Constraint Satisfaction and Scheduling

Andrew W. Moore
Professor
School of Computer Science
Carnegie Mellon University
www.cs.cmu.edu/~awm
awm@cs.cmu.edu
412-268-7599

Overview

• CSPs defined
• Using standard search for CSPs
• Blindingly obvious improvements
  • Backtracking search
  • Forward Checking
  • Constraint Propagation
• Some example CSP applications
  • Overview
  • Waltz Algorithm
  • Job Shop Scheduling
• Variable ordering
• Value ordering
• Tedious Discussion

A Constraint Satisfaction Problem
Inside each circle marked \( V_1 \) to \( V_6 \), we must assign: R, G, or B.
No two connected circles may be assigned the same symbol.
Notice that two circles have already been given an assignment.

Formal Constraint Satisfaction Problem
A CSP is a triplet \( \{ V, D, C \} \). A CSP has a finite set of variables \( V = \{ V_1, V_2, \ldots, V_n \} \).
Each variable may be assigned a value from a domain \( D \) of values.
\( C \) are binary constraints. Each member of \( C \) is a pair. The first member of each pair is a set of variables. The second element is a set of legal values which that set may take.
Example:
\( V = \{ V_1, V_2, V_3, V_4, V_5, V_6 \} \)
\( D = \{ R, G, B \} \)
\( C = \{ (V_1, V_2) : \{ (R, G), (R, B), (G, R), (G, B), (B, R), (B, G) \},
  (V_3, V_4) : \{ (R, G), (R, B), (G, R), (G, B), (B, R), (B, G) \},
  \ldots \} \)

Obvious point: Usually \( C \) isn't represented explicitly, but by a function.

How to solve our CSP?
• How about using a search algorithm?
  • Define: a search state has variables 1 ... \( k \) assigned. Values \( k+1 \) ... \( n \) as yet unassigned.
  • Start state: All unassigned.
  • Goal state: All assigned, and all constraints satisfied.
  • Successors of a state with \( V_1 \) ... \( V_k \) assigned and rest unassigned are all states (with \( V_1 \) ... \( V_k \) the same) with \( V_{k+1} \) assigned a value from \( D \).
  • Cost on transitions: 0 is fine. We don't care. We just want any solution.
DFS for CSPs

What about DFS?
Much more popular. At least it has a chance of finding an easy answer quickly.
What happens if we do DFS with the order of assignments as B tried first, then G then R?
This makes DFS look very, very stupid!

Blindingly obvious improvement – Consistency Checking: “Backtracking Search”

Don’t ever try successor which causes inconsistency with its neighbors.
Again, what happens if we do DFS with the order of assignments as B tried first, then G then R?
What’s the computational overhead for this?
Backtracking still looks a little stupid!

Obvious improvement – Forward Checking
At start, for each variable, record the current set of possible legal values for it.
When you assign a value in the search, update set of legal values for all variables. Backtrack immediately if you empty a variable’s constraint set.
Again, what happens if we do DFS with the order of assignments as B tried first, then G then R?
What’s the computational overhead?

Constraint Propagation
Forward checking computes the domain of each variable independently at the start, and then only updates these domains when assignments are made in the DFS that are directly relevant to the current variable.
Constraint Propagation carries this further. When you delete a value from your domain, check all variables connected to you. If any of them change, delete all inconsistent values connected to them, etc...
Often constraint propagation is only performed at the start. In our example it is useless. But…

Constraint Propagation being non-useless
• In this example, constraint propagation solves the problem without search … Not always that lucky!
• Constraint propagation can be done as a preprocessing step. (Cheap).
• Or it can be maintained dynamically during the search. Expensive: when you backtrack, you must undo some of your additional constraints.

Graph-coloring-specific Constraint Propagation
In the case of Graph Coloring, CP looks simple: after we’ve made a search step (instantiated a node with a color), propagate the color at that node.

PropagateColorAtNode(node, color)
1. remove color from all of “available lists” of our uninstantiated neighbors.
2. If any of these neighbors gets the empty set, it’s time to backtrack.
3. Foreach n in these neighbors: if n previously had two or more available colors but now has only one color c, run PropagateColorAtNode(n, c)
Graph-coloring-specific Constraint Propagation

In the case of Graph Coloring, CP looks simple: after we’ve made a search step (instantiated a node with a color), propagate the color at that node.

PropagateColorAtNode(node,color)

1. remove color from all of “available” lists of our uninstantiated neighbors.
2. if any of these neighbors are empty, we’ve proved there are no solutions.
3. But for General CSP problems, constraint propagation can do much more than only propagate when a node gets a unique value.

