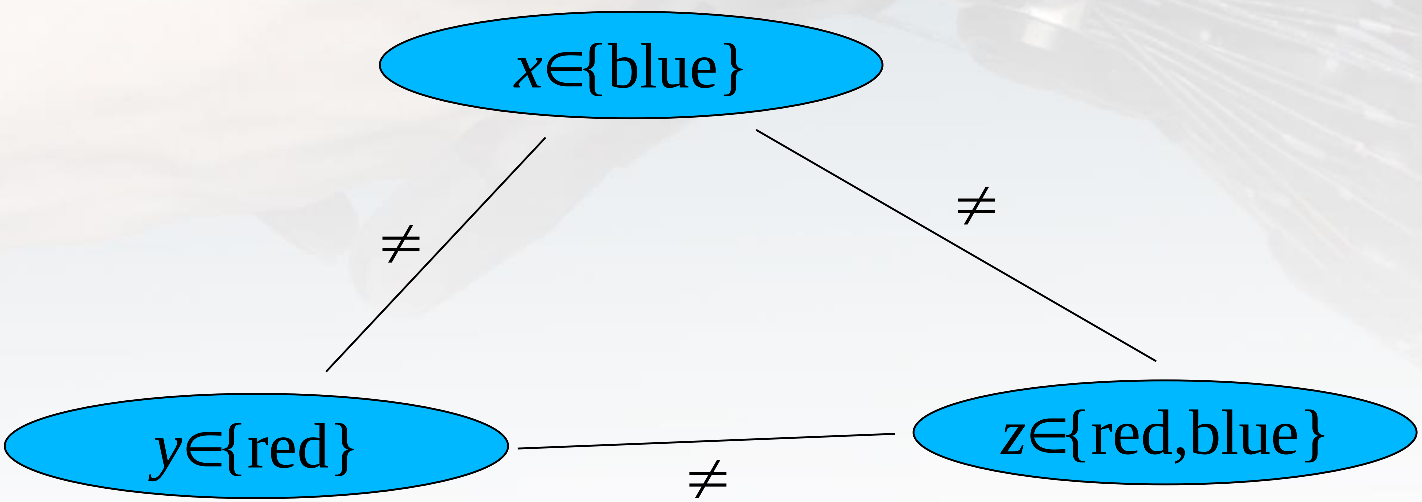
Example

- We next look at P_{xz}
- We call Revise(x,z)
- For blue in D_x, we can select red in D_z



