Constraint Satisfaction Problems

31 October 2020 15:06

Instead of reasoning explicitly in terms of states, it is typically better to describe states in terms of **features** and to reason in terms of these features. Features are described using **variables**. Often features are not independent and there are **hard constraints** that specify legal combinations of assignments of values to variables

When the goal state is defined in terms of restraints and you're looking for a solution that satisfies the constraints

- Types of CSP's
- Backtracking Search
- Arc-consistency in a constraint graph
- Domain splitting to solve
- Using CSP Problem structure
- Variable elimination for CSP

What is a constraint satisfaction problem?

- A CSP is characterized by a set of variables
- Each variable has an associated domain of possible values
- There are hard constraints on various subsets of the variables which specify legal combinations of values for the variables
- A solution to the CSP is an assignment of a value to each variable that satisfies all the constraints

CSP's as optimisation problems

- For optimisation problems there is a function that gives a cost for each assignment of a value to each variable
- A solution is an assignment of values to the variables that minimizes the cost function

Types of Variables

1. Discrete Variable – domain is finite or countably infinite
2. Binary Variable – a discrete variable with two values in its domain e.g. Boolean variable
3. Continuous Variable – not discrete e.g. a variable that corresponds to a real line

Given these variables, an assignment on the set of variable is a function from the variables into the domains of the variables:

$\{X1, X2, \dots, Xk\}$ as $\{X1 = v1, X2 = v2, \dots, Xk = vk\}$, where vi is in $dom(X_i)$

A possible world is a complete assignment of variables. That is, a function from variables into values that assigns a value to every variable

We use variables because we can reason about many worlds with just a few variables

Types of Constraints

In many domains , not all possible assignments of values to variables are permissible
A constraint specifies legal combinations of assignments of values to some of the variables.

A scope is a set of variables

A relation on a scope is a function that returns true or false to an assignment on a scope
A constraint is a scope and a relation on S. It involves each of the variables in its scope

A model is a possible world that satisfies all of the constraint

Constraints are defined by their **intension** in terms of formulas or by their **extension**, listing all the assignments that are true. Constraints defined extensionally can be seen as relations of legal assignments as in relational databases.

- Unary constraints – involve a single variable e.g. some variable can't be green or > 10
- Binary constraints – involve pairs of variables that can't be equal
- Higher-order constraints – involve 3 or more variables
- Preference or soft constraints – e.g. 1005 is better than 0805

Constraint Satisfaction Problems

- A set of variables
- A domain for each variable
- A set of constraints

A finite CSP has a finite set of variables and a finite domain for each variable

Given a CSP, we can do some useful tasks:

- Determine or not whether there is a model
- Find a model
- Count the number of models
- Enumerate all the models
- Find the best model given a measure of how good models are
- Determine whether some statement holds in all models

Real World CSP's

- Assignment problems
- Timetabling
- Hardware configuration
- Spreadsheets
- Floor planning

Solving CSP's

- A CSP can be solved by graph-searching
- A node is an assignment of values to some of the variables
- The start node is the empty assignment
- A goal node is a total assignment that satisfies the constraints

Generate-and-Test Algorithms

A finite CSP could be solved by an exhaustive algorithm. The assignment space D is the set of total assignments. The generate and test algorithm can be run to just return the first variable found.

Generate the assignment space I.e. the set of total assignments
Test each assignment with constraints

$$\begin{aligned}\mathbf{D} &= \mathbf{D}_A \times \mathbf{D}_B \times \mathbf{D}_C \times \mathbf{D}_D \times \mathbf{D}_E \\ &= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \\ &\quad \times \{1,2,3,4\} \times \{1,2,3,4\} \\ &= \{\langle 1,1,1,1,1\rangle, \langle 1,1,1,1,2\rangle, \ldots, \langle 4,4,4,4,4\rangle\}.\end{aligned}$$

How many assignments need to be tested for n variables, each with a domain of d? D to the power of n.

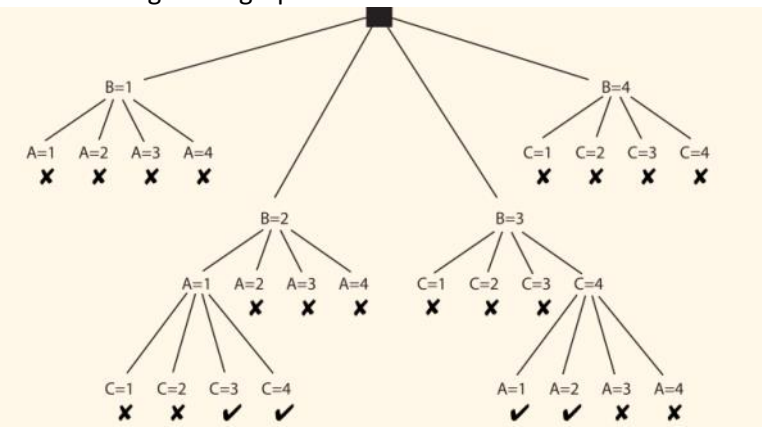
Algorithms for CSP's are trying to cut down this assignment space as it is so large.

