CS380
Artificial Intelligence for Games

Constraint Satisfaction Problems

Constraint Satisfaction Problems

- Standard search problem
  - State is a black box: any data structure that supports successor function, heuristic function, and goal test

- Constraint Satisfaction Problems
  - State is defined by variables X_i with values from domain D_i , where $D_i = \{v_1, \dots, v_k\}$ for X_i
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables.

Constraints

- Unary constraints involve a single variable, e.g. Variable X must be even: $\langle X, X\%2 = 0 \rangle$

- Binary constraints involve pairs of variables

Example: Binary Constraint

- Two variables X_1 and X_2 must have different values, where the domain for both variables are the same $\{A, B\}$
  - $\langle (X_1, X_2), [(A, B), (B, A)] \rangle$ or $\langle (X_1, X_2), X1 \neq X2 \rangle$

- X_2 must be greater than X_1, where the domain for both variables are integers $\{\dots, -2, -1, 0, 1, 2, \dots\}$
  - $\langle (X_1, X_2), [(-3,2), (-100,9), \dots] \rangle$ or $\langle (X1, X2), X_1 < X_2 \rangle$

Higher-order Constraints

- Higher-order constraints involve three or more variables,
  - e.g. The sum of variable X_1 and X_2 must equal variable X_3: $\langle (X_1, X_2, X_3), X_1 + X_2 = X_3 \rangle$
  - Global constraint: $Alldiff(\dots)$

Soft Constraints

- Preferences (soft constraints), e.g., red is better than green, which is often represented by a cost for each variable assignment and leads to constrained optimization problem.

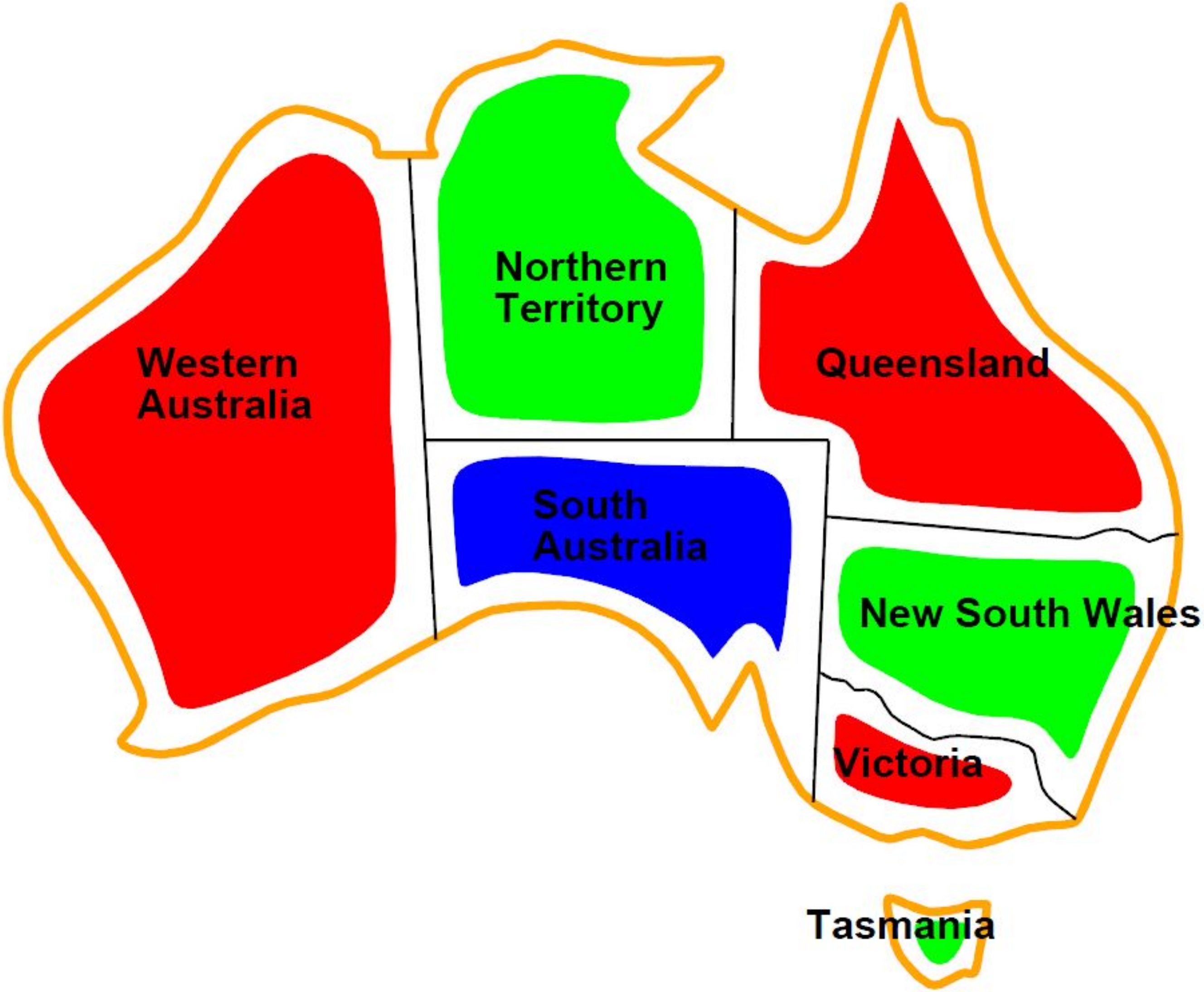
Why CSP?

