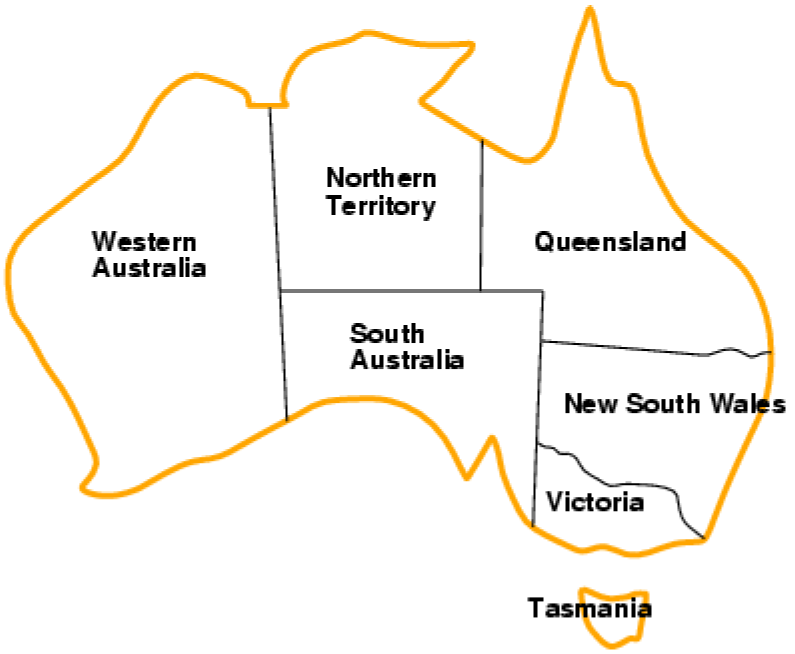
Constraint Satisfaction Problems

Constraint satisfaction problems (CSPs)

- A **constraint satisfaction problem** (or CSP) is defined by:
  - a set of **variables**, $X_1, X_2, \ldots, X_n$,
  - A set of domains, $D_1, D_2, \ldots, D_n$
  - a set of **constraints**, $C_1, C_2, \ldots, C_m$.
- An assignment that does not violate any constraints is called a **consistent** or legal assignment.
- A complete assignment is one in which every variable is mentioned.
- A **solution** to a CSP is a complete assignment that satisfies all the constraints.

Example: Map-Coloring

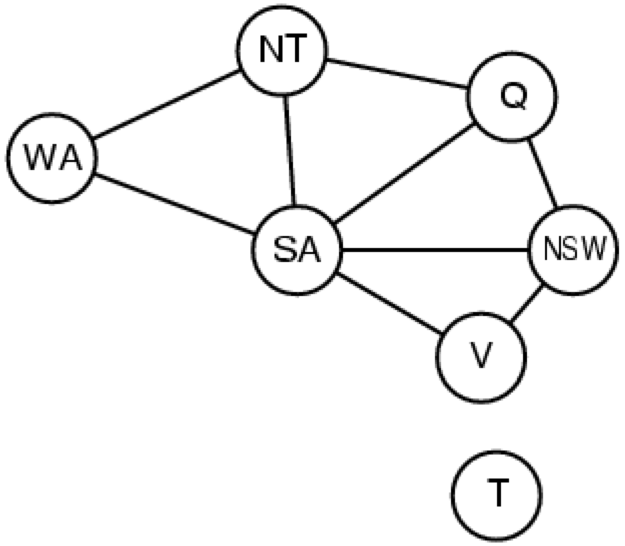


- Variables *WA, NT, Q, NSW, V, SA, T*
- Domains D_i = {red, green, blue}
- Constraints: adjacent regions must have different colors

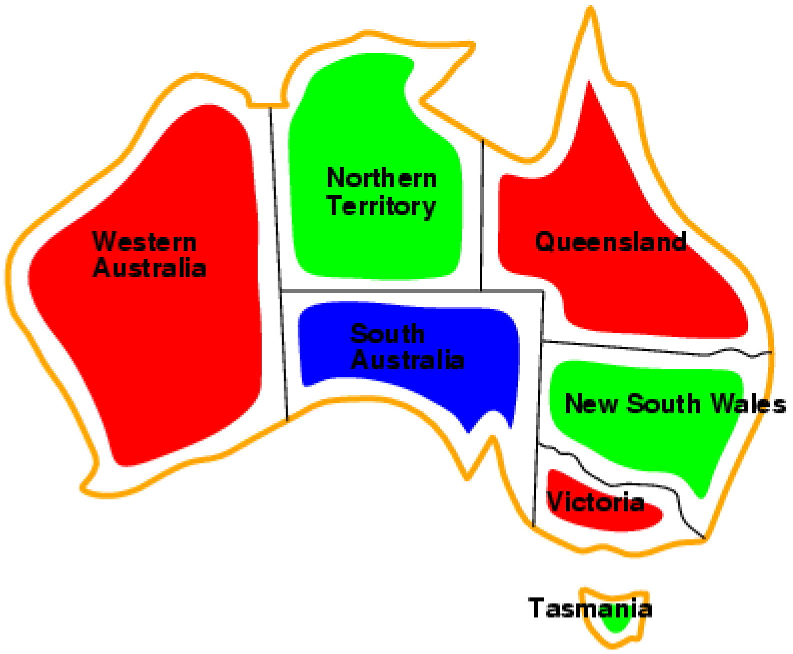
- e.g., WA ≠ NT,
- or (WA,NT) in {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}

Constraint graph

- Binary CSP: each constraint **relates** two variables
- Constraint graph: nodes are variables, arcs are constraints



Example: Map-Coloring



- Solutions are complete and consistent assignments, e.g.,
- WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
  - can be dealt with by filtering the domain of involved variables

- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA

- Higher-order constraints involve 3 or more variables

CSP as a search problem

- State: is defined by an **assignment** of variables X_i with values from domain D_i
- Initial state: the empty assignment {}, in which all variables are unassigned.
- Successor function: a value can be assigned to any unassigned variable, provided that **it does not conflict** with previously assigned variables.
- Goal test: the current assignment is complete. i.e. all variables are assigned values complying to the set of constraints
- Path cost: a constant cost (e.g., 1) for every step.

