Tufts

Lecture 07:
Constraint Satisfaction, II

Artificial Intelligence (CS 131)

1

Review: CSPs & Backtracking

- Constraint satisfaction problem (CSP):
  1. State is a set of variables $X_1,\ldots,X_n$
  2. Each variable X_i has a domain D_i of possible values
  3. *Goal test:* a set of constraints $C_1,\ldots,C_m$ (*restrictions* on possible values of the variables)
- *Solution*: a complete assignment—each variable is assigned a possible value, without violating any constraint
- Backtracking search recursively generates partial solutions, adding or removing values from variables as it looks for a consistent combination

2

Improved Backtracking

Image source: Russell & Norvig (2021)

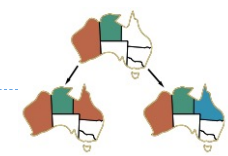
```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, assignment)
        if inferences ≠ failure then
           add inferences to csp
           result ← BACKTRACK(csp, assignment)
           if result ≠ failure then return result
           remove inferences from csp
        remove {var = value} from assignment
  return failure
```

By varying the order in which we select variables and values, we may improve algorithm efficiency dramatically

3

Variable and Value Ordering



- Variable ordering
  1. Minimum remaining values (most-constrained): expand variable with *fewest* legal *values* remaining
  2. Degree (most-constraining): expand a variable that is involved in the *largest* number of total *constraints*
- Value ordering
  - Least-constraining: choose a value that rules out the fewest choices for *other* variables

4

Most *Constrained* Variable

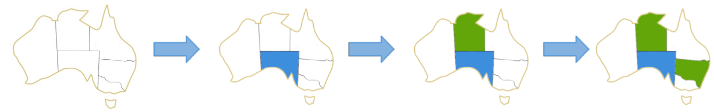
- Choose the variable with the fewest legal values:

- Ties are broken at random, or in some fixed order
- Also called minimum remaining values (MRV) heuristic
  - Like other heuristics we will see, often reduces solution time
  - This heuristic more quickly identifies possible dead ends, by narrowing down search space more quickly (early failure)

5

Most *Constraining* Variable

- Choose variable that most constrains the *others*
  - Measured by counting how many distinct unassigned variables appear in constraints with the current one

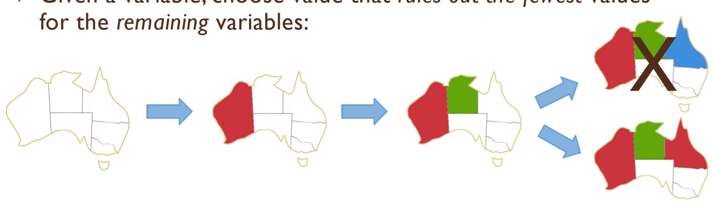


- May also find dead ends earlier by restricting search space
  - Can be used as a *tie-breaker* for the MRV heuristic
  - If there are multiple variables with the same minimal number of possible values, choose the one that affects the most others

6

Least Constraining Value

- Given a variable, choose value that *rules out the fewest* values for the *remaining* variables:



- While the other heuristics try to restrict search space to find dead-ends *faster*, this one does the opposite: it leaves the remaining search space as *open* as possible, so we don't explore *obvious* failures too often
- Combining these three heuristics together makes puzzles with sizes ≈ 1,000-Queens a feasible goal

7

Further Improvements

Image source: Russell & Norvig (2021)

