University of Strathclyde Glasgow

CS310 – AI Foundations

Andrew Abel

February 2023

Week 6: Constraint Satisfaction Problems

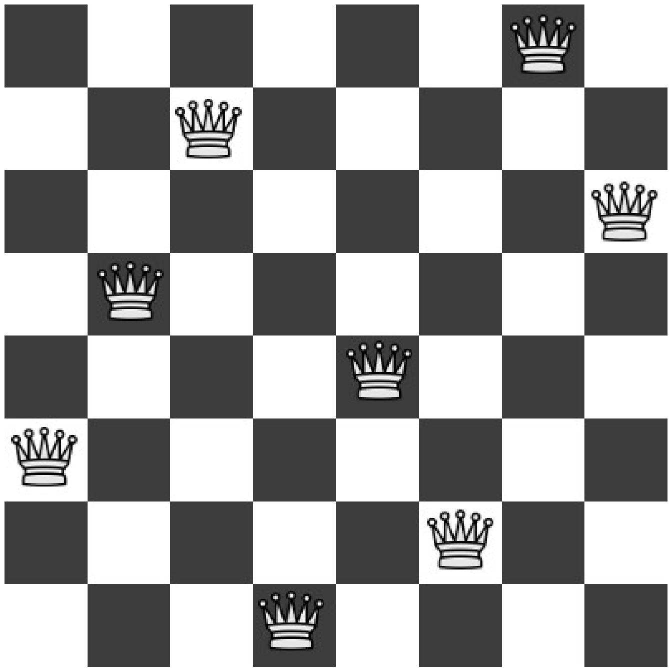
Welcome!

- Constraint Satisfaction Problems
  - This class starts out heavy but
  - A) It will make sense by the end!
  - B) Its totally worth it, its actually very interesting
  - C) You might not believe B at first, but it really is

Constraint Satisfaction Problems

Constraint Satisfaction

- Goal: find ”admissible” valuation for given set of variables
  - Given some parameters/variables,
  - Try to find solutions that meet conditions
- Example: The 8-queens problem
  - How to distribute 8 queens on a chess board such that none of them attack each other?
- Remarks:
- like planning, constraint satisfaction problems use a factored representation of states
- we take state structure into account when searching for a solution!
- Rather than seeing state as a “black box”, we now take it into account
  - general-purpose heuristics



Constraint Satisfaction Problem (formally)

- Previous slide was the "informal" definition

- A CSP consists of three components:

  - a set of variables $X = X_1, \ldots, X_n$,

  - a set of domains $D = D_1, \ldots, D_n$ (one for each variable)
    - Domain here refers to a set of possible values

  - a set C of constraints that specify allowable combinations of values.

What does a constraint look like?

- A constraint C consists of a pair < scope, rel > where:
- Scope is a tuple of variables that participate in the constraint
- Rel is a relation that defines the values that those variables can take on
- Rel can be either given explicitly or using operators, e.g. suppose two variables X_1 and X_2 both have domain {A,B}, then the constraint that X_1 is not equal to X_2 can be written in either of the following ways:
  - $< \{X_1,X_2\}, X_1 \neq X_2 >$
  - $< \{X_1,X_2\}, \{(A,B),(B,A)\} >$
    - We omit (A,A), (B,B) since they would be equal
- Often better to use operators
- We often specify the constraint by only stating the relation rel - in the example above we would simply write $X_1 \neq X_2$

