Soar Tutorial
Building Intelligent Agents Using Soar

John E. Laird

Historical Perspective

1960 1970 1980 1990 2000

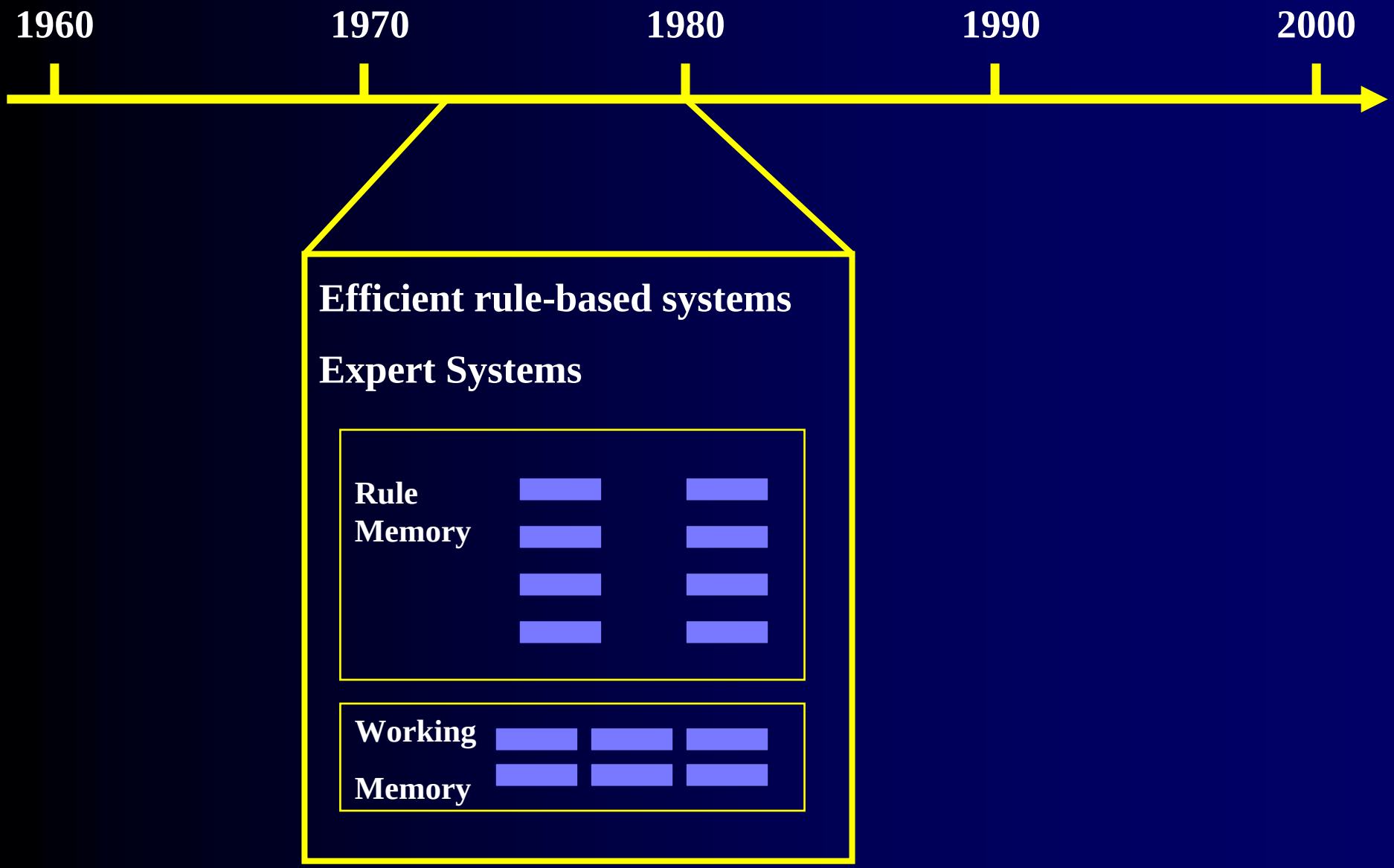
Human Problem Solving

Goal-directed search

Rule-based systems

Newell & Simon

Historical Perspective



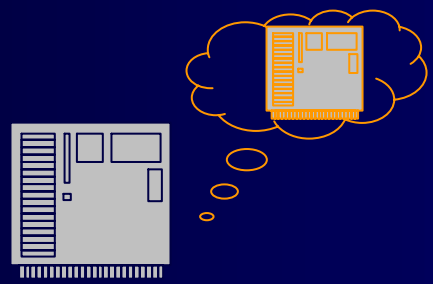
Historical Perspective

1960 1970 1980 1990 2000

Soar

Multi-method problem solving
Knowledge-based, hierarchical reasoning, search, meta-level reasoning, and learning

“Inside the head” problems
R1-Soar: Computer Configuration



Historical Perspective

1960 1970 1980 1990 2000

External environments
- Extreme efficiency
- Mobile robot control
- Stick control of simulated plane

Model human behavior
- Natural language
- Human-computer interaction
- Many forms of learning

Air-Soar

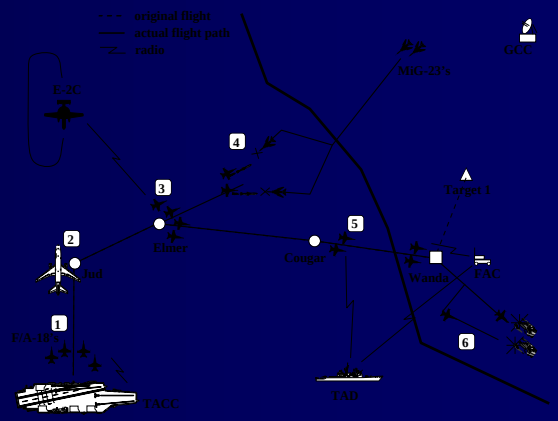
Hero-Soar

Historical Perspective

1960 1970 1980 1990 2000

Intelligent Forces for Training

- **WISSARD/IFOR (DARPA)**
- **Fixed-wing aircraft (UM)**
- **Rotary-wing aircraft (USC/ISI)**
- **STOW-E**
- **STOW-97 – 700 sorties, 100 in air**



TacAir-Soar

Historical Perspective

1960 1970 1980 1990 2000

Soar Technology, Inc.

- Develop and deploy IFORs
- More capabilities, development tools & runtime support

Computer Game AIs

Soar Quakebot

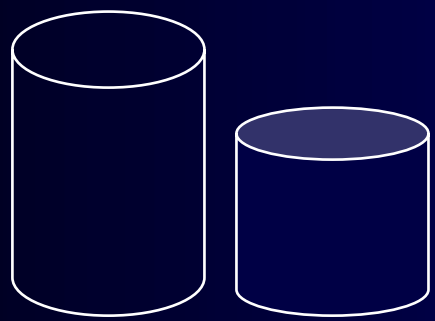
Haunt 2

Desired Behavioral Capabilities

- Interact with a complex world - limited uncertain sensing
- Respond quickly to changes in the world
- Use extensive knowledge
- Use methods appropriate for tasks
- Goal-driven
- Meta-level reasoning and planning
- Generate human-like behavior
- Coordinate behavior and communicate with others
- Learn from experience
- Integrate above capabilities across tasks
- Behavior generated with low computational expense

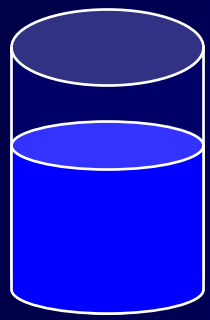
Water Jug Problem

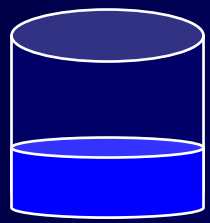
You are given two empty jugs.
One holds five gallons of water and the other holds three gallons.
There is a well that has unlimited water that you can use to completely fill the jugs.
You can also empty a jug or pour water from one jug to another.
There are no marks for intermediate levels on the jugs.
The goal is to fill the three-gallon jug with one gallon of water.



Operators and States

- Operators:
  - Fill a jug from the well.
  - Empty a jug into the well.
  - Pour water from a jug to a jug.
- States
  - Jug-a
    - Volume: 5 gallons
    - Contents: X gallons
    - Empty: Y gallons
  - Jug-b
    - Volume: 3 gallons
    - Contents: M gallons
    - Empty: N gallons





Problem Solving

- Elaborate state:
  - Entailments of current situation
    - *How much space available in jug?*
  - Select an operator
    - Propose candidate operators
      - Usually propose only if can apply
      - *If a jug has water in it, then propose empty for that jug.*
    - Compare operators:
      - Where heuristics can be used to create preferences
      - *Avoid emptying a jug after filling it.*
    - Select current operator
      - Done by the architecture based on preferences created above
  - Apply an operator
    - Change state to reflect operator actions
    - *If emptying a jug, then the contents of the jug are 0.*

- Continually checking to see if achieved goal
  - *If the three gallon jug has one gallon, then success is achieved.*
- Must create an initial state with the starting conditions: us initialization operator
  - *Jug-a: holds 0*
  - *Jug-b: holds 0*

Soar 101
Internal Problem Solving

Elaborate State → **Propose Operator** → **Compare Operators** → **Select Operator** → **Apply Operator**

Decision Procedure

Production Memory

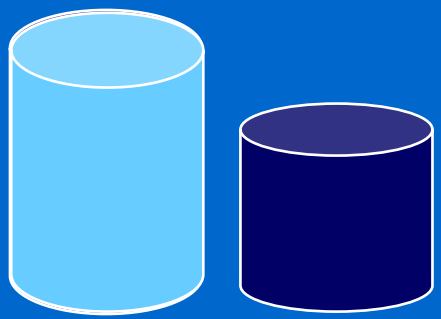
If jug <j> has content <c>, volume <v>, --> ^empty <v> - <c>

If jug <j>, empty > 0, --> propose operator initialize-state

If operator <o> empties a jug --> operator <o> <

If selected operator is initialize-state --> <j> ^contents 0 ^volume 5 <j> ^contents 0 ^volume 3

Working Memory



Operator: fill j1

Operator proposal: fill j1, fill j2

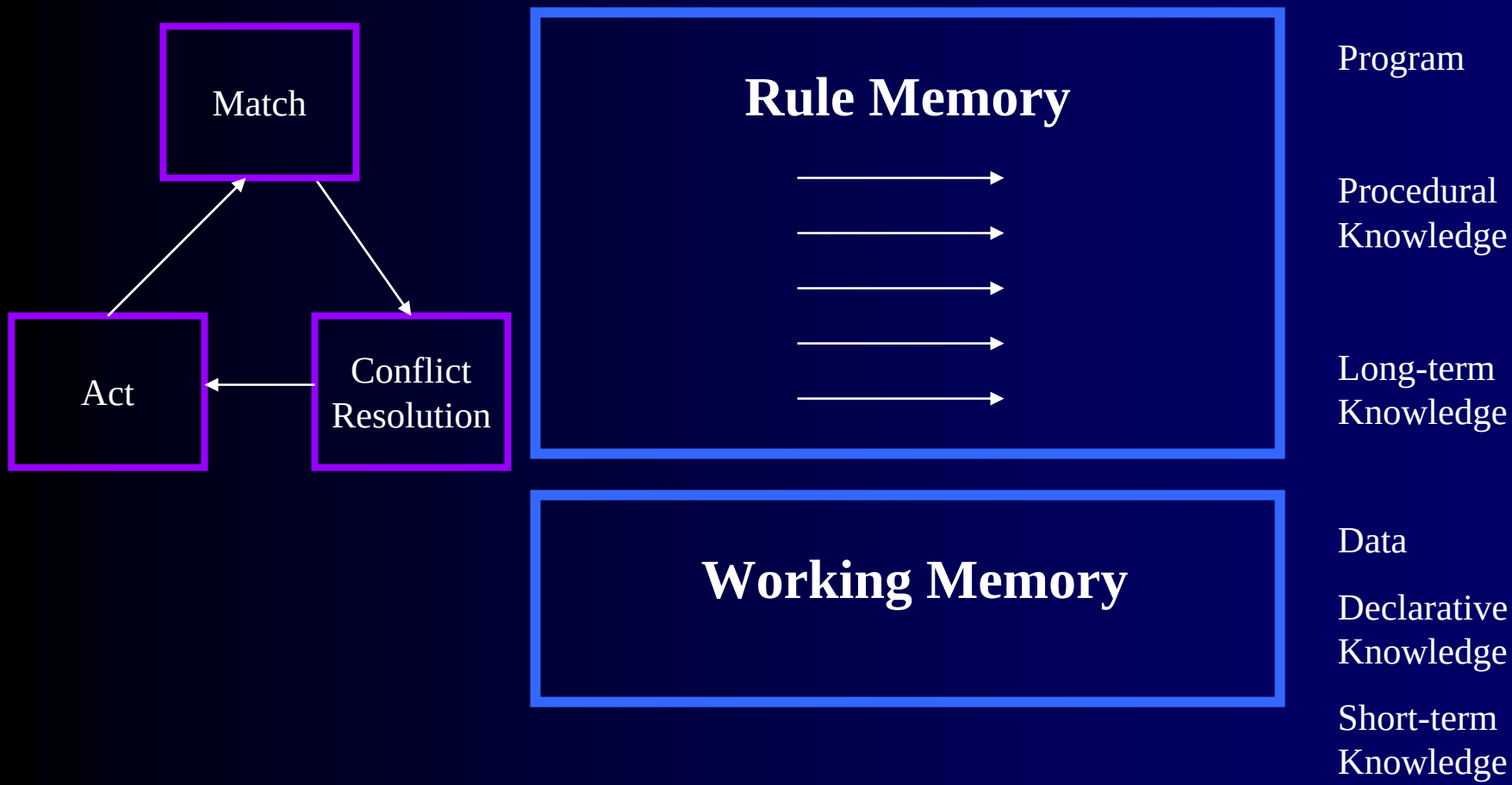
j1 ^volume 5 ^contents 0 *Persistent – o-support*

^empty 5 *Not-persistent – i-support*

j2 ^volume 3 ^contents 0 *Persistent – o-support*

^empty 3 *Not-persistent – i-support*

Rule-Based Systems Structure



Soar Syntax
Hello World Rule

If I exist,
then write “Hello World” and halt.

```
sp {hello-world
   (state <s> ^type state)
-->
   (write “Hello World”)
   (halt)}
```

