Tufts

Lecture 06:
Constraint Satisfaction

Artificial Intelligence (CS 131)

1

Constraint Satisfaction Problems

- In a search problem, a state is often a "black box"
  - Any data structure can be used
  - Even heuristic search does not pay generally pay attention to the relationships between different states and their structures beyond the basic heuristic values
- 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 $C_1,\ldots,C_m$

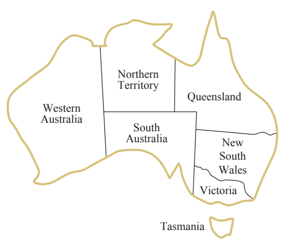
   (*restrictions* on possible values of the variables)

- *Solution*: a complete assignment—each variable is assigned a possible value, without violating any constraint

2

Example 1: Map Coloring

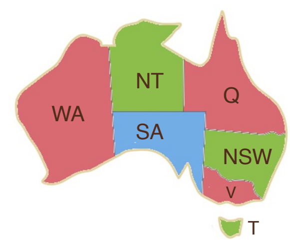
Image source: Russell & Norvig (2021)



1. *Variables*: WA, NT, Q, NSW, V, SA, T
2. *Domains*: D_i = {R, G, B}
3. *Constraints*: adjacent areas must have different colors, e.g.:
   - (WA,NT) ∈ { (R,G), (R,B), (G,R), (G,B), (B,R), (B,G) }
   - WA ≠ NT

3

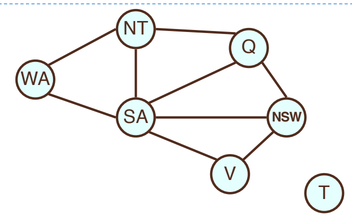
Example 1: Map Coloring



- *Solution:* a complete and consistent assignment
  - e.g., *WA = R, NT = G, Q = R, NSW = G, V = R, SA = B, T = G*

4

Constraint Graphs



- Binary CSP: each constraint involves $n \leq 2$ variables
- Constraint graph represents the problem:
  - Nodes are variables
  - Edges connect variables that occur together in any constraint; each such edge stores the necessary constraint information

Reducing to Binary Constraints

- Every CSP can be written to only involve constraints on *exactly two* variables
- Example: consider 3-ary constraint:

$$(A, B, C) \in \{ (1, 1, 0), (1, 2, 1) \}$$

- Create new variable AB with domain consisting of *pairs* from domains of A and B, and replace 3-ary version with:

$$(A, AB) \in \{ (1, (1, 1)), (1, (1, 2)) \}$$
$$(B, AB) \in \{ (1, (1, 1)), (2, (1, 2)) \}$$
$$(AB, C) \in \{ ((1, 1), 0), ((1, 2), 1) \}$$

Reducing to Binary Constraints, cont'd.

- Now consider 4-ary constraint:

$$(A, B, C, D) \in \{ (1, 1, 0, 3), (1, 2, 1, 2) \}$$

- Again, create new variable AB & replace constraint with:

$$(A, AB) \in \{ (1, (1, 1)), (1, (1, 2)) \}$$
$$(B, AB) \in \{ (1, (1, 1)), (2, (1, 2)) \}$$
$$(AB, C, D) \in \{ ((1, 1), 0, 3), ((1, 2), 1, 2) \}$$

- Now, the last 3-ary constraint can be replaced using the procedure seen before

Reducing to Binary Constraints, cont'd.

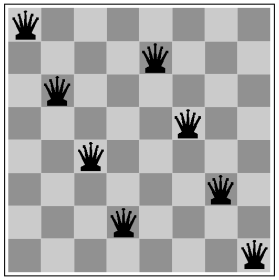
- Inductively, any n-ary constraint can be made $(n - 1)$-ary
- When all that is left are binary and unary, we can eliminate the unary ones entirely by domain reduction
- For example, if we have a variable X_i with:

Domain: $D_i = \{1, 2, 3, 4\}$ Constraint: $X_i \in \{1, 3\}$

- Simply ***replace*** the domain with the constraint itself:

Domain: $D_i = \{1, 3\}$

Example 2: *n*-Queens

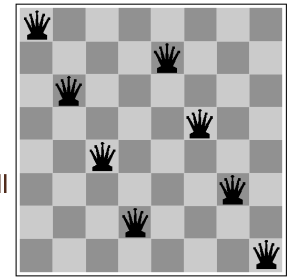


- What are the variables? Domains? Constraints?

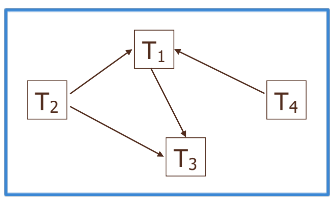
Example 2: 8-Queens

- 8 variables, one per Queen, Q_i ($i = 1,\ldots,8$)
- Domain for each is a possible position: $\{1,2,\ldots,8\}$
- Constraints are as follows for all pairs (i, j) such that $i \neq j$:

1. $Q_i \neq Q_j$
2. $|X^i - X^j| \neq |i - j|$



Example 3: Task Scheduling



1. T_1 must be done during T_3
2. T_2 must be achieved before T_1 starts
3. T_2 must overlap with T_3
4. T_4 must start after T_1 is complete

- The variables here are task start/end times
- What about the domains and constraints? How can we express something like constraint #4, above, in terms of pairs of values for the end of T_1 and the start of T_4?