CSP as a search problem (cont.)

- Every solution must be a complete assignment and therefore appears at depth ***n*** if there are ***n*** variables.
- Furthermore, the search tree extends only to depth ***n***.
- For these reasons, depth-first search algorithms are popular for CSP's.
- It is also the case that *the path by which a solution is reached is irrelevant.*

Standard search formulation (terrible) problem

Let's try a classical search on a CSP.

Suppose $|D_1| = |D_2| = \ldots = |D_n| = d$

Something terrible: the branching factor at the top level is $b=n*d$, because any

of the d values can be assigned to any of n variables.

At the next level, the branching factor is $(n-1)*d$ (In the worst case, e.g. when there aren't constraints at all)

…

We generate a tree with $n!*d^n$ leaves although there are d^n possible assignments!

Backtracking search

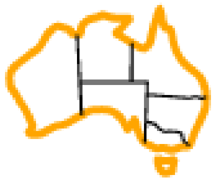
- Variable assignments are commutative, i.e.,
  [WA = red then NT = green] same as
  [NT = green then WA = red]

- So, we only need to consider **assignments to a single variable at each search node**
  - → $b = d$ and there are d^n leaves generated
    (Again, this is in the worst case, e.g. when there aren't constraints at all)

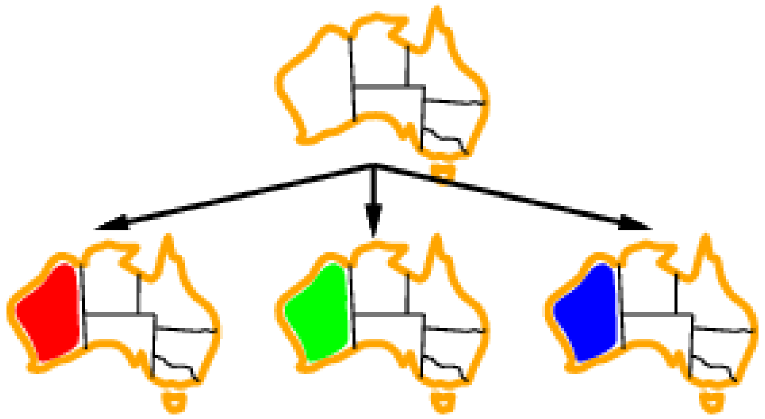
- Depth-first search for CSP 's with single-variable assignments is called backtracking search
  - Backtracking search is the basic algorithm for CSP 's

- Recall, we are looking for **any** solution
  - all the solutions are at the same depth and hence the same cost.
  - Now, by picking a good order to try the variables, we can drastically cut down search.