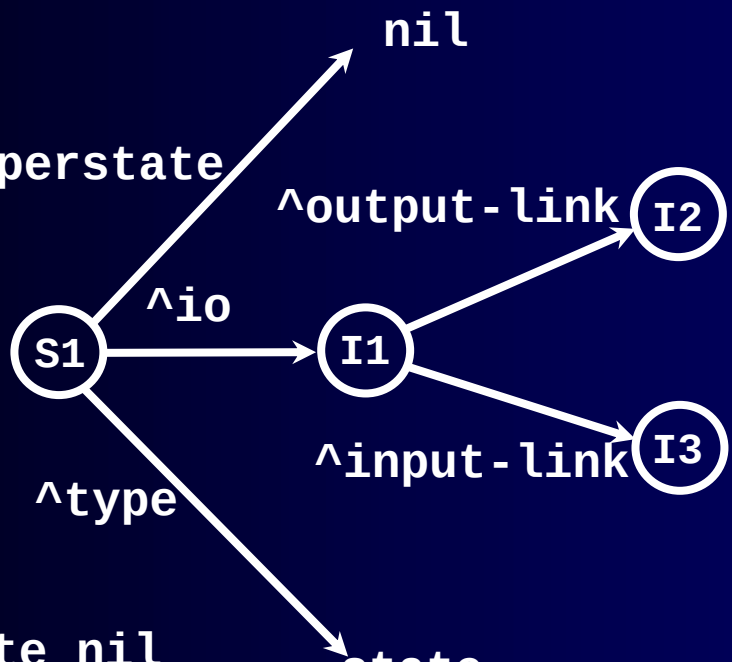
Hello World Operator

```
Propose*hello-world:
If I exist, propose the hello-world operator.

Apply*hello-world:
If the hello-world operator is selected, write “Hello World”
and halt.

sp {propose*hello-world
   (state <s> ^type state)
-->
   (<s> ^operator <o> +)
   (<o> ^name hello-world)}

sp {apply*hello-world
   (state <s> ^operator <o>)
   (<o> ^name hello-world)
-->
   (write |Hello World|)
   (halt)}
```

Creating acceptable preference for operator

Testing selected operator

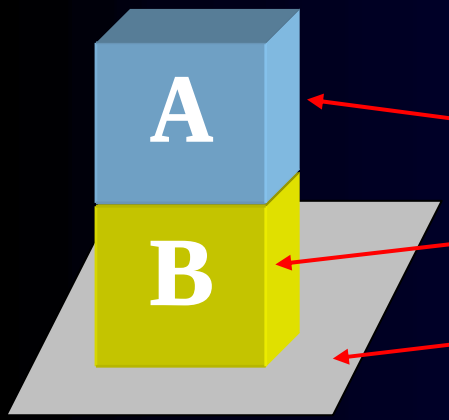
Initial Working Memory




```
S1 ^superstate nil
   S1 ^io I1
   S1 ^type state
I1 ^output-link I2
   I1 ^input-link I3
```

```
(S1 ^io I1 ^superstate nil ^type state)
(I1 ^input-link I3 ^output-link I2)
```

Working Memory




```
(s1 ^block b1 ^block b2 ^table t1)
(b1 ^color blue ^name A ^ontop b2 ^size 1
    ^type block ^weight 14)
(b2 ^color yellow ^name B ^ontop t1 ^size 1
    ^type block ^under b1 ^weight 14)
(t1 ^color gray ^shape square
    ^type table ^under b2)
```

Water Jug State Structure

- Jug-a
  - Volume: 5 gallons
  - Contents: x gallons
  - Empty: y gallons
- Jug-b
  - Volume: 3 gallons
  - Contents: m gallons
  - Empty: n gallons

```
(<s> ^5-gallon-jug-holds 0
     ^3-gallon-jug-holds 0)
```

```
(<s> ^jug <j1>
     ^jug <j2>)
(<j1> ^volume 5
      ^contents 0
      ^empty 0)
(<j2> ^volume 3
      ^contents 0
      ^empty 0)
```

multi-valued attribute

Water Jug Operators

- Initialize
- Fill a jug from the well
- Empty a jug into the well
- Pour water from a jug to a jug

- For every operator, must define at least two rules:
  - Proposal
  - Application
- Can also create selection rules, but not always necessary

Initialize

- Proposal

  If there are no jugs defined,

  then *propose* the initialize water-jug operator.

- Application

  If the initialize water-jug operator is *selected*,

  then create an empty 5 gallon jug and an empty 3 gallon jug.

```
sp {propose*initialize-water-jug
   (state <s> ^type state
              -^jug <j>)
   -->
   (<s> ^operator <o> +)
   (<o> ^name initialize)}
```

Test that no jugs exist

```
sp {apply*initialize-water-jug
   (state <s> ^operator <o>)
   (<o> ^name initialize)
   -->
   (<s> ^jug <j1>
        ^jug <j2>)
   (<j1> ^volume 5
         ^contents 0)
   (<j2> ^volume 3
         ^contents 0)}
```

Fill Jug

- Proposal

  If there is a jug that is not full, then *propose* the fill operator.

- Application

  If the fill operator is *selected* for a jug,

  then change the contents of that jug to its volume.

```
sp {propose*fill-water-jug
   (state <s> ^jug <j>)
   (<j> ^free > 0)
   -->
   (<s> ^operator <o> +)
   (<o> ^name fill
        ^jug <j>)}
```

Only match if value > 0

```
sp {apply*fill-water-jug
   (state <s> ^operator <o>)
   (<o> ^name fill
        ^jug <j>)
   (<j> ^volume <v>
        ^contents <c>)
   -->
   (<j> ^contents <v>
        ^contents <c> -)}
```

Cause WME to be removed

Instantiations

```
sp {propose*fill-water-jug
   (state <s> ^jug <j>)
   (<j> ^free > 0)
   -->
   (<s> ^operator <o> + =)
   (<o> ^name fill
        ^jug <j>)}
```

= means indifferent
(a random selection will be made)

For each set of working memory elements that successfully match the rule, an *instantiation* is created.

```
(s1 ^jug j1)          (s1 ^jug j2)
(j1 ^free 5)          (j2 ^free 3)
```

Both instantiations fire, creating two new operators and preferences:

```
Working Memory Elements:
(s1 ^operator o1 +)   (s1 ^operator o2 +)
(o1 ^name fill)       (o2 ^name fill)
(o1 ^jug j1)          (o2 ^jug j2)
Preferences:
(s1 ^operator o1 +)   (s1 ^operator o2 +)
(s1 ^operator o1 =)   (s1 ^operator o2 =)
```

The decision procedure will pick only one (randomly because they are indifferent).

Elaboration of ^free

If a jug has volume *v* and contents *c*, then it has free *v* – *c*.

```
sp {elaborate*water-jug*free
   (state <s> ^jug <j>)
   (<j> ^volume <v>
        ^contents <c>)
   -->
   (<j> ^free (- <v> <c>))}
```

Subtraction of <c> from <v>

^free is *instantiation-supported = i-support*

When this specific match of the rule retracts, the working memory element is retracted.

The rule may match new values and produce a new working memory element.

Instantiations

```
sp {elaborate*water-jug*free
   (state <s> ^jug <j>)
   (<j> ^volume <v>
        ^contents <c>)
   -->
   (<j> ^free (- <v> <c>))}
```

For each set of working memory elements that successfully match the rule, an *instantiation* is created.

```
(s1 ^jug j1)          (s1 ^jug j2)
(j1 ^volume 5)        (j2 ^volume 3)
(j1 ^contents 0)      (j2 ^contents 0)
```

Both instantiations fire in parallel, creating two new working memory elements:

```
New Working Memory Elements:
(j1 ^free 5)           (j2 ^free 3)
```

If one of the matched working memory elements in an instantiation in removed from working memory, the WME it created is removed.

Empty Jug

- Proposal

  If there is a jug that is not empty,
  then *propose* the empty operator.

- Application

  If the empty operator is *selected* for a jug,
  then change the contents of that jug to 0.

```
sp {propose*empty-water-jug
   (state <s> ^jug <j>)
   (<j> ^contents <> 0)
   -->
   (<s> ^operator <o> + =)
   (<o> ^name empty
        ^jug <j>)}
```

```
sp {apply*empty-water-jug
   (state <s> ^operator <o>)
   (<o> ^name empty
        ^jug <j>)
   (<j> ^contents <c>)
   -->
   (<j> ^contents 0
        ^contents <c> -)}
```

Pour Proposal

- Proposal

  If there is a jug that is not empty, and the other jug is not full then *propose* the pour operator.

```
sp {propose*pour-water-jug
   (state <s> ^jug <j1>
        ^jug {<> <j1> <j2>})
   (<j1> ^contents > 0)
   (<j2> ^free > 0)
   -->
   (<s> ^operator <o> + =)
   (<o> ^name pour
        ^jug <j1>
        ^into <j2>)}
```

Pour: two implementations

- If the source jug holds less than or equal to the jug being filled
- If the source jug holds more than the jug being filled

Pour Apply Case 1

If the pour operator is selected, and

the contents of the jug being poured from are less than or equal to the free amount of the into jug,

then set the contents of the source jug to 0;

set the contents of the into jug to the sum of the two jugs.

```
sp {apply*pour*not-empty-source
    (state <s> ^operator <o>)
    (<o> ^name pour
         ^jug <i>
         ^into <j>)
    (<i> ^contents { <icon> <= <jfree> })
    (<j> ^contents <jcon>
         ^free <jfree>)
    -->
    (<i> ^contents 0
                   <icon> -)
    (<j> ^contents (+ <jcon> <icon>)
                   <jcon> -)}
```

Pour Apply Case 2

If the pour operator is selected, and
  the contents of the jug being poured from are greater than
  the free amount of the into jug,
then set the contents of the source jug to its original contents minus
  the free of the destination jug, and
  set the contents of the into jug to its volume.

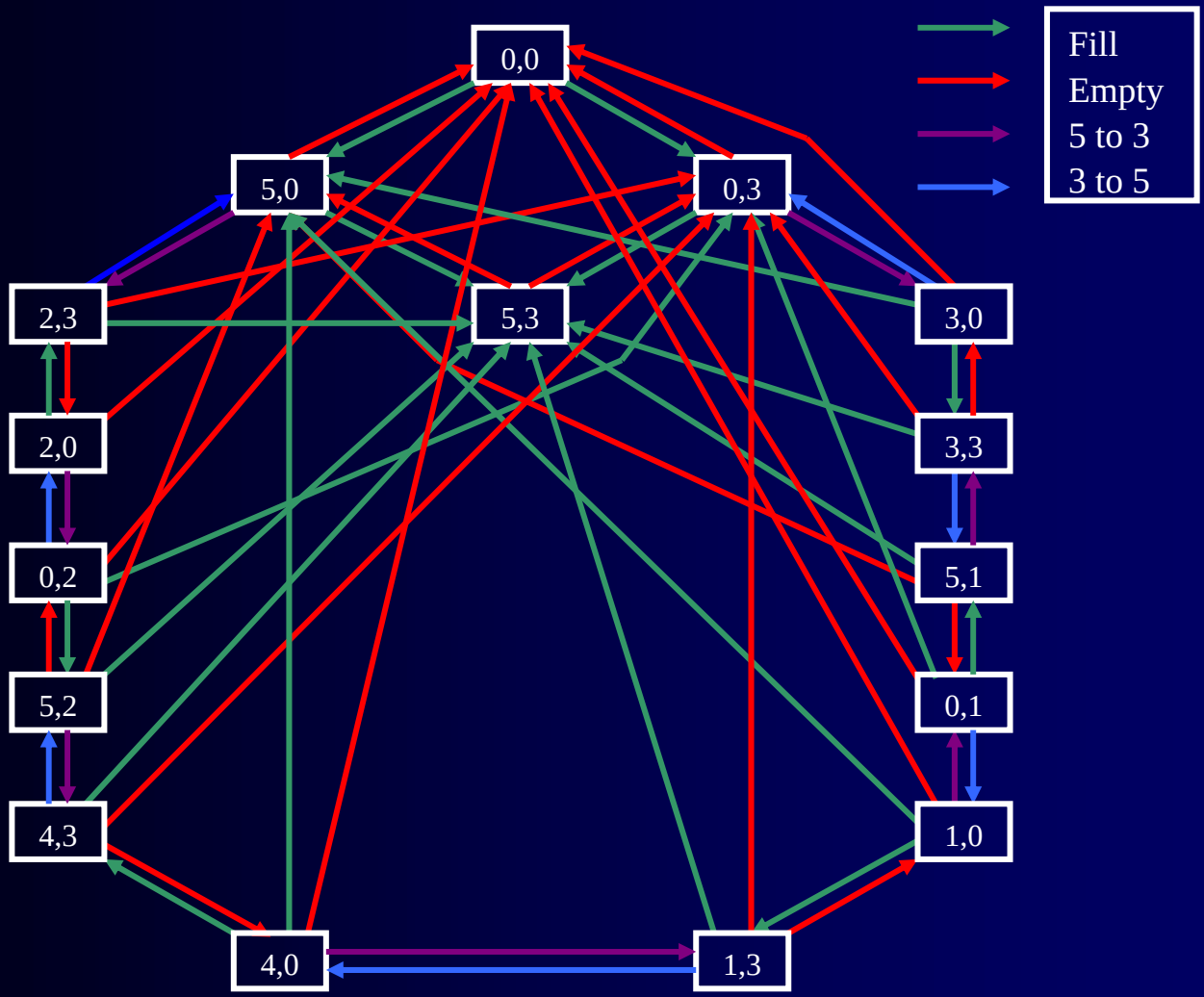
```
sp {apply*pour*empty-source
   (state <s> ^operator <o>)
   (<o> ^name pour
        ^jug <i>
        ^into <j>)
   (<i> ^contents { <icon> > <jfree> })
   (<j> ^volume <jvol>
        ^contents <jcon>
        ^free <jfree>)
   -->
   (<i> ^contents (- <icon> <jfree>)
                  <icon> -)
   (<j> ^contents <jvol>
                  <jcon> -)}
```

