JTRE/JSAINT

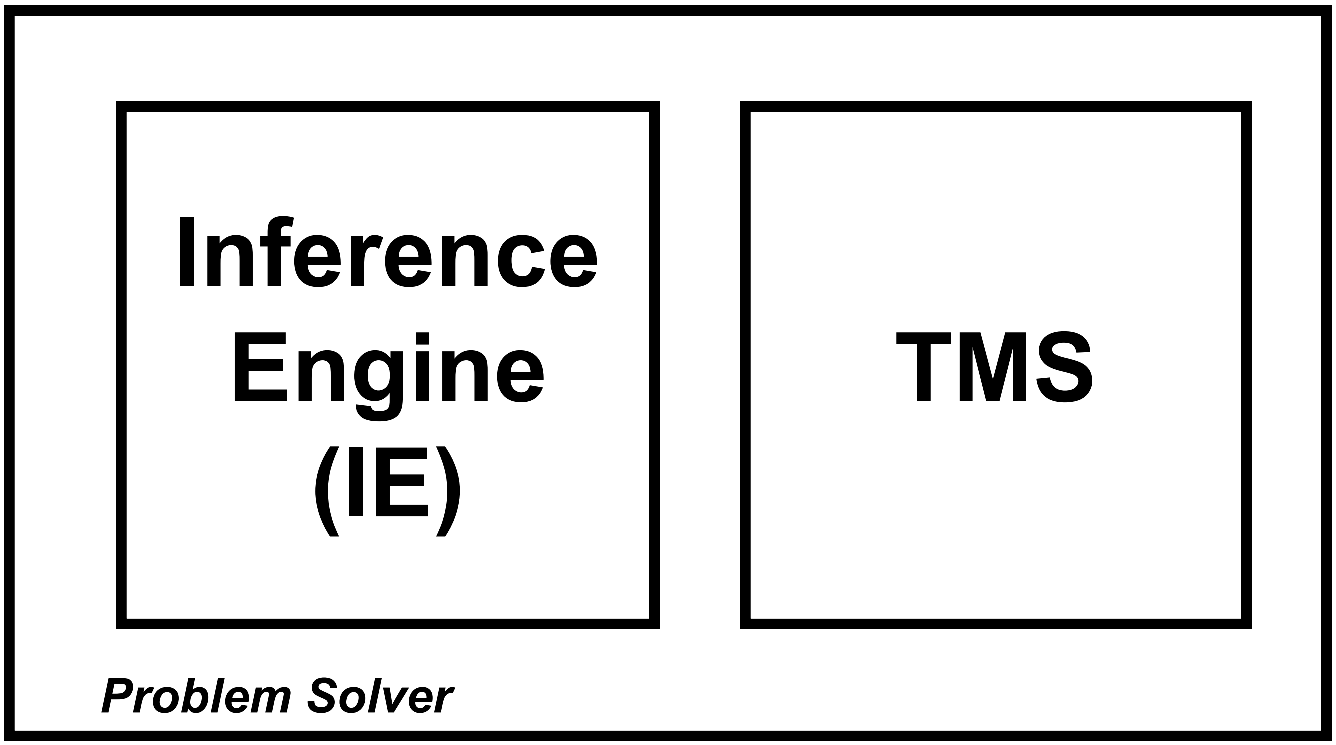
EECS 344 Winter 2008

Putting the JTMS to Work

Outline

- Interface between a JTMS and a rule engine
- Chronological Search versus Dependency Directed Search: A Playoff
- Using a TMS in a problem solver: JSAINT design issues

Review: Problem Solver = TMS + Inference Engine



The five basic actions of a TMS

- **Create Nodes**
- **Accepts records of IE deductions (as justifications)**
- **Computes the correct label for nodes and supplies them on request.**
  - **Derives consequences of assumptions & premises based on dependency network**
  - **When assumptions are retracted, their consequences are retracted**
  - **Provides explanations for belief e.g., chains of *well-founded support***
- **Detects contradictory beliefs**
  - **Based on contradiction nodes, explicit dependencies**
- **TMS accepts rules from IE to be scheduled for execution when particular belief conditions are met.**