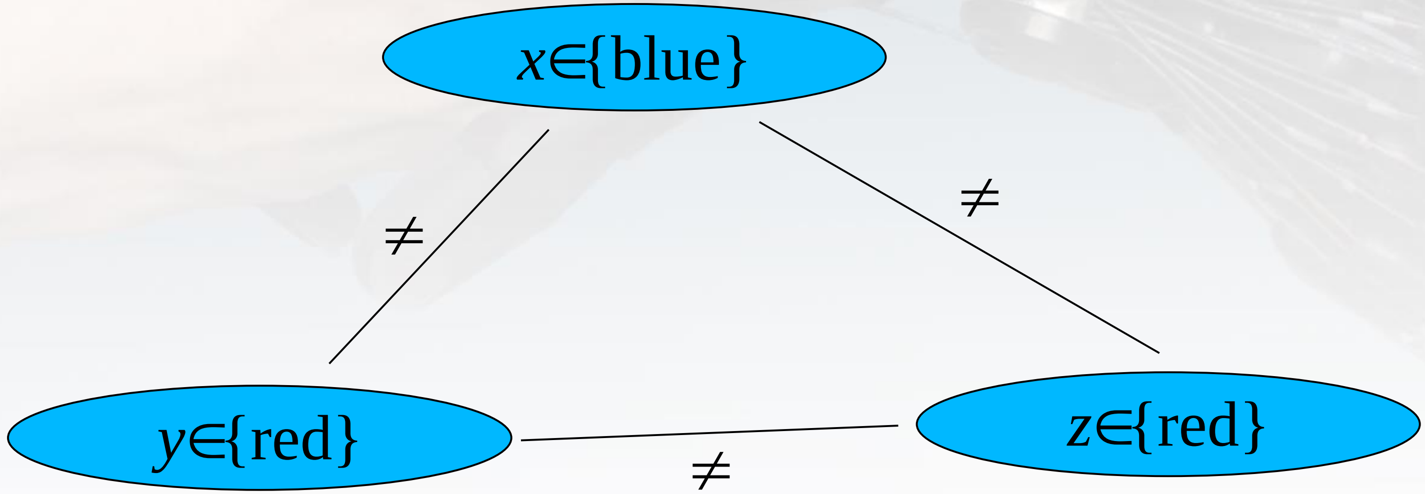
Example

- We now call Revise(z,x)
- For red in D_z, we can select blue in D_x
- For blue in D_z, there is no v in D_x such that blue $\neq$ v
- Hence, we delete blue from D_z



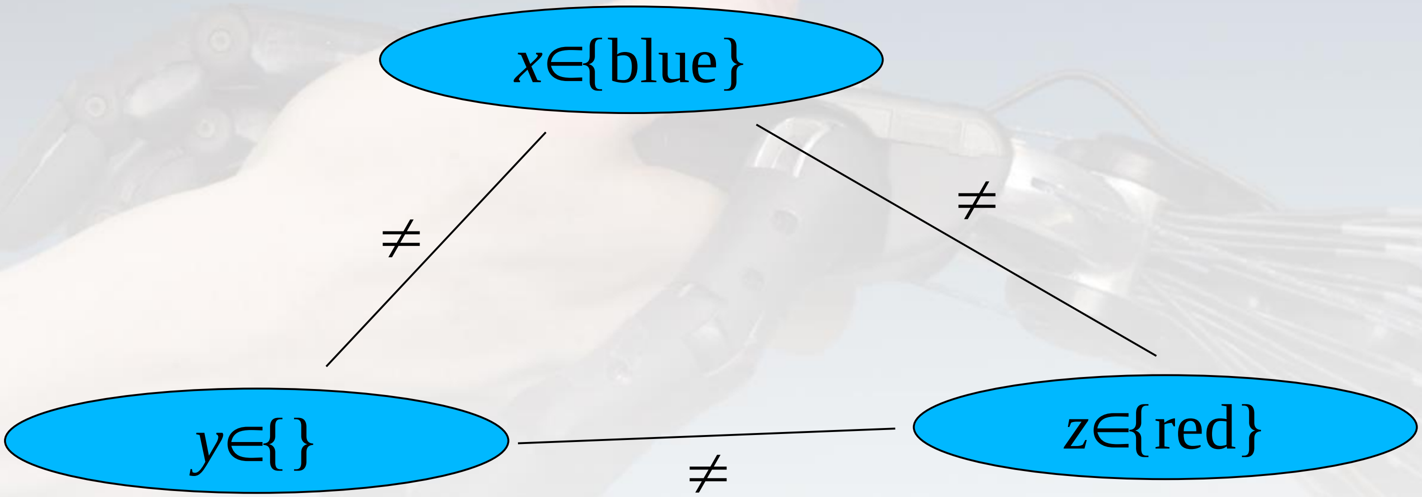
Example

- We next look at P_{yz}
- We call Revise(y,z)
- For red in D_y, there is no v in D_z such that red $\neq$ v
- Hence, we delete red from D_y