Goal Detection

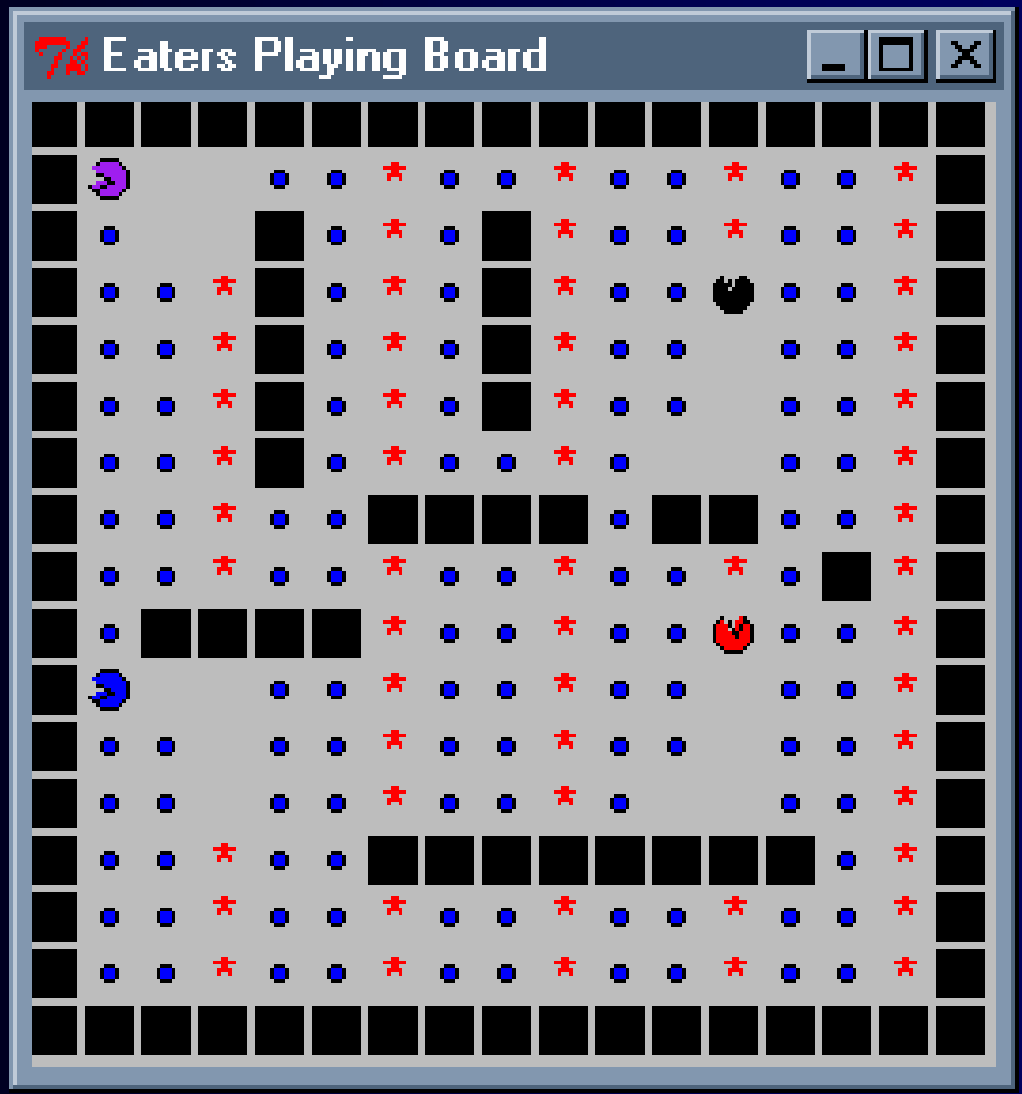
If there is a jug with volume three and contents one, then write that the problem has been solved and halt.

```
sp {waterjug*detect*goal*achieved
   (state <s> ^name waterjug
              ^jug <j>)
   (<j> ^volume 3
        ^contents 1)
-->
   (write (crlf) |The problem has been solved.|)
   (halt)}
```

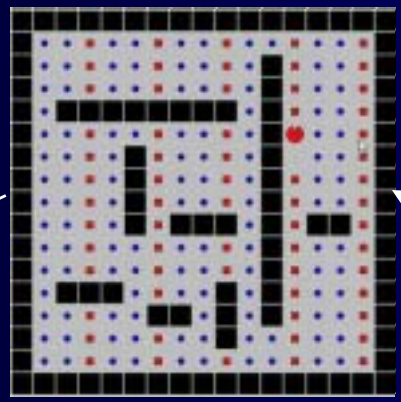
Waterjug Problem Space



Eaters



Soar 101- Eaters Style



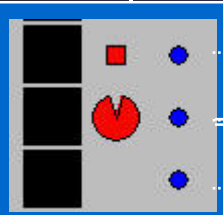
Input | **Propose Operator** | **Compare Operators** | **Select Operator** | **Apply Operator** | **Output**

If cell in direction <d> is not a wall,
-->
propose operator move <d>

If operator <o1> will move to a ~~empty cell~~ **food** and operator <o2> will move to a normal food,
--> operator <o1> > <o2>

If an operator is selected to move <d>
-->
create output move-direction <d>

Production Memory



East
North
South

`move-direction North`

North > East
South > East
North = South

Working Memory

Move North

Propose*move-north:
If I exist, propose the move-north operator.

Apply*move-north:
If the move-north operator is selected, create the command to move-north on the output-link.

```
sp {propose*move-north
   (state <s> ^type state)
-->
   (<s> ^operator <o> +)
   (<o> ^name move-north)}
```

```
sp {apply*move-north
   (state <s> ^operator <o>
              ^io <io>)
   (<io> ^output-link <ol>)
   (<o> ^name move-north)
-->
   (<ol> ^move <move>)
   (<move> ^direction north)}
```

The move command moves the eater one position in that direction

Short Cut

```
sp {apply*move-north
   (state <s> ^operator <o>
              ^io <io>)
   (<io> ^output-link <out>)
   (<o> ^name move-north)
-->
   (<out> ^move <move>)
   (<move> ^direction north)}
```

```
sp {apply*move-north
   (state <s> ^operator.name move-north
              ^io.output-link <out>)
-->
   (<out> ^move.direction north)}
```

Operator Selection

- Current operators only changes when decision changes.
- Reasons for new decision:
  - proposal instantiation no longer matches and retracts proposal
  - other operators dominate selection through preferences

Problem 1 with Move-North

- Operator is selected only once.
  - When selected, moves Eater only one step
- Operator needs to *retracts* after it has applied so can be reselected.
  - Will then generate new action.
- Need to test something that changes when operator applies

Improved Move-North

```
sp {propose*move -north
   (state <s> ^io.input-link.eater <e>)
   (<e> ^x <x> ^y <y>)
-->
   (<s> ^operator <o> +)
   (<o> ^name move-north)}
```

Persistence

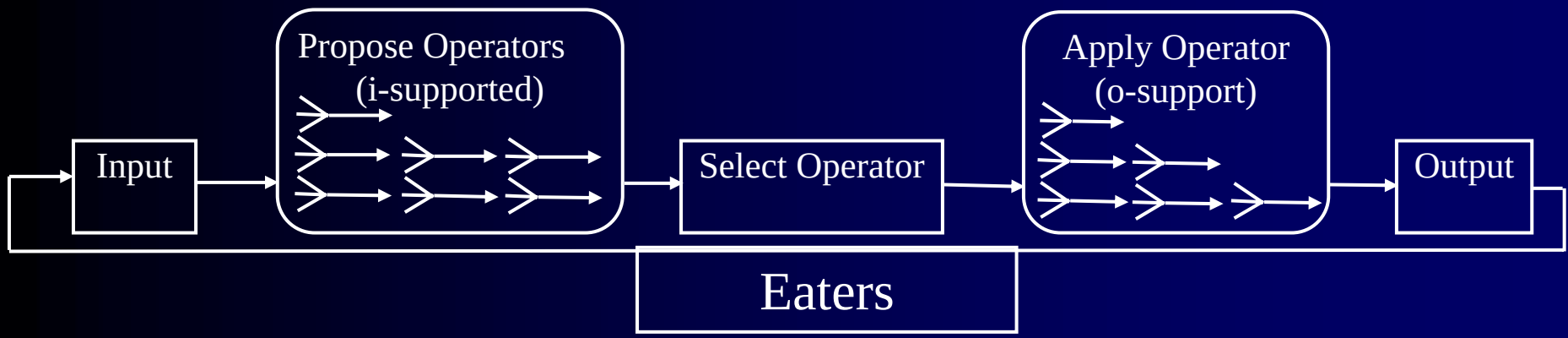
- Actions of operator application rules *persists* indefinitely
  - Otherwise actions would retract as soon as operator isn't selected
  - Operators perform non-monotonic changes to state
  - *Does rule test a selected operator and modify the state?*

- Actions of non-operator application rules *retract* when rule no longer matches
  - No longer relevant to current situation
  - Operator proposals and state elaboration
  - *Rule doesn't test operator and modify state.*

Problem 2 with Move-North

- Action command on output-link is not removed
- It persists on the state after the operator is no longer selected
- Need to remove old command from output-link
- Need to detect when action is complete.
  - ^io.output-link.move.status complete

Expanded Soar Cycle

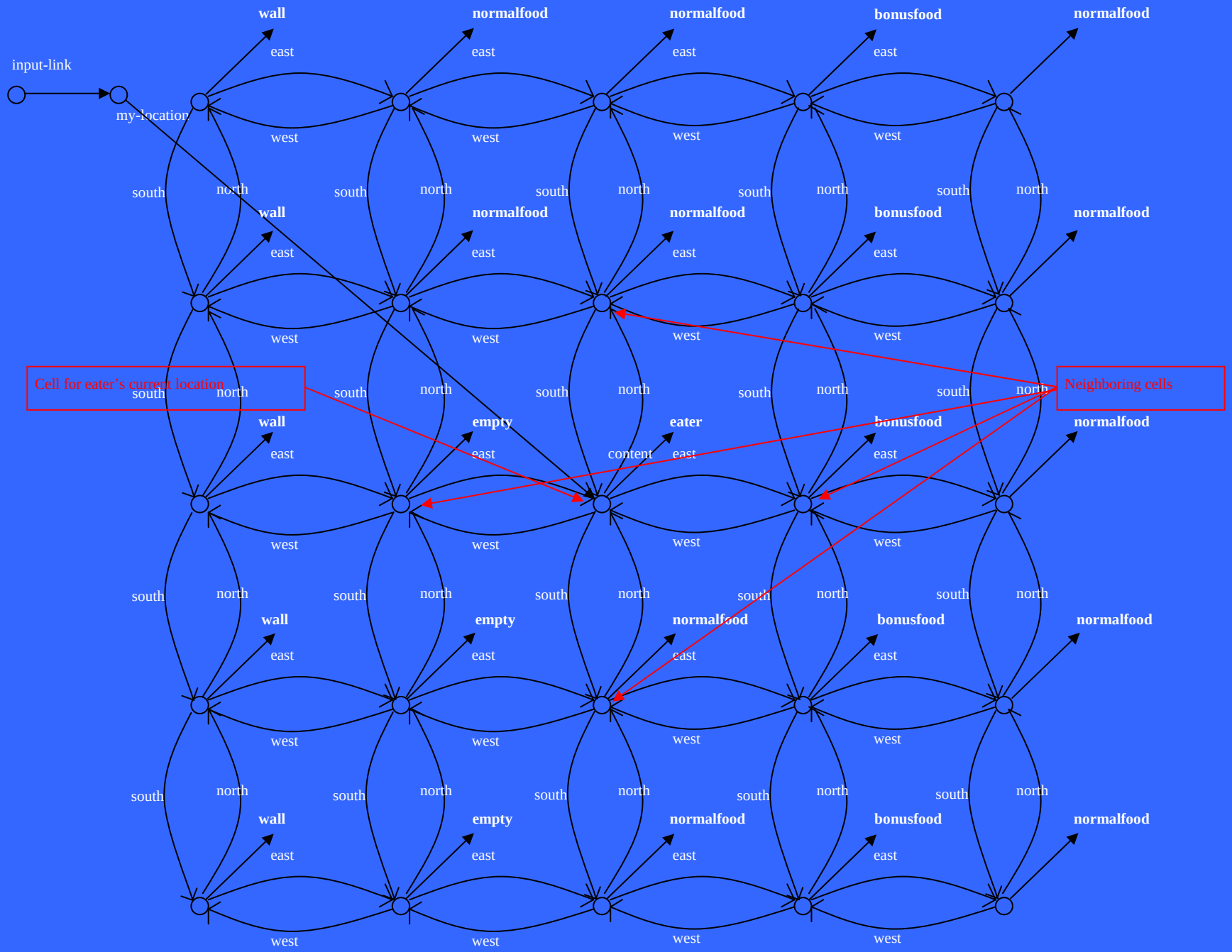


Extended Move-North

```
sp {apply*move-north*remove-move
   (state <s> ^operator.name move-north
              ^io.output-link <out>)
   (<out> ^move <move>)
   (<move> ^status complete)
-->
   (<out> ^move <move> -)}
```

Move

```
# Propose*move*normalfood
# If there is normalfood in an adjacent cell,
#    propose move in the direction of that cell
#    and indicate that this operator can be selected randomly.
#
# Propose*move*bonusfood
# If there is bonusfood in an adjacent cell,
#    propose move in the direction of that cell
#    and indicate that this operator can be selected randomly.
#
# Apply*move
# If the move operator for a direction is selected,
#    generate an output command to move in that direction.
#
# Apply*move*remove-move:
# If the move operator is selected,
#    and there is a completed move command on the output link,
#    then remove that command.
```


input-link
my-location
wall
normalfood
bonusfood
empty
eater
content
east
west
north
south
Cell for eater's current location
Neighboring cells

Move-to-food

```
sp {propose*move
   (state <s> ^io <io>)
   (<io> ^input-link <input-link>)
   (<input-link> ^my-location <my-loc>)
   (<my-loc> ^<direction> <cell>)
   (<cell> ^content normalfood)
-->
   (<s> ^operator <o> +)
   (<s> ^operator <o> =)
   (<o> ^name move
        ^direction <direction>)}


sp {propose*move*normalfood
   (state <s> ^io.input-link.my-location.<dir>.content normalfood)
-->
   (<s> ^operator <o> + =)
   (<o> ^name move
        ^direction <dir>)}
```