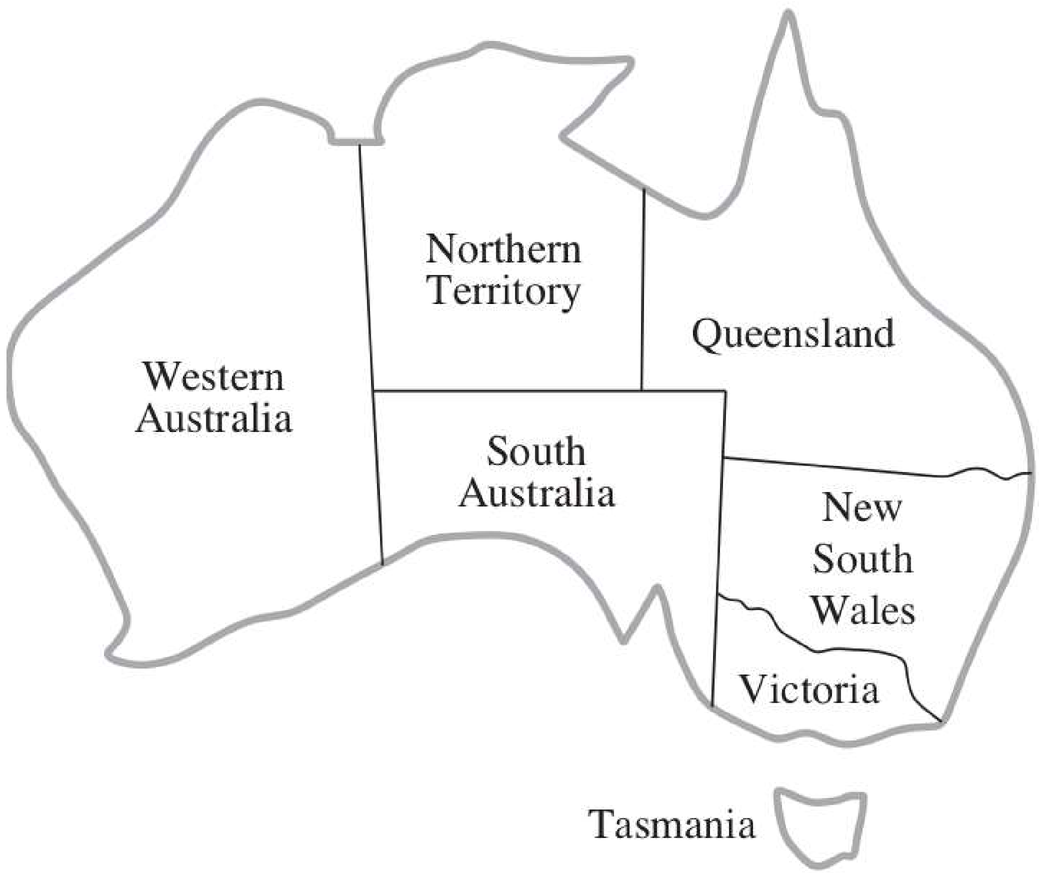
What does a solution look like?

- Given a collection $C_1,\ldots C_n$ of constraints,
  - an assignment is a map $\{ X_1 := v_1,\ldots, X_k := v_k\}$ where $X_1,\ldots, X_k$ are variables occurring in $C_1,\ldots C_n$ and for each X_i we have $v_i \in D_i$
    - Each variable assigned a value, and each value must be in the variable domain
  - a complete assignment is an assignment that provides values for all variables in $C_1,\ldots C_n$
    - i.e. all variables in our problem
    - E.g. 8 queens problem, complete assignment is one where each queen specifies a position
  - a solution is a complete assignment that ``satisfies all constraints'', ie. for each constraint $< \{X_1,\ldots,X_m\}, R(X_1, \ldots,X_m) >$ we have
  - R = the relationship, i.e. not equals from previous slide

$$(v_1,\ldots v_m) \in R$$

Example: Colouring Australia

- We want to use 3 colours to colour in map
- No two adjacent states can have same colour



Example: Colouring Australia

- Variables
- X = {WA, NT , Q, NSW , V, SA, T }

- same domain for all variables
- D={green, red, blue}

Example: Colouring Australia

- Constraint
- Adjacent tiles can not be the same colour
- constraints: C = {SA ≠ WA, SA ≠ NT , SA ≠ Q, SA ≠ NSW , SA ≠ V, WA ≠ NT , NT ≠ Q, Q ≠ NSW , NSW ≠ V }

Example formalised

- Variables X = {WA, NT , Q, NSW , V, SA, T }
- same domain for all variables D={green, red, blue}
- constraints: C = {SA ≠ WA, SA ≠ NT , SA ≠ Q, SA ≠ NSW , SA ≠ V, WA ≠ NT , NT ≠ Q, Q ≠ NSW , NSW ≠ V }

- A solution:
- {SA = green, WA = red, NT = blue}, Q = red, NSW = blue, V = red, T=green}.

8 Queens Problem Solution

Variables: $C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8$

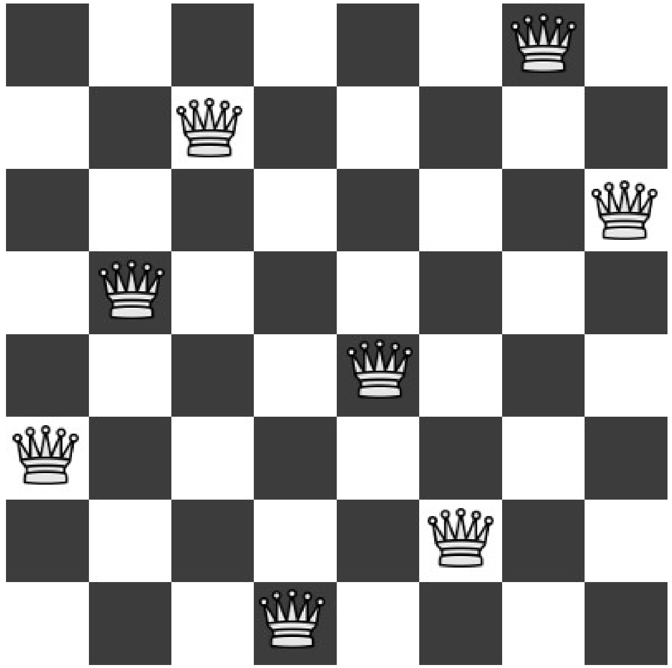
(each variable representing a column)