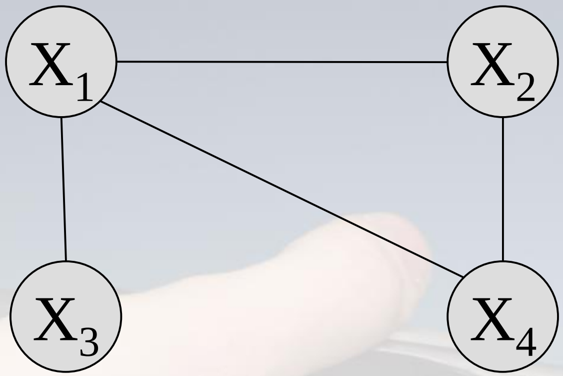
Example

- The domain of y is now empty
- Therefore, we know that the CSP is inconsistent and can stop



- Note: we found inconsistency without assigning any values

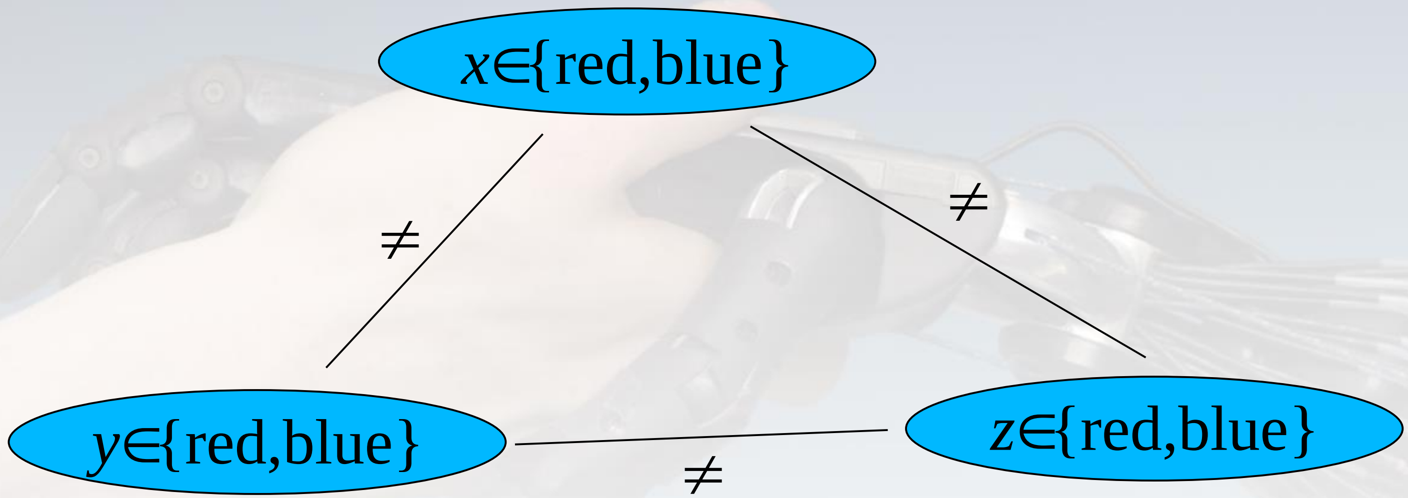
Consistency Concepts and Consistency



- If we get an empty domain, while establishing node consistency or arc consistency, our CSP must be inconsistent
- However, if a CSP is node consistent and arc consistent and has non-empty domains it is not necessarily (globally) consistent

Example

- The following CSP is both node consistent and arc consistent



- However, since there are only two values and the variables must take pairwise distinct values, the CSP is inconsistent
- Hence, node consistency and arc consistency are not sufficient for testing consistency