Move-to-food apply

```
sp {apply*move
   (state <s> ^io.output-link <ol>
              ^operator <o>)
   (<o> ^name move
        ^direction <dir>)
-->
   (<ol> ^move.direction <dir>)}
```

```
 sp {apply*move*remove-move
    (state <s> ^io.output-link <ol>
               ^operator.name move)
    (<ol> ^move <move>)
    (<move> ^status complete)
 -->
    (<ol> ^move <move> -)}
```

Short Cut: << >>

```
sp {propose*move-to-food
   (state <s> ^io.input-link.my-location.<dir>.content
                  << normalfood bonusfood >>)
-->
   (<s> ^operator <o> + =)
   (<o> ^name move
        ^direction <dir>)}
```

```
sp {monitor*move-to-food
   (state <s> ^operator <o>)
   (<o> ^name move
        ^direction <direction>)
-->
   (write |Direction: | <direction>)}
```

General Move Operator

```
# Propose*move:
# If there is normalfood, bonusfood, eater, or empty in an adjacent cell,
#    propose move in the direction of that cell, with the cell's content,
#    and indicate that this operator can be selected randomly.
```

```
 sp {propose*move*1a
    (state <s> ^io.input-link.my-location.<dir>.content
               { <content> << empty normalfood bonusfood eater >> })
 -->
    (<s> ^operator <o> + =)
    (<o> ^name move
         ^direction <dir>
         ^content <content>)}
```

General Move

```
sp {propose*move
   (state <s> ^io.input-link.my-location.<dir>.content
              { <content> <> wall })
-->
   (<s> ^operator <o> + =)
   (<o> ^name move
        ^direction <dir>
        ^content <content>)}


sp {select*move*normalfood-better-than-empty-eater
   (state <s> ^operator <o1> +
              ^operator <o2> +)
   (<o1> ^name move
         ^content normalfood)
   (<o2> ^name move
         ^content << empty eater >>)
-->
   (<s> ^operator <o1> > <o2>)}
```

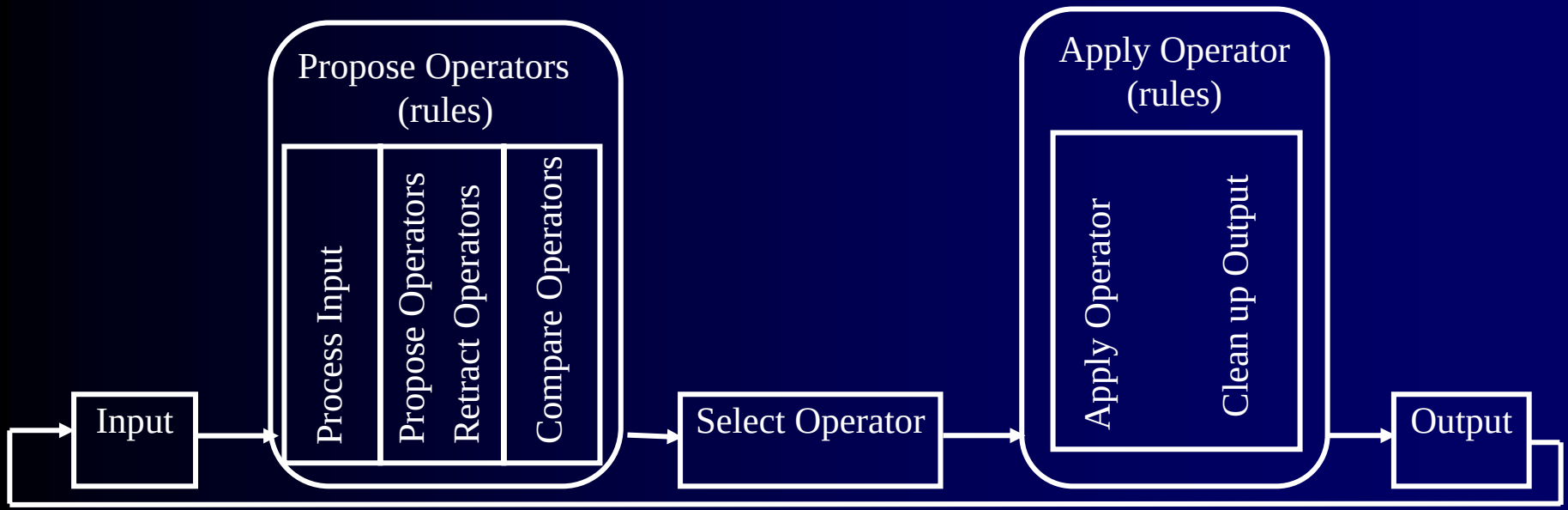
More Move Selection

```
sp {select*move*avoid-empty-eater
   (state <s> ^operator <o1> +)
   (<o1> ^name move
         ^content << empty eater >>)
-->
   (<s> ^operator <o1> <)}


sp {select*move*prefer*bonusfood
   (state <s> ^operator <o1> +)
   (<o1> ^name move
         ^content bonusfood
-->
   (<s> ^operator <o1> >)}
```

Summary of Preferences

```
Acceptable:  <o1> +
Reject:      <o1> -
Better:      <o1> > <o2>
Worse:       <o1> < <o1>
Best:        <o1> >
Worst:       <o1> <
Indifferent: <o1> = <o2>
Indifferent: <o1> =
```

Complete Soar Cycle



Record Last-Direction

```
sp {apply*move*create*last-direction
   (state <s> ^operator <o>)
   (<o> ^name move
        ^direction <direction>)
-->
   (<s> ^last-direction <direction>)}


sp {apply*move*remove*last-direction
   (state <s> ^operator <o>
              ^last-direction <direction>)
   (<o> ^direction <> <direction>
        ^name move)
-->
   (<s> ^last-direction <direction> -)}
```

Precompute Opposites

```
sp {initialize*state*directions
   (state <ss> ^type state)
   -->
   (<ss> ^directions <n> <e> <s> <w>)
   (<n> ^value north ^opposite south)
   (<e> ^value east  ^opposite west)
   (<s> ^value south ^opposite north)
   (<w> ^value west  ^opposite east)}
```

Don’t propose or reject last move

```
sp {propose*move*no-backward
   (state <s> ^io.input-link.my-location.<dir>.content <> wall
              ^directions <d>
             -^last-direction <o-dir>)
   (<d> ^value <dir>
        ^opposite <o-dir>)
-->
   (<s> ^operator <o> +, =)
   (<o> ^name move
        ^direction <dir>)}
```

```
sp {select*move*reject*backward
   (state <s> ^operator <o> +
              ^directions <d>
              ^last-direction <dir>)
   (<d> ^value <dir>
        ^opposite <o-dir>)
   (<o> ^name move
        ^direction <o-dir>)
-->
   (<s> ^operator <o> -)}
```

Jump

```
sp {propose*jump
   (state <s> ^io.input-link.my-location.<dir>.<dir>.content
                <> wall)
-->
   (<s> ^operator <o> + =)
   (<o> ^name jump
        ^direction <dir>)}
```

Jump/Move Selection

```
sp {init*elaborate*name-content-value
   (state <s> ^type state)
-->
   (<s> ^name-content-value <c1> <c2> <c3> <c4>
                            <c5> <c6> <c7> <c8>)
   (<c1> ^name move ^content empty ^value 0)
   (<c2> ^name move ^content eater ^value 0)
   (<c3> ^name move ^content normalfood ^value 5)
   (<c4> ^name move ^content bonusfood ^value 10)
   (<c5> ^name jump ^content empty ^value -5)
   (<c6> ^name jump ^content eater ^value -5)
   (<c7> ^name jump ^content normalfood ^value 0)
   (<c8> ^name jump ^content bonusfood ^value 5)}
```

Jump/Move Selection

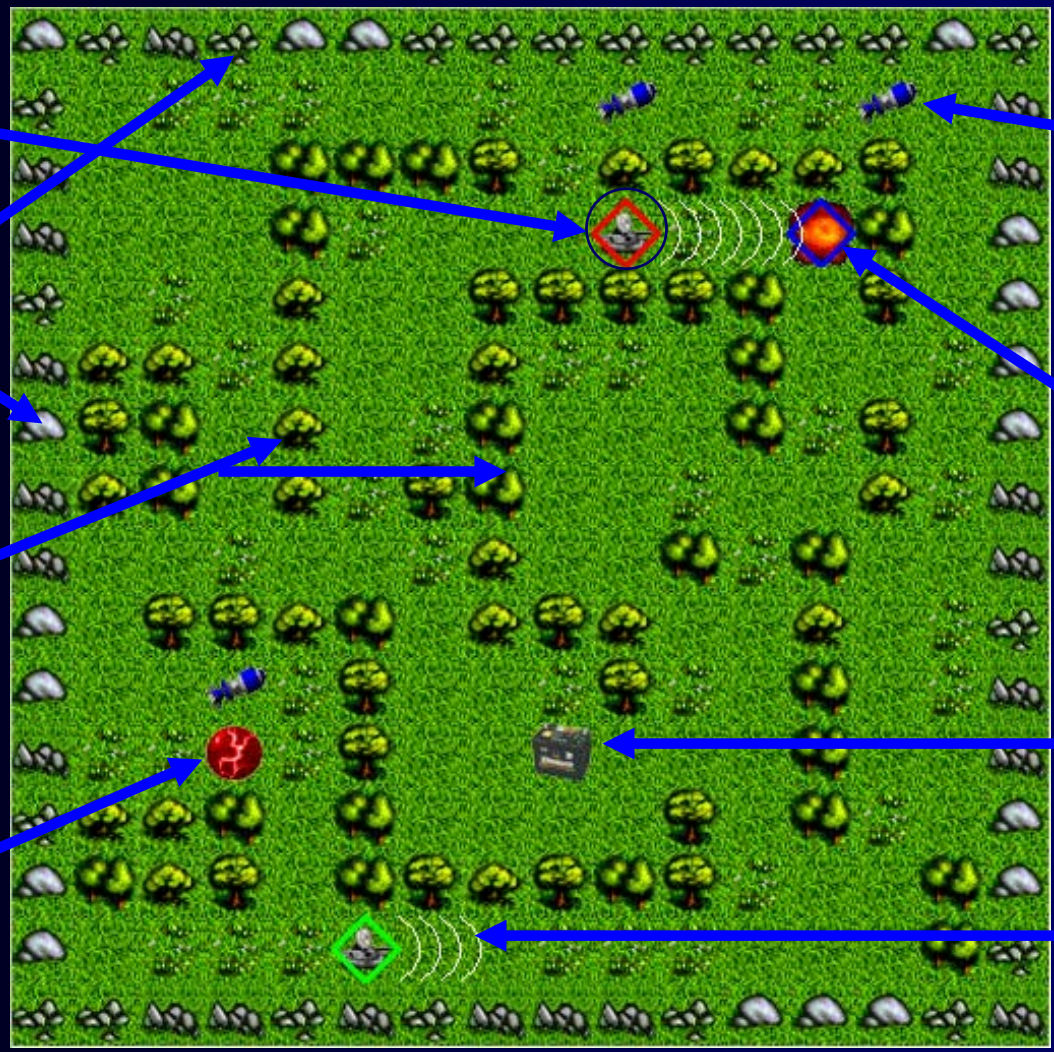
```
sp {elaborate*operator*value
   (state <s> ^operator <o> +
              ^name-content-value <ccv>)
   (<o>   ^name <name> ^content <content>)
   (<ccv> ^name <name> ^content <content> ^value <value>)
-->
   (<o> ^value <value>)}
```

```
sp {select*compare*best*value
   (state <s> ^operator <o1> +
              ^operator <o2> +)
   (<o1> ^value <v>)
   (<o2> ^value < <v>)
-->
   (<s> ^operator <o1> > <o2>)}
```

Soar Tutorial Part II

Subgoaling with TankSoar

New Environment: TankSoar

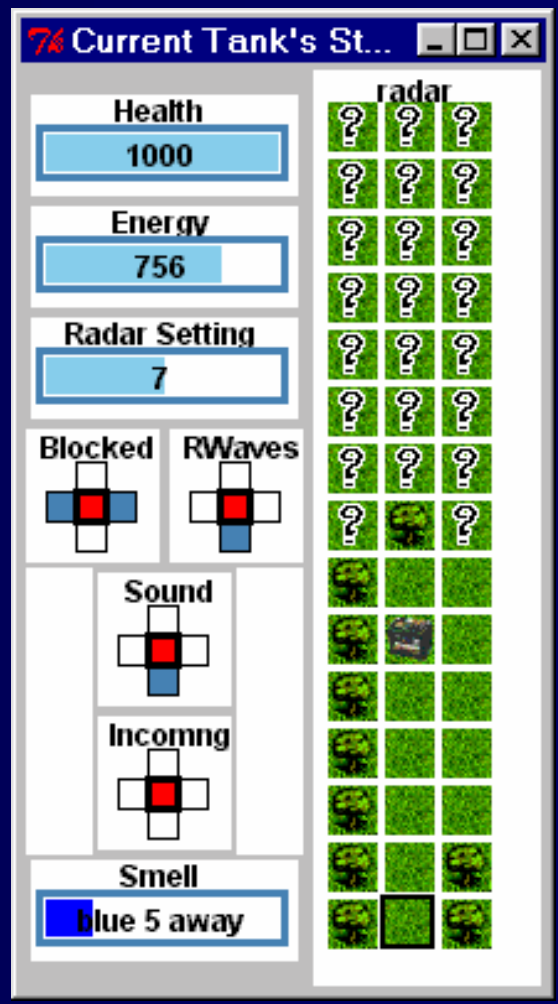


TankSoar Output

- Eaters
  - ^move.direction north/south/east/west
  - ^jump.direction north/south/east/west
- TankSoar
  - ^move.direction: left/right/forward/backward
  - ^rotate.direction: left/right
  - ^fire.weapon missile
  - ^radar.switch on/off
  - ^radar-power.setting 1-14
  - ^shields.switch on/off

TankSoar Input

- Eaters
  - ^my-location.<*dir*>.content
- TankSoar
  - ^blocked – ^incoming
  - ^radar – ^rwaves
  - ^smell – ^sound
  - ^health – ^energy
  - ^radar-setting – ^shield-status



Exercise #1: Drive a Tank (10 minutes)

1. Create a human-controlled tank
   - Hold the Ctrl key and click on an open spot on the map.
2. Experiment with the buttons “Manual Controls” window.
3. Find the “Current Tank’s Status” window.
   - Verify that you understand everything in it.
4. Have your partner create an opponent tank
   - Take turns making moves by clicking the tank in the map window and then selecting the move in the “Manual Controls” window

Game Dynamics

- Health
  - Starts at 1000. Death at zero.
  - Drive into wall/border: -100
  - Missile hits tank: -400
  - Healthcharger: +150/turn
- Missiles
  - Starts at 15
  - Firing: -1
  - Missile pack: +7
  - Missiles travel 1.3 times the speed of a tank
- Energy
  - Starts at 1000.
  - Missile hits shield: -250
  - Shields: -20/turn
  - Radar: -1-14/turn
  - Energycharger: +250/turn
- When energy reaches zero
  - Shields/radar no longer function
  - Smell, blocked, incoming, rwaves still work

Game Dynamics #2

- Tanks can run until output is generated.
- Instant death if hit while on a charger.
- Killed tanks resurrect in new, random location (unlimited lives)
- Scoring:
  - Hits: +2 points – Being Hit: -1 point
  - Kills: +3 points – Being killed: -2 points
- Winner: First tank with 50 points.