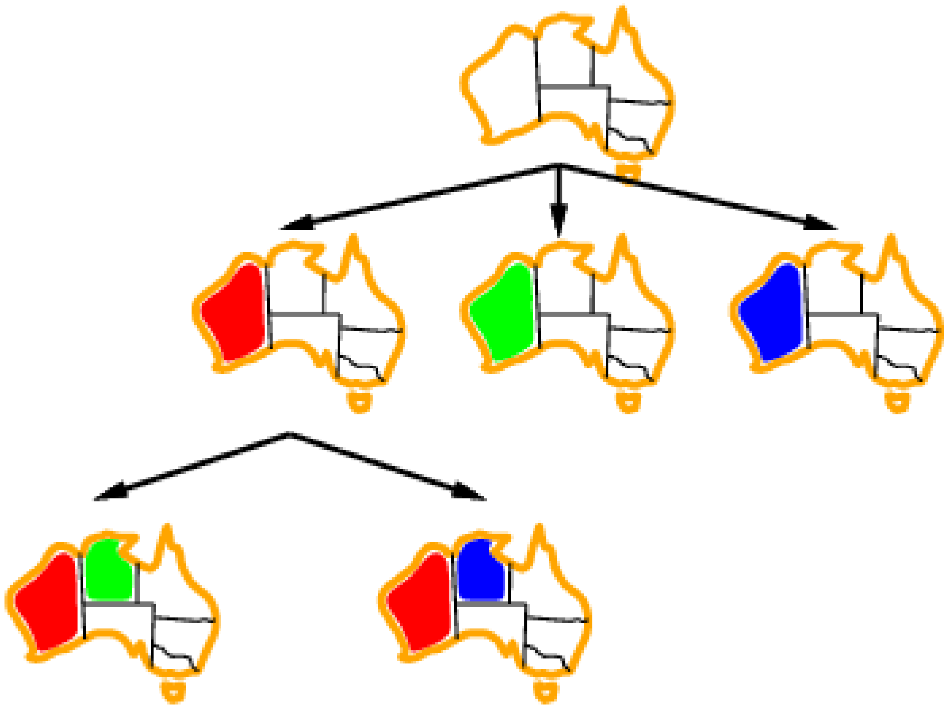
Depth-first search for CSP example



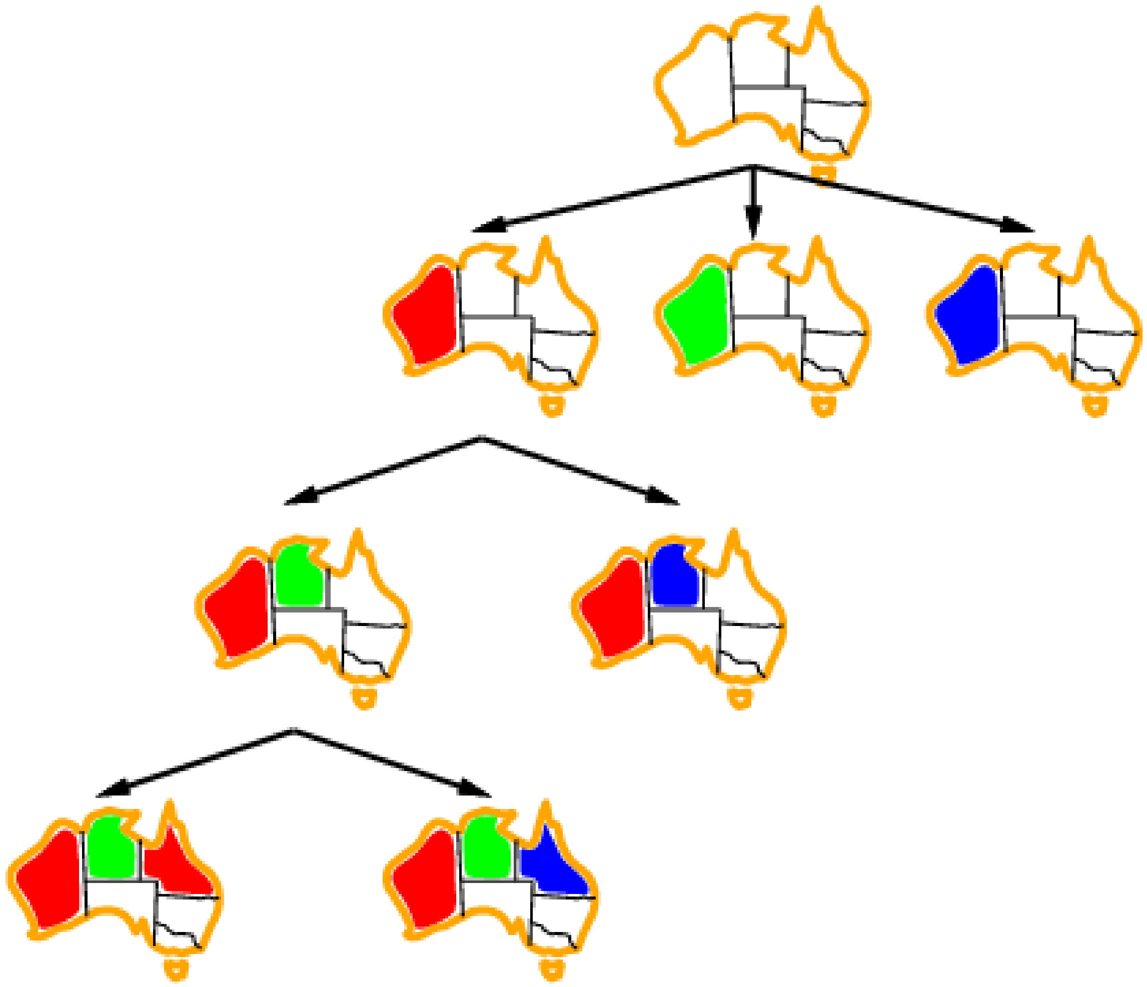
Depth-first search for CSP example



Depth-first search for CSP example



Depth-first search for CSP example



Depth-First Search for CSP

The frontier will be implemented by a **stack**. We also change the terminology a bit:

problem = csp

node = assignment

initialstate = {} //empty assignment is initial state

```
DFS-Search(frontier)
    assignment = MakeNode({});
    Insert(frontier, assignment);
    do
        if ( Empty(frontier) )
                return failure;
        assignment = Remove(frontier);
        if ( assignment is complete )
                return assignment;
        InsertAll (frontier, Expand(assignment) );
    while (true);
```

A closer look at **Expand(assignment, csp)**

- To **expand** an assignment means to assign a value at an unassigned yet variable complying to the constraints.

- **Which unassigned variable we pick** for assignment turns out to be very important in quickly finding a solution.

- In other words, the order of the list of successors returned by **Expand(…)** is very important in quickly finding a solution.

- In the next slide, we show a recursive version of DFS, where the above mentioned order is explicit.

Backtracking search

```
BacktrackingSearch()
    return Backtrack({});

Backtrack(assignment)
    if( isComplete(assignment) )
        return assignment;
    X = SelectUnassignedVariable(assignment);
    for(x : domain of X)
        if(isConsistent(X,x,assignment))
            assign(X,x,assignment);
            result = Backtrack(assignment);
            if(result!=null)
                return result;
            assignUndo(X,assignment);

    return null;
```

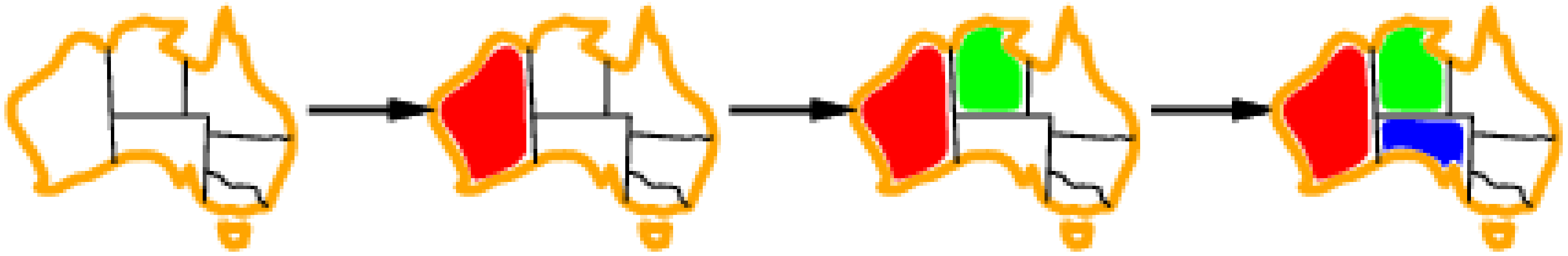
Needed for properly unwinding recursion and preparing to explore another branch of the search tree.

Improving backtracking efficiency

- Which variable should be assigned next?

Most constrained variable

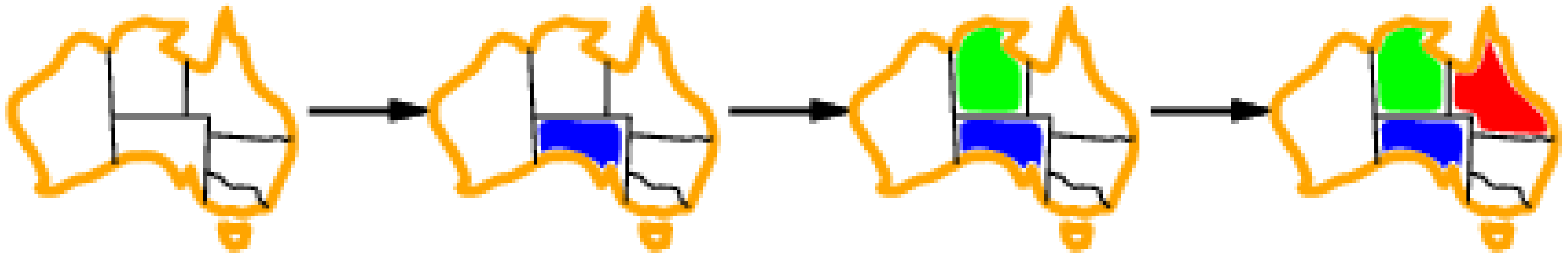
- Most constrained variable:
  choose the variable with the fewest legal values