Domains: $D_1 = D_2 = \ldots D_8 = \{1,8\}$

(position on each column)

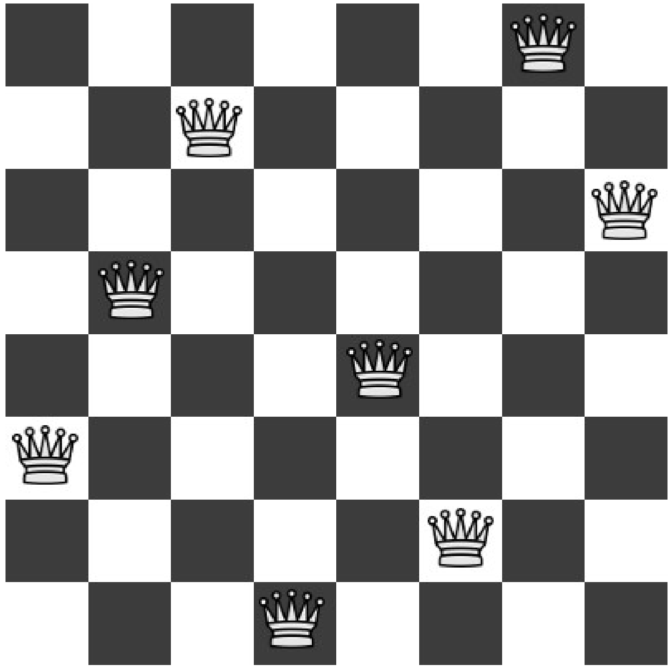
Constraints: $C_i \neq C_j$ for $i \neq j,\ i < j$

$|\ C_i - C_j\ | \neq j - i$ for $i \neq j,\ i < j$



Constraint Satisfaction

- Example: The 8-queens problem
  - How to distribute 8 queens on a chess board such that none of them attack each other?