Exercise #2: Wandering Tank

1. Copy the default agent
2. Paste in the general apply/remove rules from your Eaters agent
3. Create a Soar tank with three operators:
   - Move, Turn and Radar-Off

(Details on next 2 slides)

General apply/remove rules

```
sp {apply*operator*create-action-command
   (state <s> ^operator.actions.<att> <val>
              ^io.output-link <out>)
-->
   (<out> ^<att> <val>)}

sp {apply*operator*remove-command
   (state <s> ^operator.actions
              ^io.output-link <out>)
   (<out> ^<att> <value>)
   (<value> ^status complete)
-->
   (<out> ^<att> <value> -)}
```

Wander Operators

- Move
  - move forward if not blocked
- Turn
  - If front is blocked, rotate to clear direction, turn radar on the radar with power 13
  - If blocked on the front, left *and* right (dead-end) then turn left.
- Elaboration: Radar-off
  - If the radar is on and no objects are visible then turn the radar off
  - Piggy back on move operator

Move Operator

- Proposal:
  - If the tank is not blocked in the forward direction, propose move forward operator.

```
sp {propose*move
   (state <s> ^io.input-link.blocked.forward no)
-->
   (<s> ^operator <o> +)
   (<o> ^name move
        ^actions.move.direction forward)}
```

- Default rules will copy action to output-link
- Will terminate next cycle because blocked changes after a move

Turn Operator

- Proposal:
  - If the tank is blocked in the forward direction, propose rotate and radar operator.

```
sp {propose*turn
   (state <s> ^io.input-link.blocked.forward yes)
-->
   (<s> ^operator <o> + =)
   (<o> ^name turn
        ^actions <a>)
   (<a> ^rotate.direction left
        ^radar.switch on
        ^radar-power.setting 13)}
```

- Should turn only toward open direction

Better Turn Operator

```
sp {propose*turn
   (state <s> ^io.input-link.blocked <b>)
   (<b> ^forward yes
        ^ { << left right >> <direction> } no)
-->
   (<s> ^operator <o> + =)
   (<o> ^name turn
        ^actions <a>)
   (<a> ^rotate.direction <direction>
        ^radar.switch on
        ^radar-power.setting 13)}
```

Turn Around

```
sp {propose*turn*around
   (state <s> ^io.input-link.blocked <b>)
   (<b> ^forward yes ^left yes ^right yes)
-->
   (<s> ^operator <o> +)
   (<o> ^name turn
        ^actions.rotate.direction left)}
```

Radar-off

- Do in parallel with move if radar is on and nothing on radar.

```
sp {wander*elaborate*move*radar-off
   (state <s> ^operator <o> +
              ^io.input-link <il>)
   (<o> ^name move)
   (<il> ^radar-status on
        -^radar.<< energy health missiles tank >>)
-->
   (<o> ^actions.radar.switch off)}
```

What Next?

Rotate

Move

Fire!

Ambush

Dodge!

Recharge

Hunt

Chase

Charge!

Reload

Hide

Turn

Set Power Level

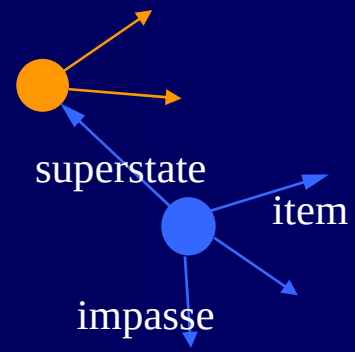
Radar on

Shields Up

SUBGOALS in Soar

Impasses and Subgoals

- Problem:
  - What to do when inconsistent of incomplete knowledge?
- Approach:
  - Detect impasses in decision procedure
  - Create substate with augmentations that define impasse
    - Superstate
    - Impasse – no-change, tie, conflict, …
    - Item – tied or conflicted operators
    - …
  - Impasse resolved when decision can be made

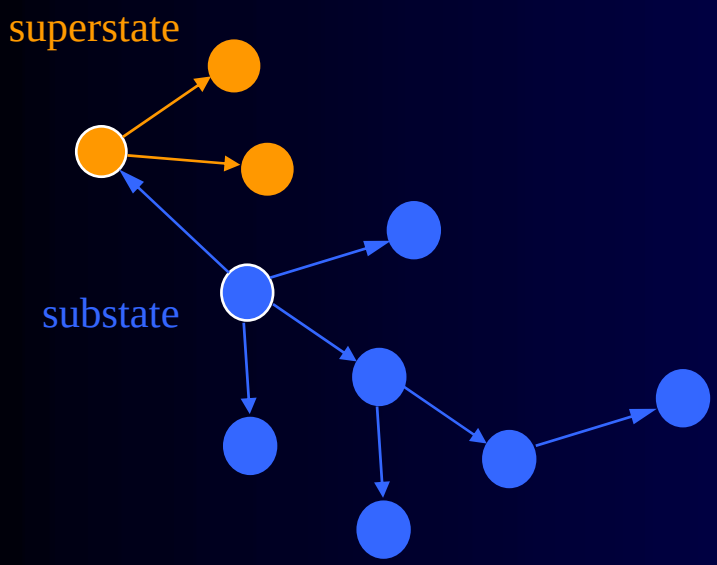


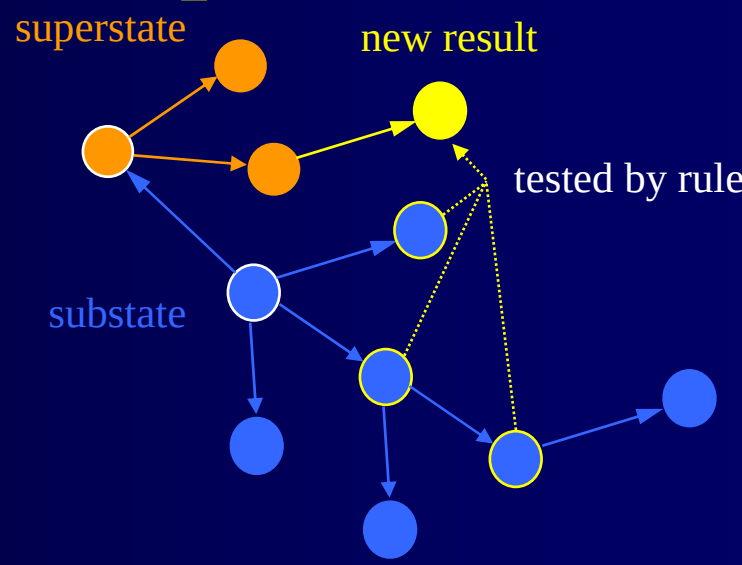
- Implications:
  - Substate is really meta-state that allows system to reflect
  - All basic problem solving functions open to reflection (and learning)
    - Operator creation, selection, application, state elaboration
  - Substate is where knowledge to resolve impasse can be found

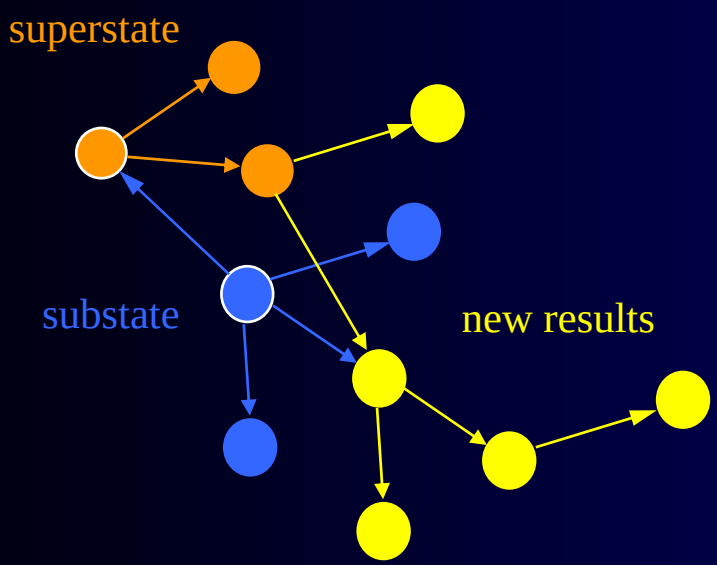
Substate Results

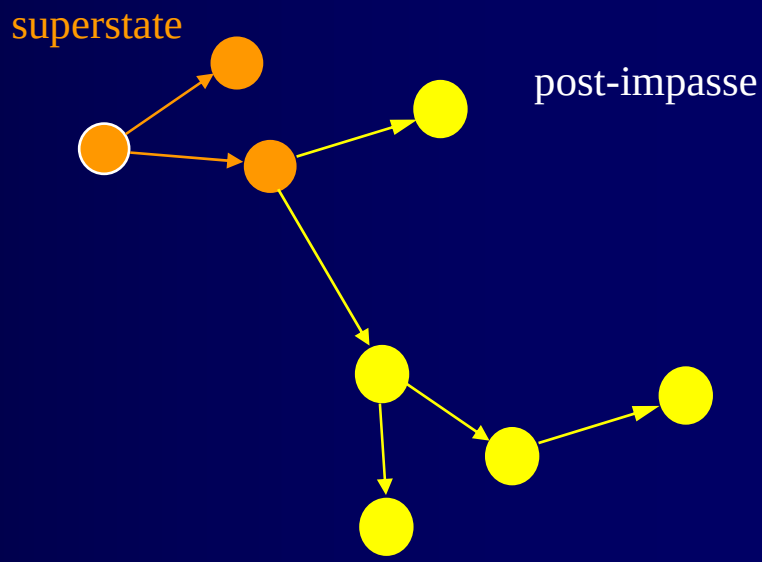
- Problem
  - What are the results of substates/subgoals?
  - Don’t want to have programmer determine via special syntax
  - Results should be side-effect of processing
- Approach
  - Results determined by structure of working memory
  - Structure is maintained based on connectivity to state stack
  - Result is
    - Structure connected to superstate but created by rule that tests substate structure
    - Structure created in substate that becomes connected to superstate
- Implications
  - Results do not necessarily resolve impasses
  - One result can cause large substate structure to become result
  - Superstate cannot be augmented with substate – substate would be result