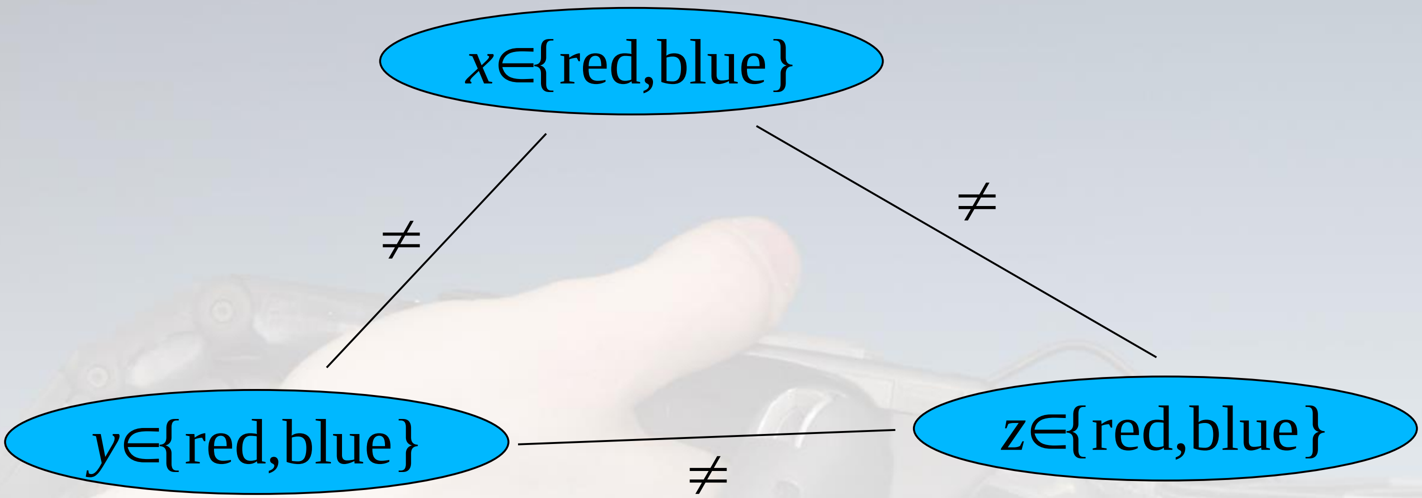
Exercise

- Suppose we have a consistency concept that can be tested in polynomial time (like node and arc consistency)
- Assuming $P \neq NP$, can this consistency concept be sufficient for consistency of the CSP?

Solution

- Assume that our consistency concept is sufficient for consistency of the CSP
- As we know from Introduction to AI, SAT (the propositional satisfiability problem) is a CSP
- Hence, we can use our consistency concept to decide SAT in polynomial time
- Since SAT is NP-hard, this implies P = NP
- This contradicts our assumption (P $\neq$ NP)
- Hence, under the usual complexity theoretical assumptions, there can be no consistency concept that is both
  - Sufficient for consistency and
  - Computable in polynomial time

Path Consistency



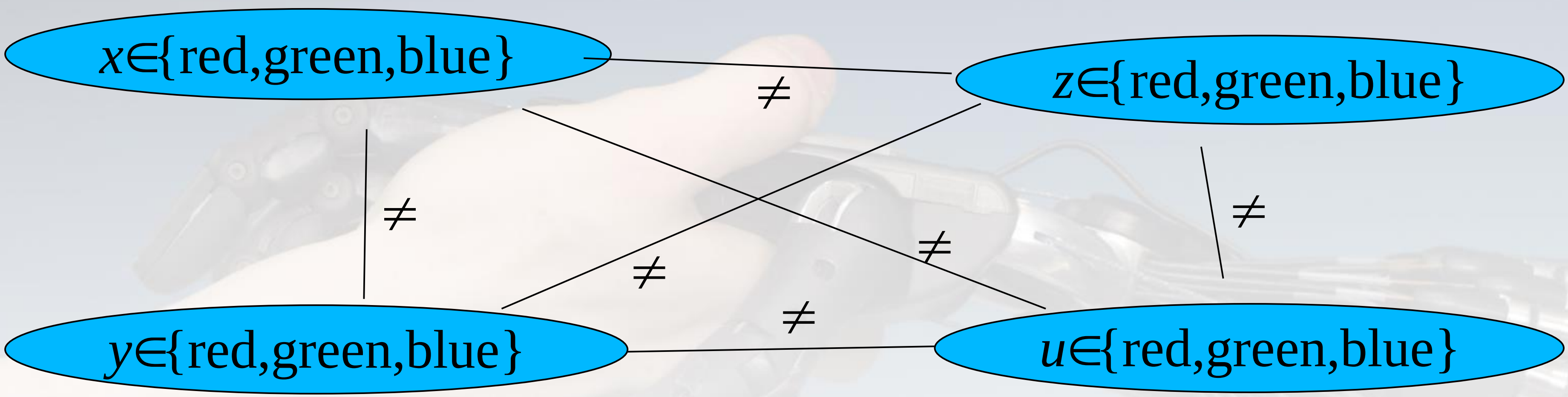
- Nodes i and j are path consistent with node m iff:

  For every arc consistent assignment of i and j, there is an assignment to m that is arc consistent with both i and j

- x and y are not path consistent with z
  - (x=red, y= blue) is arc-consistent assignment. However, z=red is not arc-consistent with x and z=blue is not arc-consistent with y

Consistency Concepts and Consistency

The following CSP is node, arc and path consistent



- However, since there are only three values and the variables must take pairwise distinct values, the CSP is inconsistent

k-Consistency and Strong k-Consistency

k-consistency

(6) Every consistent (k-1)-assignment can be extended to a consistent k-assignment

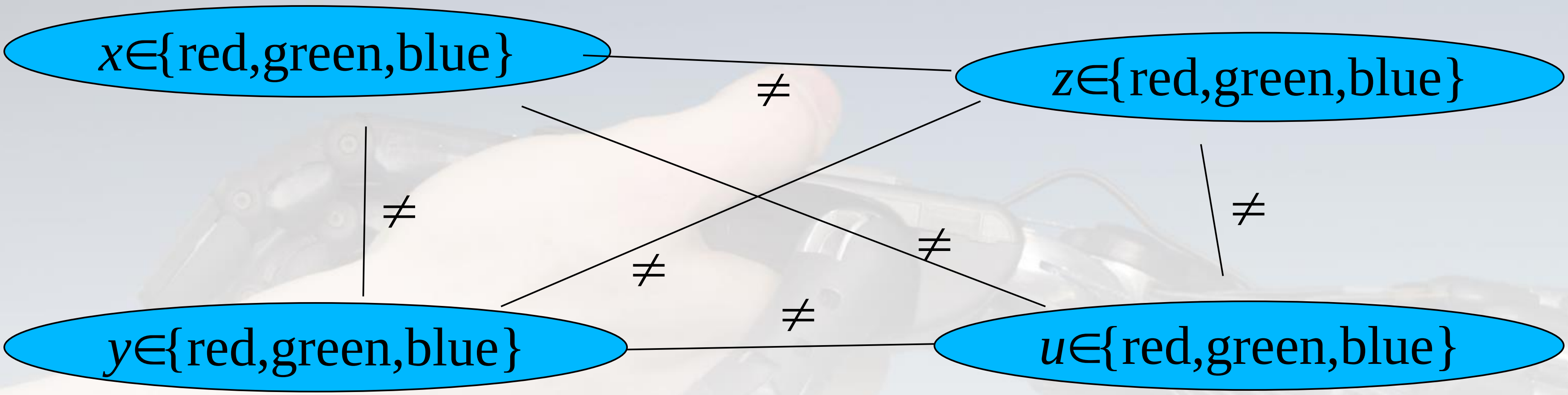
Intuitively: (1-consistency = node consistency); (2-consistency = arc consistency) and (3-consistency = path consistency) provided node consistency is guaranteed