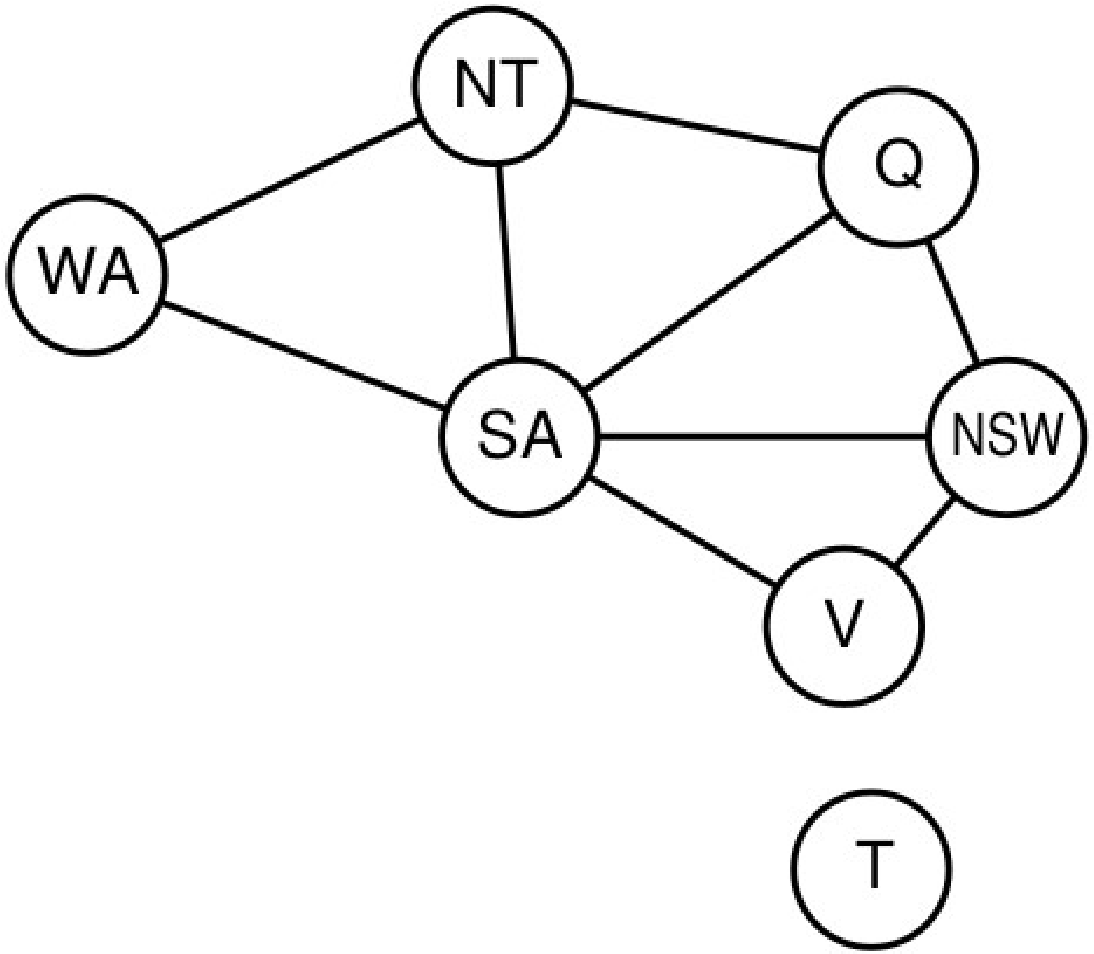
8 Queens Solution

- Have a think about it!

Structure of a CSP

The Constraint Graph

- To analyse a CSP we consider its constraint graph
- Depicts dependences between variables
- variables are nodes
- an edge indicates that the variables occur in a constraint
- This is still a map of Australia!
  - Variables are connected if they are participating in a constraint



Example: 2+2=4

- Famous arithmetic problem
- Solve for each letter between 0 and 9
- Two plus Two becomes 4 if we substitute numbers for letters!
- Each number must represent a different value

$$
\begin{array}{cccc}
 & T & W & O \\
+ & T & W & O \\
\hline
F & O & U & R
\end{array}
$$

Example: 2+2=4

- Variables are the letters. Constraints are as follows:

$$
\begin{array}{ccccc}
 & & T & W & O \\
+ & & T & W & O \\
\hline
 & F & O & U & R
\end{array}
$$

- Variables: T,W, O, F, U, R
- Domain: D = {0,..,9}

- We now have what looks like a set of obvious constraints
- O + O = R
- W + W = U
- T + T = O
- F = ?

Example: 2+2=4

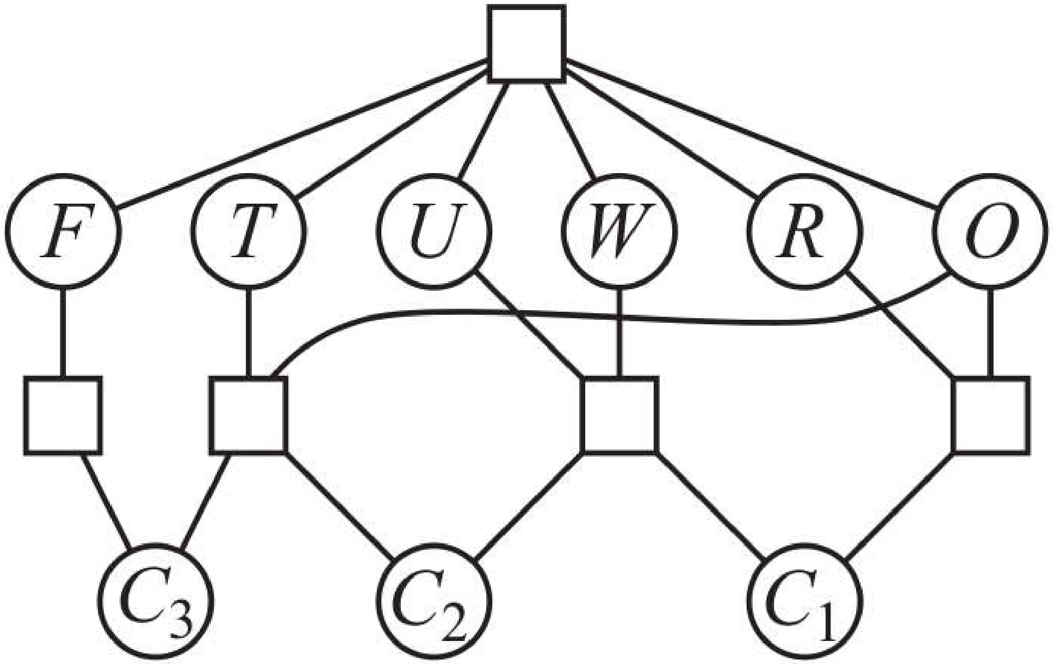
- Constraint example: O + O = R?
- But what if O≥5
  - R must be in range {0…9}
  - So we need to carry the 10!

$$
\begin{array}{cccc}
 & T & W & O \\
+ & T & W & O \\
\hline
F & O & U & R
\end{array}
$$

- $O + O = R + 10\ C_1$
- $C_1 + W + W = U + 10\ C_2$
- $C_2 + T + T = O + 10\ C_3$
- $C_3 = F$
- plus an *Alldiff* constraint that all letters represent different values!

