THE UNIVERSITY *of* ADELAIDE

Episodic memory (retrievals) as a CSP

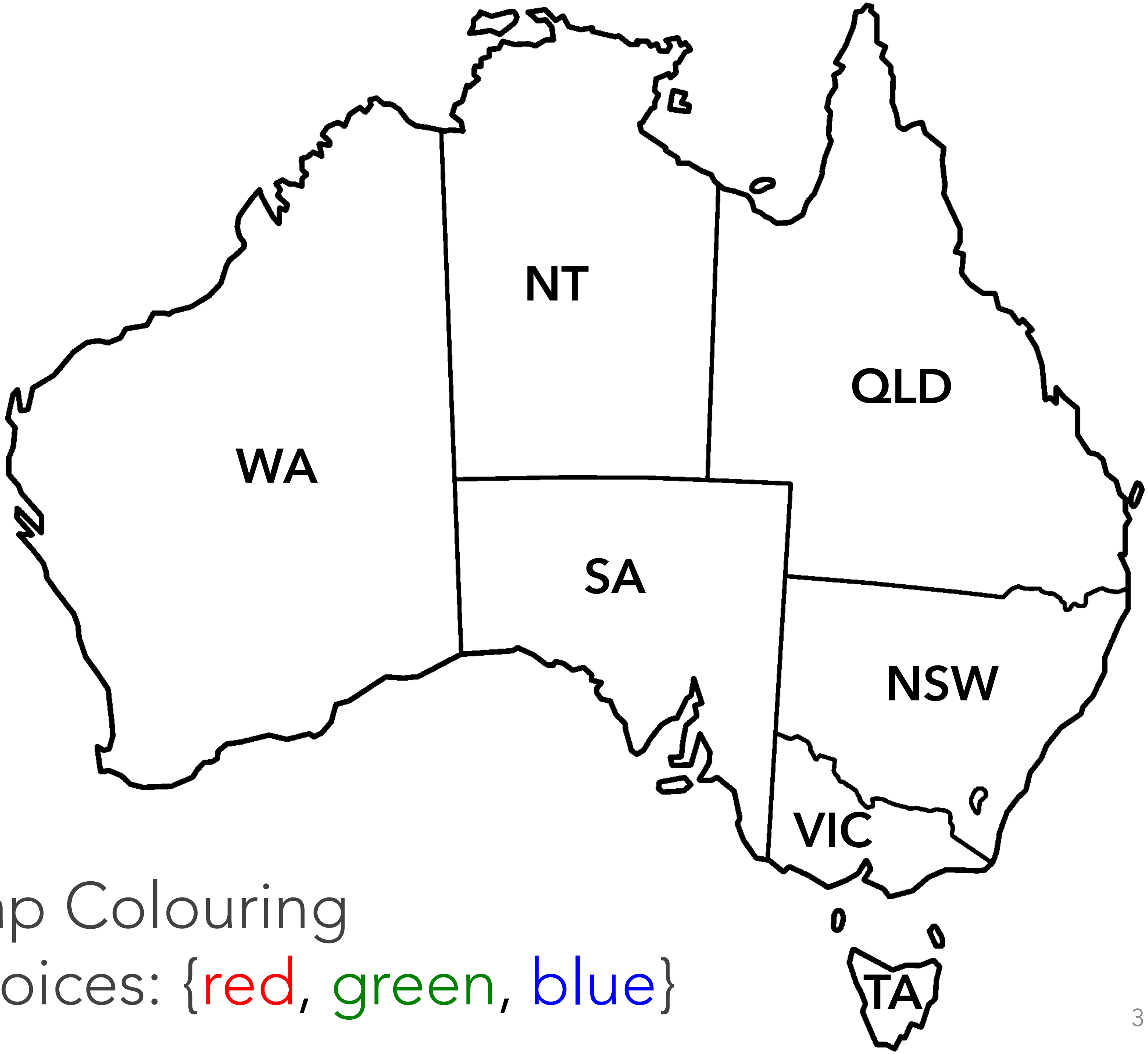
June 2014

Francis Li & Jesse Frost

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

Motivation

- Retrievals have poor time complexity
- Where do we start?
- Constraint Satisfaction Problems
  - Tap into vast research base
- What works? What doesn't?
  - We can be *very* domain-specific
- Provides easy way to model *any* changes



