Soar Basics
[45 min]

Soar Tutorial
July, 2016
The Soar Cognitive Architecture
(Laird, Newell, Rosenbloom, et al.; 1981-)
The Soar Cognitive Architecture

• Goal: General cognitive architecture
  – Focus on routine to complex behavior, learning, autonomy, …

• Inspired by psychology and neuro-science
  – Look to psychology for cognitive mechanisms and capabilities
  – A cousin of ACT-R

• Engineered using computer science
  – Look to computer science and AI for efficient and robust implementations
  – 200x faster than real-time execution over very large knowledge bases and hours of execution

• Available on all major platforms: Windows, iOS, Linux, Android
  – Open source (BSD)
  – Integrated with many robotic platforms
  – Over 100 systems implemented in Soar
Soar Users’ Institutions

Academic
• UNICAMP – State University of Campinas (Brazil)
• Brigham Young University
• Cornell University
• George Mason University
• Georgia Tech
• University of Iowa
• KAIST (South Korea)
• University of Michigan
• Pace University
• Penn State University (Applied Research Laboratory)
• Universidade Presbiteriana Mackenziem (Brazil)
• Institute for Creative Technology, University of Southern California
• Universidad Tecnologica Nacional (Argentina)
• University of Zaragoza (Spain)

Commercial
• Soar Technology, Inc. (DoD R&D)
• Lexoris Learning
• ModuleMaster (automotive electronics)
• MTH Autonomous Intelligent Systems (cyber security)

DoD Research Laboratories
• Air Force Institute of Technology
• Air Force Research Laboratories
• Naval Postgraduate School
Example Virtual Environments

- **R1-Soar**
  - Computer Configuration

- **Amber EPIC-Soar**
  - Modeling Human-Computer Interaction

- **ICT Virtual Human**
  - Natural Interaction, Emotion

- **TacAir/RWA-Soar**
  - Complex Doctrine & Tactics

- **Soar Quakebot**
  - Anticipation

- **StarCraft**
  - Spatial Reasoning & Real-time Strategy

- **Haunt**
  - AI Actors and Director

- **MOUTbot**
  - Team Tactics

- **Simulated Scout**
  - Spatial Reasoning & Mental Imagery

- **Action Games**
  - Spatial Reasoning & Reinforcement Learning

- **Liar’s Dice**
  - Probabilistic reasoning and reinforcement learning

- **Viewpoints**
  - Creative Human Interaction
Soar Robotic Platforms

1988: Robo-Soar, UM
1990: Hero-Soar, UM
1994: Adapt, Pace U
2004: Penn State
2009: Splinter, UM
2009: Soar Tech
2010: Superdroid, PSU
2011: Magic, ST
2011: rGator, ST
2012: BOLT, UM/ST
2012: Summit, ST
2013: REEM-C Pal Robotics
2013: Magic 2, UM
2014: Mindstorms, UI
2015: Penn State
2015: Magic 2, UM
Soar 9 Structure

Symbolic Long-Term Memories

- **Procedural**
  - Skills: How to stack a block; how to pickup a block; how to parse a sentence, ...
  - Reinforcement Learning
  - Chunking

- **Semantic**
  - Facts: people, places, ... Mappings from words to perceptual symbols, ...
  - Semantic Learning

- **Episodic**
  - Experiences: what happened in the past
  - Episodic Learning

Symbolic Working Memory

- Object features & relations among objects
- Active goals and actions: stack the blocks, stack the blue cylinder on the red block
- Retrievals from memory: words corresponding to colors, shapes, ...
- Internal reasoning results: ...

Spatial Visual System

- Object-based continuous metric space

Perception

- Fire rules to Propose operators

Action

- Fire rules to Apply selected operator

Predicate

- Fire rules to Evaluate operators

Output & Memory Access

- Decide
- Decide
- Decide
- Decide
Problem Spaces

- **State**: the current situation the agent is in
- **Operators**: transition to new state
  - Internal reasoning steps with changes to working memory
    - Logical deduction and inference, simple math, …
  - Retrievals from long-term semantic or episodic memory
  - Mental imagery actions
  - External motor actions
- **Goals**: states to be achieved
Soar Basic Functions

Soar represents procedural knowledge as rules

1. **Input** from environment
2. Elaborate current situation: *parallel rules*
3. Propose and evaluate operators via *preferences: parallel rules*
4. **Select operator**
5. Apply operator: Modify internal data structures: *parallel rules*
6. **Output** to motor system [and access to long-term memories]

Assumptions:
- Complex behavior arises from multiple cycles.
- Each cycle is bounded processing to maintain reactivity.
Operators and States for Colored Blocks World

States

– Objects
  • blocks [color, name]
  • paint brushes for specific colors [color]
– Initially all blocks are white

Operators:

– Initialize-blocks-world
– Paint a block with a different paint brush color

Goal:

– All blocks are red

Multiple operators can be proposed at the same time. Use *preferences* to select between them.
### Basic Soar Operation

<table>
<thead>
<tr>
<th>State</th>
<th>Operators proposed by rules creating preferences</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Proposed operators evaluated by rules</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operator selected by decision procedure</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operator applied by rule</th>
</tr>
</thead>
</table>

- If block \([X]\) is not color \([Y]\), then propose \(\text{Paint } [X] \text{ with } [Y]\).