Backtracking Algorithms

Generate-and-Test algorithms assign values to all variables before checking constraints. Because individual constraints only involve a subset of the variables, some constraints can be tested before all of the variables have been assigned values. If a partial assignment is inconsistent with the constraint then any total assignment that extends the partial assignment will also be inconsistent.

We can use backtracking instead, where we construct a search space that can be explored by the search algorithms discussed previously.

The following is the graph when A < B and B < C



1. Systematically explore D by instantiating the variables one at a time
2. Evaluate each constraint predicate as soon as all its variables are bound
3. Any partial assignment that does not satisfy the constraint can be pruned.

- Every solution appears at depth n so we can use a DFS
- Path is irrelevant so can use complete state formulation
- Branching factor b = (n-l)d at depth l hence n! x d^n leaves – top level branching factor is nd since any of d values can be assigned to any of n variables' next level branching factor si s(n-1) d and so on
- Variable assignments are commutative e.g. x=5, y=10 === y=10, x=5
- Only need to consider assignments to a single variable at each node
- Backtracking search si the basic uninformed algorithm for CSP's
- Can solve n-queens for n = 25

This is much more efficient that a generate-and-test algorithm because there we only test the constraints at the leaf nodes, whereas in a backtracking algorithm we prune all nodes immediately that violate a constraint, leaving us only with valid avenues to a possible model.

Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment,csp) returns solution/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given CONSTRAINTS[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment,csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```

Improving Search Efficiency

1. Which variable should be assigned next? MRV & degree heuristic
2. What order should we try its values? LCV
3. Can we detect inevitable failure early? Consistency Algorithms
4. Can we take advantage of the problem structure? Custset conditioning & variable elimination

Minimum remaining values (MRV)

Choose the variable with the fewest legal values

- Also called fail-first heuristic: will pick variable most likely to cause failure – if exists a variable with 0 possible assignments will pick and fail immediately

Degree Heuristic

Tie breaker among MRV variables

- Choose the variable with the most constraints on remaining variables
- Attempts to reduce branching factor of future choices

Least Constraining Value LCV

- Given a variable, choose the least constraining value: the one that rules out the fewest values in the remaining variables

Combining these we can solve n queens for n = 1000

Consistency Algorithms

Although DFS over the search space of assignments is usually a substantial improvement over generate and test, it still has various inefficiencies that can be overcome.

Idea: prune the domains as much as possible before selecting values from them

- A variable is domain consistent if no value of the domain of the node is ruled impossible by any of the constraints
- Example: if domain is Domain(B) = {1,2,3,4} and we have a rule that B can't equal 3, then D is not domain consistent

Constraint Network

1. There is a circular node for each variable

2. There is a rectangular node for each constraint
3. There is a domain of values associated with each variable node
4. There is an arc from variable X to each constraint that involves X

Arc Consistency

If there is a constraint that acts on some variables X, Y, Z, then the arc <X, c> can be called arc consistent if for each value of x in the domain(X) there are some values y and z in the domain(Y) and domain(Z) such that if we set X = x, Y = y, Z = z, the constraint is satisfied.

A network is arc-consistent if every arc is arc-consistent

What if an arc is not consistent?

If the arc <X, c> is not consistent then there are some values of X for which there are no values for Y, Z for which the constraint holds. In this case, all values of X in the domain(X) for which there are no corresponding values for the other variables can be deleted from the domain(X) to make the arc consistent.

When a value is removed from the domain of a variable it is possible that it will make some other arcs that were previously consistent, no longer consistent.

Arc Consistency Algorithm

See figure 4.3 in textbook

The arcs can be considered in turn making each arc consistent
When an arc has been made arc consistent, does it ever need to be checked again? YES – an arc needs to be revisited if the domain of one of following variables is reduced

Three possible outcomes when all arcs are consistent

- One domain is empty – no solution
- Each domain has a single value – unique solution
- Some domains have more than one value – there may or may not be a solution

Domain Splitting

Another method for simplifying the network is domain splitting, or case analysis
The idea is to split the problem into a number of disjoint cases and solve each case separately. The set of all solutions to the initial problem is the union of the solutions to each case.

