

## CONSTRAINT SATISFACTION PROBLEMS

- Standard search problem: the state is anything that supports goal test, comparison, successor.
- CSP: the state is defined by **variables**  $V_i$  with values from **domains**  $D_i$ .
  - The **goal test** is a set of **constraints**, which specifies allowable combinations of values for subsets of variables. A state is a set of variable bindings.

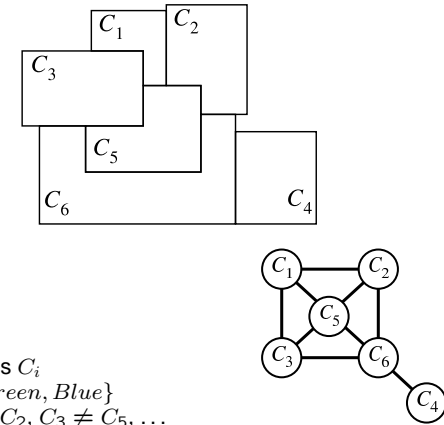
```

Variables { Shoes, Pants, Shirt }
Domains  { {SandalsRunners}, {Jeans, Blue, Grey}, {Green, White} }
Constraints { (Shoes = Sandals, Pants = Grey),
              (Shoes = Runners, Pants = Jeans),
              (Shoes = Sandals, Shirt = Green),
              (Shoes = Runners, Shirt = White),
              (Pants = Grey, Shirt = Green),
              (Pants = Jeans, Shirt = White),
              (Pants = Blue, Shirt = White) }
    
```

- This is actually an example of **binary CSP**.

## EXAMPLE: MAP COLOURING

- Colour a map so that no adjacent countries have the same colour.



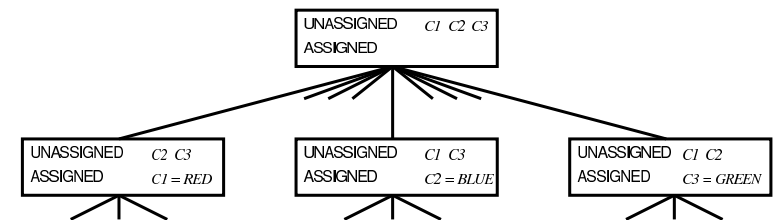
- Variables: Countries  $C_i$
- Domains:  $\{Red, Green, Blue\}$
- Constraints:  $C_1 \neq C_2, C_3 \neq C_5, \dots$

## REAL-WORLD CSP

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Hardware configuration
- Spreadsheets
- Transportation scheduling
- Factory scheduling

## SOLVING CSP BY STANDARD SEARCH METHODS

- States are defined by the variables bound so far.
  - Initial state** All variables unbound.
  - Operators** Bind one variable
  - Goal test** All variables assigned, no constraints violated



- Disadvantages?

## IMPROVING THE CSP ALGORITHM

- Order of assignment is irrelevant (many paths are equivalent)
- Further bindings cannot correct an already violated constraint
- We can use depth-first search, but
  - Fix the order of assignment
  - check for constraint violations
    - \* at the SUCCESSORS level, or immediately before expanding the state.

```
function CSP-SEARCH() returns a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE))
  loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST(node) then return node
    unless VIOLATES-CONSTRAINTS(node) do
      nodes ← APPEND(SUCCESSORS(node), nodes)
  end
```

We do not need the queue actually (why?)

## BACKTRACKING

- We fix the order of assignment, and we check for constraint violations.
- The resulting algorithm is called **backtracking**, the basic uninformed algorithm for CSP. Can solve  $n$ -queens for  $n \approx 15$ .

```
function BACKTRACKING( state, variables, domains) returns a solution, or failure
  var ← FIRST(variables)
  domain ← FIRST(domains)
  foreach val in domain do
    unless VIOLATES-CONSTRAINTS(ADD((var, val), state)) do
      if no more variables
        then return state
      else BACKTRACKING(ADD((var, val), state), REST(variables), REST(domains))
  end
  return failure
```

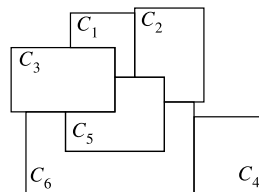
- Variant: **forward checking**, looks ahead and erases from the domains of all the variables those values that cannot be assigned without violating constraints.
  - Forward checking is a particular case of **arc consistency** working on the graph generated by the constraints.

## HEURISTICS FOR CSP

- We can make more intelligent decisions on
  - which value to choose for each variable
  - which variable to assign next

\* Given  $C_1 = Red$  and  $C_2 = Green$ ,  $C_3 = ?$

\* Given  $C_1 = Red$ ,  $C_2 = Green$ , what next?

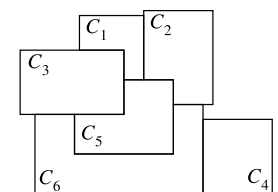


## HEURISTICS FOR CSP

- We can make more intelligent decisions on
  - which value to choose for each variable
  - which variable to assign next

\* Given  $C_1 = Red$  and  $C_2 = Green$ ,  $C_3 = ?$   
 $C_3 = Green$ , **the least constraining value**

\* Given  $C_1 = Red$ ,  $C_2 = Green$ , what next?  
Choose  $C_5$ , **the most constrained variable**



- Can solve  $n$ -queens for  $n \approx 1000$