- If operator has color \([\text{Red}]\), then make best preference

- If \(\text{Paint block } [X] \text{ with color } [Y]\) selected, then change \([X]\) color to \([Y]\).

**O1+. Paint \([A]\) \([\text{Red}]\)**

**O2+. Paint \([A]\) \([\text{Blue}]\)**

**O3+. Paint \([A]\) \([\text{Green}]\)**

**O4+. Paint \([B]\) \([\text{Red}]\)**

---

(A)
Example Working Memory

Working memory is a graph. All working memory elements must be “linked” directly or indirectly to a
state.

State: blocks [color, name]

paint brushes for specific colors [color]

```
(S1 ^block B14)
(S1 ^block B23)
(S1 ^color red)
(S1 ^color blue)
(S1 ^color green)
(B14 ^type block)
(B14 ^name A)
(B14 ^color white)
(B23 ^type block)
(B23 ^name B)
(B23 ^color white)
```

```
(S1 ^block B14 B23
 ^color red blue green)
(B14 ^type block ^name A
 ^color white)
(B23 ^type block ^name B
 ^color white)
```
Defining Task in Soar

Create rules for:

- Initialize-color-block operator
  - Propose initialize-color-block
  - Apply initialize-color-block

- Color-block operator
  - Propose color-block
  - Select color-block
  - Apply color-block

- Detect goal achieved
Propose initialize-color-block

If there the top state does not have the name “color-block” then propose the operator to initialize-color-blocks.

\[
\text{sp \{propose*initialize-color-blocks}
\]
\[
\text{(state <s> ^superstate nil}
\]
\[
\text{^-^name color-block)}
\]
\[
\text{-->}
\]
\[
\text{(<s> ^operator <o> +)}
\]
\[
\text{(<o> ^name initialize-color-blocks)}\}
\]

Apply initialize-color-block

If the initialize-color-blocks operator is selected, then add the name to the state and add the colors, and create the blocks A, B, and C.

\[
\text{sp \{apply*initialize-color-blocks}
\]
\[
\text{(state <s> ^operator.name initialize-color-blocks)}
\]
\[
\text{-->}
\]
\[
\text{(<s> ^name color-block}
\]
\[
\text{^-color red green blue}
\]
\[
\text{^-block <b1> <b2> <b3>)}
\]
\[
\text{(<b1> ^type block}
\]
\[
\text{^-color white}
\]
\[
\text{^-name A)}
\]
\[
\text{(<b2> ^type block}
\]
\[
\text{^-color white}
\]
\[
\text{^-name B)}
\]
\[
\text{(<b3> ^type block}
\]
\[
\text{^-color white}
\]
\[
\text{^-name C)}}\}
Propose color-block
If there is a block that has a color different than an existing color, then propose the operator to color that block that color, also create an indifferent preference.

sp {propose*color-block
   (state <s> ^color <color>
    ^block <block>)
   (<block> ^color <> <color>)
   -->
   (<s> ^operator <o> +)
   (<s> ^operator <o> =)
   (<o> ^name color-block
    ^color <color>
    ^block <block>)}

Apply color-block
If there is an operator selected to color a block a color, color that block that color.

sp {apply*color-block
   (state <s> ^operator <o>)
   (<o> ^name color-block
    ^color <color>
    ^block <block>)
   (<block> ^name <name>
    ^color <old-color>)
   -->
   (write (crlf) |Paint block | <name> | | <color>)
   (<block> ^color <old-color> -
    ^color <color>)}
# If an operator is proposed that will color red, then create a best preference for it.
sp {prefer*color-red
   (state <s> ^operator <o> +)
   (<o> ^color red)
   -->
   (<s> ^operator <o> >))}

# If an operator will color red and another operator will color green or blue, then create a better preference.
sp {prefer*color-red-to-blue
   (state <s> ^operator <o1> +
     ^operator <o2> +)
   (<o1> ^color red)
   (<o2> ^color << green blue >>)
   -->
   (<s> ^operator <o1> > <o2>))}
Goal Detection

# If all blocks are color "red" then halt.

sp {detect*color-red
   (state <s> ^block <a> <b> <c>)
   (<a> ^name A ^color red)
   (<b> ^name B ^color red)
   (<c> ^name C ^color red)
   -->
   (halt)}
Persistence!