In the simplest case, imagine we have a variable X with a domain of {t,f}. All solutions either have X = t, or X = f. One way to find the solutions is to set X = t, find all of the solutions with this assignment, then assign X = f, and find all those solutions.

If we make the problem into a smaller set of subproblems, we can get massive improvements in efficiency:

- Suppose each subproblem has c of n total variables
- There are n/c subproblems each of which takes at most d^c to solve. Worse case solution is therefore n/c x d^c, I.e linear in n

If the variable has > 2 elements, we can split it in a number of ways.

- Split it into a case for each value
- Always split the domain down the middle into 2 disjoint sets

We can be more efficient by interleaving arc consistency with the search
We would solve a CSP by using arc consistency to simplify the network before each step of domain splitting

After domain splitting, we do not need to start arc consistency from scratch. We can simply check the arcs that are possibly no longer arc consistent as a result of the split.

Complexity of Generalized Arc Consistency Algorithm

- If there are c binary constraints, and the domain of each variable is of size d. There are 2c arcs.
- Checking an arc <X, r(X,Y)> involves in the worst case iterating through each value in the domain of Y for each value in the domain of X, which takes d^2 time.
- This arc may need to be checked once for every element in the domain of Y, thus GAC for binary variables can be done in time O(cd^3) which is linear in C – the number of constraints
- The space used is O(nd) where n is the number of variables and d is the domain size.

Hard and Soft Constraints

Given a set of variables, assign a value to each variable that either:

- Satisfies some set of constraints – satisfiability problems – hard constraints
- Minimizes some cost function where each assignment of values to variables has some cost – optimisation problems – soft constraint
- Many problems are a mix of hard and soft constraints

Tree-Structured CSP's

Theorem: If the constraint graph has no loops, the CSP can be solved in O(nd^2)
Compare this to general CSP's, where worst case time is O(d^n)
This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.

Algorithm

Choose variable as root, order variables from root to leaves such that ever nodes parent precedes it in the ordering
For j from n down to 2, remove inconsistent domain elements for the parent.
For j from 1 to n, assign X consistently with Parent

Nearly Tree-Structured CSP's

- Conditioning – instantiate a variable, prune its neighbours domains, I.e. assign variable so remained is a tree.
- Cutset conditioning: instantiate a set of variables such that the remaining constraint graph is a tree
- Cutset size c -> runtime O(d^c x (n-c)d^2) very fast for small c.

Variable Elimination

Arc consistency simplifies the network by removing values of variables. A complementary method is variable elimination which simplifies the network by removing variables.

The idea is to remove the variables one by one. When removing a variable X, VE constructs a new constraint on some of the remaining variables, reflecting the effects of X on all the other variables.

This new constraint replaces all of the constraints that involve X, forming a reduced network that does not involve X.

When we eliminate X, the influence of X on the remaining variables is through the constraint relations that involve X. First, the algorithm collects all of the constraints that involve X.

Variable Elimination Algorithm

If there is only one variable return the intersection of the unary constraints that contain it

- Select a variable x
- Join the constraints in which X appears, forming constraint R1
- Project R1 onto its variables other than X, forming R2
- Replace all of the constraints in which X appears by R2
- Recursively solve the simplified problem, forming R3
- Return R1 joined with R3

- When there is a single variable remaining, if it has no values, the network was inconsistent
- The variables are eliminated according to some elimination ordering
- Different elimination orderings result in different size intermediate constraints

Figure 4.6 gives a recursive algorithm for variable elimination to find all the solutions for a CSP.

The number of variables n the largest relation returned for a particular variable ordering is called the **treewidth of the graph for that variable ordering**. The **treewidth of a graph is the minimum treewidth for an ordering**.

The complexity of VE is exponential in treewidth and linear in the number of variables.

There are some heuristics that exist that can help us get the minimum treewidth:

1. Min-factor – at each stage, select the variable that results in the smallest relation
2. Minimum deficiency or minimum fill – at each stage, select the variable that adds the fewest arcs to the remaining constraint network. The intuition is that it is okay to remove a variable that results in a large relation as long as it does not make the network more complicated.

Example: eliminating C

$r_1 : C \neq E$

C	E
3	2
3	4
4	2
4	3

$r_2 : D < C$

C	D
3	2
4	2
4	3

$r_3 : r_1 \bowtie r2$

C	D	E
3	2	2
3	2	4
4	2	2
4	2	3
4	3	2
4	3	3

$r_4 : \pi_{\{D,E\}} r_3$

D	E
2	2
2	3
2	4
3	2
3	3

↪ new constraint

NB: $r_1 \bowtie r_2$ = join of r_1 and r_2, $\pi_S(r)$ = projection of r onto S.