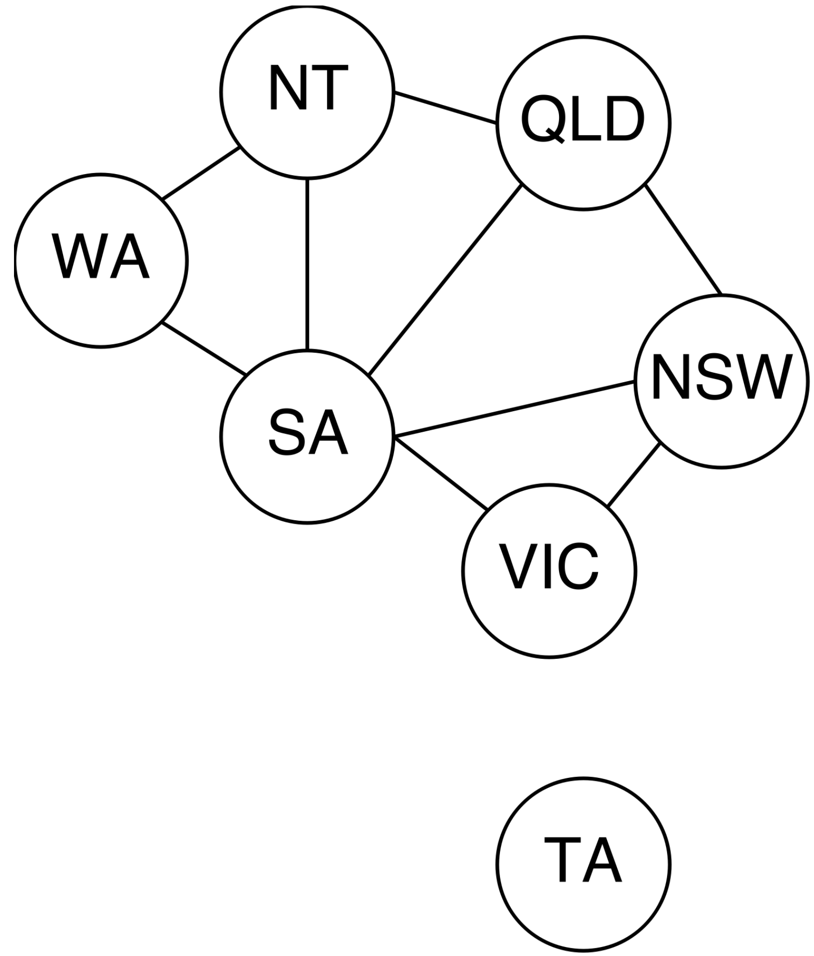
Map Colouring

Choices: {red, green, blue}

A Solution

Primal Constraint Graph

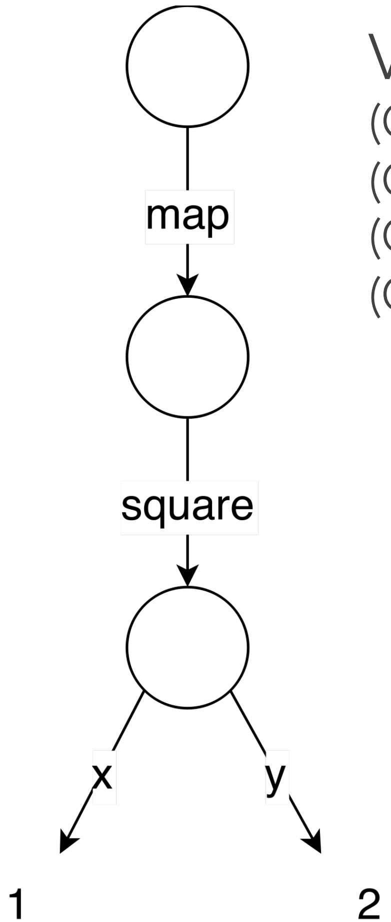
- Nodes are variables
- Domain is {red, green, blue}
- Arcs are binary constraints
- $C_{x,y}$ = (r,g),(r,b),(g,b),(g,r),(b,g),(b,r)
- or $C_{x,y} = (x \neq y)$
- We can exploit this structure



Stuff you probably know

- We have:
  - A cue as a set of WMEs
  - A set of every unique element that has ever appeared in working memory (the WMG)
  - A list of intervals for each element representing the times they were active

