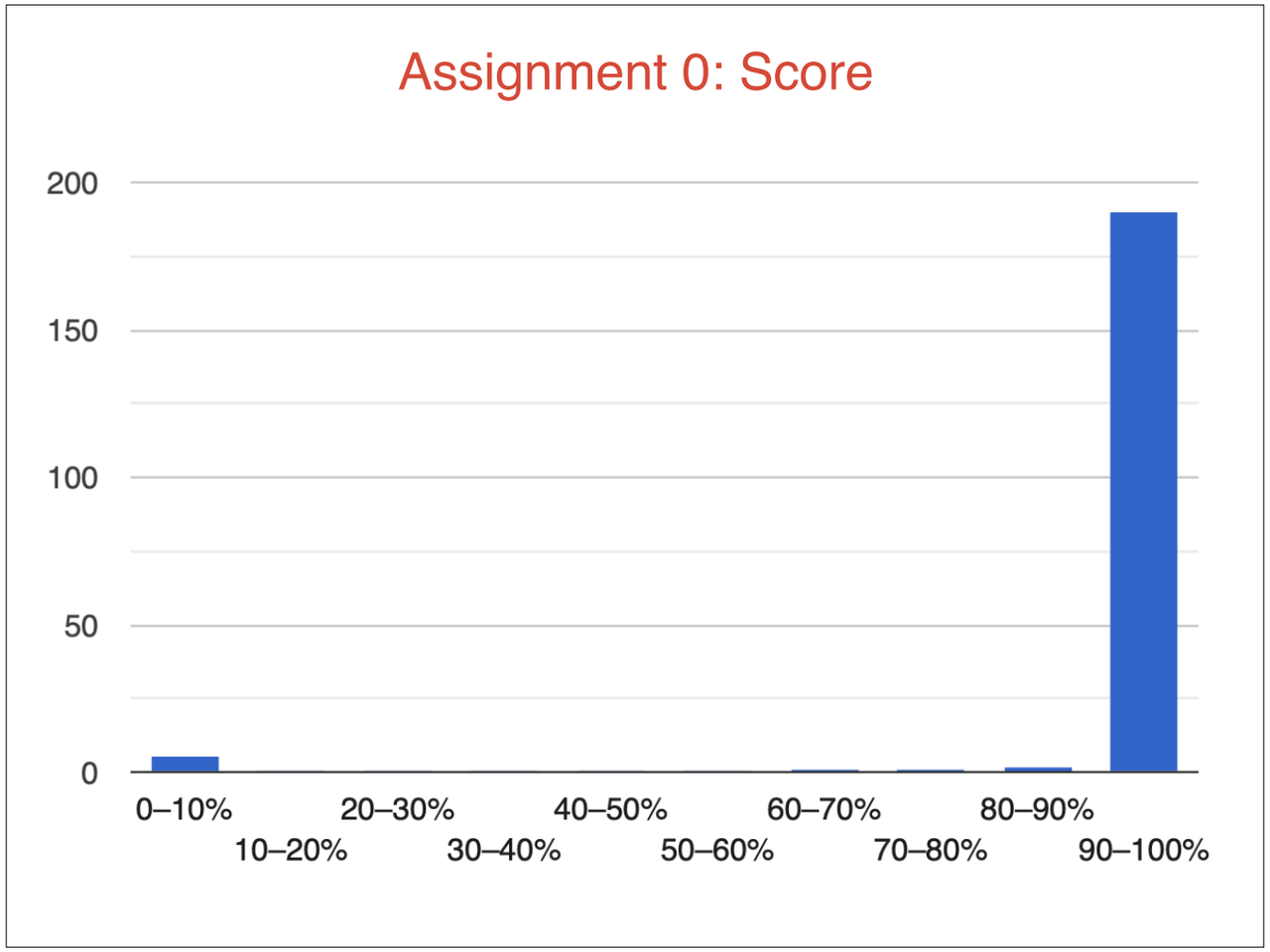
Chapter 6: Constraint satisfaction problems

W4-Wed:

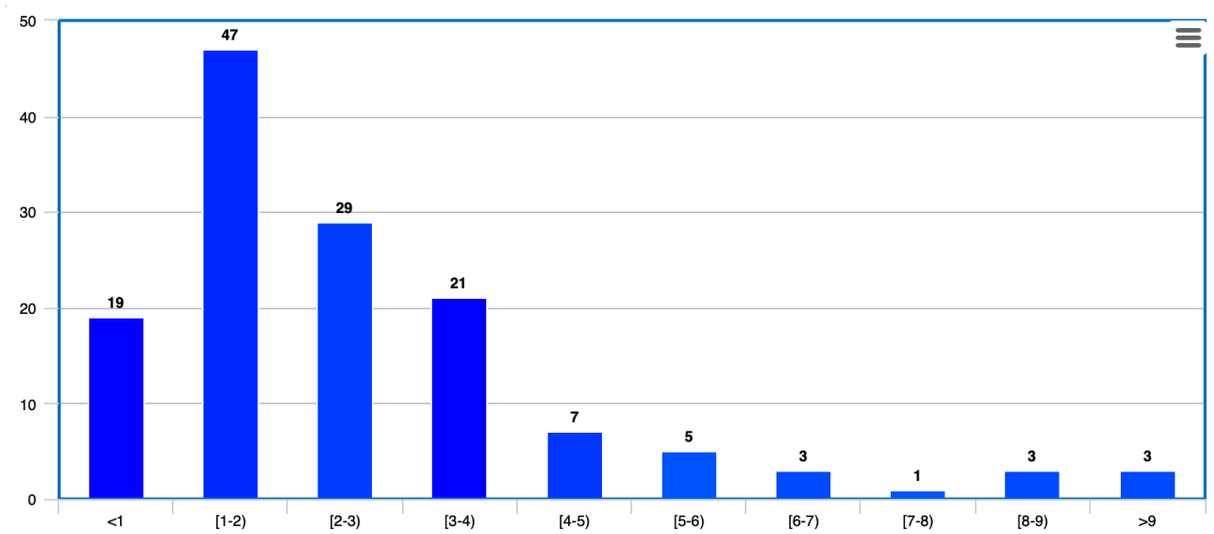
- Adv search problem from last time.
- A0 recap
- A1 intro.
- Friday:

- We will consider a special case of observable, deterministic problems.
- Previously, we viewed each state (unique configuration of the world) as a black box -- if two configurations of the world are different in any way, they count as different states.
- Factored representation: set of variables, each of which has a value.
- Constraint satisfaction problems: We just care about finding an assignment to variables that satisfies a bunch of constraints.

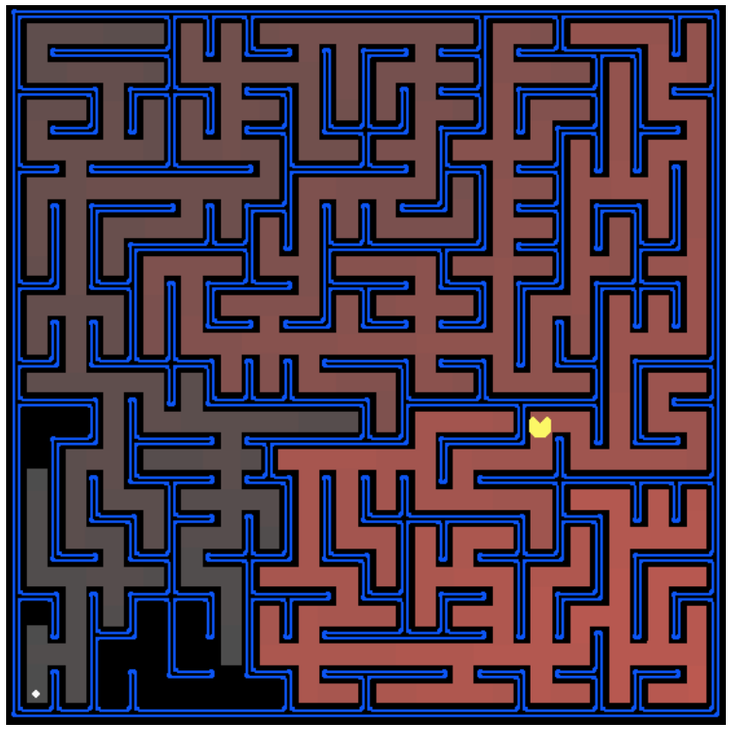


- Scores: Everyone did well.
- Time: Good. A1 will be much longer -- we aim for 20 hours per assignment.
- Feedback.