Constraint Hypergraph

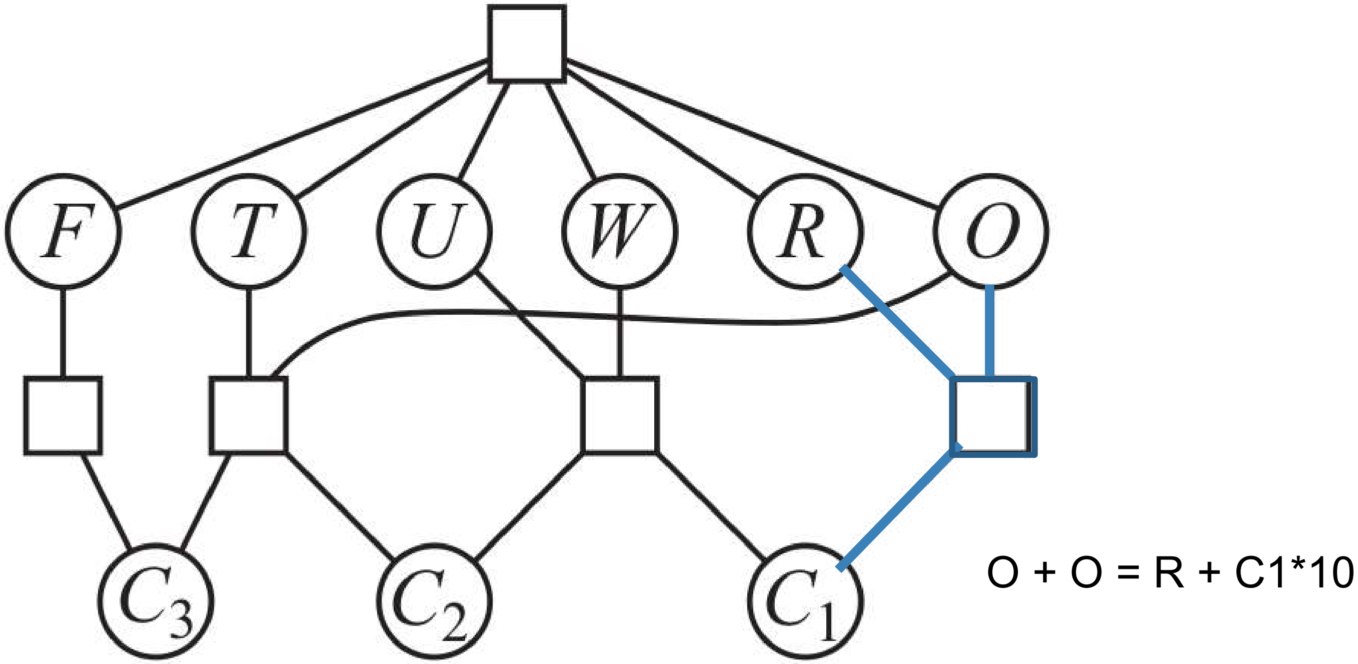
- Hypergraph has ``hyperedges'' that connect more than two nodes:



- Non-binary constraints can always be replaced by binary ones (cf. Norvig & Russell 6.1.3).

Constraint Hypergraph

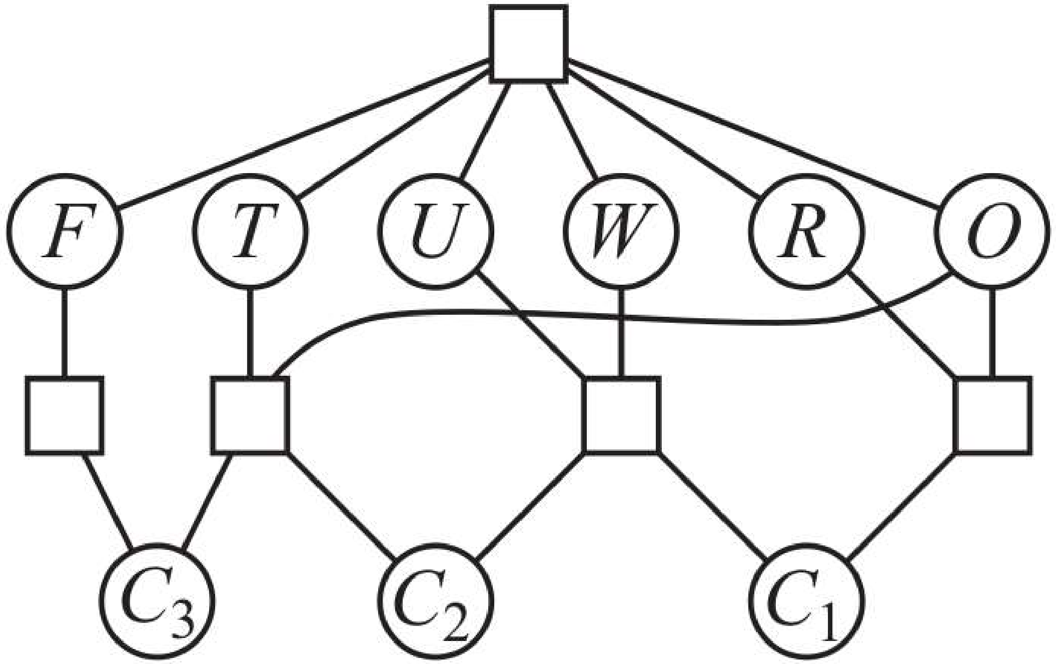
- Hypergraph has ``hyperedges'' that connect more than two nodes:



- Non-binary constraints can always be replaced by binary ones (cf. Norvig & Russell 6.1.3).

Constraint Hypergraph

- In this Graph, all letter variables are connected to each other via the box at the top of the graph (the hyperedge)
  - This is the “alldiff” constraint
- $C_2 + T + T = O + 10\ C_3$
  - C_2, T, O, C_3 all participating in constraint
  - All connected
- Can this help us solve problems more efficiently?



Types of constraints

- a unary constraint is a constraint involving only one variable
  - i.e. $x < 5$, only variable mentioned is x
- a binary constraint is a constraint involving two variables
  - i.e. $X_3 > X_4$
- a higher-order constraint is a constraint involving more than two variables
- We don’t need all the hyper edges and graphs
  - Any CSP can be written into a CSP involving only binary constraints!

Binary constraints suffice

- Any CSP can be rewritten into a CSP involving only binary constraints.
- Remove unary constraints by restricting the domain of the relevant variable.
- for each higher-order constraint $R(X_1, \ldots ,X_n)$
  - introduce a new variable Y, and
  - define its domain D_Y by
    - $D(Y) = \{ (d_1,\ldots,d_n) \mid R(d_1,\ldots,d_n) \text{ is true.}\}$
- for each X_i add the constraint that $\pi[Y] = X_i$, i.e., the ith component of the value of Y has to be equal to X.
- Remove the higher-order constraint $R(X_1,\ldots ,X_n)$.

Binary constraints suffice

- Remove unary constraints by restricting the domain of the relevant variable.
  - So if $X_1 < 5$, just restrict D_1 to {0,..4} and remove constraint
- for each higher-order constraint $R(X_1, \ldots, X_n)$
  - introduce a new variable Y, and
  - define its domain D_Y by
    - $D(Y) = \{ (d_1,\ldots,d_n) \mid R(d_1,\ldots,d_n) \text{ is true.} \}$
- for each X_i add the constraint that $\pi[Y] = X_i$, i.e., the ith component of the value of Y has to be equal to X.
- Remove the higher-order constraint $R(X_1,\ldots ,X_n)$.

Binary constraints suffice

- for each higher-order constraint $R(X_1, \ldots, X_n)$
  - Here we have n variables, and R is the relationship that should be satisfied for those n variables
  - This is a form of constraint
  - introduce a new variable Y
    - define its domain D_Y exactly by the tuples
    - $D(Y) = \{ (d_1,\ldots,d_n) \mid R(d_1,\ldots,d_n) \text{ is true.}\}$
  - We only keep the combination of the domain elements where the relation holds
  - Result is a set of elements d_1 to d_n which satisfy this constraint
  - Have a new variable Y with a specific domain, the exact combinations allowed!

Binary constraints suffice

- for each X_i add the constraint that $\pi[Y] = X_i$, i.e., the ith component of the value of Y has to be equal to X.
  - Whenever we have values for X_i and a value for Y
  - Then the X_i's are exactly the ith component of the value for Y
  - Xi will satisfy the relation, because all Xi's will create an element of Y and the elements of Y satisfy the constraints
    - $\pi[Y] = X_i$

- Remove the higher-order constraint $R(X_1,\ldots,X_n)$.
  - By using Y and restricting domain, we have only binary constraints

Real-world examples

- 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
- Floorplanning
  - Layout of buildings
- Notice that many real-world problems involve real-valued variables