General Constraint Propagation

Propagate(A₀, A₁, … Aᵢ)

finished = FALSE

while not finished

finished = TRUE

foreach constraint C

Assume C concerns variables Vᵰ, Vᵱ, … Vᵢ

Set NewAᵰ = {… , NewAᵱ = {… , NewAᵢ = {…

Foreach assignment (Vᵰ=α, Vᵱ=β, … Vᵢ=γ) in C

If xᵰ in Aᵰ and xᵱ in Aᵱ and … xᵢ in Aᵢ

Add xᵰ to NewAᵰ, xᵱ to NewAᵱ, … xᵢ to NewAᵢ

for i = 1, 2 … k

Aᵰ := Aᵰ intersection NewAᵰ

If Aᵰ was made smaller by that intersection

finished = FALSE

If Aᵰ is empty, we’re toast. Break out with “Backtrack” signal.

Details on next slide

A New CSP (where fancier propagation is possible)

- The semi magic square
- Each variable can have value 1, 2 or 3

V₁ V₂ V₃

This row must sum to 6

V₄ V₅ V₆

This row must sum to 6

V₇ V₈ V₉

This row must sum to 6

This column must sum to 6

This column must sum to 6

This diagonal must sum to 6

We’ll keep iterating until we do a full iteration in which none of the availability lists change. The “finished” flag is just to record whether a change took place.

Details on next slide
Propagate on Semi-magic Square

- The semi magic square
- Each variable can have value 1, 2 or 3

<table>
<thead>
<tr>
<th>1</th>
<th>123</th>
<th>123</th>
<th>This row must sum to 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
<td>This row must sum to 6</td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
<td>This row must sum to 6</td>
</tr>
</tbody>
</table>

This column must sum to 6
This column must sum to 6
This column must sum to 6
This diagonal must sum to 6

(V1,V2,V3) must be one of
(1,2,3)
(1,3,2)
(2,1,3)
(2,2,2)
(2,3,1)
(3,1,2)
(3,2,1)

New Alv1 = {1}
New Alv2 = {2,3}
New Alv3 = {2,3}

After doing first row constraint...

<table>
<thead>
<tr>
<th>1</th>
<th>23</th>
<th>23</th>
<th>This row must sum to 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>123</td>
<td>123</td>
<td>This row must sum to 6</td>
</tr>
<tr>
<td>23</td>
<td>123</td>
<td>123</td>
<td>This row must sum to 6</td>
</tr>
</tbody>
</table>

This column must sum to 6
This column must sum to 6
This column must sum to 6
This diagonal must sum to 6

After doing all row constraints and column constraints...

<table>
<thead>
<tr>
<th>1</th>
<th>23</th>
<th>23</th>
<th>This row must sum to 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>123</td>
<td>123</td>
<td>This row must sum to 6</td>
</tr>
<tr>
<td>23</td>
<td>123</td>
<td>123</td>
<td>This row must sum to 6</td>
</tr>
</tbody>
</table>

This column must sum to 6
This column must sum to 6
This column must sum to 6
This diagonal must sum to 6

And after doing diagonal constraint...

<table>
<thead>
<tr>
<th>1</th>
<th>23</th>
<th>23</th>
<th>This row must sum to 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>23</td>
<td>123</td>
<td>This row must sum to 6</td>
</tr>
<tr>
<td>23</td>
<td>123</td>
<td>23</td>
<td>This row must sum to 6</td>
</tr>
</tbody>
</table>

This column must sum to 6
This column must sum to 6
This column must sum to 6
This diagonal must sum to 6

CP has now iterated through all constraints once. But does it make further progress when it tries iterating through them again?
And after doing another round of constraints...

This row must sum to 6

This column must sum to 6

This diagonal must sum to 6

YES! And this showed a case of a constraint applying even when none of the variables involved was down to a unique value.

So... any more changes on the next iteration?

CSP Search with Constraint Propagation

CPSearch(A₁, A₂, ..., Aᵣ)

Let i = lowest index such that Aᵢ has more than one value

foreach available value x in Aᵢ

Define Aᵢ' = { x }

Call Propagate(A₁', A₂', ..., Aᵣ')

If no "Backtrack" signal

If A₁', A₂', ..., Aᵣ' are all unique we're done!

Recursively Call CPSearch(A₁', A₂', ..., Aᵣ')

Details on next slide

Slide 26

CSP Search with Constraint Propagation

CPSearch(A₁, A₂, ..., Aᵣ)

Let i = lowest index such that Aᵢ has more than one value

foreach available value x in Aᵢ

Define Aᵢ' = Aᵢ

Aᵢ' := { x }

Call Propagate(A₁', A₂', ..., Aᵣ')

If no "Backtrack" signal

If A₁', A₂', ..., Aᵣ' are all unique we're done!