Assignment 0: Hours spent

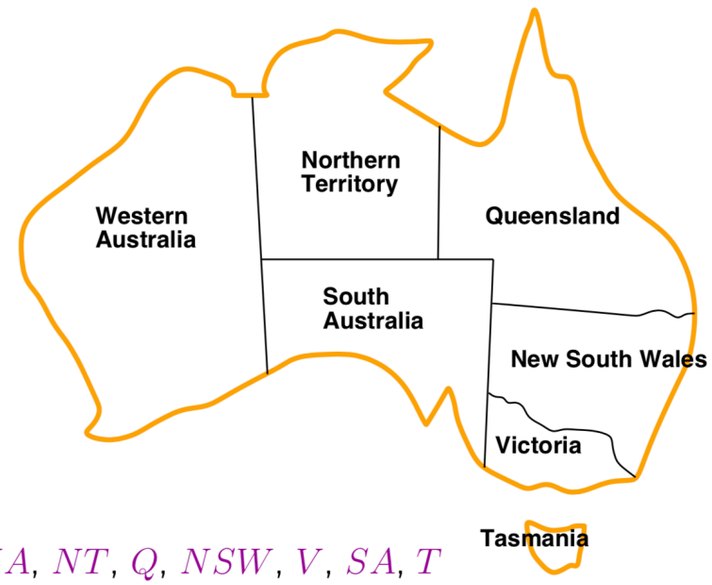


Assignment 1



- Repeated state checking. "Graph search".
    - What is a state for Pacman finding one goal? With multiple food squares?
- Heuristic
    - Consistency
        - Consistent:
            - The drop in h() for a node to its successor is no greater than the cost of that action.
            - Formally: For node n1 and successor n2 (generated by action a): h(n2) >= h(n1) - c(a)
        - Example showing why consistency is needed: Two paths to the same node with costs 99 and 100.
    - What is a consistent and admissible heuristic for pacman with one goal?
    - For multiple food squares: It's not right to calculate the sum of Manhattan distances from Pacman's position to each food square. Why?
    - Your heuristic should be constant in the size of the maze. The number of food squares is constant.
- You need to use Python 3.

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., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

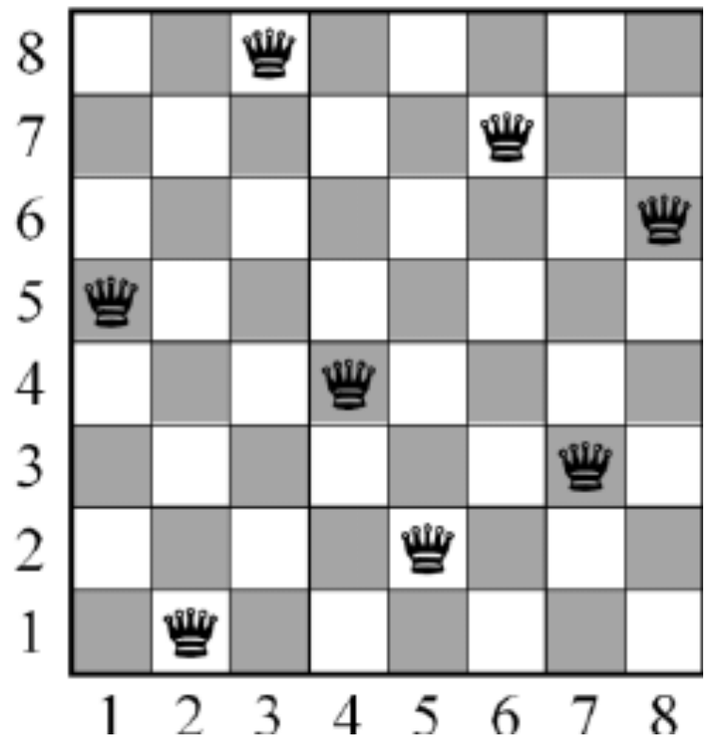
Satisfying assignment:
{WA=red,NT =green,Q=red,NSW =green,V =red,SA=blue,T =green}

Definition of a CSP

Three parts:

1) **Variables X1 ... Xn**
2) **Domains D1 ... Dn <- allowed values**
3) **Constraints.** Ex: (X1, X2), X1 != X2 -> ((R,B), (R,G), (B,G))

Example: 8-queens



Variables: Notice that each column must have exactly one queen. Variables: q_i = location of queen in column i.
Domains: {1..8} for all.
Constraints -- not allowed:
q_1,q_3 not allowed:
(1,1), (1,3),
(2,2), (2,4),
(3,1), (3,3), (3,5)
...
(8,8), (8,6)

Example: Sudoku

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

Variables: X_11 ... X_99 -- 81 total
Domains: {1, ... , 9} for each variable
Constraints:
- Informally: Every row, column and box
- Formally:
    - Row k: x_ki != x_kj for i,j in {1 ... 9}
    - (Same for columns, boxs)

Question 2.4

For each of the following activities, give a PEAS description of the task environments and characterize it in terms of properties.

a) Exploring the subsurface oceans of Titan.

b) Playing a tennis match.

c) Practicing tennis against a wall.

d) Knitting a sweater.