Result Examples





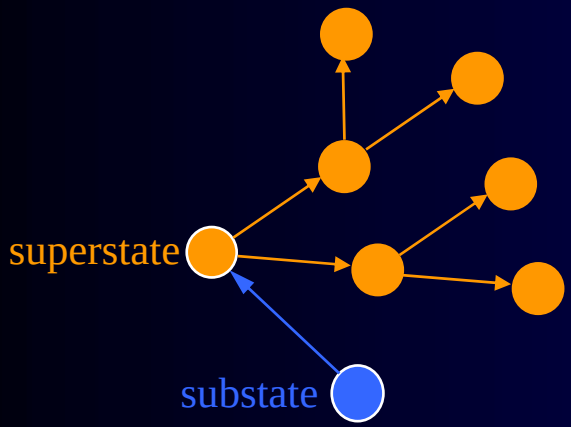


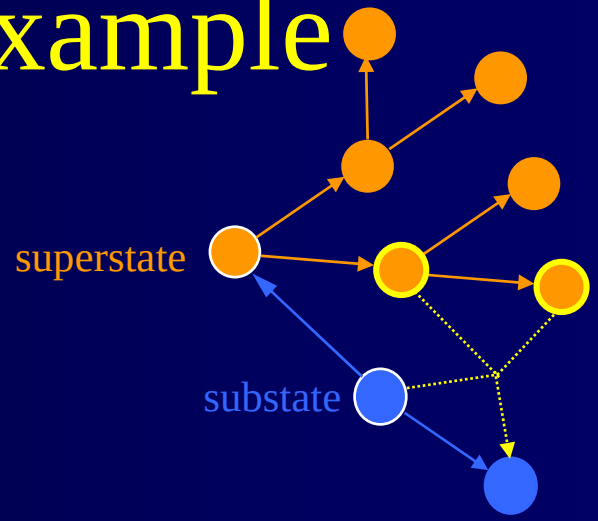


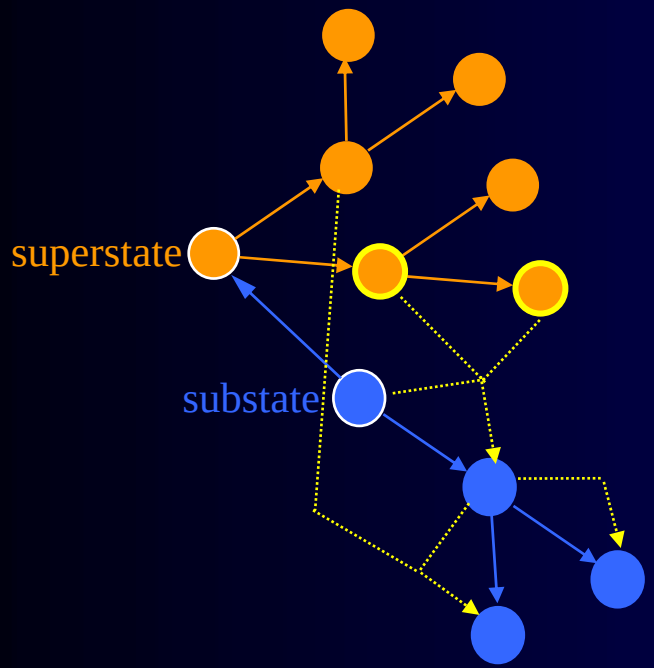
Persistence of Results

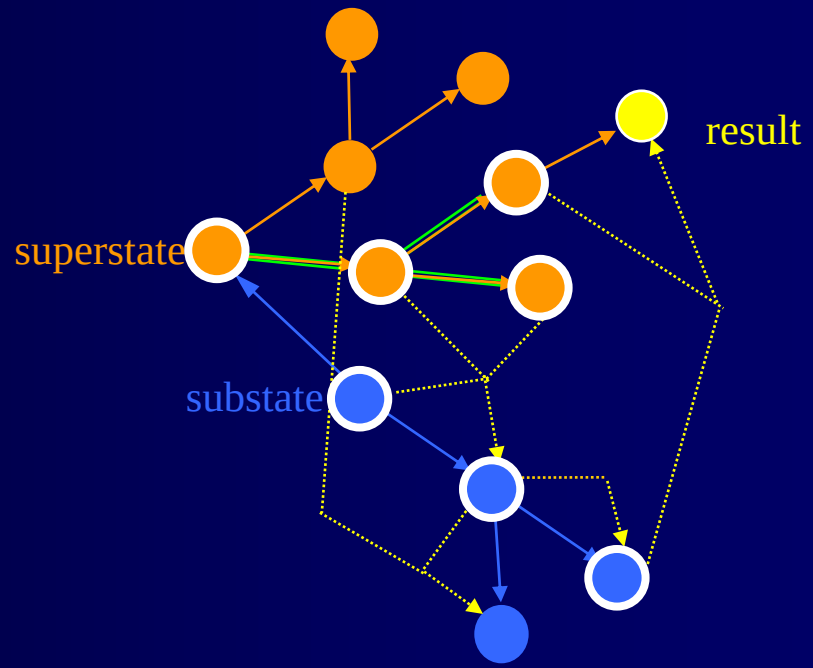
- Problem:
  - What should be the persistence of results?
  - Based on persistence of structure in subgoal?
  - Could have different persistence before and after chunking
    - Operator in subgoal could create elaboration of superstate
  - How maintain i-support after substate removed?
- Approach:
  - Build justification that captures processing
  - Analyze justification
    - Elaborate, propose, select, apply
    - Assign o/i-support
  - Maintain justification for i-support until result removed

Justification Example



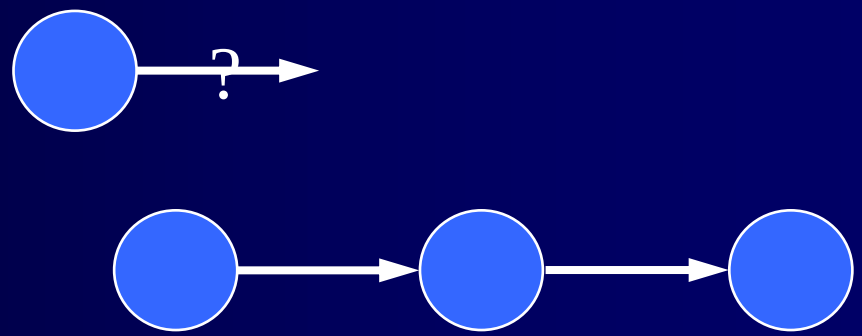






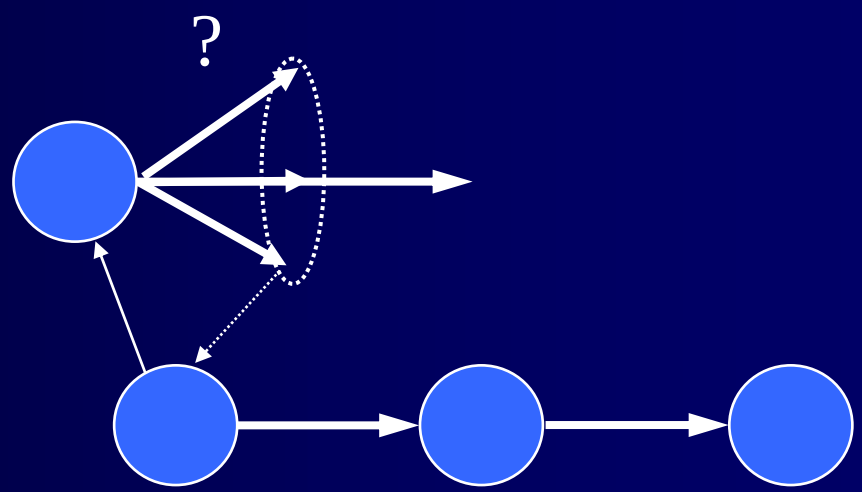
Example Substates: State no-change

- Reason for impasse
  - State is not appropriately elaborated
  - Operator not proposed
- Types of problem solving in substate
  - Analyze superstate structure to find some missing patterns that will stimulate existing operator proposal
  - Analyze superstate to determine which operators are legal to apply.
  - Generate operators.
- Results
  - State elaborations
  - Operator proposals



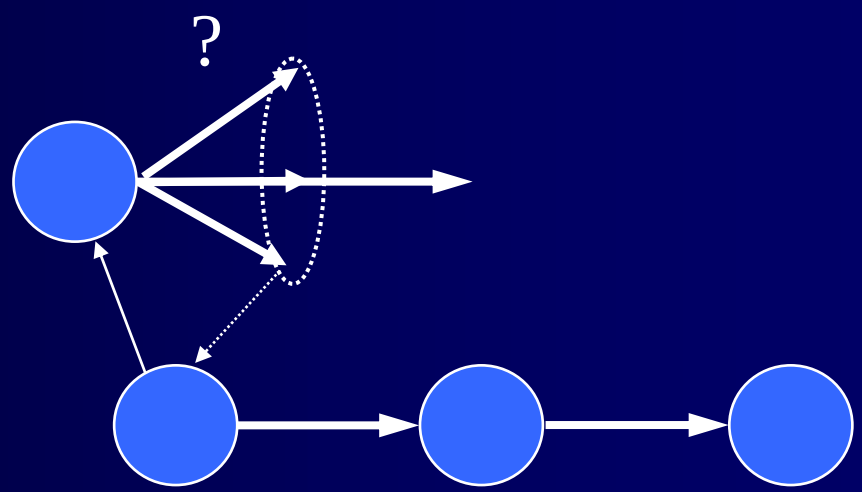
Example Substates: Operator Tie

- Reason for impasse
  - Insufficient preferences
- Types of problem solving in substate
  - Analyze superstate structure and proposed operators to generate additional preferences
  - Selection problem space
  - Meta-reasoning
- Results
  - Operator preferences
  - State elaborations



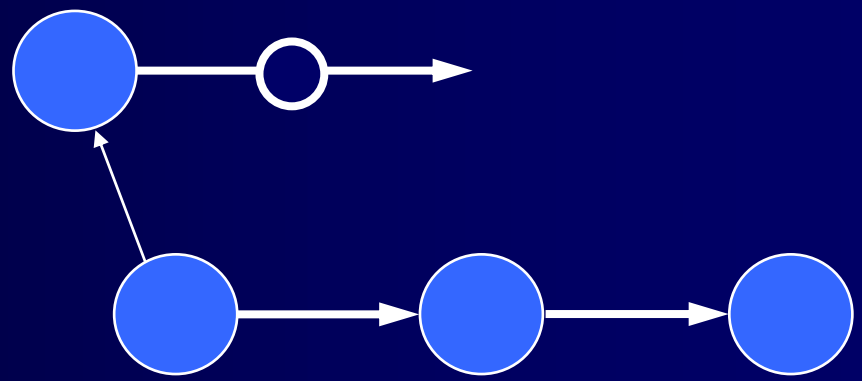
Example Substates: Operator Conflict

- Reason for impasse
  - Conflicting preferences
- Types of problem solving in substate
  - Analyze superstate structure and proposed operators to generate additional preferences
  - Selection problem space
  - Meta-reasoning
- Results
  - Operator reject preferences
- Non-standard
  - Elaborate state

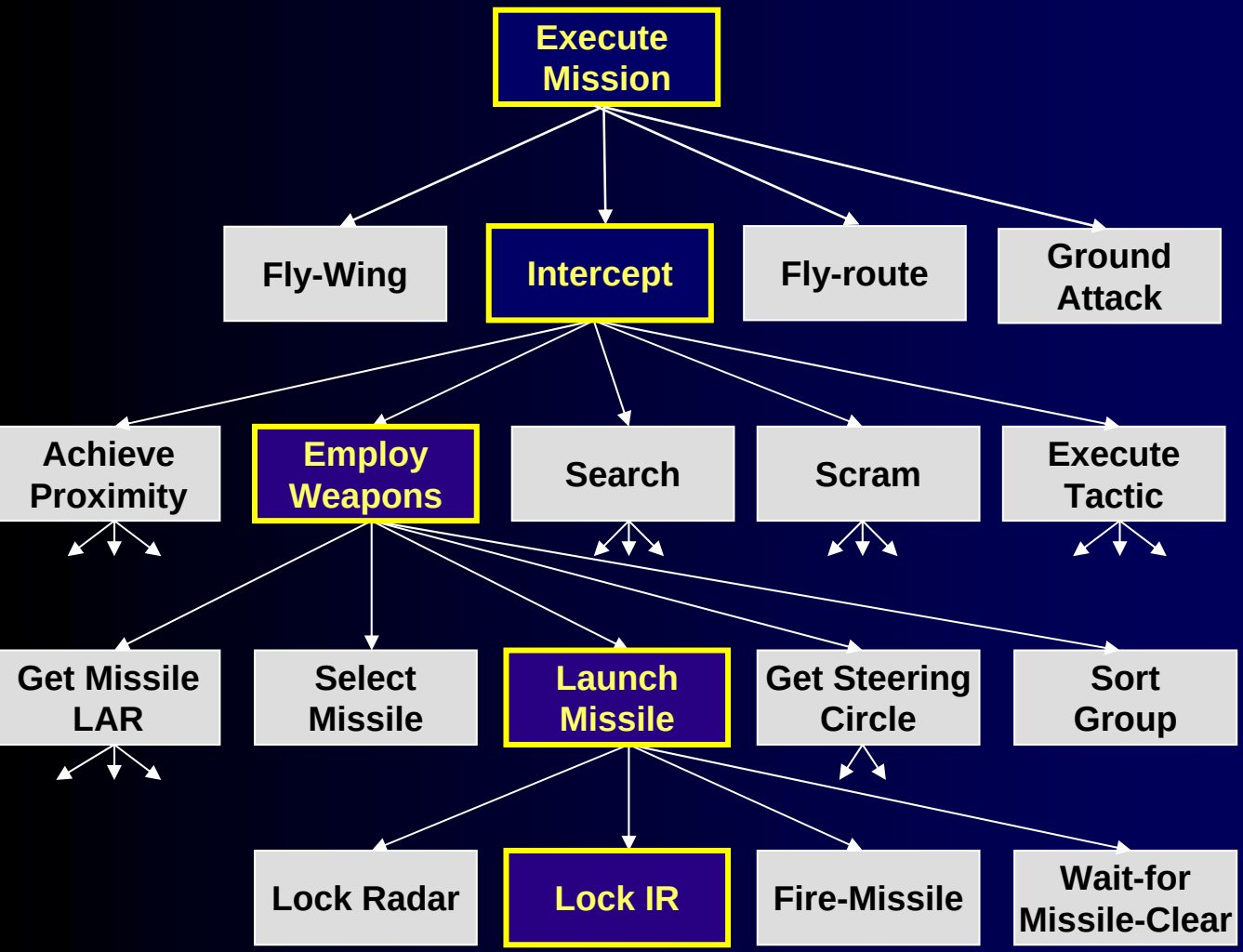


Example Substates: Operator No-change

- Reason for impasse
  - Insufficient knowledge to apply operator
- Types of problem solving in substate
  - Apply operator bit by bit
  - Task decomposition
- Results
  - O-supported state changes
- Non-standard
  - Select another operator



Soar 102:
Dynamic Task Decomposition



If instructed to intercept an enemy then propose intercept

If intercepting an enemy and the enemy is within range ROE are met then propose employ-weapons

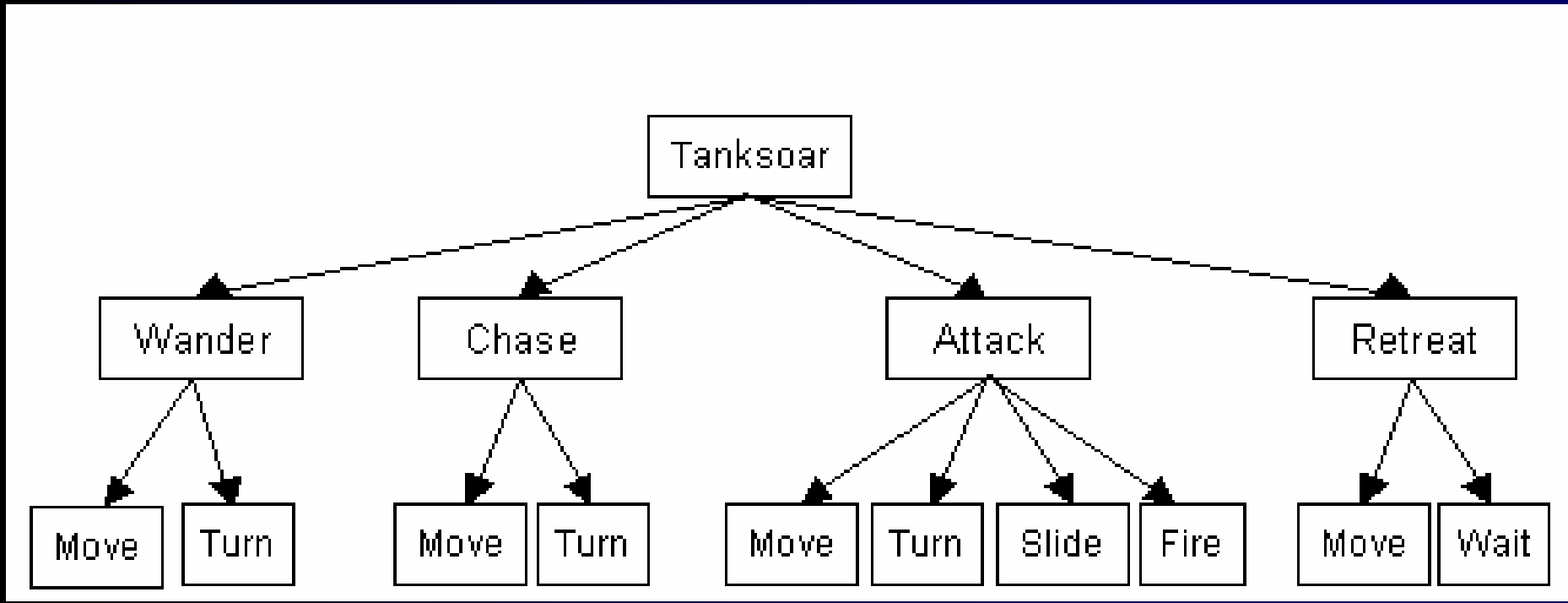
If employing-weapons and missile has been selected and the enemy is in the steering circle and LAR has been achieved, then propose launch-missile