- CSPs yield a natural representation for a wide variety of problems
  - Assignment problems
    - e.g., who teaches what class
  - Timetabling problems
    - e.g., which class is offered when and where?
  - Transportation scheduling
  - Factory scheduling
  - Map coloring problem (AKA Graph coloring)

Example: Map-Coloring

Color the map such that no two adjacent regions are of the same color

Example: Map-Coloring Solution

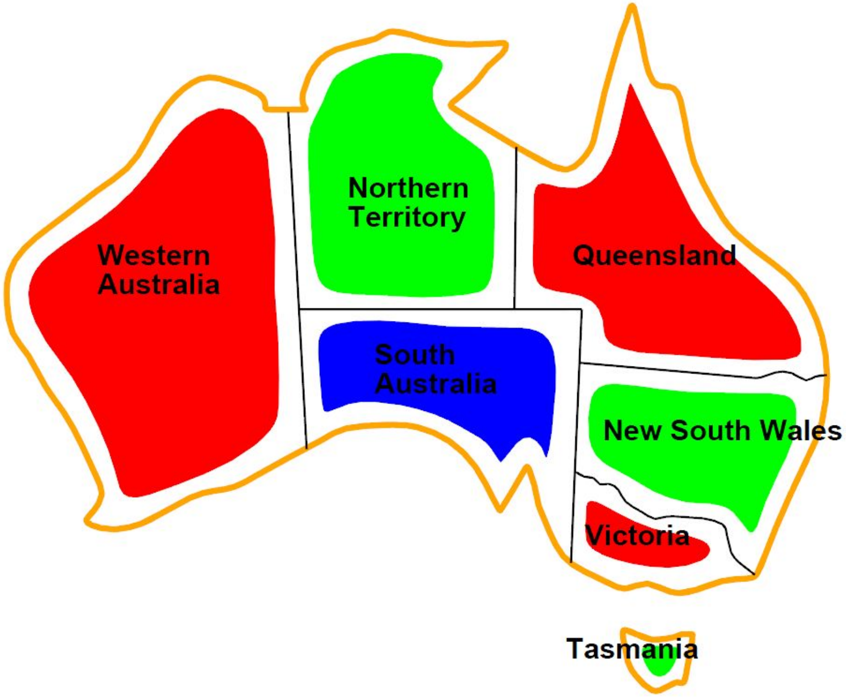


Example: Map-Coloring

- Variables?
  - $X = \{WA, NT, Q, NSW, V, SA, T\}$
- Domains?
  - $D_i = \{red, green, blue\}$ for $i = 1,2,3,4,5,6,7$
- Constraints?
  - Adjacent regions must have different colors
  - $\langle (WA, NT, Q, NSW, V, SA, T), WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, SA \neq Q, Q \neq SW, SA \neq SW, SW \neq V, SA \neq V \rangle$