W4-Fri:

- Today: example problems.
- First: consistency example

Question 3.6

Give a complete problem formulation for each of the following. Choose a formulation that is precise enough to be implemented.

a) Using only four colors, you have to color a planar map in such a way that no two adjacent regions have the same color.

b) A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. He would like to get the bananas. The room contains two stackable, movable climbable 3-foot-high crates.

c) You have a program that outputs the message "illegal input record" when fed a certain file of input records. You know that processing of each record is independent of the other records. You want to discover what record is illegal.

d) You have three jugs, measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet. You can fill the jugs up or empty them out from one to another or onto the ground. You need to measure out exactly one gallon.

Question

Give a state space representation for the following problem:

A farmer wants to cross a river with his wolf, goat and cabbage. There is a boat that can carry the farmer plus one thing. If the wolf is left alone with the goat, it will eat the goat. Same for the goat and cabbage. How can they cross without anything getting eaten?

States: Defined by the items that have gone across -- none, F, W, G, C, FW, FG, FC, ..., FWGC.
Not allowed states: GC, WG, FW, FC, F, WGC.
Start: none. Goal: FWGC.
Actions: From where the farmer is, move the farmer plus one item: none -> FW. FWG -> G.
Solution: FG; G; FWG; W; FWC; WC; FWGC

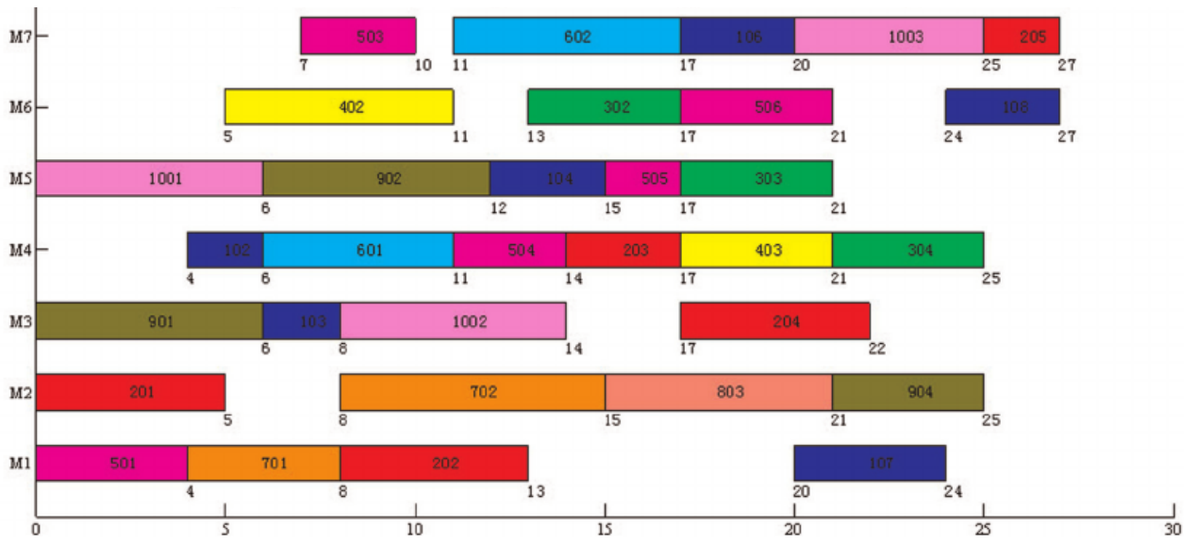
Question 5.1

Suppose you have an oracle that correctly predicts the opponent's move in any state. Using this, formulate the definition of a game as a (single-agent) search problem. Describe an algorithm for finding the optimal move.

Question 5.7

Prove the following assertion: For every game tree, the utility obtained by MAX using minimax decisions against a suboptimal MIN will never be lower than the utility obtained playing against an optimal MIN. Can you come up with a game tree in which MAX can do even better using a suboptimal strategy against a suboptimal MIN?