- Actions of non-operator application rules *retract* when rule no longer matches
  - No longer relevant to current situation
  - Operator proposals and state elaboration
  - Instantiation-support = i-support
  - Rule doesn’t test the selected operator and modify state.
    - Elaborate state
    - Propose operator
    - Create operator preferences

- Actions of operator application rules *persists* indefinitely
  - Otherwise actions retract as soon as operator isn’t selected
  - Operators perform non-monotonic changes to state
  - Operator-support = o-support
  - Rule tests the selected operator and modifies the state
    - Operator application
Simple Eater

Actions:
- forward: move one cell
- rotate: turn right

State:
- sensory data: input-link
- internally maintained: state

Get points for eating food.
-1 for each forward/rotate.
Input/Output in Soar

• All input and output happens through working memory.
• Input is added by perception during input phase:
  – ($<s>$ ^io.input-link <input>)

• Output commands are created by rules on:
  • ($<s>$ ^io.output-link <output>)
    – Sent to motor system in output phase
Propose and apply initialize-eater

If there the top state does not have the name “eater” then propose the operator to initialize-eater.

sp {propose*initialize-eater
    (state <s> ^superstate nil
    --^name eater)
  -->
    (<s> ^operator <o> +)
    (<o> ^name initialize-eater)}

sp {apply*initialize-eater
    (state <s> ^operator.name initialize-eater)
  -->
    (<s> ^name eater)}
Simple Eater Input-link

(<s> ^io.input-link <input>)
(<input> ^east red          # absolute directions and contents
  ^north wall
  ^south red          # these change with forward
  ^west purple

  ^back purple       # relative directions and contents
  ^front red
  ^left wall         # these change with rotate or forward
  ^right red

  ^orientation east  # this changes with rotate
  ^score 0
  ^score-diff 0
  ^food-remaining 10 # 0 when eaten all food

  ^x 1            # these change with forward
  ^y 2

  ^time 1     # this changes with rotate/forward)
Propose and Apply Forward

# if the task is eater and there is something in front,
# then propose moving forward
sp {random*propose*forward
  (state <s> ^name eater
    ^io.input-link.front <f>)}  # will blink

  -->
  (<<s> ^operator <op> + =)
  (<op> ^name forward)}

# if operator forward is selected, then put the forward
# command on the output-link
sp {apply*forward
  (state <s> ^operator <op>
    ^io.output-link <out>))
  (<op> ^name forward)

  -->
  (<out> ^forward <f>)}
Propose and Apply Rotate

# if the task is eater and there is something in front,
# then propose rotate
sp {random*propose*rotate
    (state <s> ^name eater
        ^io.input-link.front) # will blink
    -->
        (<s> ^operator <op> + =)
        (<op> ^name rotate)}

# if operator rotate is selected, then put the rotate
# command on the output-link
sp {apply*rotate
    (state <s> ^operator <op>
        ^io.output-link <out>)
    (<op> ^name rotate)
    -->
        (<out> ^rotate <r>)}
Cleaning up output-link

Need to remove structures on the output-link. Can do this when an operator is selected and get back `^status complete`.

```plaintext
sp {apply*cleanup*output-link
  (state <s> ^operator <op>
     ^io.output-link <out>)
  (<out> ^<cmd> <id>)
  (<id> ^status complete)
  -->
  (<out> ^<cmd> <id> -)
}
```
Detecting Completion

If there is no food remaining, halt.

\[
\text{sp \{ task*complete \}
\begin{align*}
\text{(state \langle s \rangle \ ^\text{name eater}} \\
\text{\quad ^\text{io.input-link.food-remaining 0)} \\
\end{align*}
\]

\[
\quad \rightarrow \\
\quad (\text{halt})
\]

\}
Smarter Eater

• Reject moving forward into walls
• Avoid moving forward into empty cells
Reject wall, Avoid empty

# If forward is proposed and there is a wall in front,
# then reject that operator
sp {eater*reject*forward*wall
  (state <s> ^operator <o> +
   ^io.input-link.front wall)
  (<o> ^name forward)
-->
  (<s> ^operator <o> -)}

# If forward is proposed and there is an empty cell in front,
# and there is a non-empty, non-wall in some other direction
# then prefer turning to forward
sp {eater*avoid*forward*empty
  (state <s> ^operator <o1> +
   ^operator <o2> +
   ^io.input-link <input>)
  (<input> ^<< left right back >> { <> empty <> wall })
  (<o1> ^name forward)
  (<o2> ^name rotate)
-->
  (<s> ^operator <o1> < <o2>)}