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 $V_1 .. V_6$ 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 $V = \{V_1, V_2, .. V_n\}$.
Each variable may be assigned a value from a domain $D$ of values.
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:
$V = \{V_1, V_2, V_3, V_4, V_5, V_6\}$
$D = \{R, G, B\}$
$C = \{(V_1, V_2) : \{(R,G), (R,B), (G,R), (G,B), (B,R) (B,G)\},$
$\quad \{(V_1, V_3) : \{(R,G), (R,B), (G,R), (G,B), (B,R) (B,G)\},$
$\quad \vdots \}$

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 V1 … Vk assigned and rest unassigned are all states (with V1 … Vk the same) with Vk+1 assigned a value from D.
- Cost on transitions: 0 is fine. We don’t care. We just want any solution.

What search algorithms could we use?
It turns out BFS is not a popular choice. Why not?
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!
Example: [http://www.cs.cmu.edu/~awm/animations/constraint/9d.html](http://www.cs.cmu.edu/~awm/animations/constraint/9d.html)

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...

In the above example it is useless

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.

\[
\text{PropagateColorAtNode}(\text{node}, \text{color})
\]

1. remove color from all of "available lists" of our uninstantiated neighbors.
2. If any of these neighbors get the empty set, it's time to backtrack.
3. For each neighbor \(n\) in these neighbors: if \(n\) previously had two or more available colors but now has only one color \(c\), run PropagateColorAtNode(n, c).

But for General CSP problems, constraint propagation can do much more than only propagating when a node gets a unique value…

A New CSP (where fancier propagation is possible)

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>This row must sum to 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V_1)</td>
<td>(V_2)</td>
<td>(V_3)</td>
<td></td>
</tr>
<tr>
<td>(V_4)</td>
<td>(V_5)</td>
<td>(V_6)</td>
<td></td>
</tr>
<tr>
<td>(V_7)</td>
<td>(V_8)</td>
<td>(V_9)</td>
<td></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
General Constraint Propagation

Propagate($A_1, A_2, \ldots, A_n$)

finished = FALSE
while not finished

finished = TRUE
foreach constraint C

Assume C concerns variables $V_1, V_2, \ldots, V_k$

Set New$A_{V_1} = \emptyset$, New$A_{V_2} = \emptyset$, \ldots, New$A_{V_k} = \emptyset$

Foreach assignment $(V_1=x_1, V_2=x_2, \ldots, V_k=x_k)$ in C

If $x_1$ in $A_{V_1}$ and $x_2$ in $A_{V_2}$ and \ldots $x_k$ in $A_{V_k}$

Add $x_1$ to New$A_{V_1}$, $x_2$ to New$A_{V_2}$ \ldots $x_k$ to New$A_{V_k}$

for $i = 1, 2 \ldots k$

$A_{V_i} := A_{V_i}$ intersection New$A_{V_i}$

If $A_{V_i}$ was made smaller by that intersection

finished = FALSE

If $A_{V_i}$ is empty, we’re toast. Break out with “Backtrack” signal.

Details on next slide
General Constraint Propagation

Propagate($A_1, A_2, ..., A_n$)

1. finished = FALSE
2. while not finished
3. finished = TRUE
4. foreach constraint $C$
   1. Assume $C$ concerns variables $V_1, V_2, ..., V_k$
   2. Set $NewA_{V_1} = \{\}$, $NewA_{V_2} = \{\}$, ..., $NewA_{V_k} = \{\}$
   3. Foreach assignment $(V_i=x_1, V_2=x_2, ..., V_k=x_k)$ in $C$
      1. If $x_1$ in $A_{V_1}$ and $x_2$ in $A_{V_2}$ and ... $x_k$ in $A_{V_k}$
         1. Add $x_1$ to $NewA_{V_1}$, $x_2$ to $NewA_{V_2}$, ..., $x_k$ to $NewA_{V_k}$
      2. for $i = 1, 2, ..., k$
         1. $A_{V_i} := A_{V_i}$ intersection $NewA_{V_i}$
         2. If $A_{V_i}$ was made smaller by that intersection
            1. finished = FALSE
   4. If $A_{V_i}$ is empty, we're toast. Break out with "Backtrack" signal.

After we've finished all the iterations of the foreach loop, $NewA_i$ contains the full set of possible values of variable $V_i$, taking into account the effects of constraint $C$.

If this test is satisfied that means that there's at least one value $q$ such that we originally thought $q$ was an available value for $V_i$ but we now know $q$ is impossible.

If $A_{V_i}$ is empty we've proved that there are no solutions for the availability-lists that we originally entered the function with.
Propagate on Semi-magic Square

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>123</th>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>123</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
<td></td>
</tr>
</tbody>
</table>

This row must sum to 6
This row must sum to 6
This row must sum to 6

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

(V_1,V_2,V_3) 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)

Propagate on Semi-magic Square

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>123</th>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>123</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
<td></td>
</tr>
</tbody>
</table>

This row must sum to 6
This row must sum to 6
This row must sum to 6

This column must sum to 6
This column must sum to 6
This column must sum to 6
This diagonal must sum to 6
Propagate on Semi-magic Square

• Each variable can have value 1, 2 or 3

This row 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...

This row 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></th>
<th>23</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>23</td>
<td>123</td>
<td>123</td>
</tr>
</tbody>
</table>

This row 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></th>
<th>23</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>123</td>
</tr>
<tr>
<td>23</td>
<td>123</td>
<td>23</td>
</tr>
</tbody>
</table>

This row must sum to 6
This row must sum to 6
This row must sum to 6
This column must sum to 6
This column must sum to 6
This column 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...