- minimum remaining values (MRV) heuristic

Degree Heuristic

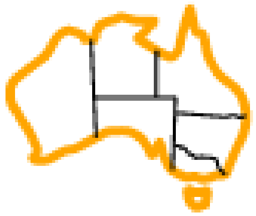
- The MRV heuristic doesn’t help at all in choosing the first region to color in Australia,
  - because initially every region has three legal colors.
- In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is **involved in the largest number of constraints** on other unassigned variables.
- Not only the first time: The MRV heuristic is usually a more powerful guide, but the degree heuristic can be useful as a **tie-breaker**.

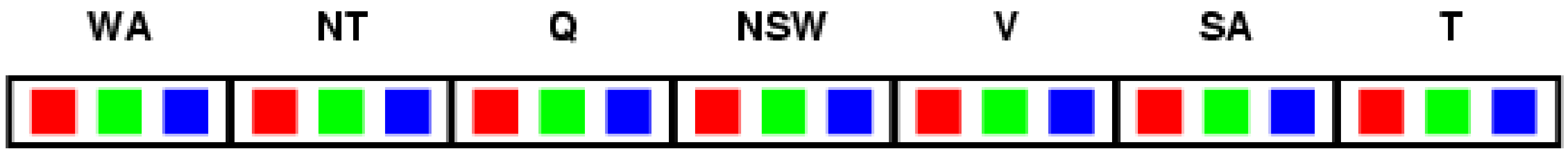


Forward checking

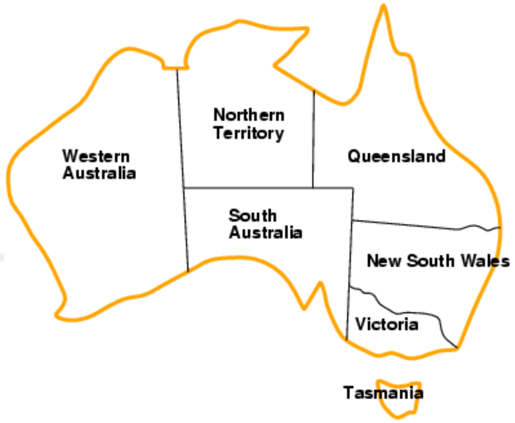
Western Australia
Northern Territory
Queensland
South Australia
New South Wales
Victoria
Tasmania