Looking for a solution I : Propagating Constraints

Key idea

- Solving the problem!
  - Cut down on the search space
- focus on ``local consistency''
  - Look at one constraint at a time
  - Remove values that will violate the constraints we are trying to satisfy
- simplest form is node consistency: remove values from domain D_i that obviously violate a constraint that X_i is involved in
  - E.g. if $X_i < 5$ and d = {2,3,4,5,6,7,8} , can remove 5,6,7,8
- arc-consistency – where 2 variables are involved

- path-consistency, k-consistency - covering whole path, not covered here!
  - see Section 6.2 in Norvig & Russell

Arc-consistency

- consider binary constraints (ie constraints with ≤ 2 variables)
- a variable is arc-consistent if every value in its domain satisfies the variable's binary constraints
- A network is arc-consistent if every variable is arc-consistent with every other variable.
- Concretely: X_i is arc-consistent with respect to X_j if for every value in $d_i \in D_i$ there exists a value in $d_j \in D_j$ such that the pair d_i,d_j satisfies the constraint on (X_i,X_j).
- Example: Let X and Y be variables with domain {1,…, 10} and consider the constraint $X^2=Y$. Then we have to restrict the values of X to {1,2,3} to make X arc-consistent.

Arc-consistency

- Example: Let X and Y be variables with domain {1,…, 10} and consider the constraint $X^2=Y$. Then we have to restrict the values of X to {1,2,3} to make X arc-consistent.
- So now X is arc-consistent with Y
  - But now we have to make Y arc consistent with X
  - So we have to restrict values of Y to {1,4,9}
- Imagine now we have a whole graph of connected nodes…
- How can we achieve overall arc-consistency for whole program

The AC-3 Algorithm

- Taken from Russel and Norvig

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

The AC-3 Algorithm

- Returns true if it looks like a solvable problem
- Returns false if an inconsistency is found, i.e. it can't be solved

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
   inputs: csp, a binary CSP with components (X, D, C)
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (X_i, X_j) ← REMOVE-FIRST(queue)
      if REVISE(csp, X_i, X_j) then
         if size of D_i = 0 then return false
         for each X_k in X_i.NEIGHBORS - {X_j} do
            add (X_k, X_i) to queue
   return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
   revised ← false
   for each x in D_i do
      if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
         delete x from D_i
         revised ← true
   return revised
```

The AC-3 Algorithm

- Returns true if it looks like a solvable problem
- Returns false if an inconsistency is found, i.e. it can't be solved
- Input is the CSP
- Queue of arcs created
- Arc is a directed connection between 2 variables
    - i.e. pair X -> Y
    - Pair Y -> X

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X,\ D,\ C)$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, X_i, X_j) **then**
      if size of $D_i\ =\ 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to $queue$
  return $true$

---

function REVISE(csp, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised $\leftarrow$ *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised $\leftarrow$ *true*
  return *revised*

The AC-3 Algorithm

- Remove each arc and check it
- Revise function removes any inconsistent values
- Check if the domain is empty
  - If empty, no possible values, so inconsistent
- If revised, have to add anything that is connected to X_i back into the queue

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X,\ D,\ C)$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, $X_i,\ X_j$) **then**
      if size of $D_i\ =\ 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to $queue$
  return $true$

---

function REVISE(csp, $X_i,\ X_j$) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

The AC-3 Algorithm

- If algorithm produces at least one empty domain the CSP is unsolvable!
- Next, more detailed algorithm and an example!

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REVISE(csp, Xi, Xj) then
      if size of Di = 0 then return false
      for each Xk in Xi.NEIGHBORS - {Xj} do
        add (Xk, Xi) to queue
  return true
```

```
function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
  revised ← false
  for each x in Di do
    if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
      delete x from Di
      revised ← true
  return revised
```

The AC-3 Algorithm (simpified)

We had a queue of arcs…

1. Turn each binary constraint into two arcs e.g. $X_1 < X_2$ becomes (X_1,X_2) and (X_2,X_1).
   1. Important to have both directions, X_1 is constrained by X_2, and X_2 is also constrained by X_1
2. Add all arcs to an agenda.
3. Repeat until agenda empty:
   - take an arc (X_i,X_j) of the agenda and check it
   - for every value of D_i there must be some value of D_j
   - remove any **inconsistent** values from D_i
   - If domain of X_i has changed, add all arcs of the form (X_k,X_i) to the agenda
     - Only add the ones that are not there already

- This procedure can be done for pre-processing or after each (partial) assignment.
  - If we do partial assignment and then check and find inconsistencies, then we know our partial assignment does not work!

Complexity of AC-3

- assume n variables, domain size at most d and value $c=n^2$
  - c is measure of number of possible arcs in problem
- checking one arc for consistency can be done in $O(d^2)$
  - For every element in domain, we need to find an element in the other domain
- checking all arcs once is $O(cd^2)$
- an arc can be added at most d times to the queue
- -> overall worst-case complexity is $O(cd^3)$.