If launching a missile and it is an IR missile and there is currently no IR lock then propose lock-IR

>250 goals, >600 operators, >8000 rules

TankSoar Hierarchy

The Soar Tutorial's full Hierarchy for TankSoar:



Soar 103: Subgoals

Input | Propose Operator | Compare Operators | Select Operator | Apply Operator | Output

```
If enemy not
sensed, then wander
```

Wander

Move

Soar 103: Subgoals

Input | Propose Operator | Compare Operators | Select Operator | Apply Operator | Output

```
If enemy is sensed,
then attack
```

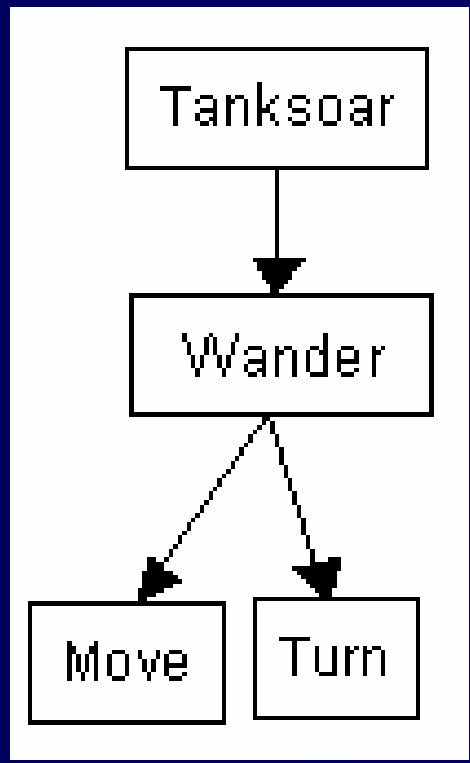
Attack

Shoot

Exercise #3

Let’s start simple…

1. Elaborate the top state with the name ‘tanksoar’.
2. Wander: If you can’t see a tank on the radar, propose wander
3. Use VisualSoar to drag and drop the existing move and turn operators as sub-operators of Wander.
4. Modify the move and turn proposal rules to fire only if the current state is named ‘wander’.



Default Rules to Support Subgoals

- Copy down the ^io pointer from ^superstate.io and ^top-state pointer to every substate.
- Name each substate with superoperator name.
  - NOTE: This rule is provided by default by VisualSoar

Elaborations

```
sp {elaborate*task*tanksoar
   (state <s> ^superstate nil)
-->
   (<s> ^name tanksoar)}
```

```
sp {elaborate*state*superstateio
   (state <s> ^superstate.io <io>)
-->
   (<s> ^io <io>)}
```

Propose Wander

```
sp {propose*wander
   (state <s> ^name tanksoar
              ^io.input-link <io>)
   (<io> ^sound silent
         -^radar.tank
         -^incoming.<dir> yes)
-->
   (<s> ^operator <o> +)
   (<o> ^name wander)
}
```

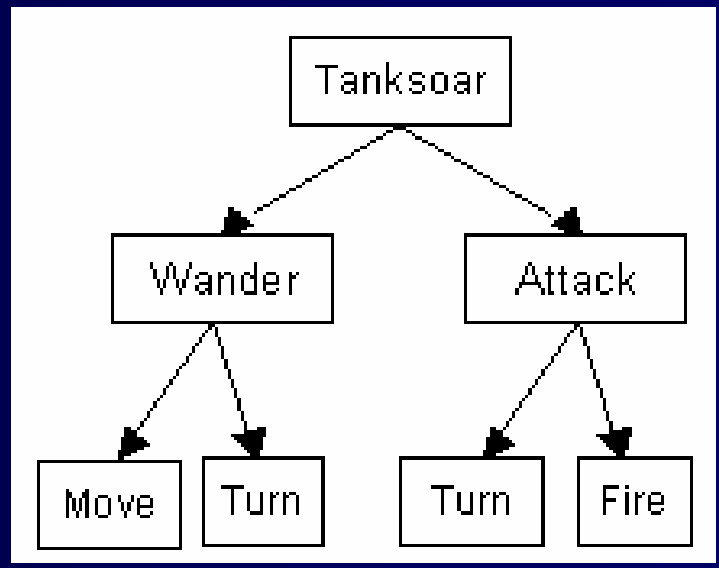
Revised Move

```
sp {wander*propose*move
   (state <s> ^name wander
              ^io.input-link.blocked.forward no)
-->
   (<s> ^operator <o> + =)
   (<o> ^name move
        ^actions.move.direction forward)}
```

Attack!

Add the attack operator:

1. Propose an Attack operator when you can see a tank on the radar
2. Create two operators for the attack subgoal



- Fire: If I see a tank on radar ahead of me in the center, fire a missile.
- Turn: If there is a tank next to me, turn and fire.

Fire Missile

```
sp {attack*propose*fire-missile
   (state <s> ^name attack
              ^io.input-link <io>)
   (<io> ^radar.tank.position center
         ^missiles > 0)
-->
   (<s> ^operator <o> + >)
   (<o> ^name fire-missile
        ^actions.fire.weapon missile)}
```

- Note: The rule must test the number of missiles, otherwise it will not retract after the operator is applied.

“State of the Art”: Simple Tank

- Expanded Attack
- Chase
- Retreat
- Shield Control Rules
- Wait operator (next slide)
- Improved Sound Detection

Wait

- Prevents multiple state no changes

```
sp {propose*wait
   (state <s> ^attribute state
              ^choices none
             -^operator.name wait)
-->
   (<s> ^operator <o> + <)
   (<o> ^name wait)}
```

Wait Operator

- How it works
  - Detects a state no change (via ^choices none)
  - Proposes wait operator **only** if one is not selected
  - Wait operator is selected
  - Proposal rule no longer matches and is retracted before application

Performance Issues

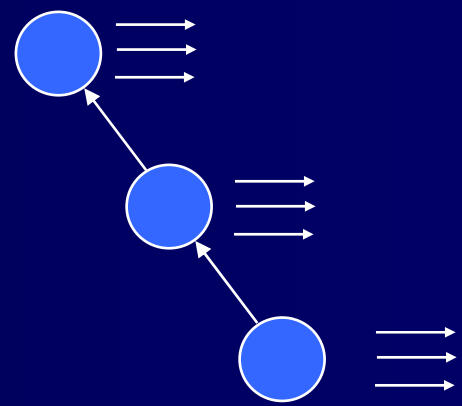
- How to find and correct
- Memories
  - Anything greater than 500 should be looked at seriously
- Firing-counts (fc)
  - What can you do to remove the top rule?
  - Negation?

Soar Summary

- AI engine to support multi-method problem solving
  - Applied to wide variety of tasks and methods
  - Combines reactive, deliberative, reflective, and learning
- Meta-level reasoning through preference-based decisions & subgoals
  - All decisions open to knowledge-based control or deliberate problem solving
  - Can always add more knowledge to refine decisions
- Proposed unified theory for modeling human cognition
  - Natural language understanding and generation, HCI tasks, simple puzzles, syllogistic reasoning, new task acquisition, concept acquisition, video game playing, software debugging, robotic control, learning by instruction, learning by experience, correcting incorrect knowledge, integration of many capabilities together for a single task, …
- Supports very large bodies of knowledge
  - >100,000 rules
- Optimized implementation in ANSI C
- In the public domain

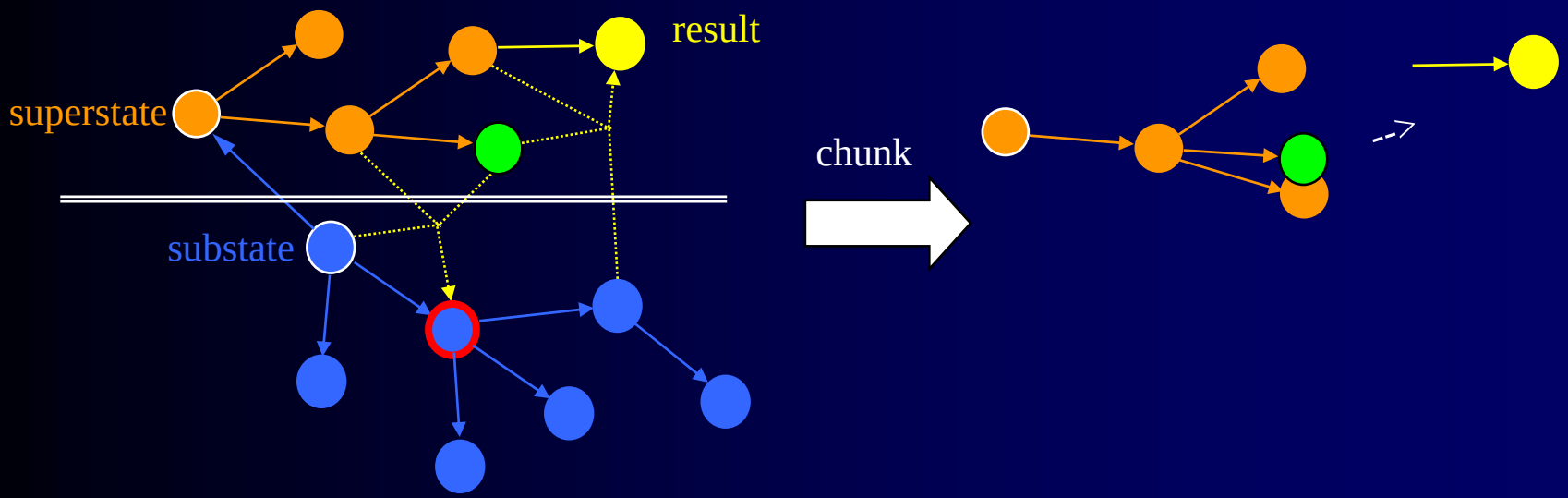
Processing Across Substates

- Problem:
  - Rules can fire in substates even though impasse is about to be resolved
  - Run-away substate can generate results that are invalid
- Approach
  - Cascade rules from oldest to newest substate
  - Remove substates if impasse is resolved
  - Recompute match set after each substate processed
- Implications
  - Avoids firing rules that will be irrelevant
  - Avoids some race conditions



Persistence of Substate Structures: Problem

- O-supported structure in subgoals can become *inconsistent*
  - Future behavior is no longer reactive to changes in the context
    - Non-reentrant – results would be different if rentered subgoal
  - Chunks have conditions that can never match
    - Test mutually exclusive values of same attribute
    - Non-contemporaneous



Analysis

- Whenever the substate WMEs cannot be recreated from superstate WMEs using existing rules.
- Occurs from changes to input and returning results.
- Only a problem for o-supported structures and their entailments
  - Not a problem for i-supported structures

Possible Approach

- Remove any substate WME that becomes inconsistent
  - One detail of Soar makes this very nasty
    - WMEs don't "blip" when there is a change in i-support
    - If an i-supported WME loses support, but at exact same time, same WME is created with new i-support, WME doesn't changes

```
(<s> ^sensor-a < 20) --> (<s> ^enemy near)
```