Example: Map-Coloring

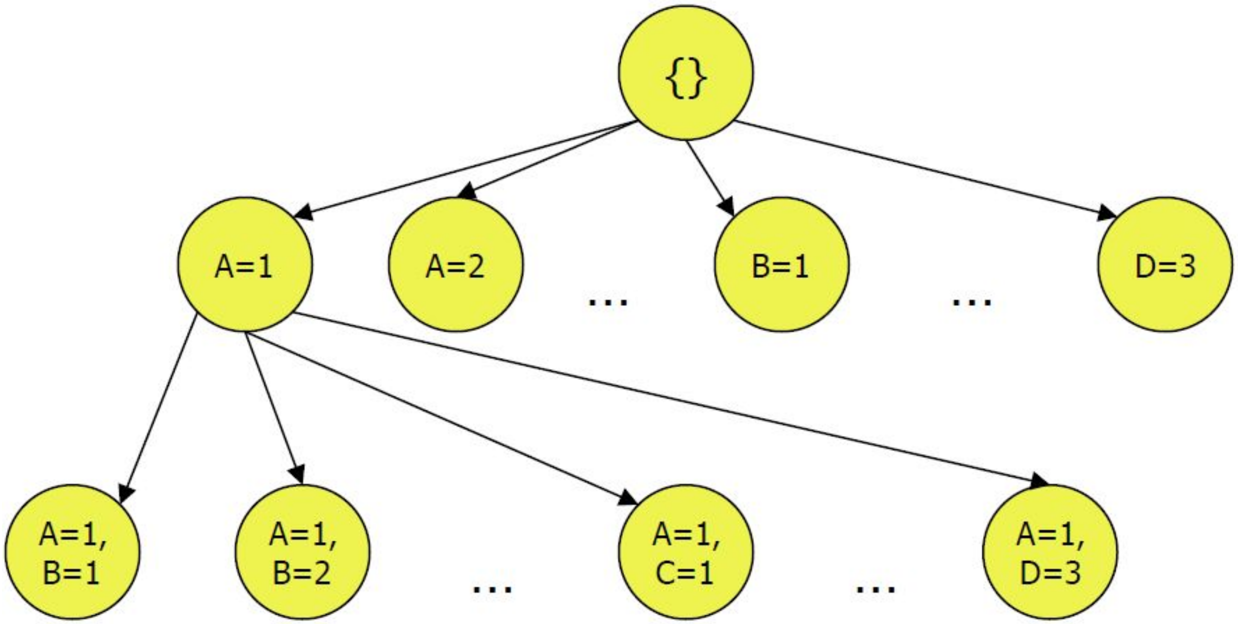
- Solutions are complete and consistent assignments:
- E.g. {WA = red, NT = green, Q = red, NSW =green, V = red, SA = blue, T = green}



Straightforward Search

- States are defined by the values assigned so far.
- Initial state: the empty assignment {}
- Successor function: assign a value to an unassigned variable that <u>does not conflict with current assignment</u>
- Goal test: Is the current assignment complete and consistent?

Straightforward Search



Depth 1: 4 variables × 3 values = 12 states

Depth 2: 3 variables × 3 values = 9 states

Depth 3: 2 variables × 3 values = 6 states

Depth 4: 1 variable × 3 values = 3 states (leaf level)

Variables: A, B, C, D
Domains: 1,2,3

Backtracking Search

- Variable assignments are commutative, i.e., "$WA = red$ then $NT = green$" is the same as "$NT = green$ then $WA = red$"

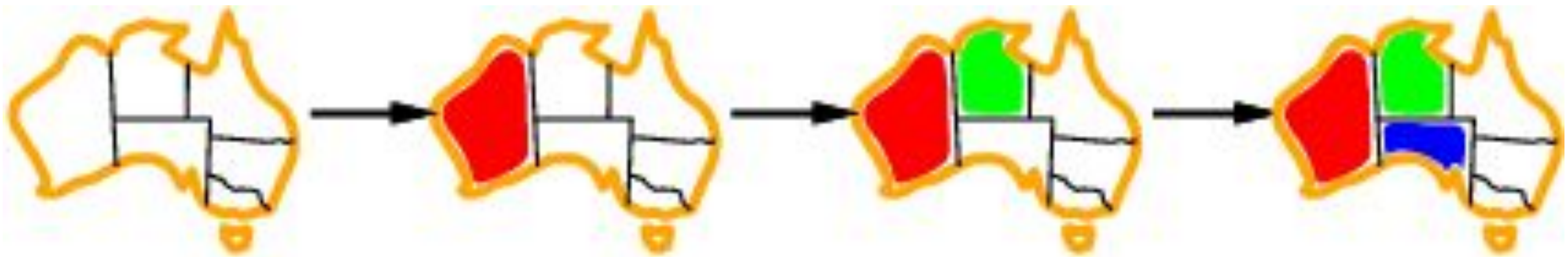
- Only need to consider assignments to a single variable at each node
  - Fix an order in which we'll examine the variables

- Depth-first search for CSPs with single-variable assignments is called backtracking search

Improving Backtracking Efficiency

Q: Which variable should be assigned next?