Example

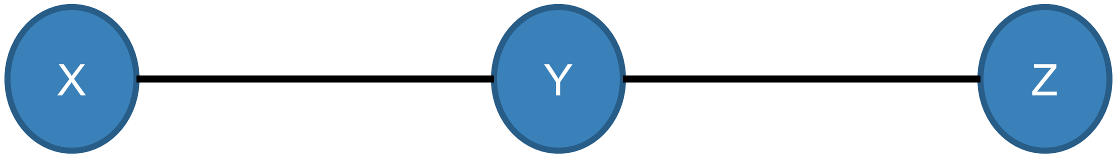
- $D_X=D_Y=D_Z= \{1,2,3,4\}$
- $X > Y$ and $Y > Z$

1. Draw the constraint graph!
2. Make this problem arc-consistent via AC-3.
3. Solve the remaining instance.

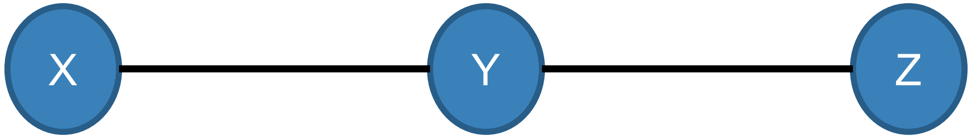
AC3 – Algorithm Example, 1 – Draw Constraint Graph

- Variables: X, Y, Z
- Domains $D_X = D_Y = D_Z = \{1,2,3,4\}$
- Constraints X > Y, Y > Z

- Constraint graph



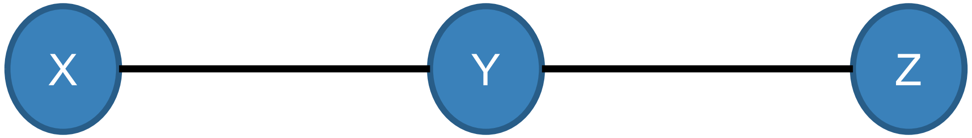
AC3 – Algorithm Example, 2 – Make problem arc-consistent



- Add all arcs to agenda
- Agenda = { (X,Y), (Y,X), (Y,Z), (Z,Y) }

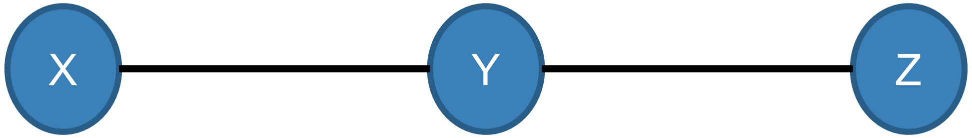
- Remove (X,Y): X>Y
  - D_X = { 2,3,4}   (Removed 1)
  - Add (Y,X) to agenda **if not there** (it is, so we do not!)
  - Agenda = { (Y,X), (Y,Z), (Z,Y) }
- Remove (Y, X): Y<X
  - D_Y = {1, 2,3}   (4 is not smaller than anything, so we remove it)
  - Add (X,Y) to agenda **if not there**
  - Agenda = { (Y,Z), (Z,Y), (X,Y) }

AC3 – Algorithm Example, 2 – Make problem arc-consistent



- Remove (Y,Z): Y>Z
  - D_Y= { 2,3}   (Removed 1)
  - Add (Z,Y) to agenda **if not there**
  - Agenda = { (Z,Y), (X,Y) }
- Remove (Z, Y): Z<Y
  - D_Z = {1, 2}
  - Add (Y,Z) to agenda **if not there**
  - Agenda = { (X,Y), (Y,Z) }
- Remove (X,Y): X>Z
  - DX = {3,4}
  - Add (Y,X) to agenda **if not there**
  - Agenda = {(Y,Z), (Y,X)}

AC3 – Algorithm Example, 2 – Make problem arc-consistent



- Remove (Y,Z): Y>Z
  - D_Y= { 2,3}   (No change)
  - Agenda = { ((Y,X) }
- Remove (Y,X): Y<X
  - D_Y= { 2,3}   (No change)
  - Agenda = { } (empty)
- Algorithm Terminates:
  - D_X = {3,4}, D_Y = {2,3}, D_Z = {1,2}

Example

- Another excellent video by John Levine
- Runs through AC-3 Algorithm as a detailed example

Summary

- This week was about learning to solve problems!
- Some of the theory was a bit heavy
    - Unary, Binary, and Higher-order constraints
    - Hypergraphs
    - Theory of binary constraints
- Constraint satisfaction with the AC3 Algorithm
    - Very useful way of solving problem
- Some very real applications
    - Useful for programming!