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, assignment)
      if inferences ≠ failure then
        add inferences to csp
        result ← BACKTRACK(csp, assignment)
        if result ≠ failure then return result
        remove inferences from csp
      remove {var = value} from assignment
  return failure
```

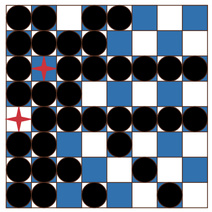
In simplest possible version, this step *does nothing* (can be deleted).

Instead, every time we add a new value-assignment, we can do some reasoning to determine what sorts of other assignments are or are not still possible

8

Constraint Propagation

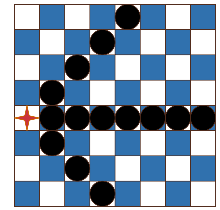
- The process of determining how the possible values of one variable affect the possible values of other variables



Given the current positions of the Queens (red),
we can rule out others (black)
without even having to check them

9

Forward Checking



- After variable X is assigned value v :
  1. Consider each *unassigned* variable Y that shares some constraint with X
  2. Delete any value inconsistent with v from the domain of Y
- Reduces branching factor and helps identify failures early

10

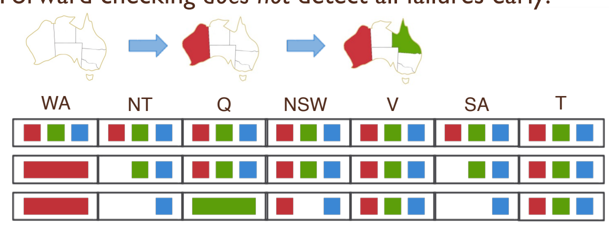
Forward Checking (Example)

- Keep track or remaining legal values for *unassigned* variables
  - Terminate search when any variable has no values left

11

Constraint Propagation

- Forward checking *does not* detect all failures early:



  - From this stage in the search, there is no point in continuing, since NT and SA cannot both be blue
- Better forms of constraint propagation *repeatedly* enforce constraints to detect such situations

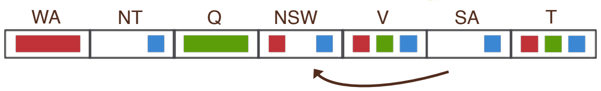
12

Arc Consistency

- Simplest form of propagation: make each constraint pair consistent
  - Instead of only checking forward, check backward as needed
- $X \rightarrow Y$ is consistent if and only if: for *every* value x of X, there remains *some* legal value y of Y

At this point, SA has only one possible value (Blue).

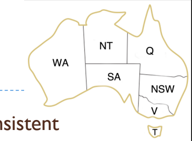
We check all neighbors, starting with NSW.



  - If X loses a value, neighbors of X must to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a *preprocessor* or *after* each assignment

13

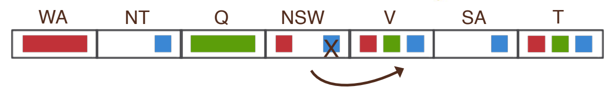
Arc Consistency



- Simplest form of propagation: make each constraint pair consistent
  - Instead of only checking forward, check backward as needed
- $X \rightarrow Y$ is consistent if and only if: for *every* value x of X, there remains *some* legal value y of Y

This means that NSW has lost a value (can't also be Blue).

We now check *its* neighbors, starting with V.



  - If X loses a value, neighbors of X must to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a *preprocessor* or *after* each assignment

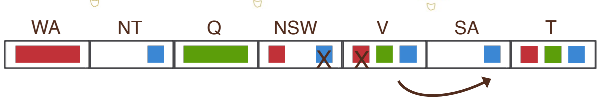
14

Arc Consistency

- Simplest form of propagation: make each constraint pair consistent
  - Instead of only checking forward, check backward as needed
- $X \rightarrow Y$ is consistent if and only if: for *every* value x of X, there remains *some* legal value y of Y

Now, V has lost a value (can't also be Red).

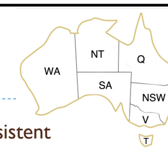
We now check *its* remaining neighbors, namely SA itself.



  - If X loses a value, neighbors of X must to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a *preprocessor* or *after* each assignment

15

Arc Consistency



- Simplest form of propagation: make each constraint pair consistent
  - Instead of only checking forward, check backward as needed
- $X \rightarrow Y$ is consistent if and only if: for *every* value x of X, there remains *some* legal value y of Y

V has lost another value (can't also be Blue). It has no more neighbors.

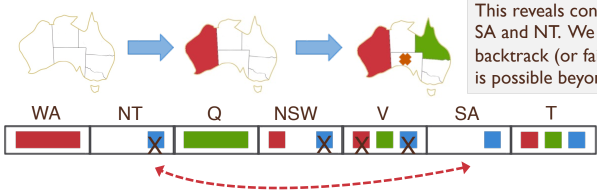
We return to SA and check next neighbor, NT.

  - If X loses a value, neighbors of X must to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a *preprocessor* or *after* each assignment

16

Arc Consistency

- Simplest form of propagation: make each constraint pair consistent
  - Instead of only checking forward, check backward as needed
- $X \rightarrow Y$ is consistent if and only if: for *every* value x of X, there remains *some* legal value y of Y



This reveals conflict between SA and NT. We would now backtrack (or fail): no solution is possible beyond this point.

  - If X loses a value, neighbors of X must to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a *preprocessor* or *after* each assignment

AC-3 for Propagation

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← POP(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
```

arc: a binary constraint

fails: if any variable has no legal values

re-checks: if X_i loses any value, will look at all neighbors again

Checks if X_i loses *at least* one value based upon legal values of X_j

AC-3 for Propagation

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← POP(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
```

$O(n^2 d)$

$O(d^2)$

Overall time complexity: $O(n^2 d^3)$

n: number of variables $\quad$ d: size of largest domain

Solving a CSP

- *Search*: can find good solutions, but can examine failed non-solutions along the way
- *Constraint Propagation*: can rule out non-solutions, but this is not the same as finding solutions
- *Interleaving* constraint propagation and search: perform constraint propagation *after* each step of search

Search for next value-variable assignment $\rightleftarrows$ Propagate the constraints over all variables again

Another Approach: Local Search for CSPs

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen conflicted variable from csp.VARIABLES
    value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)
    set var = value in current
  return failure
```

- Start state is some *random complete* assignment of values
  - *May violate* some constraints
- Successor: change value of one variable
- Improve: use heuristic to reduce number of conflicts
  - Min-conflicts: choose a value that minimizes the number of remaining conflicts
- Can solve million-queens problem in 50 steps on average!