A: Choose the **most constrained variable** with minimum remaining values

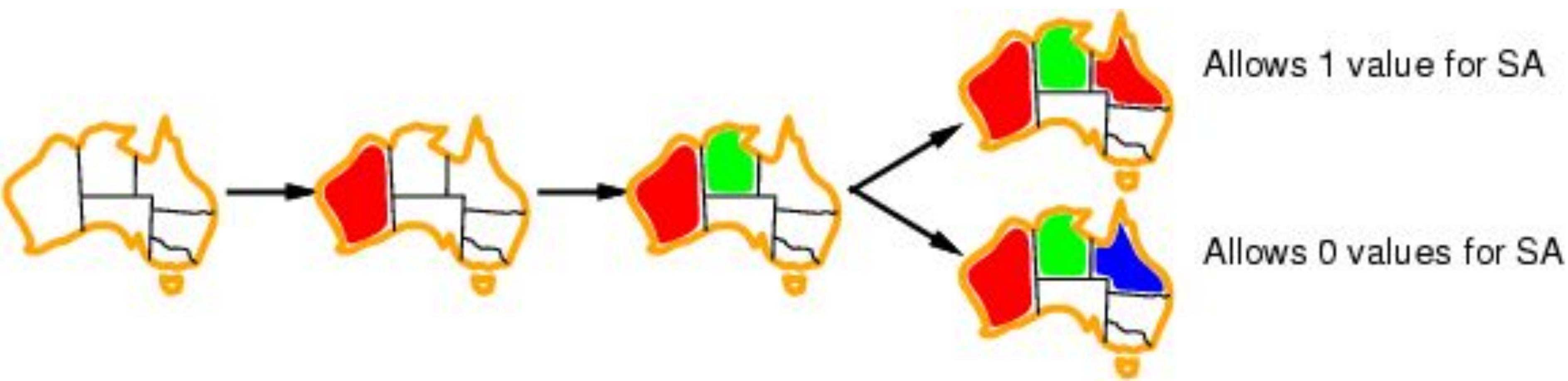


Improving Backtracking Efficiency

Q: In what order should its values be tried?

A: Given a variable, choose the **least constraining value**:

- the one that rules out the fewest values in the remaining variables of its neighbors



Sudoku Example

- Variables?
- Domains?
- Constraints?

		3		2		6		
9			3		5			1
		1	8		6	4		
		8	1		2	9		
								8
		6	7		8	2		
		2	6		9	5		
8			2		3			9
		5		1		3		