Recursively Call CPSearch(A₁', A₂', ..., Aᵣ')

What's the top-level call?

Call with that Aᵢ = complete set of possible values for Vᵢ.

Slide 28

CSP Search with Constraint Propagation

CPSearch(A₁, A₂, ..., Aᵣ)

Let i = lowest index such that Aᵢ has more than one value

foreach available value x in Aᵢ

Define Aᵢ' := Aᵢ

Aᵢ' := { x }

Call Propagate(A₁', A₂', ..., Aᵣ')

If no "Backtrack" signal

If A₁', A₂', ..., Aᵣ' are all unique we're done!

Recursively Call CPSearch(A₁', A₂', ..., Aᵣ')

What's the top-level call?

Call with that Aᵢ = complete set of possible values for Vᵢ.

Slide 29

CSP Search with Constraint Propagation

CPSearch(A₁, A₂, ..., Aᵣ)

Let i = lowest index such that Aᵢ has more than one value

foreach available value x in Aᵢ

Define Aᵢ' := Aᵢ

Aᵢ' := { x }

Call Propagate(A₁', A₂', ..., Aᵣ')

If no "Backtrack" signal

If A₁', A₂', ..., Aᵣ' are all unique we're done!

Recursively Call CPSearch(A₁', A₂', ..., Aᵣ')

What’s the top-level call?

Call with that Aᵢ = complete set of possible values for Vᵢ.
Semi-magic Square

CPSearch Tree

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>23</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
</tr>
</tbody>
</table>

In fact, we never even consider these because we stop at first success.

Some real CSPs

- Graph coloring is a real, and useful, CSP. Applied to problems with many hundreds of thousands of nodes. Not very AI-esque.
- VLSI or PCB board layout.
- Selecting a move in the game of "minesweeper".

Which squares have a bomb? Squares with numbers don’t. Other squares might. Numbers tell how many of the eight adjacent squares have bombs. We want to find out if a given square can possibly have a bomb.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

"Minesweeper" CSP

\[ V = \{ V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8 \} \]
\[ D = \{ (\text{bomb}), (\text{space}) \} \]
\[ C = \{ (V_i, V_j): \{ (\text{bomb}, \text{space}), (\text{space}, \text{space}), (\text{space}, \text{bomb}) \} \} \]

The Waltz algorithm

One of the earliest examples of a computation posed as a CSP.

The Waltz algorithm is for interpreting line drawings of solid polyhedra.

The junctions must be labeled so that lines are labeled consistently at both ends.

Waltz Alg. on simple scenes

Assume all objects:
- Have no shadows or cracks
- Three-faced vertices
- "General position": no junctions change with small movements of the eye.

18 legal kinds of junctions

Given a representation of the diagram, label each junction in one of the above manners.

The junctions must be labeled so that lines are labeled consistently at both ends.

Can you formulate that as a CSP? **FUN FACT:** Constraint Propagation always works perfectly.
Scheduling

A very big, important use of CSP methods.
- Used in many industries. Makes many multi-million dollar decisions.
- Used extensively for space mission planning.
- Military uses.

People really care about improving scheduling algorithms!

Problems with phenomenally huge state spaces. But for which solutions are needed very quickly.

Many kinds of scheduling problems e.g.:
- Job shop: Discrete time; weird ordering of operations possible; set of separate jobs.
- Batch shop: Discrete or continuous time; restricted operation of ordering; grouping is important.
- Manufacturing cell: Discrete, automated version of open job shop.

Job Shop scheduling

At a job-shop you make various products. Each product is a "job" to be done. E.G.
- Job_1: Make a polished-thing-with-a-hole
- Job_2: Paint and drill a hole in a widget

Each job requires several operations. E.G.
- Operations for Job_1: Polish, Drill
- Operations for Job_2: Paint, Drill

Each operation needs several resources. E.G.
- Polishing needs the Polishing machine
- Drilling needs the Drill

Scheduling Formalized

A Job Shop problem is a pair (J, RES)
- J is a set of jobs \( J = \{ j_1, j_2, \ldots, j_n \} \)
- RES is a set of resources \( RES = \{ R_1, R_2, \ldots \} \)

Each job \( j_i \) is specified by:
- a set of operations \( O_i = \{ O_{i1}, O_{i2}, \ldots, O_{im} \} \)
- and must be carried out between release-date \( rd_i \) and due-date \( dd_i \).
- and a partial order of operations: \( O_{ij} \) before \( O_{jk} \), etc...