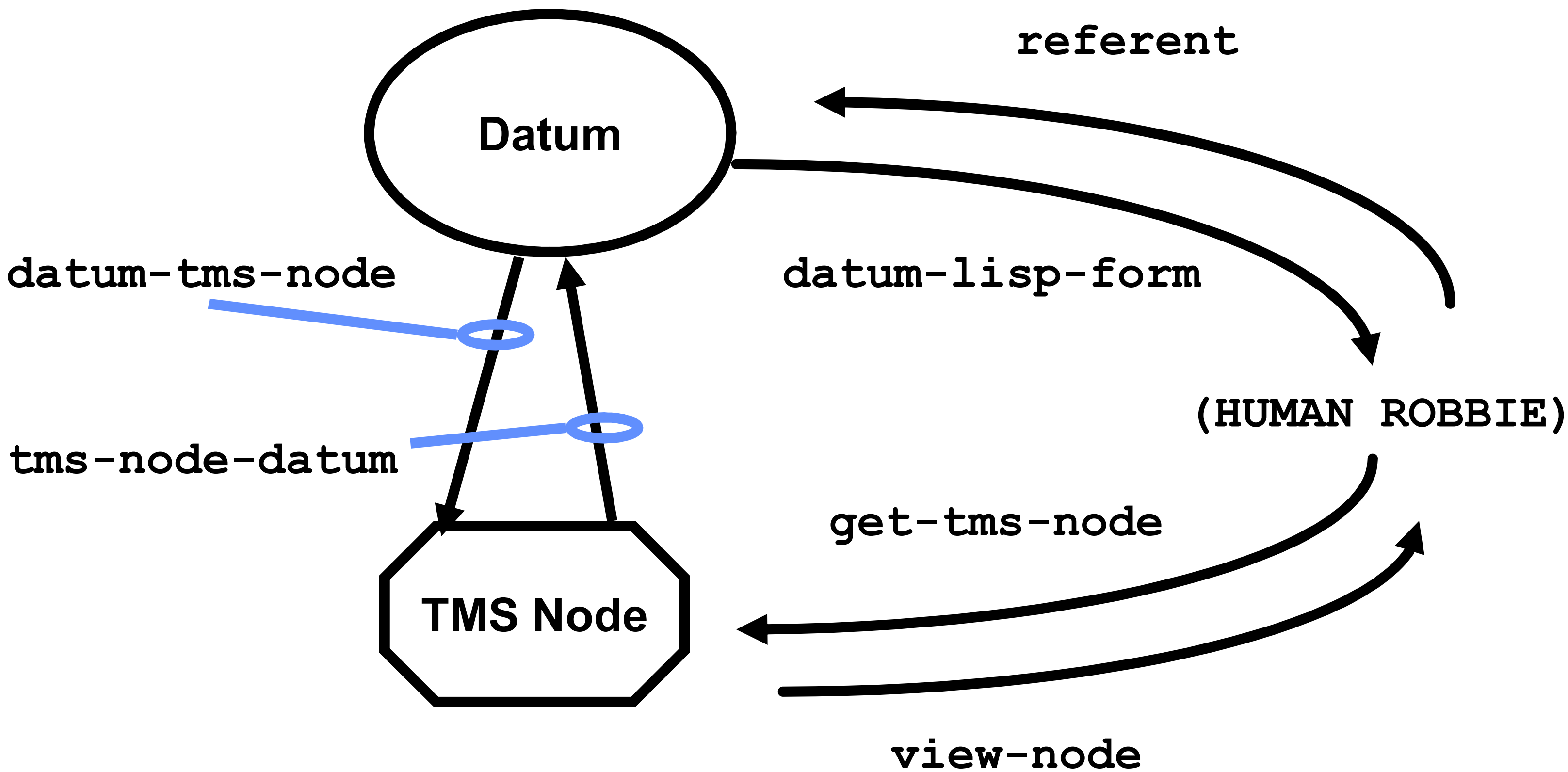
Constraints on the IE

1. Provide mapping between IE and TMS data structures
   – IE must inform TMS when a new node is needed
   – Must be able to retrieve the TMS node associated with an assertion.
2. Provide facilities for changing beliefs and expressing dependency relations.
   – Marking assertions as PREMISEs or ASSUMPTIONs, and for enabling/retracting assumptions.
   – Provide facilities for representing justifications.
3. Provide facilities for inspecting system's beliefs (node labels)
4. Provide facilities for contradiction handling.
5. Provide methods for tying the execution of rules to belief states.
   – Allow including constraints on beliefs in conditions for rules
   – Ensure both belief constraints and syntactic matching constraints are met before rules are run.

Inference Engine services

- Provides *reference mechanism*
  - e.g., assertions, pattern matching
- Provides *procedures*
  - e.g., rules
- Provides *control strategy*

1. Mapping Assertions to TMS nodes



2. Justifying assertions in terms of other beliefs

- `(assert! <fact> (<informant> . <antecedents>))` installs a justification
- `(assert! <fact> <Anything else>)` makes a premise
- `(assume! <fact> <reason>)` makes an assumption
- `rassume!, rassert!` as before
- `retract!` disables an assumption
- `(contradiction <fact>)` installs a contradiction

3. Queries concerning Belief States

- `in?`
- `out?`
- `why?`
- `assumptions-of`
- `fetch`
- `wfs`

4. Handling Contradictions

- ```
  (with-contradiction-handler
              <jtms> <handler>
             . <body>)
  ```
- **We'll see example with N-queens problem**
  ```

5. Tying rule execution to belief states

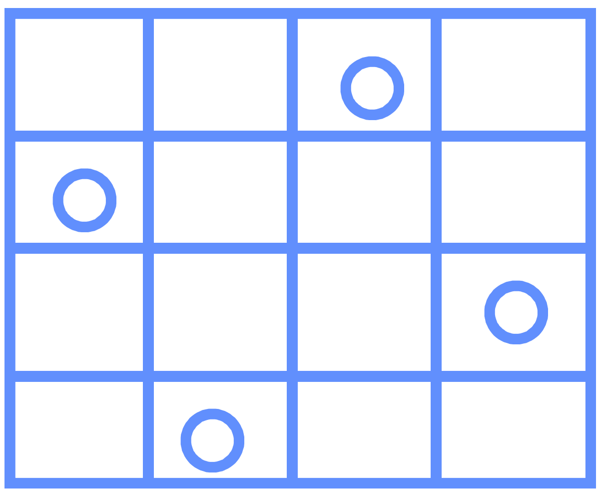
- `(rule` *`<list of triggers>`* *`<body>`*`)`
- **Triggers are** `(`*`<condition>`* *`<pattern>`*`)`
- **Types of conditions**
  - `:IN`
  - `:OUT`
  - `:INTERN`
- **Trigger options**
  - `:VAR`
  - `:TEST`

Examples of rules

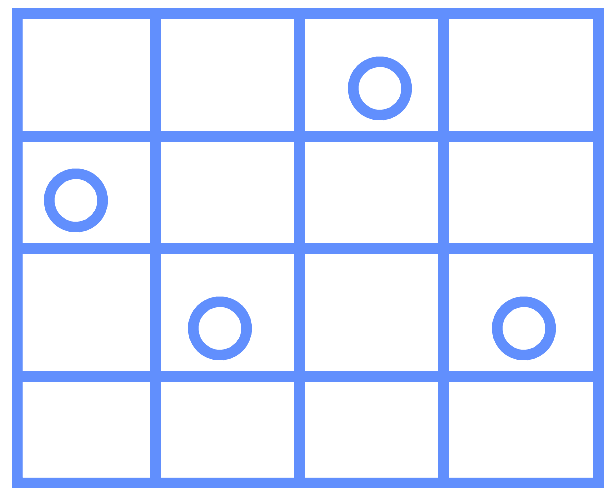
```
(rule ((:in (implies ?p ?q) :var ?f1)
        (:in ?p))
      (rassert! ?q (CE ?f1 ?p)))