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

CSP Search with Constraint Propagation

CPSearch($A_1, A_2, \ldots, A_n$)

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

foreach available value $x$ in $A_i$

foreach $k$ in 1, 2.. $n$

Define $A'_k := A_k$

$A'_i := \{ x \}$

Call Propagate($A'_1, A'_2, \ldots, A'_n$)

If no “Backtrack” signal

If $A'_1, A'_2, \ldots, A'_n$ are all unique we’re done!

Recursively Call CPSearch($A'_1, A'_2, \ldots, A'_n$)
CSP Search with Constraint Propagation

CPSearch($A_1, A_2, \ldots, A_n$)

Let $i = \text{lowest index such that } A_i \text{ has more than one value}$

foreach available value $x$ in $A_i$

foreach $k$ in $1, 2, \ldots, n$

Define $A'_k := A_k$

$A'_i := \{ x \}$

Call Propagate($A'_1, A'_2, \ldots, A'_n$)

If no “Backtrack” signal

If $A'_1, A'_2, \ldots, A'_n$ are all unique we’re done!

Recursively Call CPSearch($A'_1, A'_2, \ldots, A'_n$)

Specification: Find out if there’s any combination of values in the combination of the given availability lists that satisfies all constraints.

At this point the A-primes are a copy of the original availability lists except $A'_i$ has committed to value $x$.

This call may prune away some values in some of the copied availability lists.

Assuming that we terminate deep in the recursion if we find a solution, the CPSearch function only terminates normally if no solution is found.

What’s the top-level call?
CSP Search with Constraint Propagation

CPSearch(A_1, A_2, ..., A_n)
   Let i = lowest index such that A_i has more than one value
   foreach available value x in A_i
       foreach k in 1, 2.. n
           Define A'_k := A_k
           A'_i := { x }
       Call Propagate(A'_1, A'_2, ..., A'_n)
   If no “Backtrack” signal
       If A'_1, A'_2, ..., A'_n are all unique we’re done!
       Recursively Call CPSearch(A'_1, A'_2, ..., A'_n)

What’s the top-level call?
Call with that A_i = complete set of possible values for V_i.
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...
“Minesweeper” CSP

\[ V = \{ V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8 \}, D = \{ B \text{ (bomb)}, S \text{ (space)} \} \]

\[ C = \{(V_1, V_2) : \{ (B, S), (S, B) \}, (V_1, V_2, V_3) : \{ (B, S, S), (S, B, S), (S, S, B) \}, \ldots \} \]

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.

Look at all intersections.

What kind of intersection could this be? A concave intersection of three faces? Or an external convex intersection?

Adjacent intersections impose constraints on each other. Use CSP to find a unique set of labelings. Important step to “understanding” the image.
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.

Then each line on image is one of the following:
- Boundary line (edge of an object) (<) with right hand of arrow denoting “solid” and left hand denoting “space”
- Interior convex edge (+)
- Interior concave edge (-)

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.
Waltz Examples
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.

\[ \text{Job}_1 = \text{Make a polished-thing-with-a-hole} \]
\[ \text{Job}_2 = \text{Paint and drill a hole in a widget} \]

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
Polishing needs Pat (a Polishing expert)
Drilling needs the Drill
Drilling needs Pat (also a Drilling expert)
Or Drilling can be done by Chris

Some operations need to be done in a particular order (e.g. Paint after you’ve Drilled)
Job Shop 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 \(RES = \{R_1, \ldots, R_m\}\)

Each job \(j_i\) is specified by:
- a set of operations \(O^i = \{O^i_1, O^i_2, \ldots, O^i_n(i)\}\)
- and must be carried out between release-date \(rd_i\) and due-date \(dd_i\)
- and a partial order of operations: \((O^i_j, \text{before}, O^i_k), (O^i_j, \text{before}, O^i_l), \text{etc…}\)

Each operation \(O^i_j\) has a variable start time \(st^i_j\) and a fixed duration \(du^i_j\) and requires a set of resources. e.g.: \(O^i_j\) requires \(\{R^i_{j1}, R^i_{j2}, \ldots\}\).

Each resource can be accomplished by one of several possible physical resources, e.g. \(R^i_{j1}\) might be accomplished by any one of \(\{r^i_{j11}, r^i_{j12}, \ldots\}\). Each of the \(r^i_{jk}\)'s are a member of \(RES\).

Job Shop Example

\(j_1 = \text{polished-hole-thing} = \{O^1_1, O^1_2\}\)

\(j_2 = \text{painted-hole-widget} = \{O^2_1, O^2_2\}\)

\(RES = \{\text{Pat}, \text{Chris}, \text{Drill}, \text{Paint}, \text{Drill}, \text{Polisher}\}\)

\(O^1_1 = \text{polish-thing: need resources…}\)
  \(\{R^1_{j1} = \text{Pat}, R^1_{j2} = \text{Polisher}\}\)

\(O^1_2 = \text{drill-thing: need resources…}\)
  \(\{R^1_{j2,1} = \text{Pat or } r^1_{j2,1} = \text{Chris}, R^1_{j2,2} = \text{Drill}\}\)

\(O^2_1 = \text{paint-widget: need resources…}\)
  \(\{R^2_{j1} = \text{Paint}\}\)

\(O^2_2 = \text{drill-widget: need resources…}\)
  \(\{R^2_{j1,1} = \text{Pat or } r^2_{j1,1} = \text{Chris}, R^2_{j2,2} = \text{Drill}\}\)

Precedence constraints: \(O^2_2\) before \(O^2_1\). All operations take one time unit \(du^i_j\) = 1 for all \(j, i\),. 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 $st_i$
- The resources $R_{ij}$ (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.
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.
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 Chap 5 of the Russell handout. Winston’s “Artificial Intelligence” book has good discussion of constraint satisfaction and Waltz algorithm.*