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

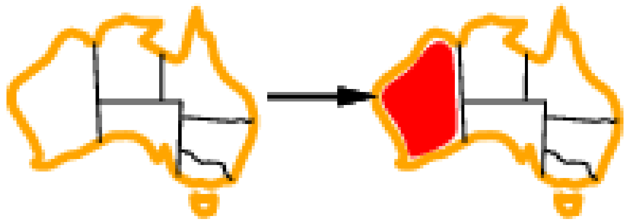




Forward checking

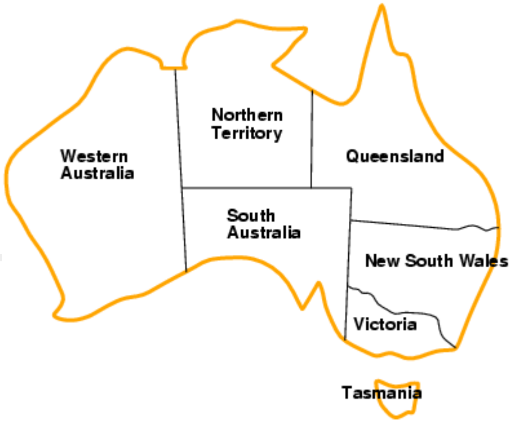


- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

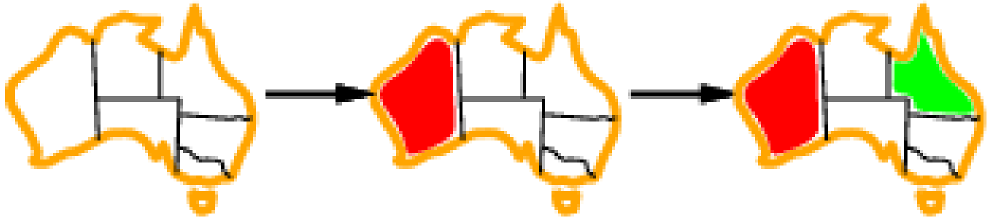


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦

Forward checking

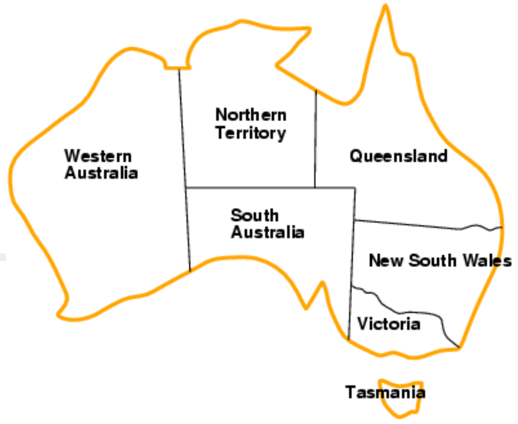


- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

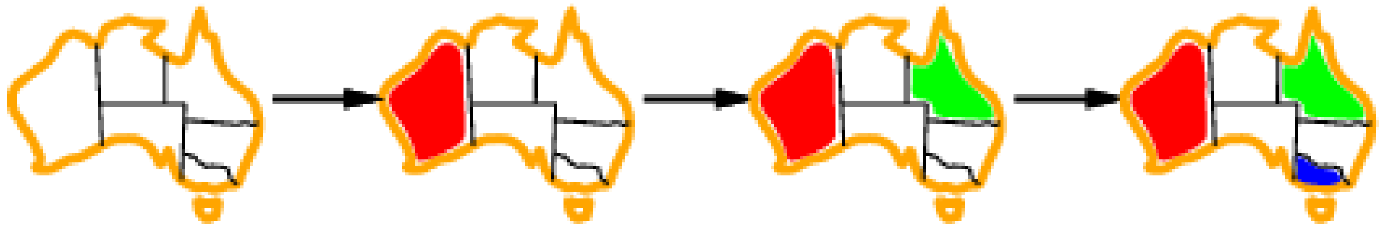


WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥 🟦	🟥 🟩 🟦	🟦	🟥 🟩 🟦

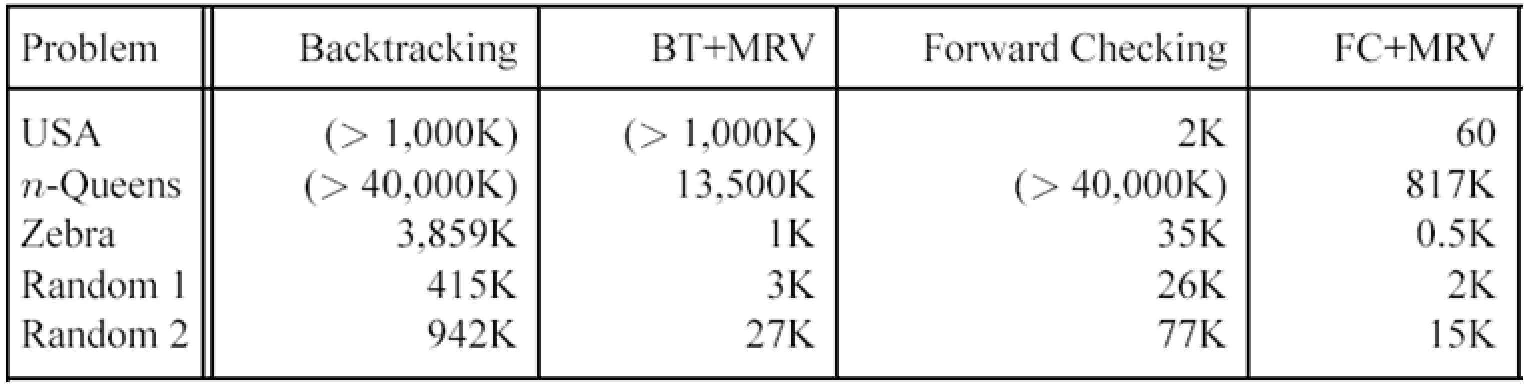
Forward checking



- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦



Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV
USA	(> 1,000K)	(> 1,000K)	2K	60
n-Queens	(> 40,000K)	13,500K	(> 40,000K)	817K
Zebra	3,859K	1K	35K	0.5K
Random 1	415K	3K	26K	2K
Random 2	942K	27K	77K	15K

CSP class

```
public abstract class CSP {
        //variable-domain
        Map<Object,Set<Object>> D = new TreeMap<Object,Set<Object>>();

        //variable-variable
        Map<Object,Set<Object>> C = new TreeMap<Object,Set<Object>>();

        public void addBidirectionalArc(Object X, Object Y) {
                addArc(X,Y);
                addArc(Y,X);
        }

        public void addArc(Object X, Object Y) {
                if(!C.containsKey(X))
                        C.put(X, new TreeSet<Object>());

                C.get(X).add(Y);
        }
```

CSP class

```
        public void addDomain(Object X, Object[] values) {
                Set<Object> s = new TreeSet<Object>();

                for(Object v : values)
                        s.add(v);

                D.put(X, s);
        }

        public abstract boolean isGood(Object X, Object Y, Object x, Object y);
}
```

This what you need to implement for each problem.
It should return "true" if there is no violation of any constraint involving variables X and Y with values x and y.

CSPGraphColoring class

```
public class CSPGraphColoring extends CSP {

        public boolean isGood(Object X, Object Y, Object x, Object y) {
                //if X is not even mentioned in by the constraints, just return true
                //as nothing can be violated
                if(!C.containsKey(X))
                        return true;

                //check to see if there is an arc between X and Y
                //if there isn't an arc, then no constraint, i.e. it is good
                if(!C.get(X).contains(Y))
                        return true;

                //not equal constraint
                if(!x.equals(y))
                        return true;

                return false;
        }
```

CSPGraphColoring class

```
public static void main(String[] args) throws Exception {
        CSPGraphColoring csp = new CSPGraphColoring();

        String[] vars = {"WA", "NT", "Q", "NSW", "V", "SA", "T"};
        String[] colors = {"r", "g", "b"};
        String[][] pairs = {{"WA","NT"}, {"NT","Q"}, {"Q","NSW"}, {"NSW","V"},
                        {"SA", "WA"}, {"SA", "NT"}, {"SA", "Q"},
                        {"SA", "NSW"}, {"SA", "V"}};

                for(Object X : vars)
                        csp.addDomain(X, colors);

                for(Object[] p : pairs)
                        csp.addBidirectionalArc(p[0], p[1]);

                Search search = new Search(csp);
                System.out.println(search.BacktrackingSearch());
}
```

Search class

```
public class Search {
        CSP csp;
        public Search(CSP csp) { this.csp = csp; }

        //returns an assignment, variable-value map
        public Map<Object, Object> BacktrackingSearch() {
                //create an empty assignment
                Map<Object, Object> assignment = new TreeMap<Object, Object>();

                return Backtrack(assignment);
        }
```

Search class

```
Map<Object, Object> Backtrack(Map<Object, Object> assignment) {
        if( isComplete(assignment) )
                return assignment;
        Object X = SelectUnassignedVariable(assignment);
        for(Object x : csp.D.get(X)) {
                if(isConsistent(X,x,assignment)) {
                        assign(X,x,assignment);

                        Map<Object, Object> result = Backtrack(assignment);
                        if(result!=null)
                                return result;

                        assignUndo(X,assignment);
                }
        }
        return null;
}
```

Search class

```
boolean isConsistent(Object X, Object x, Map<Object, Object> assignment) {
        for(Object Y : assignment.keySet()) {
                Object y = assignment.get(Y);

                if(!csp.isGood(X,Y,x,y))
                        return false;

                if(!csp.isGood(Y,X,y,x))
                        return false;
        }
        return true;
}
```