(rule ((:in (show ?p) :var ?f1)
         :test (not
           (logical-connective? ?p)))
  (rassert! ((show ?p) Indirect-Proof
                   :PRIORITY Low)
             (BC-IP ?f1)))
```

Search Example: The N-Queens problem



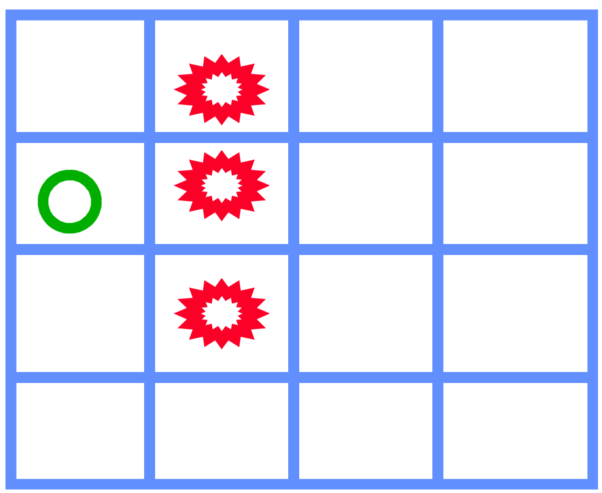
Good solution



Bad solution

Chronological Search solution

- **Given NxN board**
  - Create a choice set for placing a queen in each column
  - Unleash rules that detect captures
  - Systematically search all combinations of choices

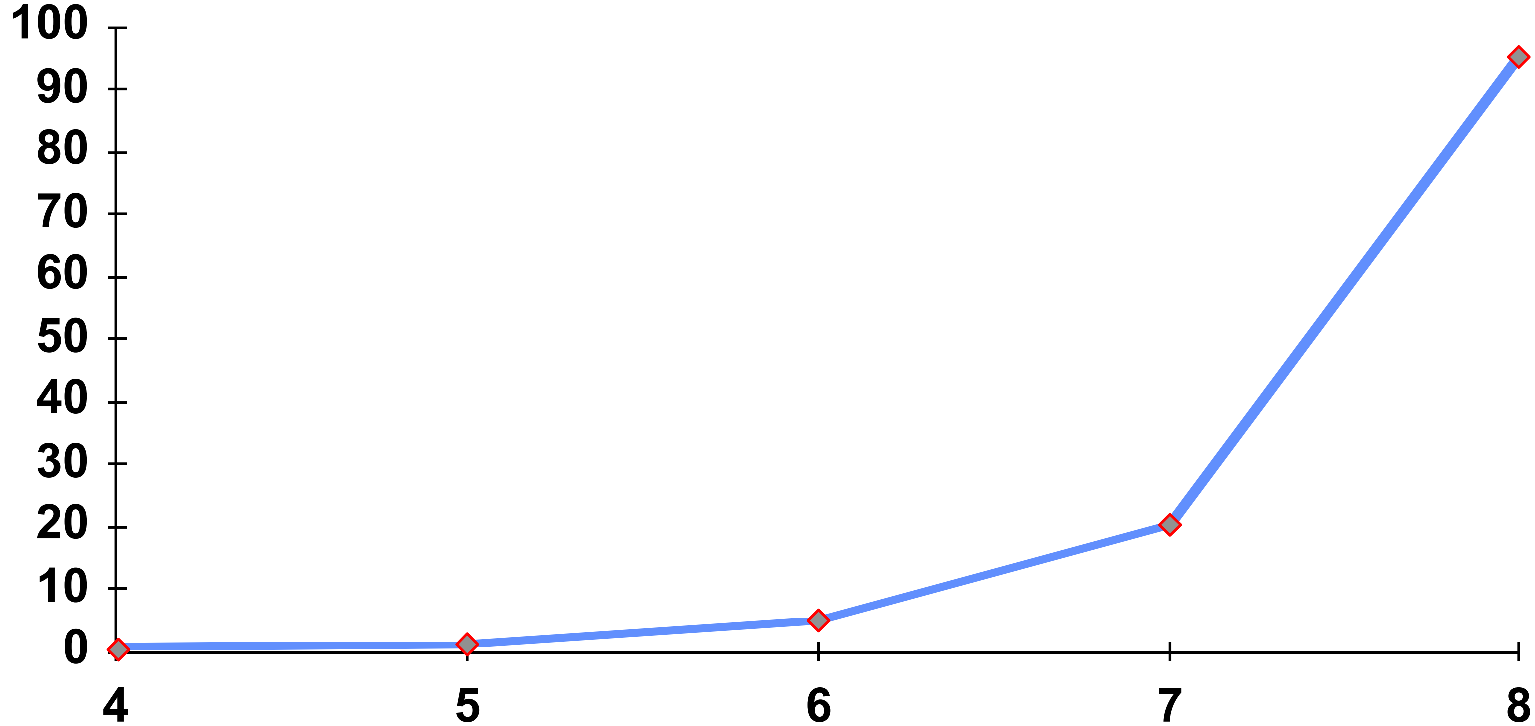


Dependency Directed Search Solution

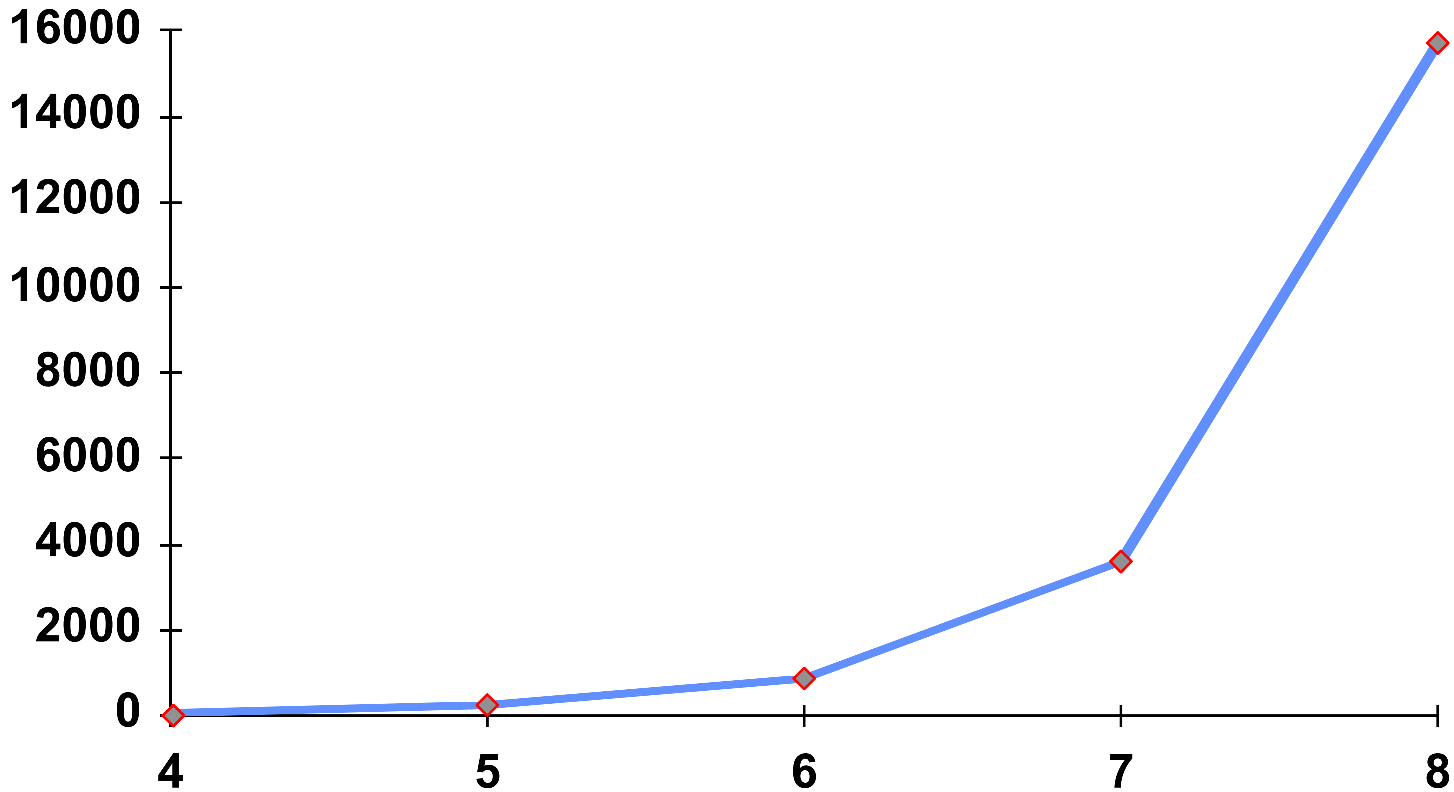
- **Like chronological search solution, but**
  - **When inconsistent combination found, assert negation of queen statement. (Creating a *nogood*)**
  - **When searching, check for a nogood before trying an assumption.**

Chronological Search: Time required

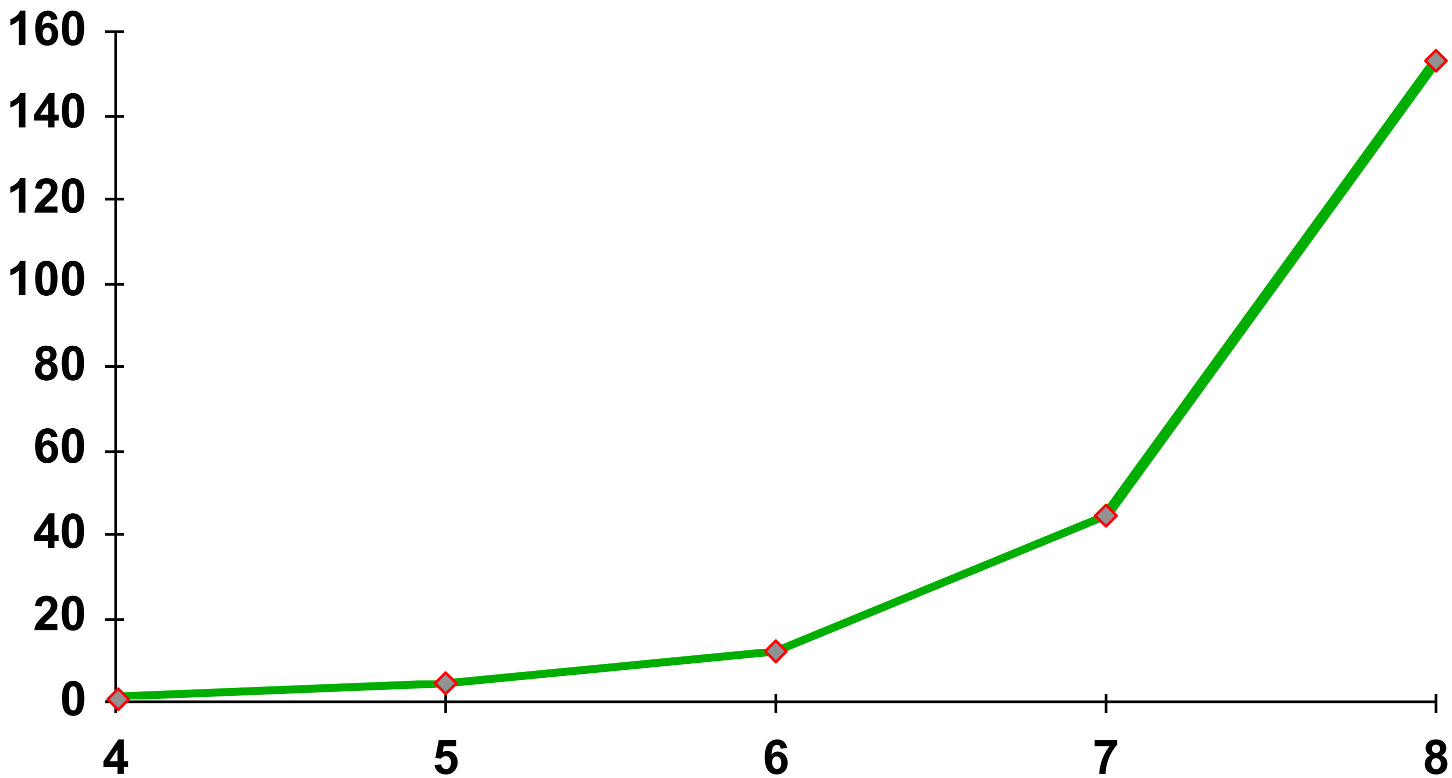
- **IBM RT, Model 125, 16MB RAM, Lucid CL**



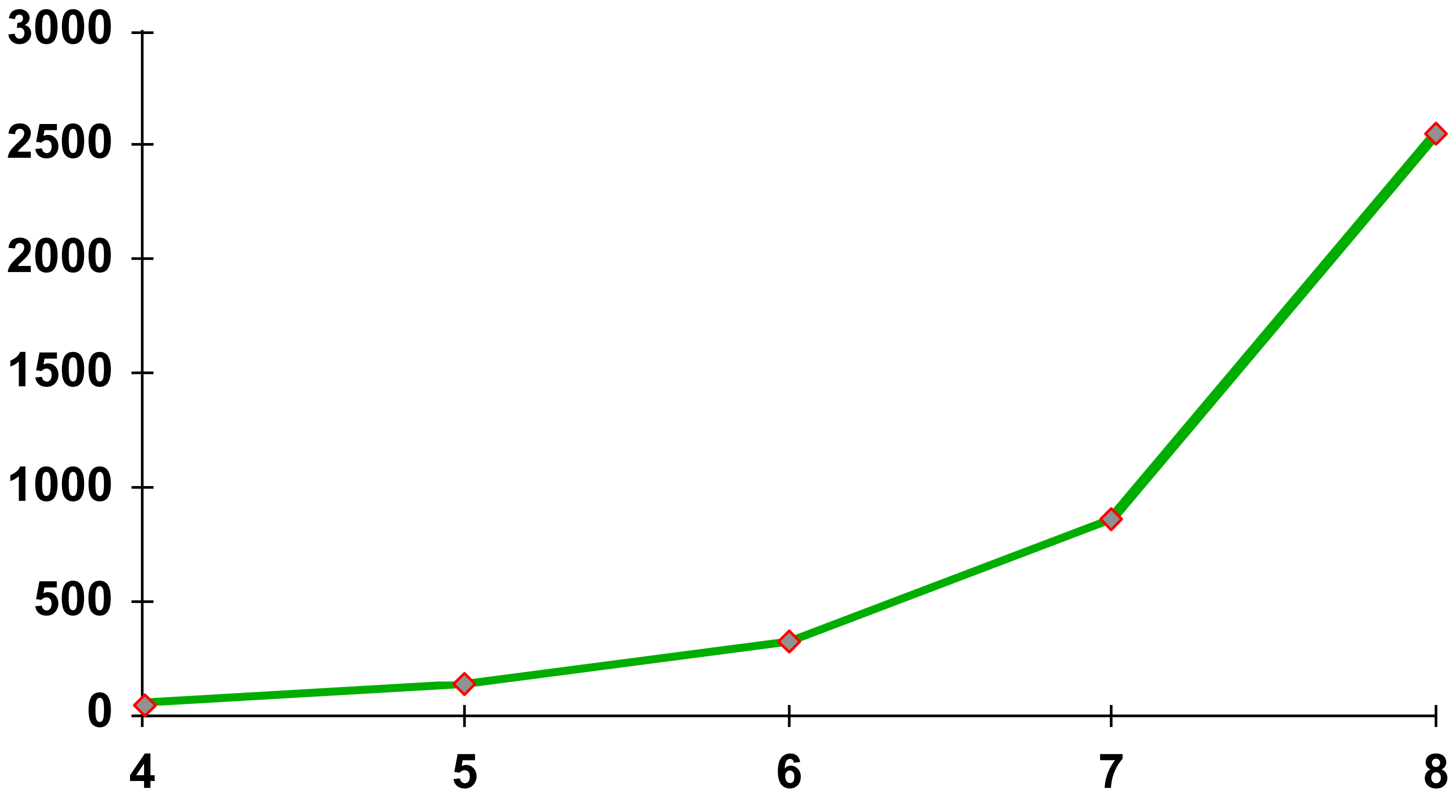