Example: Class scheduling



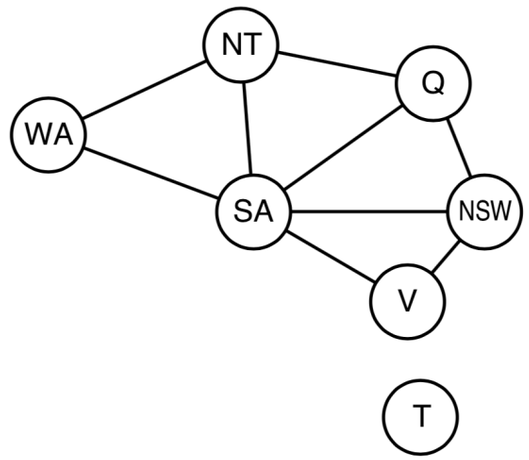
- Two options:
  - Variables = classes, domain = rooms.
  - Variables = rooms, domain = classes + none.
- Constraints:
  - Some rooms can't fit some classes.
  - No two classes can be in the same room; each class
  - Some classes can't be at the same time .

Other examples of CSPs

- What can you think of?
- Most logic puzzles: Sudoku, crossword, cross-sums.
- Scheduling:
    - Assembling a car: Each person/machine can only do one thing at a time. Some things need to be done before others.
    - Flights, transportation, assembly.
- Configuration: floor plan, hardware.

Binary CSPs

Binary CSP: each constraint involves at most two variables.



Binary CSP: each constraint involves at most 2 variables
We can represent variables that have a constraint with edges on a graph.

Using search to solve a CSP

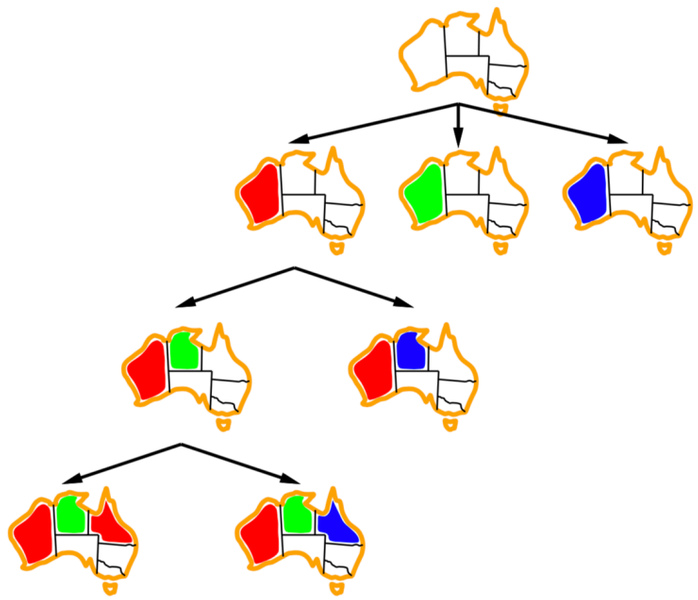
- We can use search to solve a CSP.
- How?
  - Initial state: empty assignment
  - Actions: Assign a value to a variable that does not result in violating constraints.
  - Goal: All variables assigned.
  - Costs: All zero.
- Any search algorithm works. Solution is always at depth n -> use DFS.
- How long does this take?
  - n = number of variables
  - d = size of domains
  - Number of actions <= remaining variables * d <= n*d
  - Depth of solution = n
  - O(n! d^n)
- That running time gives a hint of how to do better. There are only d^n solutions; the extra factor of n! comes from considering all possible orderings.
- We will do better: backtracking search

Backtracking search

```
def backtracking_search(csp, assignment={}):
    if complete(assignment):
        return assignment
    var = select_variable(csp, assignment)
    for val in order_values(csp, assignment, var):
        new_assn = assignment + {var=val}
        if not csp.consistent(new_assn): continue
        result = backtracking_search(csp, new_assn)
        if result != "failure": return result
    return "failure"
```

- Similar to DFS in that it uses recursion/stack.
- Primary difference: when we choose a variable we commit to trying all values for it. Notice that this means we test each assignment only once.

Backtracking search example



Properties of backtracking search

Complete:

Optimal:

Time:

Space:

n = number of variables
d = size of domain (number of values)

- Complete: Yes, same as DFS.
- Optimal: N/A. All solutions are the same.
- Time: O(d^n)
- Space: O(d*n)

Works for n-queens with n~=25. We will use heuristics to go faster.
We get to change select_variable() and order_values().