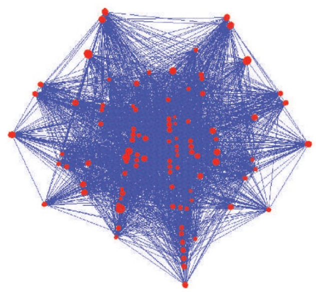
Finite vs. Infinite Domains

- Finite: *n*-Queens, matching mates, job assignment
  - Constraints can, if we wish, be ***listed explicitly***, in terms of all the possible pairs of values between constrained variables
- Infinite: job scheduling
  - Cannot usually just enumerate all the possibilities
  - We need a constraint language to express things concisely:

    $\text{Start-Job}_1 + 5 \leq \text{Start-Job}_2$
  - The choice of this language affects the complexity of checking constraint satisfaction
  - Programs may need significant computation to *check* whether or not some assignment satisfies all constraints

Real-World CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling



Bad Solution 1: Generate and Test

- A naïve algorithm:
  1. Generate *all possible combinations* of values (nested loops)
  2. Test each one to see if it satisfies constraints
  3. Terminate on first satisfying assignment or *fail* when all checked
- The complexity of this is far too high; if we have:
  1. A set of n variables
  2. A range of d possible domain values (e.g. values $1, 2, 3, \ldots, d$)

d^n possible value combinations ✖ n^2 possible binary constraints =

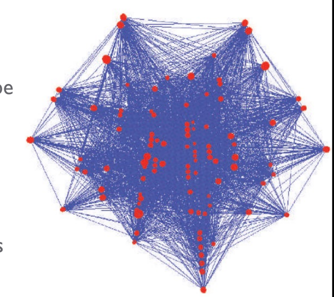
Bad Solution 2: Naïve Search

- Simple search in a tree of (partial/complete) solutions:
  1. Initial start state is set of unassigned variables
  2. Each branch chooses one unassigned variable and some value for it that does not cause a constraint failure
  3. Depth-First search, where each leaf is a *complete or failed* assignment
- Again, complexity of this is far too high, since we have:
  1. $(n \times d)$ choices for (variable, value) choices at first level
  2. $((n-1) \times d)$ choices at level 2, $((n-2) \times d)$ choices at level 3, etc.

$n!d^n$ possible paths down to leaves that must be explored =

Solving CSPs: What else is needed?

- We need more than a successor function and a goal test:
  1. Way to propagate constraints: checking how changes to values of one variable affect those that can be placed on all the others
  2. Early failure test, so we don't explore dead ends in our search
- Thus, we need:
  1. Precise way to *represent* constraints
  2. *Algorithms* to check constraint conflict, agreement, and satisfaction



One Improvement: Using Commutativity

- Naïve search ignores commutativity of CSPs
  - Whether a solution exists *does not depend upon the order* in which we assign variables, since all we care about is a final solution that is consistent
- Backtracking Search: a basic CSP solution algorithm
  - Exploit commutative structure by only changing single variable at each level (X_1 at first search level, X_2 at second, etc.)
  - This reduces total number of search paths: $n!d^n \Rightarrow d^n$
  - Allow search to *back up* by removing a variable assignment when failure occurs, rather than starting all over again

17

Backtracking Search

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
```

For any value chosen, we *recurse*, moving onto next variable in our ordering.

Failure test checks any result returned from lower levels.

- A simple recursive version of back-tracking
  - Even in this limited version, can solve n-Queens for $n \leq 25$

18

Backtracking Search

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
```

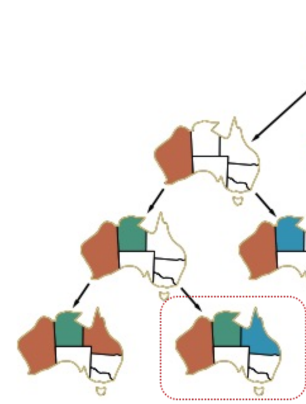
If a solution is not consistent, then the latest assignment is *removed* from the current set.

The algorithm repeats until either we succeed, or all values are used up and we *fail*.

19

Backtracking example

Image source: Russell & Norvig (2021)



Any search from this point on will fail, so this branch will back up to the parent node above it as necessary.

20

Improving Backtracking

- Depending upon how we answer the following questions, our search can go in a number of different ways:
  - *Which variable* should be assigned next and in *what order* should the values be tried?
  - What results does the current variable assignments have on *other* variables?
  - How to detect *dead-ends* early?
  - When a path fails, can we *avoid repeating* this failure in later search paths?

21

Improving Performance of Backtracking

```
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

22

Backtracking Search

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* (and can be deleted).

More complex versions will do some computation to enforce certain other properties on an assignment (to be covered next lecture).

23