Chronological Search: Assumptions Explored



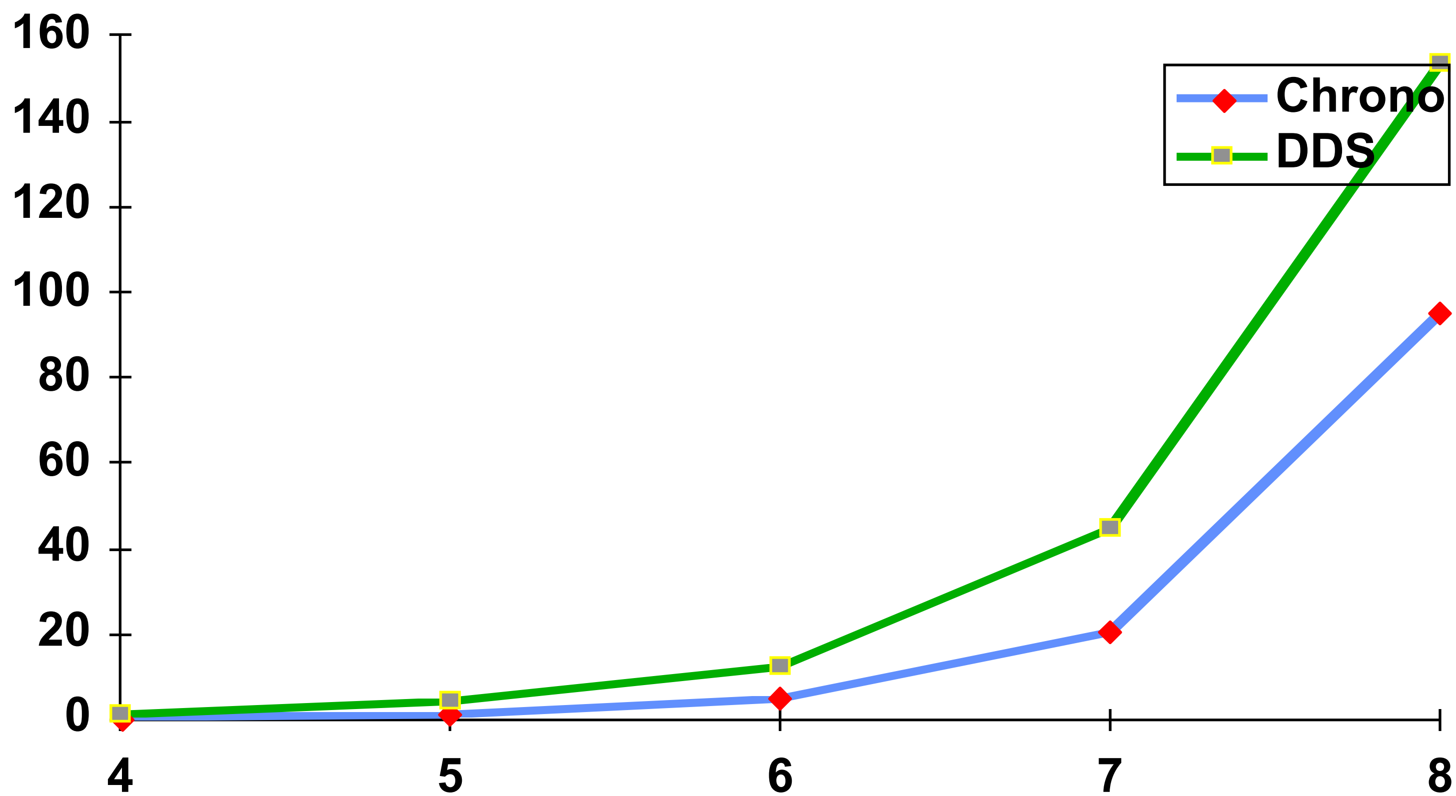
Dependency Directed Search: Time used



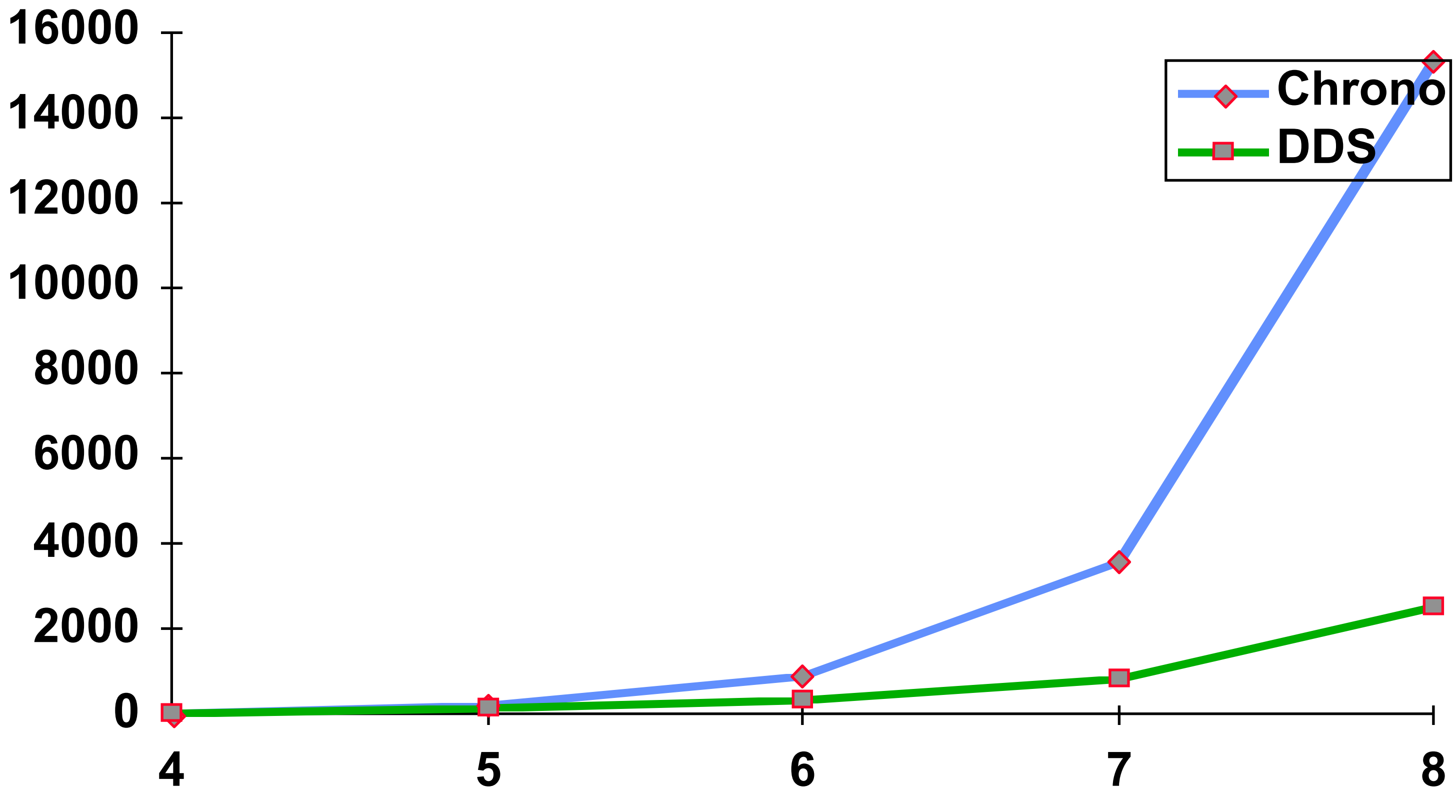
Dependency-Directed Search: Assumptions Explored



Comparing the results
Time in seconds



Comparing the results
Assumptions Explored



Implications

- Neither strategy changes the exponential nature of the problem
- Dependency-directed search requires extra overhead per state explored
- The overhead of dependency-directed search pays off on large problems when the cost of exploring a set of assumptions is high

Using a TMS in problem solving

Case study: JSAINT

JSAINT: Its task

- **Input: An indefinite integration problem**
- **Output: An expression representing the answer**

$$\int \left[4e^{2x} + 3.2\sin(1.7x) + 0.63 \right] dx$$

`JSAINT` returns

$$2e^{2x} - 1.88\cos(1.7x) + 0.63x$$

Issues in JSAINT design

- Explicit representation of control knowledge
- Suggestions Architecture
- Special-purpose higher-level languages
- Explanation generation

Issue 1: Explicit representation of control knowledge

- The use of show assertions in KM* is only the beginning!
- Recording control decisions as assertions enables
  - Control knowledge to be expressed via rules
  - keeping track of what is still interesting via the TMS
  - Explaining control decisions
  - Provides grist for debugging and learning
- Key part of JSAINT design is a *control vocabulary*

Issue 2: Control via suggestions

- **Problem: Local methods cannot detect loops, combinatorial explosions**
- **Solution: Decompose problem-solving operations into two kinds:**
  - **Local operations for "obvious" tasks, making relevant suggestions**
  - **Global operations for choosing what to do**
- ***Suggestions Architecture* is a very useful way to organize problem solvers**

Issue 3: Special-purpose higher-level languages

- Problem: Rules still too low-level for many purposes.
- Solution: Design special-purpose language to meet domain experts half-way