Strong *k-consistency*

The problem is *j-consistent* for all $j \leq k$

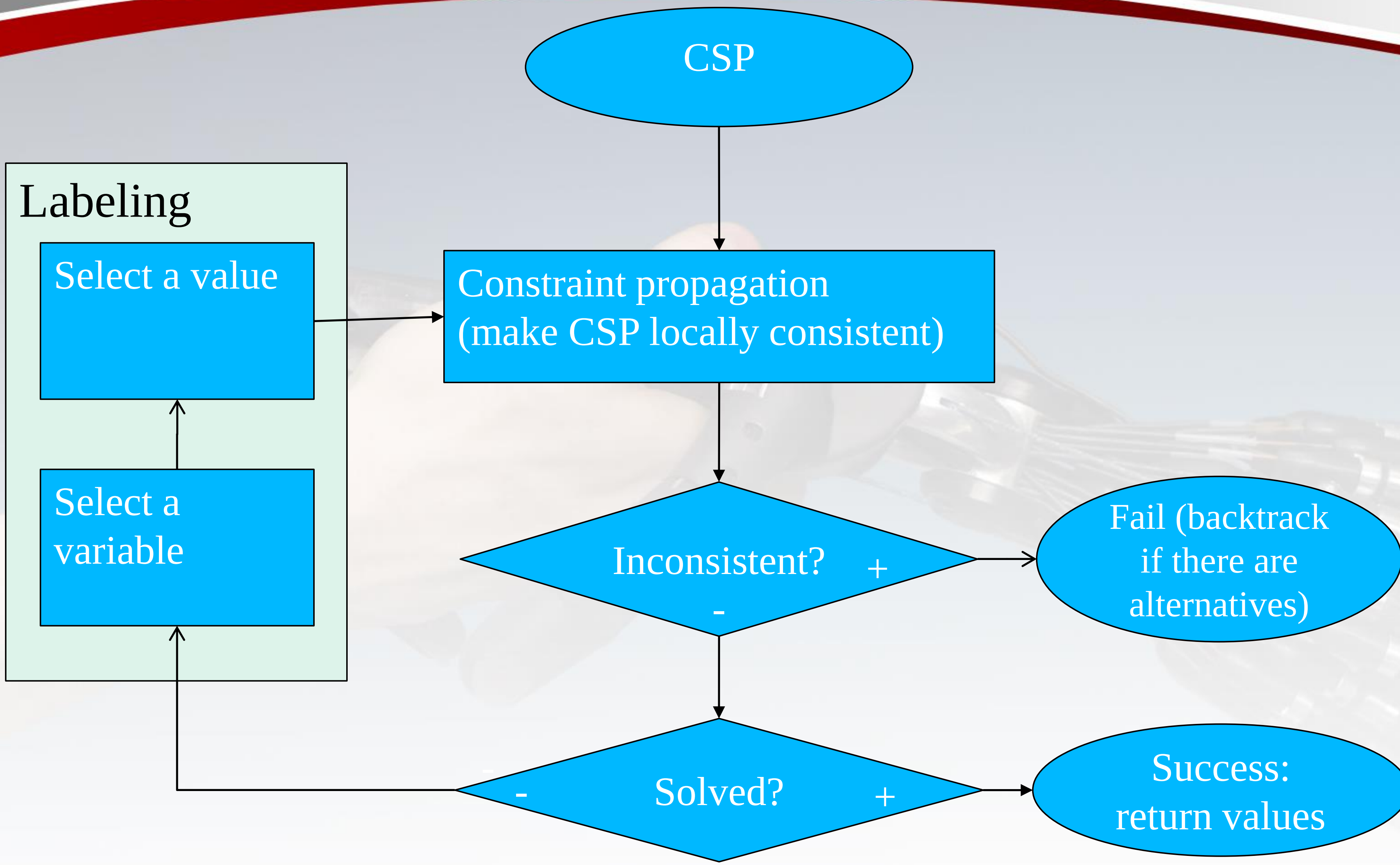
Consistency Concepts and Consistency

The following CSP is strong 3-consistent



- Is strong k-consistency for any k sufficicient for consistency?
- No, just consider complete graph over k+1 variables, where each edge corresponds to inequality constraint and each variable has domain {1,…,k}
- Then, CSP cannot be consistent (only k values for k+1 variables), but is k-consistent (for each (k-1)-assignment, there is a k-th value)