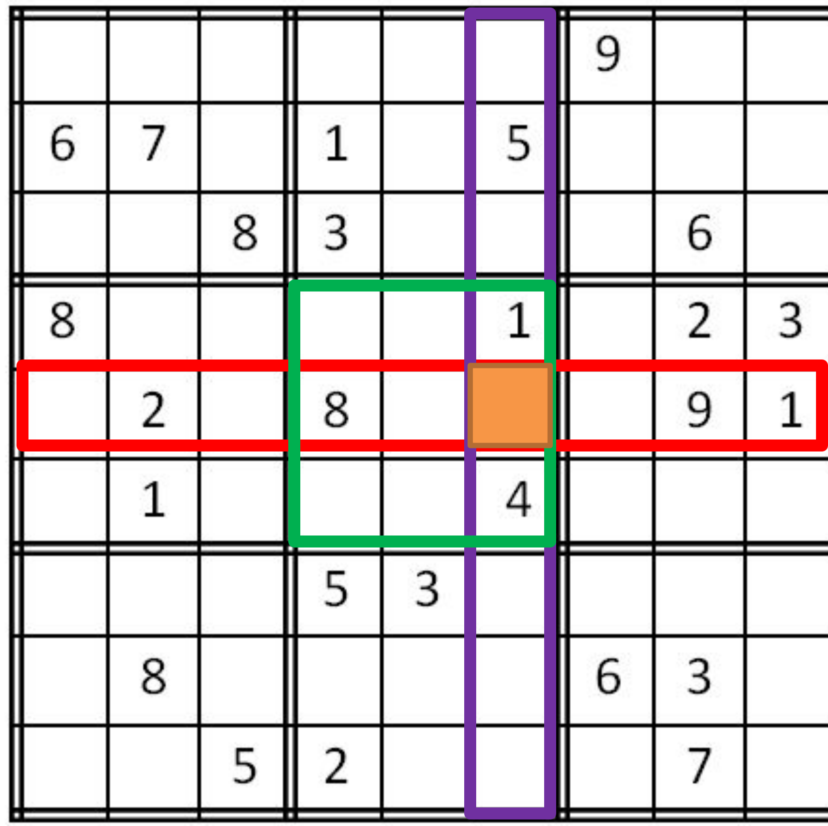
Sudoku Example

- Variables: A[0][0], …, A[8][8]
- Domains: {1,…,9}
- Constraints:
  - for i=0 to 8: Alldiff(A[i][0], A[i][1], A[i][2], A[i][3], A[i][4], A[i][5], A[i][6], A[i][7], A[i][8])
  - for i=0 to 8: Alldiff(A[0][i], A[1][i], A[2][i], A[3][i], A[4][i], A[5][i], A[6][i], A[7][i], A[8][i])
  - for i=0 to 2, for j=0 to 2, Alldiff(A[i][j])
  - …
  - for i=6 to 8, for j=6 to 8, Alldiff(A[i][j])

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Sudoku Example

- From the Sub-grid constraint we can remove {1,4 and 8}.
- From the Row constraint we can remove {2 and 9}.
- From the Column constraint we can remove {5}.
- The domain of A[4][5] will be simplified to {3,6,7}.



4		1			8	7	5	
		5		4	7	6		9
6		8		5			3	4
1	2			6		8	9	
5		7	8	9				3
	6	9		7		4	1	
	1	2		8	4			6
	5				6	3	2	8
		6	7	2	3			1

4	$\boxed{3}$	1			8	7	5	
		5		4	7	6		9
6		8		5			3	4
1	2			6		8	9	
5		7	8	9				3
	6	9		7		4	1	
	1	2		8	4			6
	5				6	3	2	8
		6	7	2	3			1

4	3	1	$\boxed{2}$		8	7	5	
		5		4	7	6		9
6		8		5			3	4
1	2			6		8	9	
5		7	8	9				3
	6	9		7		4	1	
	1	2		8	4			6
	5				6	3	2	8
		6	7	2	3			1

4	3	1	$\boxed{6}$		8	7	5	
		5		4	7	6		9
6		8		5			3	4
1	2			6		8	9	
5		7	8	9				3
	6	9		7		4	1	
	1	2		8	4			6
	5				6	3	2	8
		6	7	2	3			1

4	3	1	$\boxed{9}$		8	7	5	
		5		4	7	6		9
6		8		5			3	4
1	2			6		8	9	
5		7	8	9				3
	6	9		7		4	1	
	1	2		8	4			6
	5				6	3	2	8
		6	7	2	3			1

4	$\boxed{9}$	1			8	7	5	
		5		4	7	6		9
6		8		5			3	4
1	2			6		8	9	
5		7	8	9				3
	6	9		7		4	1	
	1	2		8	4			6
	5				6	3	2	8
		6	7	2	3			1

```
Backtracking(next_location, next_candidate) {
  var stack = new Stack(); // Set the stack to start the search
  var location = next_location();
  if (location.notFound()) return true;
  stack.push(location);
  while(true) {
    var location = stack.top();
    if (location.notFound()) return true;
    var candidate = next_candidate(location);
    if (candidate) {                        // If there is a new candidate
      location.setValue(candidate);         // for the current location,
      var next = next_location();           // set it and find next
      if (next.notFound()) return true;     // location to check in the next
      stack.push(next);                     // call of this function
    } else {                                // If a new candidate not found,
      location.clearValue();                // backtrack by cleaning
      stack.pop();                          // location and popping it from
    }                                       // the stack
    // Solution not found yet, so return false to repeat the process
    return false;
  }
}
```

```
// This is a domain specific class that is used to find a solution
// for a one-dimensional Sudoku by using Backtracking algorithm.
// run() returns next location which is not occupied yet on the map.
// If no location found, returns Location(null, 0).
class NextLocation_Sudoku1D
{
  constructor(map)
  {
    this.map = map;
  }

  run()
  {
    for (var i=0; i<this.map.width; ++i)
      if (this.map.base[i]==0)
        return new Location(this.map.base,i);
    return new Location(null, 0);
  }
}
```

```
// This is a domain specific class that is used to find a solution
// for a one-dimensional Sudoku by using Backtracking algorithm.
// run() returns next candidate for a specified location on the map.
// If no candidate found, returns null.
class NextCandidate_Sudoku1D
{
  constructor(map)
  {
    this.map = map;
  }
  run(location)
  {
    var v = this.map.base[location.i];
    while (v<this.map.width)
    {
      v++;
      if (this.map.base.indexOf(v)==-1)
        return v;
    }
    return null;
  }
}
```