```
(defIntegration Move-Constant-Outside
  (Integral (* ?const ?nonconst) ?var)
  :test (and (not (occurs-in? ?var
                                 ?const))
             (occurs-in? ?var ?nonconst))
  :subproblems ((?int
                  (Integrate
                   (Integral ?nonconst ?var))))
  :result (* ?const ?int))
```

Issue 4: Explanation generation

- Want to know how a solution was obtained
  - Dependencies involving the data provide this
- Want to know what went wrong when JSAINT can't solve the problem
  - Dependencies involving the control assertions provide this

How SAINT Worked

1. Is problem a standard form?
   If so, substitute & return answer
2. Find potentially applicable transformations.
   For each transformation, create the
   subproblem of solving the transformed
   problem.

- SAINT used 26 standard forms, 18 transformations
- Also used many special-purpose procedures

Knowledge about Integration

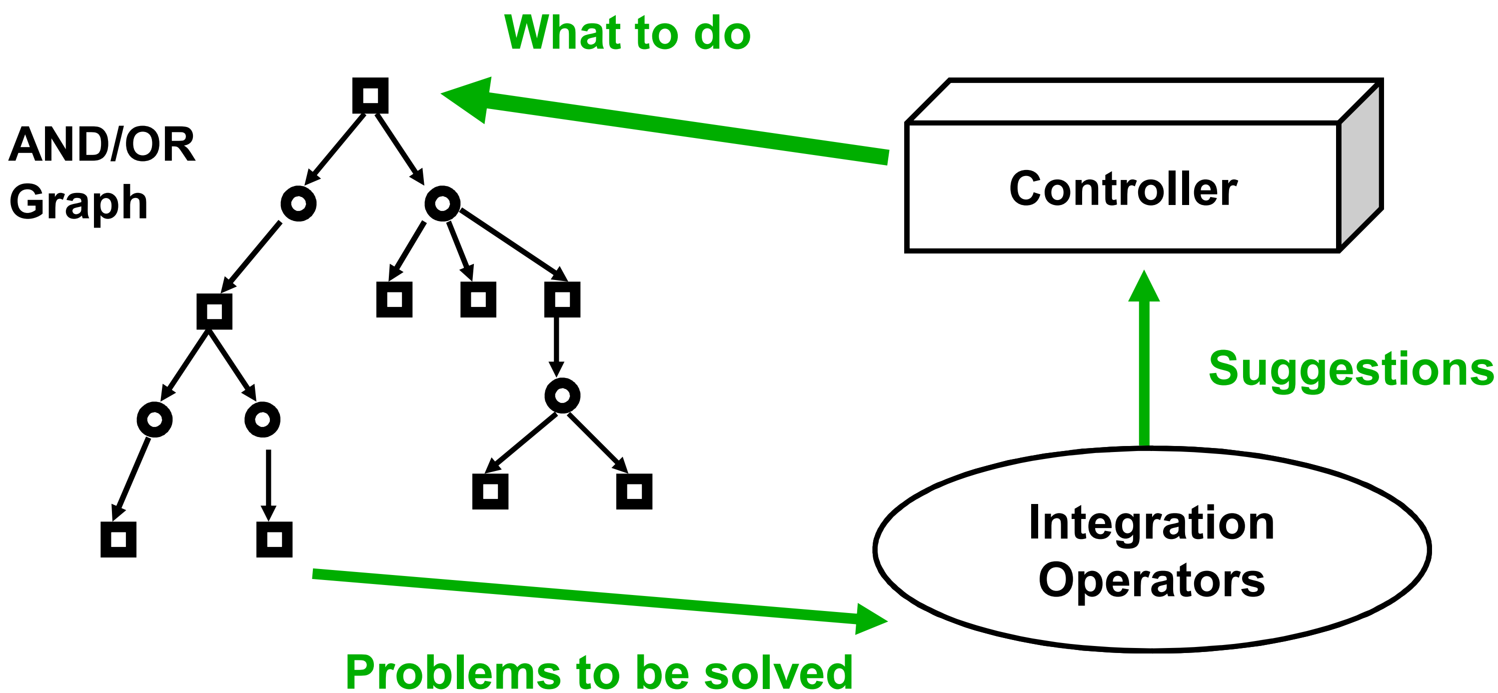
- **Standard forms**

$$\int v dv \rightarrow \frac{1}{2} v^2$$

- **Transformations**

$$\int c g(v) dv \rightarrow c \int g(v) dv$$

JSAINT Architecture

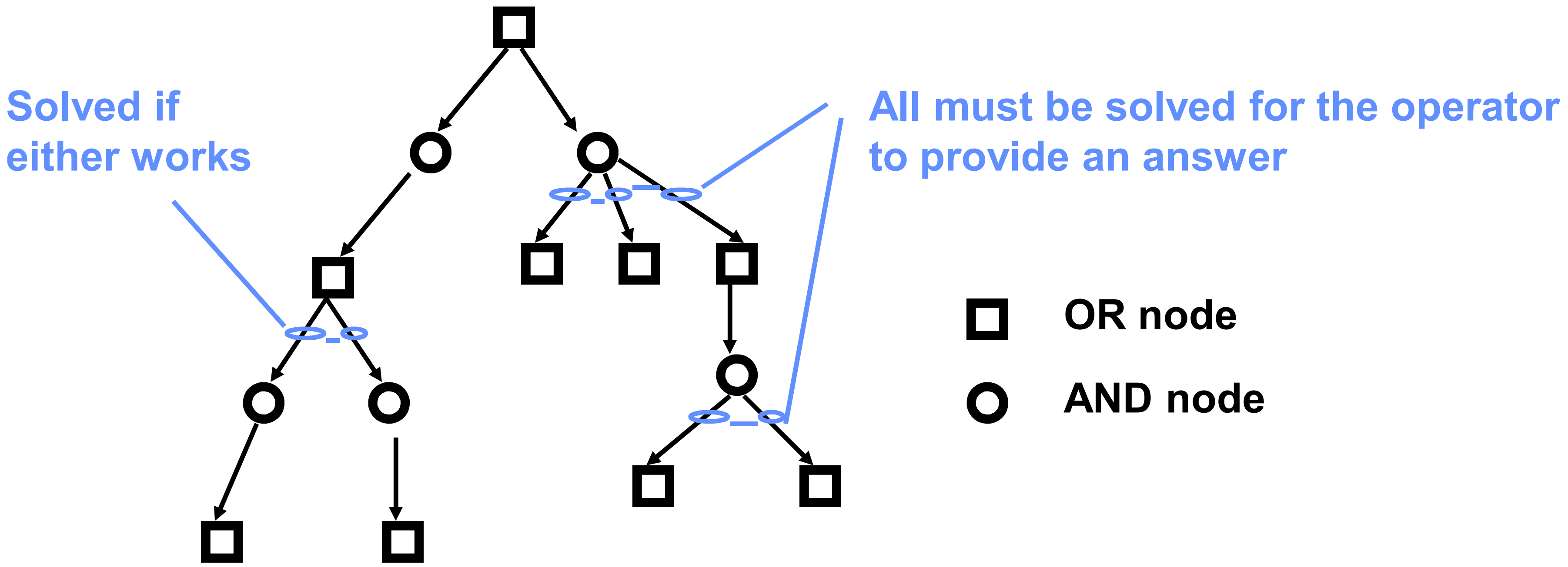


Central Controller

- Gathers suggestions about particular subproblems
- Selects what subproblem to work on next
- Ensures that resource limits aren’t exceeded

AND/OR Trees

•


Solved if
either works
All must be solved for the operator
to provide an answer
OR node
AND node

AND/OR Graph

- Maintains status of work on problems and subproblems
- Detects when problems are solved
- Detects when problems cannot be solved

Integration Operators

- **Provide direct solutions to simple problems (analogously to SAINT's *standard forms*)**
- **Suggests ways of decomposing problems into simpler problems**

JSAINT in operation

1. If original problem has been solved,
   or clearly cannot be solved,
   or if resource bounds have been reached,
   quit.

2. Select best subproblem *P* to work on.

3. If P can be directly solved, do it.

4. Otherwise, gather suggestions for how to solve *P* and extend the AND/OR graph accordingly.

Representations

- **Mathematics is the easy part**

$$\int (x+5)dx$$

is represented as