Search class

```
Object SelectUnassignedVariable(Map<Object, Object> assignment) {
        //Implements minimum remaining values (MRV)
        int min = Integer.MAX_VALUE;
        Object Xmin = null;

        for(Object X : csp.D.keySet()) {
                if (assignment.containsKey(X))
                        continue;

                if (csp.D.get(X).size() < min) {
                        min = csp.D.get(X).size();
                        Xmin = X;
                }
        }

        return Xmin;
}
```

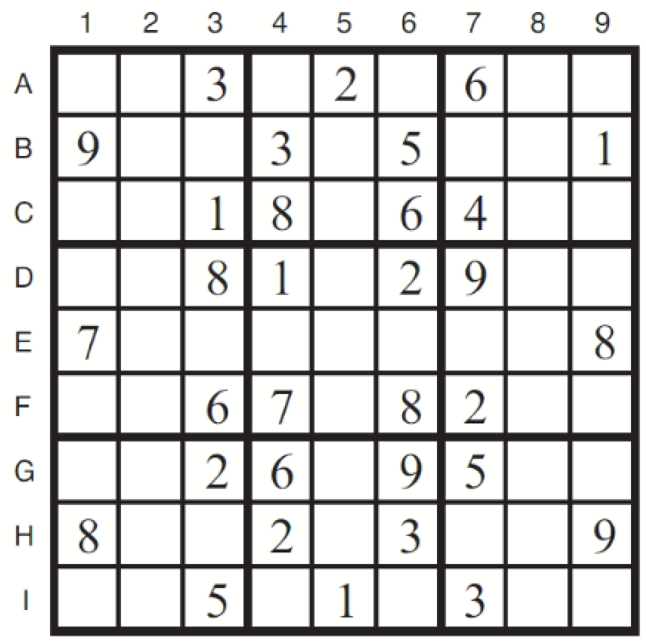
Search class

```
//assigns x to X and does Forward Checking
void assign(Object X, Object x, Map<Object, Object> assignment) {
        assignment.put(X, x);

        //now do forward checking and record values to be deleted in a stack
        //of var-set maps of deleted values
        ...
}

void assignUndo(Object X, Map<Object, Object> assignment) {
        assignment.remove(X);

        //Now undo value deletions done by forward checking
        ...
}
```

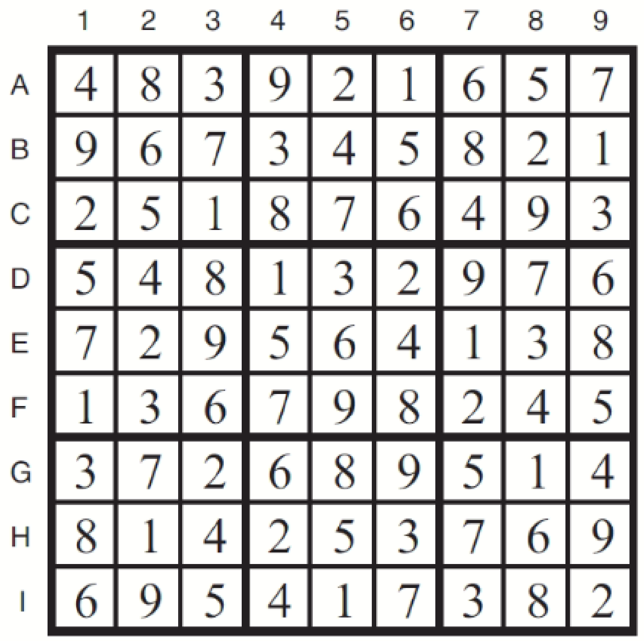
Constraint Satisfaction Problems

Examples

Sudoku



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		



	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

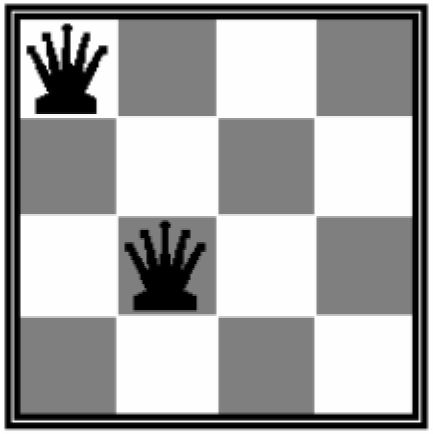
A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9.

The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3×3 box.

Sudoku

- 81 variables, one for each square.
  - A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row.
- Domains:
  - Empty squares have domain
    - {1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Prefilled squares have a domain consisting of a single value.
- Constraints: 27 “all different” constraints one for each row, column, and box of 9 squares.
  - Alldiff (A1,A2,A3,A4,A5,A6, A7, A8, A9)
  - Alldiff (B1,B2,B3,B4,B5,B6,B7,B8,B9)
  - . . .
  - Alldiff (A1,B1,C1,D1,E1, F1,G1,H1, I1)
  - Alldiff (A2,B2,C2,D2,E2, F2,G2,H2, I2)
  - . . .
  - Alldiff (A1,A2,A3,B1,B2,B3,C1,C2,C3)
  - Alldiff (A4,A5,A6,B4,B5,B6,C4,C5,C6)

N-Queens



- Variables: Qi
- Domains: Di = {1, 2, 3, 4}
- Constraints:
  - Qi≠Qj (cannot be in same row)
  - |Qi - Qj | ≠ |i - j| (or same diagonal)

- Valid values for (Q1 , Q2) are (1,3) (1,4) (2,4) (3,1) (4,1) (4,2)

Crossword Puzzles

- Constructing crossword puzzles: **fitting words into a rectangular grid**.

b	i	s	h	o	p
	n	▨			
	t		▨		
	e				
	l				

Crossword Puzzles

- **Variables**: one for each line, and one for each column.
  - H1, H2, H3, H4, H5
  - V1, V2, V3, V4, V5, V6
- **Domains**: Subsets of English words,
  - E.g. D_{H2} is the set of all 5-letter English words.
- **Constraints**: One constraint between each two variables that intersect
  - E.g. H1→V2 saying that H1[2] = V2[1]

	V1	V2	V3	V4	V5	V6
H1	b	i	s	h	o	p
H2		n	▨			
H3		t		▨		
H4		e				
H5		l				

Rectilinear floor-planning:

- **Problem**: Find non-overlapping places in a large rectangle for a number of smaller rectangles.
  - Let's assume that the floor is a **grid**.