Properties of backtracking search

Question 5.12

Describe how the minimax and alpha-beta algorithms change for two-player, non-zero-sum games in which each player has a distinct utility function and both utility functions are known to both players. If there are no constraints on the two terminal utilities, is it possible for any node to be pruned by alpha-beta?

What if the players' utility functions on any state differ by at most a constant k, making the game almost cooperative?

Question 6.4

Give precise formulations for the following CSPs:

a) Rectilinear floor-planning: find non-overlapping places in a large rectangle for a number of smaller rectangles.

b) 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. Each professor has a list of classes that he or she can teach.

c) Hamiltonian tour: Given a network of cities connected by roads, choose an order to visit all cities without repeating any.

Improvements to backtracking search

select_variable()

- Minimum remaining values
- Maximum degree

order_values()

- Least constraining value

is_consistent()

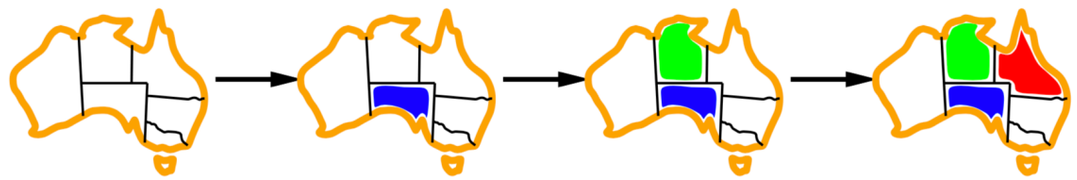
- Forward checking

- W40-Wed:
  - A1 due Friday
  - No class the next two Mondays: Oct 7 and Oct 14.

- Normal backtracking search is good enough for 25 queens.
- We can improve backtracking search with good choices.
- These are all heuristics -- no asymptotic speedup.

Minimum remaining values

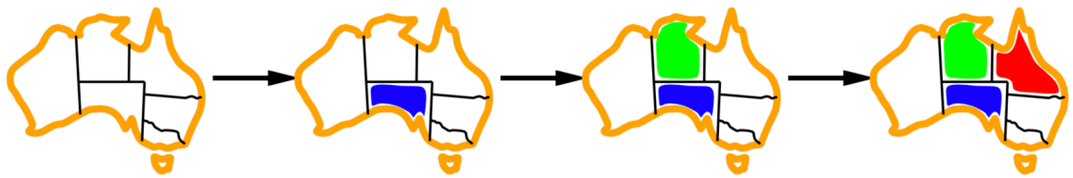
Minimum remaining values: Choose the variable with the fewest legal values



- 1: All the same
- 2: All neighbors have 2 possibilities, pick one of those.
- 3: Queensland has just 1 possibility, pick that.

Maximum degree

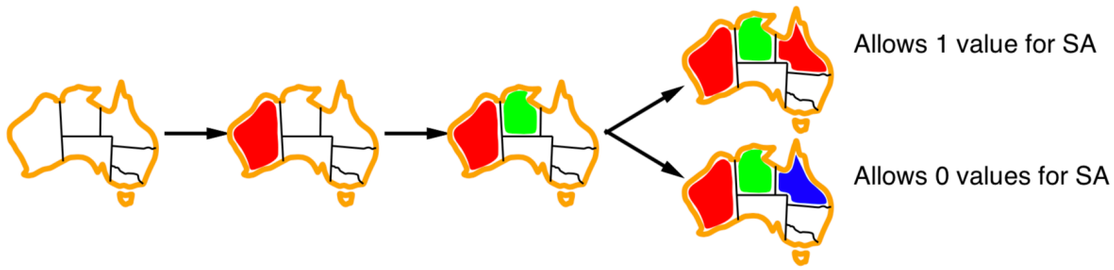
Maximum degree: Tiebreaker among MRV variables. Choose the variable with the most constraints on remaining variables.



- Degree of a variable is the number of constraints it participates in.
- Tiebreaker after MRV: pick the variable with maximum degree.
- SA = 5. WA = 2. NT = 3.