```
(integral (+ x 5) x)
```

- **Representing control knowledge is harder**

How detailed?

- **Implicit**
  `(integral (+ x 5) x)`
- **Make operations to perform explicit**
  `(integrate (integral (+ x 5) x))`
- **Make nature of goal explicit**
```
(solve
   (integrate (integral (+ x 5) x)))
```
- **Make nature of activity explicit**
```
(do (solve
      (integrate
        (integral (+ x 5) x))))
```

Tradeoffs

- Implicit often means fast & simple
  - Fewer assertions means less storage, fewer justifications
  - Avoid hunting polar bears in the desert
- Explicit often means flexible & maintainable
  - Recording decisions in dependency network makes them available to both the program and its users
  - Avoid killing dead bears

JSAINT Decisions

- Won't explicitly represent goal versus problem versus task distinction
- Only kind of goal: `TRY`

```
(TRY (integral-of-sum
        (integral (+ x 5) x)))
```

Success or failure of problems

`(solved <P>)` is believed exactly when problem *P* has been solved

`(failed <P>)` is believed exactly when *P* cannot be solved by JSAINT given what it knows.

`(solution-of <P> <A>)` holds exactly when *A* is the result of solving problem *P*

Representing Goals

- JSAINT uses the form of the goal itself
```
(integrate (integral (+ x 5) x))
```
- Advantage: Easy to recognize recurring subproblems
  - Actually an AND/OR graph rather than an AND/OR tree
- Alternative: Reify goals
```
(goal GOAL86)
(GOAL86 form-of
  (try (risch-algorithm
         (integrate
          (integral CENSORED )))))
```

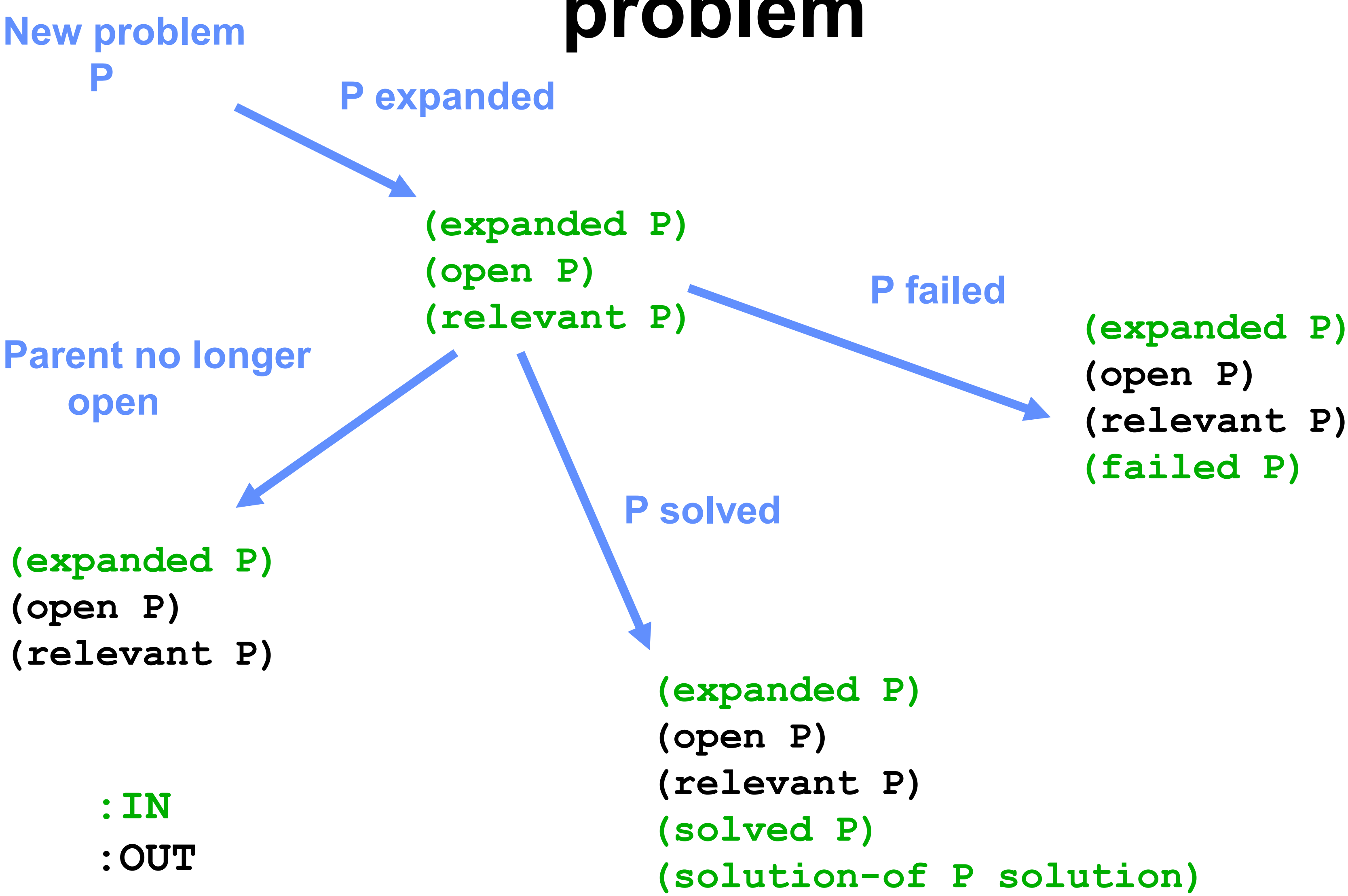
Representing progress

`(expanded P)` **is believed exactly when work has begun on P**

`(open P)` **is believed exactly when P has been expanded but is not yet solved or known to be unsolvable.**

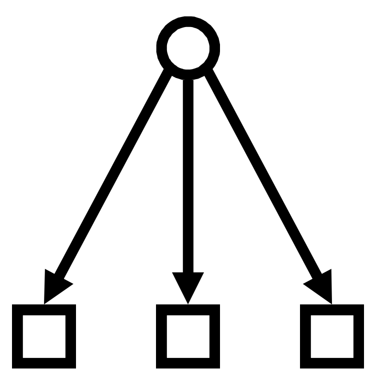
`(relevant P)` **is believed exactly when P is still potentially relevant to solving the original problem.**

The natural history of a problem



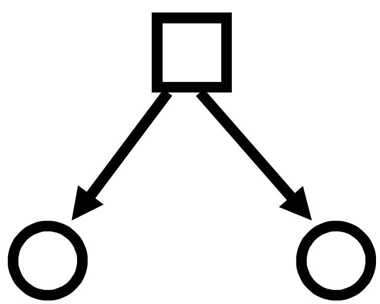
Semantics of success and failure for AND nodes

- Failure of single child means failure of parent
- Success of all children means success of parent



Semantics of success and failure for OR nodes

- Failure of all children means failure of parent
- Success of any child means success of parent



Closed-World assumptions in JSAINT

- Implicit in structure of system

1. All possible relevant suggestions are available when a problem is first posed.
2. Every operator succeeds if its conjunctive subgoals succeeds

- However: Any node can gain parents at any time.

Design issues for operators

- An operator must
  - look for relevant problems
  - make suggestions when it finds them
  - apply itself when selected by the controller
  - justify an answer when it succeeds
- This requires using the control vocabulary in a reasonable protocol

A typical operator