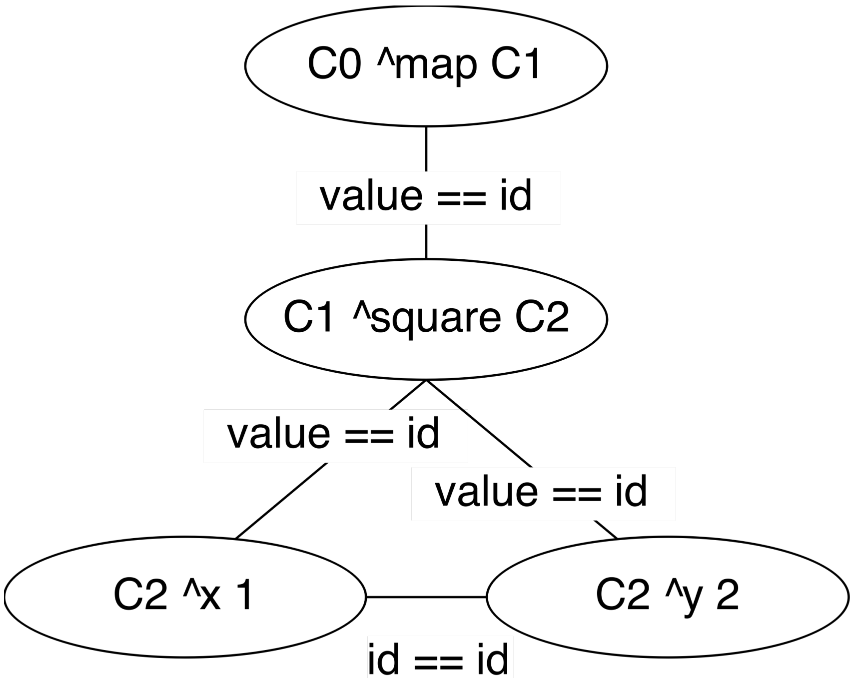
Epmem as a CSP



Variables:
(C0 ^map C1)
(C1 ^square C2)
(C2 ^x 1)
(C2 ^y 2)

- Domain of each variable is every element in the WMG where the constant values match
- Can further restrict for cues starting at root
- For example:
  - Domain of (C2 ^x 1) is every value with attribute *x* and value *1*

Epmem Primal Constraint Graph

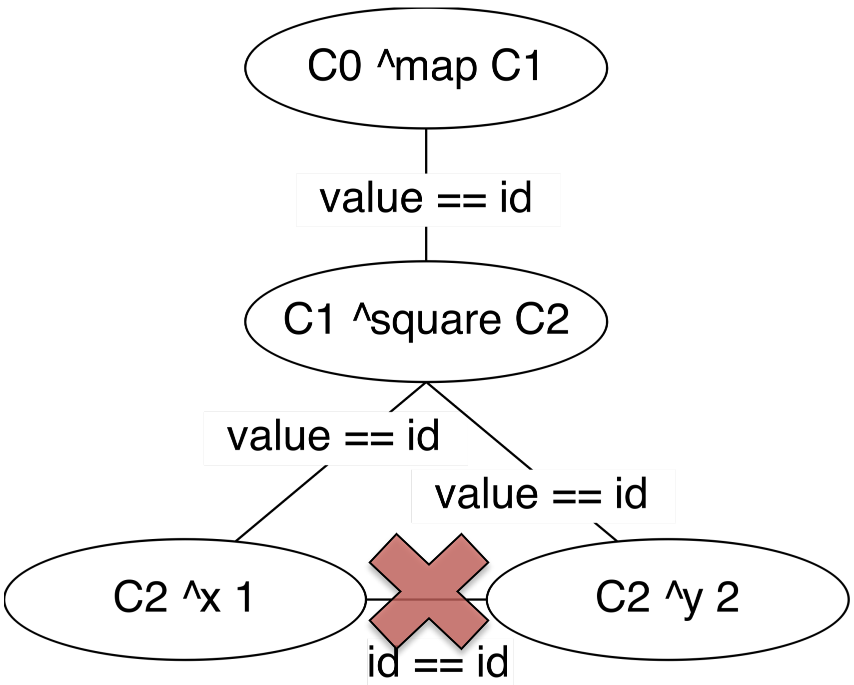


- Any variables that share ids are constrained
- Also n-ary temporal constraint (not shown)
- How do we solve this?