Each operation \( O_{ij} \) has a variable start time \( st_i \) and a fixed duration \( du_i \) and requires a set of resources. e.g. \( O_{ij} \) requires \( \{ R_{ij1}, R_{ij2}, \ldots \} \).

Each resource can be accomplished by one of several possible physical resources, e.g. \( R_{ij1} \) might be accomplished by any one of \( \{ r_{ij11}, r_{ij12}, \ldots \} \). Each of the \( r_{ijk} \)s are a member of RES.

Job Shop Example

\( j_1 \) = polished-hole-thing \( = \{ O_{11}, O_{12} \} \)
\( j_2 \) = painted-hole-widget \( = \{ O_{21}, O_{22} \} \)

RES = \{ Pat, Chris, Drill, Paint, Drill, Polisher \}

\( O_{11} \) = polish-thing: need resources...
- \( R_{111} = \) Polishing needs the Polishing machine

\( O_{12} \) = drill-thing: need resources...
- \( R_{121} = \) Drilling needs the Drill

\( O_{21} \) = paint-widget: need resources...
- \( R_{211} = \) Paint needs the Paint machine

\( O_{22} \) = drill-widget: need resources...
- \( R_{221} = \) Drilling needs the Drill

Precedence constraints: \( O_{12} \) before \( O_{21} \). All operations take one time unit \( du_i = 1 \) for all \( i \). Both jobs have release-date \( rd_i = 0 \) and due-date \( dd_i = 1 \).
Job-shop: the Variables and Constraints

Variables
• The operation state times $s_i$.
• The resources $R_i$ (usually these are obvious from the definition of $O_i$. Only need to be assigned values when there are alternative physical resources available, e.g. Pat or Chris for operating the drill).

Constraints:
• Precedence constraints. (Some $O_i$s must be before some other $O_j$s).
• Capacity constraints. There must never be a pair of operations with overlapping periods of operation that use the same resources.

Non-challenging question. Can you schedule our Job-shop?

A slightly bigger example


4 jobs. Each 3 units long. All jobs have release date 0 and due date 15. All operations use only one resource each.

Further CSP techniques

Let’s look at some other important CSP methods. Keep the job-shop example in mind.

Here’s another graph-coloring example (you’re now allowed R, G, B and Y)

General purpose Variable Ordering Heuristics

1. Most constrained variable.
2. Most constraining variable.

General purpose Value Ordering Heuristics

A good general purpose one is “least-constrained-value”. Choose the value which causes the smallest reduction in number of available values for neighboring variables

General purpose CSP algorithm

(From Sadeh+Fox)

1. If all values have been successfully assigned then stop, else go on to 2.
2. Apply the consistency enforcing procedure (e.g. forward-checking if feeling computationally mean, or constraint propagation if extravagant. There are other possibilities, too.)
3. If a deadend is detected then backtrack (simplest case: DFS-type backtrack. Other options can be tried, too). Else go on to step 4.
4. Select the next variable to be assigned (using your variable ordering heuristic).
5. Select a promising value (using your value ordering heuristic).
6. Create a new search state. Cache the info you need for backtracking. And go back to 1.
Job-shop example. Consistency enforcement

Sadeh claims that generally forward-checking is better, computationally, than full constraint propagation. But it can be supplemented with a Job-shop specific TRICK.

The precedence constraints (i.e. the available times for the operations to start due to the ordering of operations) can be computed exactly, given a partial schedule, very efficiently.

Reactive CSP solutions

- Say you have built a large schedule.
- Disaster! Halfway through execution, one of the resources breaks down. We have to reschedule!
- Bad to have to wait 15 minutes for the scheduler to make a new suggestion.

  Important area of research: efficient schedule repair algorithms.

- Question: If you expect that resources may sometimes break, what could a scheduling program do to take that into account?
- Unrelated Question: Why has none of this lecture used A*?

Other approaches. And What You Should Know

Other Approaches:
- Hill-climbing, Tabu-search, Simulated annealing, Genetic Algorithms. (to be discussed later)

What you should know:
- How to formalize problems as CSPs
- Backtracking Search, Forward Checking, Constraint Propagation
- The Waltz algorithm
- You should understand and appreciate the way job-shop scheduling is formalized. It is an excellent representative example of how important well-studied constraint satisfaction problems are represented.
- Understand examples of Variable ordering and Value ordering heuristics

In those cases where your lecturer or these handouts are too incomprehensible, consult the Russell and Norvig. Winston's "Artificial intelligence" book also has good discussion of constraint satisfaction and Waltz algorithm.