```
(defIntegration Integral-of-Sum
  (integral (+ ?t1 ?t2) ?var)
  :SUBPROBLEMS
   ((?int1 (integrate
            (integral ?t1 ?var)))
     (?int2 (integrate
             (integral ?t2 ?var))))
  :RESULT (+ ?int1 ?int2))
```

Looking for relevant problems

- Look for `expanded` assertions that match

```
(expanded (integrate (+ x y) x))
```

Making suggestions

- Happens antecedently

```
(suggest-for
 (integrate (integral (+ x y) x))
 (integral-of-sum
     (integral (+ x y) x)))
```

Controller communicates its wishes

- Operator spawns rule that looks for the signal to start working:

```
(expanded
 (try (integral-of-sum
        (integral (+ x y) x))))
```

How the Controller Works

1. Check the original problem
   If solved, then halt & report success
   If failed, then halt & report failure
2. If agenda is empty, halt & report failure
3. If resource allocation exceeded, halt & report failure
4. Select simplest subproblem on the agenda and work on it
5. Return to Step 1

The Agenda

- Unlike TRE queues, not everything will be executed.
- Items on the agenda consist of
  - A subproblem
  - An estimate of its difficulty
- Difficulty estimates depend only on the structure of the problem, not its history

Working on a subproblem

1. Assert `EXPANDED` and assume `OPEN`
2. Run JTRE queues to completion
3. If `SOLUTION-OF` found, then finish.
4. Fetch all suggestions for the problem
5. If no suggestions, mark `FAILED`.
6. Otherwise, install `TRY` assertions as OR children of the problem