CSP solving techniques/heuristics

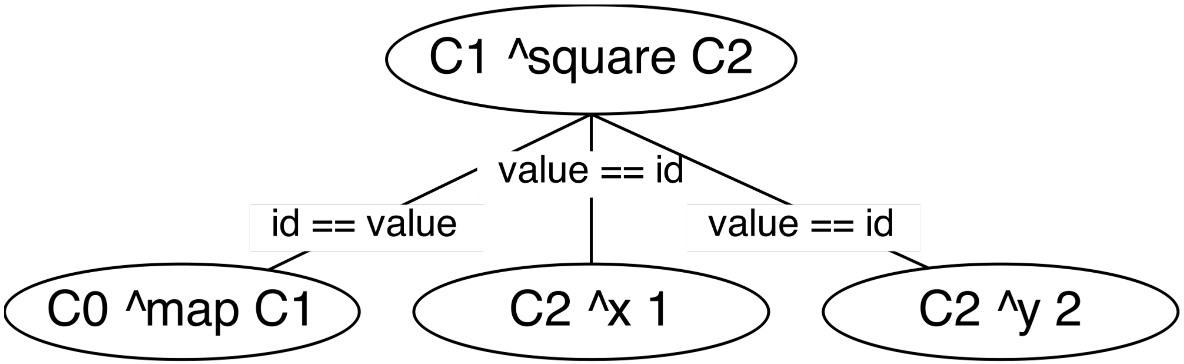
- Search (generating solutions)
  - Backtracking (naïve, intelligent, look-ahead)
- Inference (preprocessing/filtering)
  - Prune the search space
  - Create a tightened, but equivalent, problem
- Variable ordering
- Exploitation of constraint graph structure

Exploiting structure



- We want width-1 tree structures
- Want to maximise graph degree centrality (we think)
- Must identify redundant constraints

Directional arc consistency (DAC)



- When we backtrack, pick an ordering:
  - Instantiate parent (C1 ^square C2) first
  - Instantiate children after (heuristics define child order)
- Before we backtrack:
  - Delete values in the domain of the parent which don't satisfy a constraint with all of its children

“Revise” example (id == value)

Parent

Domain[**C1** ^square C2]

- (**O3** ^square O57)
- (**O0** square O247)
- (**O3** ^square O75)
- (**O0** ^square O68)
- (**O3** ^square O34)

Child

Domain[C0 ^map **C1**]

- (O0 ^map **O3**)

Implemented using sets (of value ids in this case)

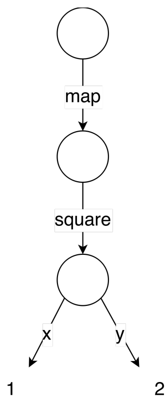
O(k), k = max(|Dom|)

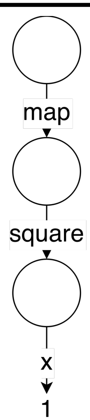
- Delete unsupported values (where id ≠ O3)
- Repeat for all children
- If domain becomes empty at any point -> no solutions
- All values in the parent participate in a solution

Arc consistency

- Backtrack-free for one solution
- For all solutions, do DAC again (down the tree)
- This establishes arc consistency (AC)
- All values participate in a solution
- Better complexity than standard AC algorithms (controlled propagation)

Empirically (tanksoar: ~30000eps)

<table>
<tr><th>Cue</th><th>Metric</th><th>Naive</th><th>DAC</th><th>AC</th></tr>
<tr><td rowspan="6">

1 result</td><td>Node visits O(ek)</td><td>547</td><td>7</td><td>7</td></tr>
<tr><td>Consistency checks O(1)</td><td>17512</td><td>103</td><td>3</td></tr>
<tr><td colspan="2">Set adds for revise O(1)</td><td>128</td><td>131</td></tr>
<tr><td colspan="2">Set membership checks for revise O(1)</td><td>408</td><td>536</td></tr>
<tr><td>Sum of constant checks</td><td>17512</td><td>639</td><td>670</td></tr>
<tr><td>CPU time (ms)</td><td>32.7</td><td>.774</td><td>.706</td></tr>
<tr><td rowspan="6">

16 results</td><td>Node visits O(ek)</td><td>515</td><td>35</td><td>35</td></tr>
<tr><td>Consistency checks O(1)</td><td>16034</td><td>545</td><td>152</td></tr>
<tr><td colspan="2">Set adds for revise O(1)</td><td>64</td><td>96</td></tr>
<tr><td colspan="2">Set membership checks for revise O(1)</td><td>410</td><td>474</td></tr>
<tr><td>Sum of constant checks</td><td>16034</td><td>1019</td><td>722</td></tr>
<tr><td>CPU time (ms)</td><td>32.0</td><td>4.01</td><td>3.37</td></tr>
</table>

Generating solutions

1. **Process cue**
   - Remove redundant constraints
   - Obtain highly branched tree under some ordering
   - Dealing with cycles is a bit more complex

Generating solutions

1. Process cue
   - Remove redundant constraints
   - Obtain highly branched tree under some ordering
   - Dealing with cycles is a bit more complex
2. **Run DAC up the tree, then down**