- **Variables**: one for each of the small rectangles,
  - with the value of each variable being a **4-tuple** consisting of the coordinates of the upper-left and lower-right corners of the place where the rectangle will be located.
- **Domains**:
  - for each variable it is the set of 4-tuples that are the right size for the corresponding small rectangle and that fit within the large rectangle.
- **Constraints**: say that no two rectangles can overlap;
  - E.g. if the value of variable X1 is (0,0,5,8), then no other variable can take on a value that overlaps with the (0,0,5,8) rectangle.

Class-Scheduling

- There is
  - a fixed number of professors and classrooms,
  - a list of classes to be offered, and
  - a list of possible time slots for classes.
- Each professor has a set of classes that he or she can teach.

- **Variables**: C_i : one for each class.
  - Values are triples (classroom, time, professor)
- **Domains**: For each C_i
  - D_i is the set of all the possible triples after *filtering out* those triples with third element a professor that doesn't teach C_i.
- **Constraints**: one for each pair of variables ($C_i \rightarrow C_j$) saying:

  $\neg$ (classroom$_i$=classroom$_j$ $\wedge$ time$_i$=time$_j$) $\wedge$

  $\neg$ (professor$_i$=professor$_j$ $\wedge$ time$_i$=time$_j$)

Zebra Problem

- Consider the following logic puzzle:

  In **five houses**, each with a different **color**, live **five persons** of different **nationalities**, each of whom prefers a different brand of **cigarette**, a different **drink**, and a different **pet**.

  Given the facts in the next slide, the question to answer is

- **“Where does the zebra live, and in which house do they drink water?”**

Zebra Problem

https://en.wikipedia.org/wiki/Zebra_Puzzle

1. The Englishman lives in the red house.
2. The Spaniard owns a dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is directly to the right of the ivory house.
6. The Old-Gold smoker owns snails.
7. Kools are being smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house on the left.
10. The Chesterfield smoker lives next to the fox owner.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky-Strike smoker drinks orange juice.
13. The Japanese smokes Parliament.
14. The Norwegian lives next to the blue house.

Zebra Problem

- **Variables**: five variables for each house, one with the domain of colors, one with pets, and so on. Total 25 variables. i.e.
  - color1, …, color5,
  - drink1, …, drink5
  - nationality1, …, nationality5
  - pet1, …, pet5
  - cigarette1, …, cigarette5
- **Domains**:
  - Blue, Green, Ivory, Red, Yellow
  - Coffee, Milk, Orange-Juice, Tea, Water
  - Englishman, Japanese, Norwegian, Spaniard, Ukrainian
  - Dog, Fox, Horse, Snails, Zebra
  - Chesterfield, Kools, Lucky-Strike, Old-Gold, Parliament

House:	1	2	3	4	5
Colour:	Blue Green Ivory Red Yellow	Blue Green Ivory Red Yellow	Blue Green Ivory Red Yellow	Blue Green Ivory Red Yellow	Blue Green Ivory Red Yellow
Drink:	Coffee Milk Orange Juice Tea Water	Coffee Milk Orange Juice Tea Water	Coffee Milk Orange Juice Tea Water	Coffee Milk Orange Juice Tea Water	Coffee Milk Orange Juice Tea Water
Nationality:	Englishman Japanese Norwegian Spaniard Ukrainian	Englishman Japanese Norwegian Spaniard Ukrainian	Englishman Japanese Norwegian Spaniard Ukrainian	Englishman Japanese Norwegian Spaniard Ukrainian	Englishman Japanese Norwegian Spaniard Ukrainian
Pet:	Dog Fox Horse Snails Zebra	Dog Fox Horse Snails Zebra	Dog Fox Horse Snails Zebra	Dog Fox Horse Snails Zebra	Dog Fox Horse Snails Zebra
Cigarette:	Chesterfield Kools Lucky-Strike Old-Gold Parliament	Chesterfield Kools Lucky-Strike Old-Gold Parliament	Chesterfield Kools Lucky-Strike Old-Gold Parliament	Chesterfield Kools Lucky-Strike Old-Gold Parliament	Chesterfield Kools Lucky-Strike Old-Gold Parliament

Zebra Problem

- **Constraints**:
  - **Unary**: Rules 8 (*Milk is drunk in the middle house*) and 9 (*The Norwegian lives in the first house on the left*):
    - drink3=Milk
    - nationality1=Norwegian
  - We filter the corresponding domains

House:	1	2	3	4	5
Colour:	Blue Green Ivory Red Yellow	Blue Green Ivory Red Yellow	Blue Green Ivory Red Yellow	Blue Green Ivory Red Yellow	Blue Green Ivory Red Yellow
Drink:	Coffee ~~Milk~~ Orange Juice Tea Water	Coffee ~~Milk~~ Orange Juice Tea Water	~~Coffee~~ Milk ~~Orange Juice~~ ~~Tea~~ ~~Water~~	Coffee ~~Milk~~ Orange Juice Tea Water	Coffee ~~Milk~~ Orange Juice Tea Water
Nationality:	~~Englishman~~ ~~Japanese~~ Norwegian ~~Spaniard~~ ~~Ukrainian~~	Englishman Japanese ~~Norwegian~~ Spaniard Ukrainian	Englishman Japanese ~~Norwegian~~ Spaniard Ukrainian	Englishman Japanese ~~Norwegian~~ Spaniard Ukrainian	Englishman Japanese ~~Norwegian~~ Spaniard Ukrainian
Pet:	Dog Fox Horse Snails Zebra	Dog Fox Horse Snails Zebra	Dog Fox Horse Snails Zebra	Dog Fox Horse Snails Zebra	Dog Fox Horse Snails Zebra
Cigarette:	Chesterfield Kools Lucky-Strike Old-Gold Parliament	Chesterfield Kools Lucky-Strike Old-Gold Parliament	Chesterfield Kools Lucky-Strike Old-Gold Parliament	Chesterfield Kools Lucky-Strike Old-Gold Parliament	Chesterfield Kools Lucky-Strike Old-Gold Parliament