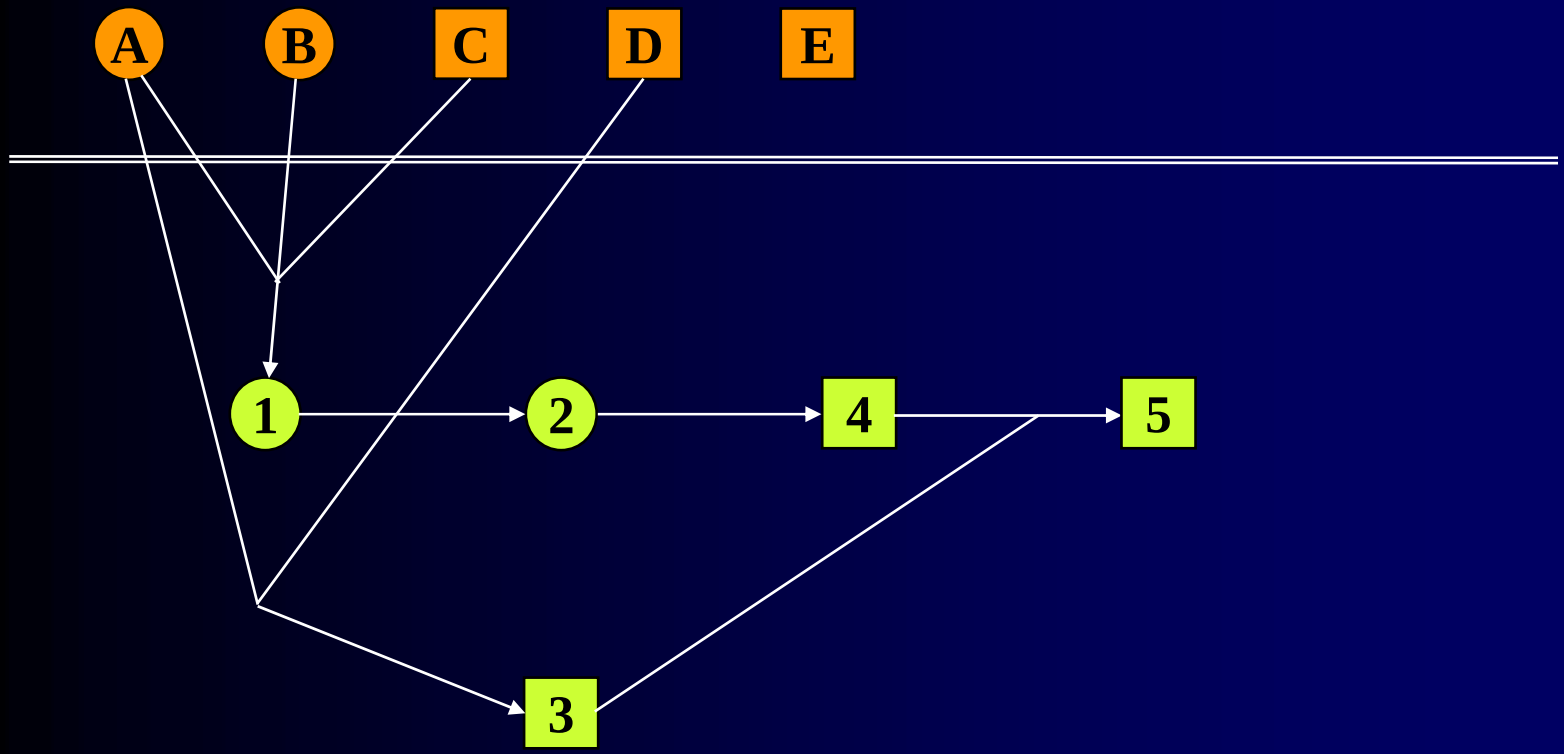
  - Can't maintain derivation information with every WME
    - Because it can change
  - Must dynamically compute derivation information
- Very expensive to maintain and compute

Approach

- A substate is regenerated whenever higher state WMEs become *inconsistent* with substate's internal processing
- Regenerated = all substate structure removed from WM and new substate created.
- Each substate maintains a *goal dependency set (GDS)*
  - All superstate WMEs tested in creating o-supported WMEs in substate
- If anything changes in GDS, substate is regenerated.

GDS Example

superstate



substate

GDS= [] GDS= [A,D] GDS= [A,B,C,D] GDS= [A,B,C,D]

= i-support

= o-support

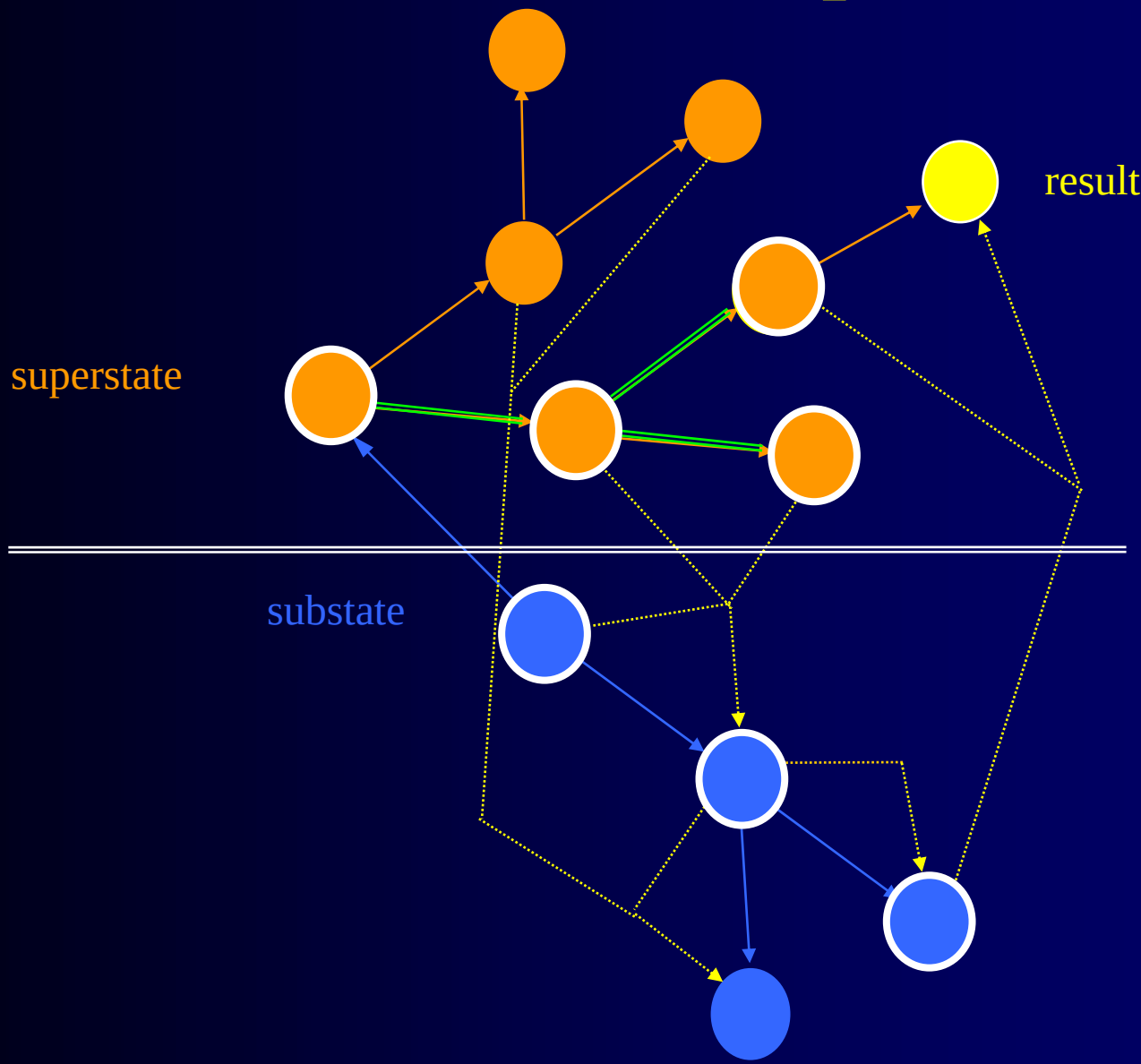
Implications

- Only an issue for o-supported structures in substates.
- Can’t create o-supported structures based on changing sensors.
  - Can’t create counters of external events in substates
- O-supported structures in substates are steps in that problem space.
  - Look-ahead search
- Can avoid regeneration by maintaining “fragile” o-support structure on top-state.

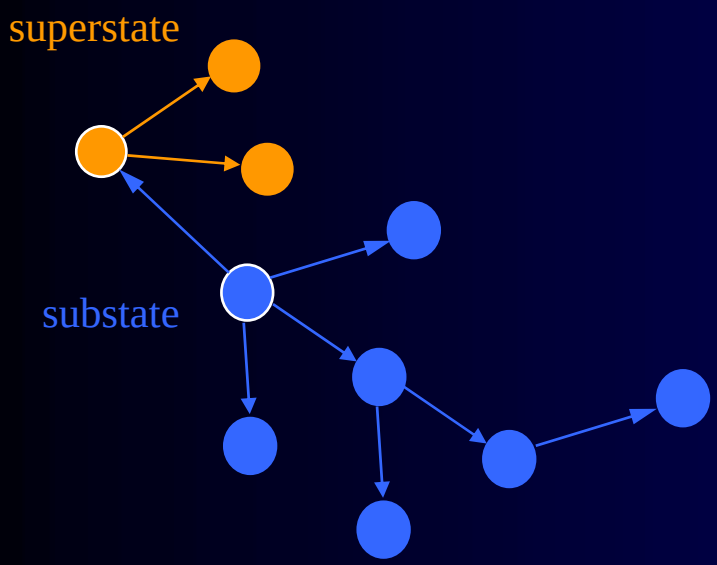
Learning/Chunking

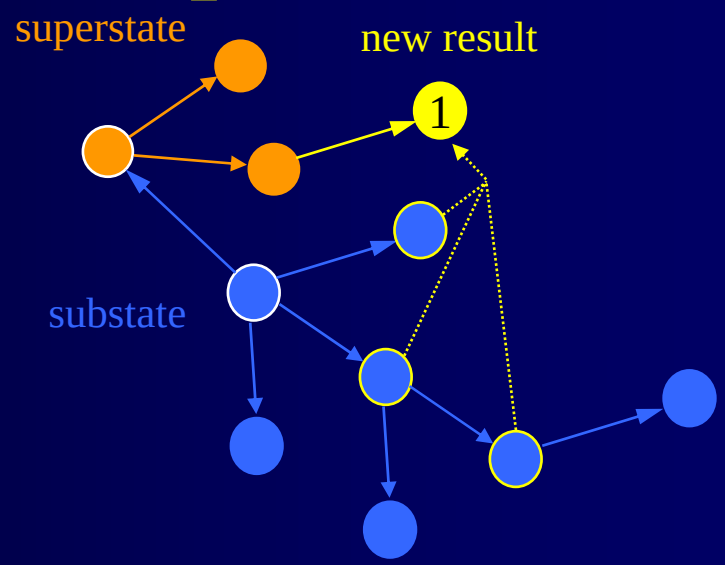
- Problem:
  - Subgoals “discover” knowledge to resolve impasses but it is lost after each problem solving episode
- Approach
  - Automatically build rules that summarize processing
    - Variablize justifications = chunks
    - Variablizes only identifiers – no constants
    - Loses >, <, … tests between constants
    - Conditions include only those test required to produce result = implicit generalization
  - Chunks are built as soon as a result is produced
    - Immediate transfer is possible
  - One chunk for each result, where a result consists of connected WMEs that become results at the same time
    - Different results can lead to very different conditions
    - Improves generality of chunks

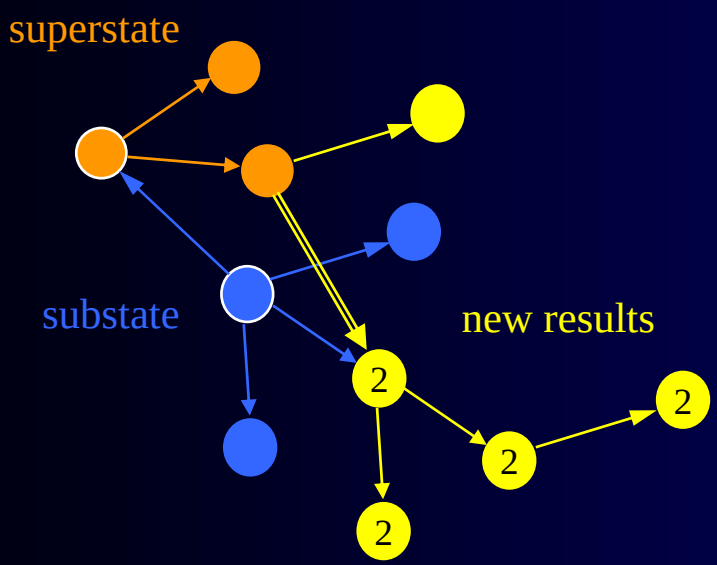
Chunk Example



Action Examples







Key Feature about Chunking

- Chunk over problem solving necessary to produce result
  - Search control should affect efficiency of problem solving, not correctness
  - Do not include search control in analysis for chunks
    - Search control = non-acceptable preferences
      - Except require and prohibit
- Implications
  - Should not use search control to avoid invalid states
  - Should incorporate goal tests in search control
  - Goal tests can be in preconditions of proposals for operators

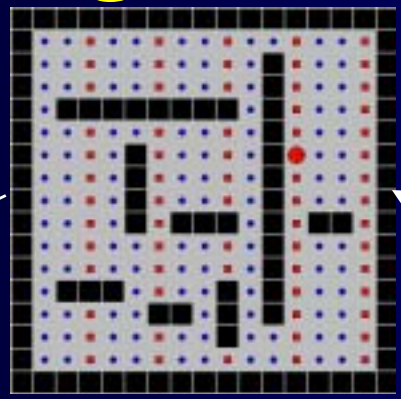
Testing for Impasse

- If problem solving result depends on the fact that there is an impasse, then should not chunk.
- All substates have ^quiescence true
- If tested on path to result, no chunk will be built.
- A bit of a hack for disabling chunking.

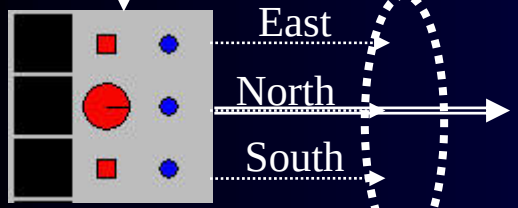
Chunking Analysis

- Converts deliberate reasoning/planning to reaction
- Generality of learning based on generality of reasoning
  - Leads to many different types learning
  - If reasoning is inductive, so is learning
- Soar only learns what it thinks about
- All learning is impasse driven
  - Learning arises from a lack of knowledge

Soar 104: Subgoals and Chunking



Input | Propose Operator | Compare Operators | Select Operator | Apply Operator | Output



Tie Impasse

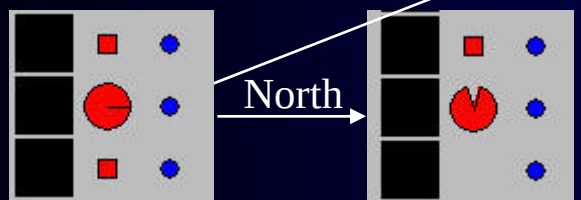
North > East
South > East
North = South

Chunking creates rules that create preferences based on what was tested

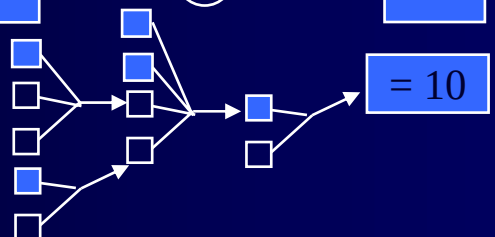
Evaluate-operator (North) = 10

Evaluate-operator (South) = 10

Evaluate-operator (East) = 5



= 10



Chunking creates rule that applies evaluate-operator

Learning Results