Generating solutions

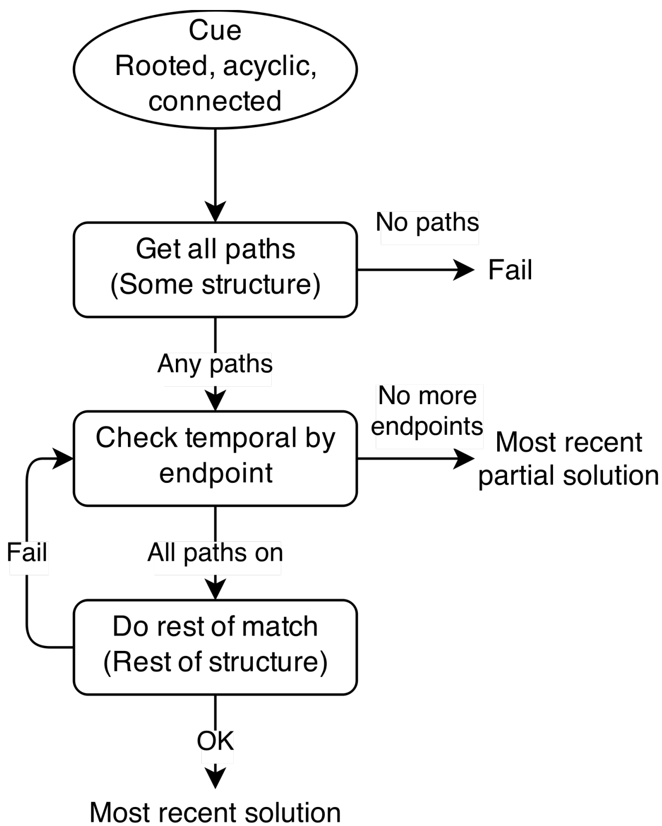
1. Process cue
   - Remove redundant constraints
   - Obtain highly branched tree under some ordering
   - Dealing with cycles is a bit more complex
2. Run DAC up the tree, then down
3. **Backtrack down the tree,** **maintaining AC**

Generating solutions

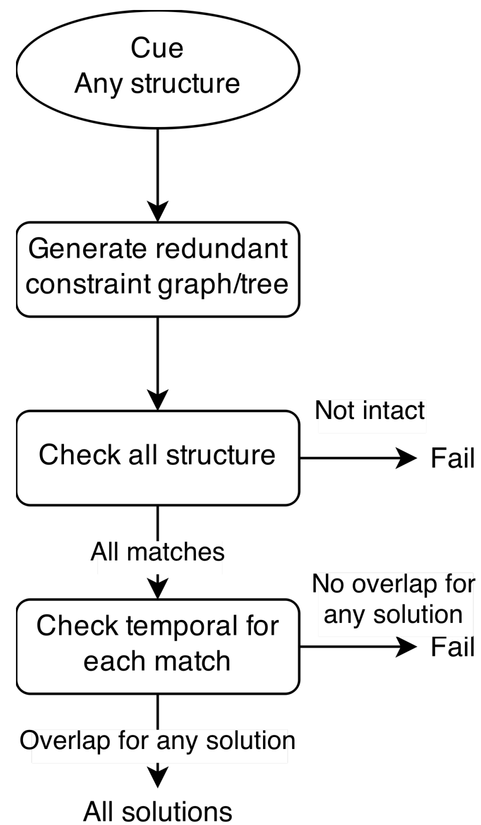
1. Process cue
   - Remove redundant constraints
   - Obtain highly branched tree under some ordering
   - Dealing with cycles is a bit more complex
2. Run DAC up the tree, then down
3. Backtrack down the tree, maintaining AC
4. **Check temporal overlap by merging pairs of interval lists for each solution [O(nm)]**

Comparison

Soar Implementation



This Implementation



Pluses

- Few (if any) restrictions on cue structure (can be disjoint, cyclic, non-rooted)
- Easy to model extensions `(C2 ^x >0)`
- Structure is most constrained
- No possibility of multiple complex graph matches
- Retrieve all solutions
- Parallelisation is fine-grained
- Same principles for production matcher

Needs work

- Retrieval is still unbounded
- Poor when many solutions
- Partial solutions not considered (yet)
- More investigation needed
  - Variable ordering
  - Dealing with cycles
  - Look-ahead

BUT

- The problem is defined.
- All extensions can use the CSP formulation
- We just tweak the techniques

Thank you

QUESTIONS?