Zebra Problem

- **Constraints**:
  - **Binary**:
    - The uniqueness: for each $i \neq j$, $i,j=1,\ldots,5$ set the following constraints:

      $(color_i \neq color_j)$

      $(drink_i \neq drink_j)$

      $(nationality_i \neq nationality_j)$

      $(pet_i \neq pet_j)$

      $(cigarette_i \neq cigarette_j)$

Zebra Problem

- **Constraints**
  - **Examples**:
    - Rule 1 (*The Englishman lives in the red house*): $\forall i=1,\ldots,5$:
      - If $nationality_i$ and $color_i$ both have assigned values then

        $(nationality_i=Englishman \wedge color_i=red) \vee$

        $(nationality_i \neq Englishman \wedge color_i \neq red)$

    - Rule 2 (*The Spaniard owns a dog*): $\forall i=1,\ldots,5$:
      - If $nationality_i$ and pet_i both have assigned values then

        $(nationality_i = Spaniard \wedge pet_i = dog) \vee$

        $(nationality_i \neq Spaniard \wedge pet_i \neq dog)$

Zebra Problem

- **Constraints**
  - Rule 10 (*The Chesterfield smoker lives next to the fox owner*): Not easy to represent!

Zebra Problem – change in vars

- **Variables**: (var-name, house-number)
  - (color,1), …, (color,5),
  - (drink,1), …, (drink,5)
  - (nationality,1), …, (nationality,5)
  - (pet,1), …, (pet,5)
  - (cigarette,1), …, (cigarette,5)

- **Domains**:
  - ∀i=1,…,5, (Blue,i), (Green,i), (Ivory,i), (Red,i), (Yellow,i)
  - ∀i=1,…,5, (Coffee,i), (Milk,i), (Orange-Juice,i), (Tea,i), (Water,i)
  - ∀i=1,…,5, (Englishman,i), (Japanese,i), (Norwegian,i), (Spaniard,i), Ukrainian,i)
  - ∀i=1,…,5, (Dog,i), (Fox,i), (Horse,i), (Snails,i), (Zebra,i)
  - ∀i=1,…,5, (Chesterfield,i), (Kools,i), (Lucky-Strike,i), (Old-Gold,i), (Parliament,i)

Zebra Problem

- **Constraints**
  - Rule 10 (*The Chesterfield smoker lives next to the fox owner*):

    $(cigarette_i = Chesterfield \wedge pet_j = fox \wedge i-j=1) \vee$

    $(cigarette_i = Chesterfield \wedge pet_j = fox \wedge i-j=-1) \vee$

    …

Zebra Problem

Alternate representation:

- **Variables**: one variable for each color, drink, nationality, pet, and cigarette, i.e.
  - blue, green, ivory, red, yellow,
  - coffee, milk, orange, juice, tea, water
  - englishman, japanese, norwegian, spaniard, ukrainian
  - dog, fox, horse, snails, zebra
  - chesterfield, kools, lucky-strike, old-gold, parliament
- **Domains**:
  - For each one of the variables the domain is {1,2,3,4,5} i.e. the house number.

Zebra Problem

- **Constraints**:
  - **Unary**: Rules 8 (*Milk is drunk in the middle house*) and 9 (*The Norwegian lives in the first house on the left*):
    - milk=3
    - norwegian=1
  - i.e. we filter the corresponding domains

Zebra Problem

- **Constraints**:
  - **Binary**:
    - The uniqueness:
      - (blue≠ivory) …

    - Rules, e.g. rule 14 (*The Norwegian lives next to the blue house*):

      (|norwegian-blue| =1)

Zebra Problem

- **Constraints**:
  - **Binary**:
    - Rule 1 (*The Englishman lives in the red house*):

      (englishman=red)

    - Rule 2 (*The Spaniard owns a dog*):

      (spaniard =dog)

Zebra

```
public class CSPZebra2 extends CSP {

  static Set<Object> varCol = new HashSet<Object>(
    Arrays.asList(new String[] {"blue", "green", "ivory", "red", "yellow"}));

  static Set<Object> varDri = new HashSet<Object>(
    Arrays.asList(new String[] {"coffee", "milk", "orange-juice", "tea", "water"}));

  static Set<Object> varNat = new HashSet<Object>(
    Arrays.asList(new String[] {"englishman", "japanese", "norwegian", "spaniard", "ukrainian"}));

  static Set<Object> varPet = new HashSet<Object>(
    Arrays.asList(new String[] {"dog", "fox", "horse", "snails", "zebra"}));

  static Set<Object> varCig = new HashSet<Object>(
    Arrays.asList(new String[] {"chesterfield", "kools", "lucky-strike", "old-gold", "parliament"}));
```

Zebra

```
public boolean isGood(Object X, Object Y, Object x, Object y) {
        //if X is not even mentioned in by the constraints, just return true
        //as nothing can be violated
        if(!C.containsKey(X))
                return true;

        //check to see if there is an arc between X and Y
        //if there isn't an arc, then no constraint, i.e. it is good
        if(!C.get(X).contains(Y))
                return true;

        //The Englishman lives in the red house.
        if(X.equals("englishman") && Y.equals("red") && !x.equals(y))
                return false;
        …
        //Uniqueness constraints
        if(varCol.contains(X) && varCol.contains(Y) && !X.equals(Y) && x.equals(y))
                return false;
        …
        return true;
}
```

Zebra

```
public static void main(String[] args) throws Exception {
        CSPZebra2 csp = new CSPZebra2();

        Integer[] dom = {1,2,3,4,5};

        for(Object X : varCol)
                csp.addDomain(X, dom);
        …

        //unary constraints: just remove values from domains
        …

        //binary constraints: add constraint arcs
        //The Englishman lives in the red house.
        csp.addBidirectionalArc("englishman", "red");
        …
        //Uniqueness constraints
        for(Object X : varCol)
                for(Object Y : varCol)
                        csp.addBidirectionalArc(X,Y);
        …
```

Zebra

```
                //Now let's search for solution

                Search search = new Search(csp);
                System.out.println(search.BacktrackingSearch());
        }
}
```