Least constraining value

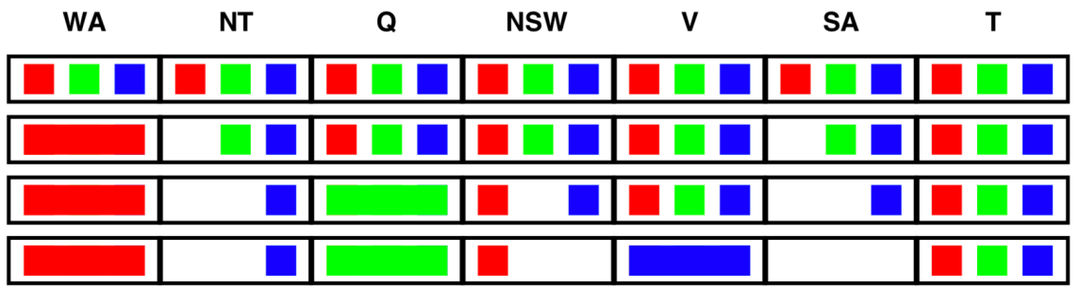
Given a variable, choose the value that rules out the fewest values in remaining variables.



- Combining these makes 1000-queens feasible.
- Note: All these heuristics take O(n) to compute. We're okay doing anything that is sub-exponential, since backtracking search is exponential.

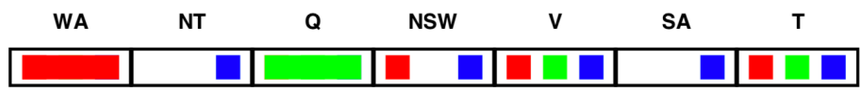
Forward checking

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



Arc consistency

$X \rightarrow Y$ is consistent iff
for **every** value x of X there is **some** allowed y



- Run arc consistency on X (with neighbors of X) every time X loses a legal value. When assigning to Y, will run on all of its neighbors.
- Just assigned green to Q. SA loses a legal value, so run arc consistency on that.
  - SA -> NSW: No problem, blue is okay.
  - NSW -> SA: Blue doesn't work, no value for SA. NSW just lost a value, now need to run with its neighbors again.
  - NSW -> V: All good
  - V -> NSW: Red doesn't work.
  - SA -> NT: Blue doesn't work. This solution is inconsistent.
- This is expensive: O(n^2 d^2). It's still helpful because it reduces the search depth of backtracking.

-

Local search

General framework:

1. Initialize: random assignments to variables.
2. Iterate: Choose a variable, change it.
3. Stop when solution is satisfactory.

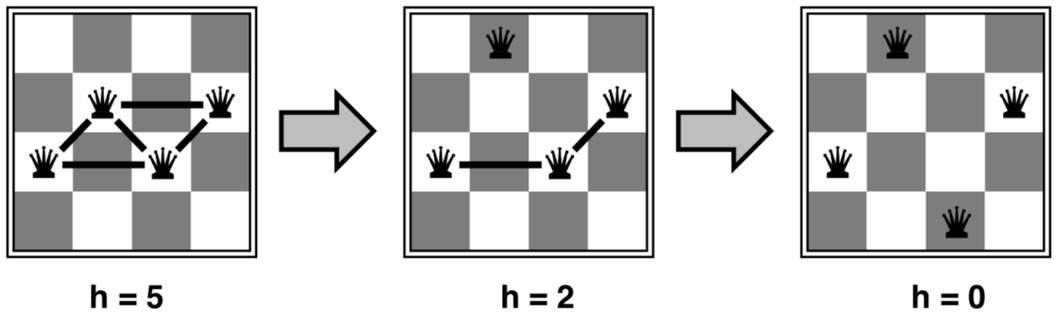
- So far we have looked at algorithms that start with an empty solution and build the solution until it satisfies all constraints. Tree search (DFS/BFS etc), backtracking search.
- We will look at a different type of algorithm: Local search algorithms start with a full solution (that violates constraints) and slowly improves it until it is satisfactory.
- It's easy to see that changing values randomly will eventually find a solution. (Perhaps very slowly.)
- We can do better by intelligently changing step 2.