Solving CSPs with Backtracking Search



Local Search for CSPs

Excercise: CSPs and Local Search

- Give a local search formulation for a general CSP
- Given variables with corresponding domains and constraints, define
  - State space
  - Neighborhood
  - Objective function

Solution: CSPs and Local Search

- State space: each complete variable assignment is a state
- Neighborhood: neighbors of a state (variable assignment) are obtained by changing the assignment of an arbitrary variable
- Hence, we have one neighbor for each
  - Variable and
  - each value in the domain of the variable (other than the current one)
- Objective function: value of state (variable assignment) is defined as
  - Number of constraints that are satisfied in state

Excercise: Genetic Algorithm for CSPs

- define a genetic algorithm for a general CSP
- Given variables with corresponding domains and constraints, define
  - Genes
  - Chromosomes
  - Fitness (value) of individuals
  - Mutation operation
- Illustrate reproduction by means of a small example

Solution: Genetic Algorithm for CSPs

- Genes: domain elements of all variables
- Chromosomes correspond to variable assignments
- Fitness: number of constraints that are satisfied
- Mutation operation: change a random gene
- Illustration: see Local Search slides (8Queens, Graph Coloring)

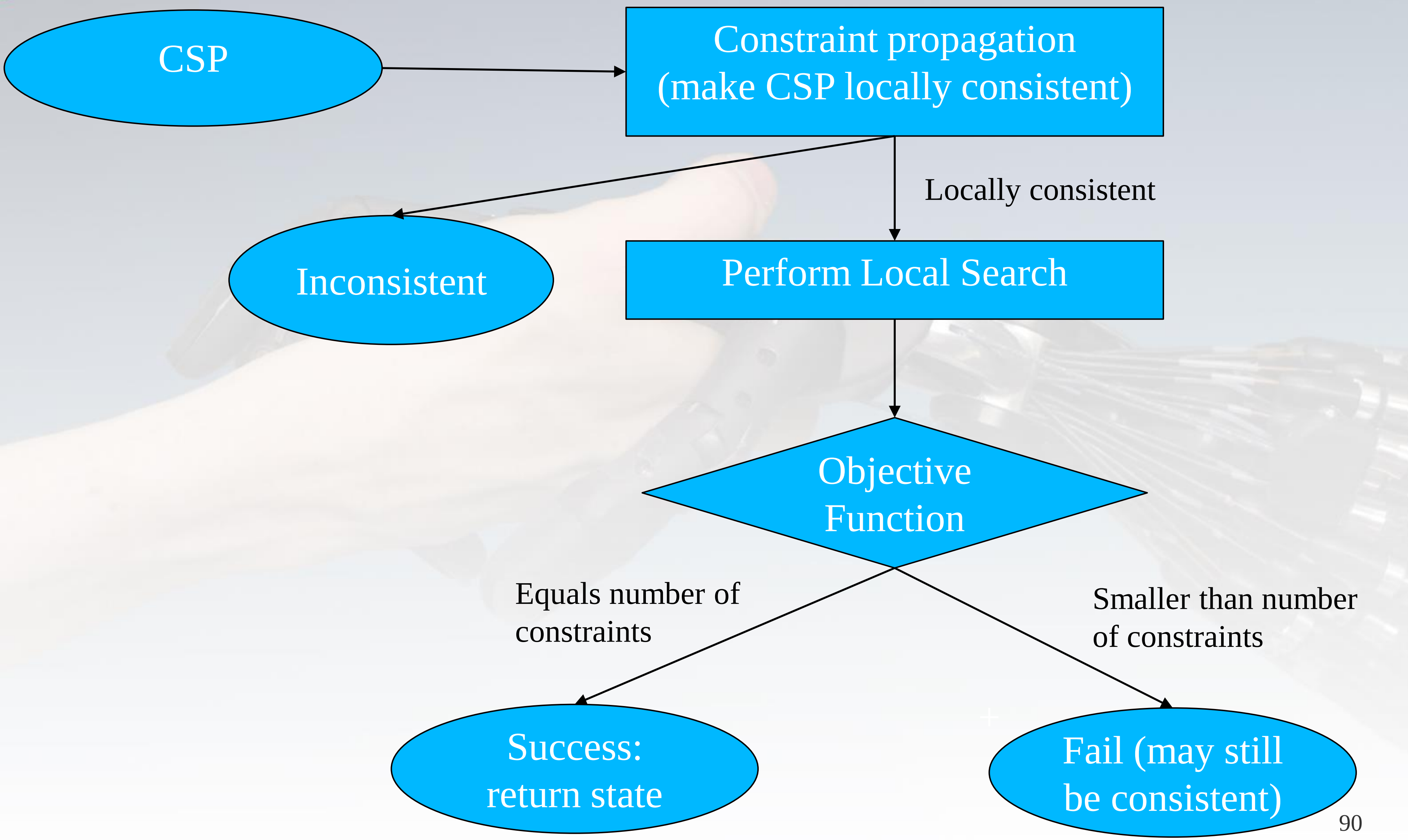
Excercise: Local Search Solution

- What can you conclude if local search algorithm finds
  - Globally optimal solution? ($f(s)$ = number of constraints)
  - Locally optimal solution? ($f(s)$ < number of constraints)

Solution: Local Search Solution

- Globally optimal solution
  - globally optimal solution satisfies all constraints by definition
  - Hence, we found a solution for the CSP (CSP is consistent)
- Locally optimal solution
  - Locally optimal solution violates some constraints
  - The CSP may be inconsistent, but we may also just got stuck in a local optimum
  - We cannot conclude anything in this case
  - However, solution may be sufficient (soft constraints)

Solving CSPs with Local Search



Summary

Summary: CSPs

- A constraint satisfaction problem (CSP) is given by
  - A set of *variables* $\{X_1,\ldots,X_n\}$
  - a *domain* D_i for each variable(the possible values), where the whole space $D = D_1 \times \ldots \times D_n$ is the assignment space
  - A set of *constraints*, i.e. relations $R_k \subseteq D_{k_1} \times \ldots \times D_{k_m}$ for some domains $D_{k_1}, \ldots, D_{k_m}$
- The goal is to find an assignment that satisfies all constraints
- If no such assignment exists, the CSP is called inconsistent (and consistent otherwise)

Summary: Algorithms

- Generate the whole search tree and test
  - usually not practical because of tree size
- Backtracking Search
  - start with empty assignment
  - Systematically extend assignment using heuristics
- Local Search
  - Move through space of complete variable assignment
  - Maximize number of satisfied constraints
- Consistency Concepts
  - can be applied to simplify problem initially (preprocessing)
  - can be applied to simplify problem during backtracking search

Summary: Consistency Concepts

- Node consistency: $\forall x : x \in D_i \rightarrow P_i(x)$
- Arc consistency: $\forall x : x \in D_i \rightarrow (\exists y \in D_j : P_{i,j}(x,y))$
- Path consistency:
  $\forall x \in D_i, z \in D_j : P_{i,j}(x,z) \rightarrow (\exists y \in D_m : P_{i,m}(x,y) \wedge P_{m,j}(y,z))$
- *k*-consistency
  - Any solution for *k*-1 variables can be extended to a solution for *k* variables
- Strong *k*-consistency
  - The problem is *j*-consistent for all $j \leq k$

Further Readings

Most topics of this week can be found in:

Russell, S., Norvig, P. Artificial Intelligence - A modern approach. Pearson Education: 2010.

More details on representing and solving CSPs can be found in:

Dechter, R. Constraint processing. Morgan Kaufmann: 2003.

Rossi, F., Van Beek, P., & Walsh, T. Handbook of constraint programming. Elsevier: 2006.