Hill climbing search

```
def hill_climbing(csp, max_steps):
    current = choose_assignment(csp)
    for i in 1 .. max_steps:
        if csp.satisfies(current):
            return current
        var = choose_variable(csp, assignment)
        val = min_conflicts(csp, assignment, var)
        current[var] = val
    return "failure"
```

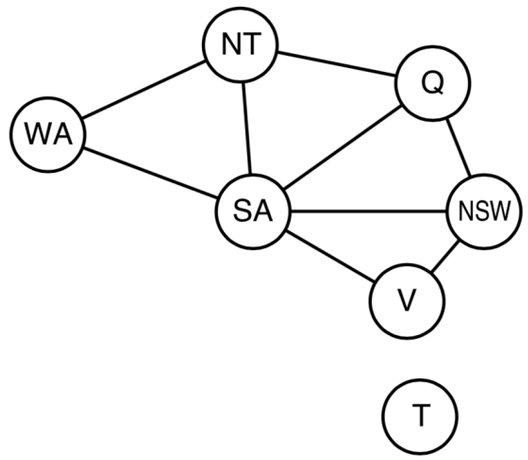
- We have choices in choose_variable,
- choose_variable: Should be a variable that violates some constraints.
- min_conflicts: whichever value violates the fewest assignments

Hill climbing search



- It turns out that hill climbing search solves n-queens very fast. Roughly constant (!) in n. Intuitively, that is because solutions are dense.
- We'll see more local search algorithms in logic, machine learning.
- Downside: local optima. We can get stuck where all variables cannot be changed without increasing number of violated constraints.
- Lots of possible improvements:
    - It can be best to sometimes just use a random variable+value. We'll talk about this a bit more in the next chapter.
    - Constraint weighting: prioritize important constraints.
    - Avoid recently-visited solutions.
    - Random restarts to avoid local optima.
- Which should you use?
    - Local search is often better for really hard problems. But it requires a lot of fine-tuning. You can never guarantee that it will work. These problems are still NP-hard -- local search can't solve them quickly in general.
    - Backtracking is much more

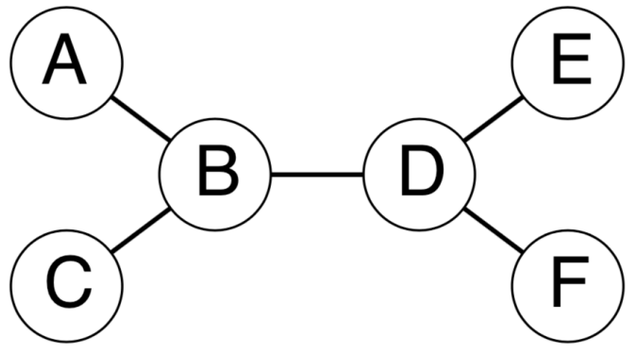
Structure of problems: independent subproblems



Tasmania and mainland are independent subproblems

- We can improve our algorithms if our problems have structure.
- One example: independent subproblems.
- **10 million nodes/sec , # variables = 80 , domain size = 2 , 4 independent subproblems.**
    - **All together: 2^80 -> 4 billion years**
    - **Separately: 4 * 2^20 -> 0.4 seconds**

Structure of problems: tree-structured CSPs



- **No loops -> O(n d^2)**
- Algorithm is in the book.
- We will see another example of a structured problem when we get to probability.

Question 6.6

Show how a single ternary constraint such as "A + B = C" can be turned into three binary constraints using an auxiliary variable. You may assume finite domains.

Hint: Consider a new variable that takes on values that are pairs of other values.

To express the ternary constraint on A, B and C that A + B = C, we first introduce a new variable, AB. If the domain of A and B is the set of numbers N, then the domain of AB is the set of pairs of numbers from N , i.e. N × N . Now there are three binary constraints, one between A and AB saying that the value of A must be equal to the first element of the pair-value of AB; one between B and AB saying that the value of B must equal the second element of the value of AB; and finally one that says that the sum of the pair of numbers that is the value of AB must equal the value of C. All other ternary constraints can be handled similarly.

Now that we can reduce a ternary constraint into binary constraints, we can reduce a 4-ary constraint on variables A,B,C,D by first reducing A,B,C to binary constraints as shown above, then adding back D in a ternary constraint with AB and C, and then reducing this ternary constraint to binary by introducing CD.

By induction, we can reduce any n-ary constraint to an (n – 1)-ary constraint. We can stop at binary, because any unary constraint can be dropped, simply by moving the effects of the constraint into the domain of the variable.

Question 6.9

Why is it a good heuristic to choose the *most* constrained variable but the *least* constrained value?

The most constrained variable makes sense because it chooses a variable that is (all other things being equal) likely to cause a failure, and it is more efficient to fail as early as possible (thereby pruning large parts of the search space). The least constraining value heuristic makes sense because it allows the most chances for future assignments to avoid conflict.