The Selection Space

John E. Laird
32nd Soar Workshop
Overview One-step Look-ahead
Using Selection Problem Space
Selection Space

• Important state structures created by Soar
  – ^impasse tie, ^item 01 02 ...

• Evaluate-operator
  1. Instantiated with every item (every tied operator) that has not been evaluated
     (<s> ^operator <o>)
     (<o> ^name evaluate-operator
        ^superoperator <so>)
  2. Usually randomly select between them (some exceptions)
  3. Create ^evaluation structure on selection state
Evaluate State Structure

• When evaluate-operator is selected, create:
  – (<s> ^evaluation <e>)
  – (<e> ^superoperator <i>)
  – (<o> ^evaluation <e>        # on evaluate-operator
  – ^superstate <ss>       # task state
  – ^superproblem space <ps>)

• Evaluate-operator terminates when a value is created on the associate evaluation
  – (<e> ^value true)
Evaluate-operator Substate

• Create a *copy* of the task state
  – Includes ^name, ^desired
  – ^problem-space determines how to create copy
    • Many flags to control what to copy and how deep
    • ^default-state-copy yes is default

• If don’t create copy, original state will change
Evaluate-operator Processing

1. Force selection of a copy of the operator being evaluated
2. Operator application rule should fire and generate new state
   – Requires *action model*: operator application rule for simulating operator
   – If doesn’t, will eventually get impasses that lead to a failed evaluation.
3. If there is state evaluation knowledge, it adds augmentation to state
   – \(^\text{numeric-value}\), \(^\text{symbolic-value}\), \(^\text{expected-value}\)
   – Copied up to the evaluation structure in the selection space
   – Leads to evaluate-operator terminating
   • By default, elaboration rules aggressively convert evaluations to preferences.
     – Evaluates only as many operators as necessary to generate preferences to break the tie.
   • Chunks are learned for computing evaluations and preferences
Overview One-step Look-ahead Using Selection Problem Space

- move(C, B)
- move(C, Table)
- move(B, C)

Evaluate-operator(move(C, Table))

Evaluation = 1

Evaluation = 0

Evaluation = 0

Evaluation = 1
Requirements to Use Selection Space

• Source in selection.soar!
  – Explains the following requirements
• Have a $^\text{problem-space}$ structure on the state
• Have a $^\text{desired}$ structure on the state
• Include rules that compute failure/success/evaluation.
• Have rules that simulate action of operators
  – This is an action model
  – Only apply when in state with
    $^\text{name evaluate-operator}$
Depth-First Search in Soar

- If no evaluation of the state, continues in substate
  - If sufficient knowledge, selects and applies operator
  - If insufficient knowledge, get a tie impasse and recursively get depth-first search.
- The state “open” list is represented as the stack of substates.
- Elaboration rules pass success up the stack to avoid extra search.
- No guarantee of finding shortest path.
- Chunking is necessary to avoid repeated search.
Overview One-step Look-ahead Using Selection Problem Space

- move(C, B)
- move(C, Table)
- move(B, C)

Evaluate-operator(move(C, Table))

Tie Impasse

copy

Goal

move(C, Table)

Move(C, Table)

Tie Impasse

move(C, B)
move(A, B)
move(B, C)
Iterative Deepening

• Include an evaluation-depth in the selection space
• Evaluate all of the task operators to that depth
  – Start with depth = 1
  – In each recursive selection substate, decrement depth
• Terminate if achieve goal
• Increment depth when all task operators have been evaluated
Deep Search in Soar: Iterative A*

- Assumes task state structure
  - Graph structure of waypoints, with a current-location
- Every evaluation maintains
  - Path-cost: $g(x)$
  - Estimated-cost: $h(x)$
  - Total-estimated-cost: $f(x) = g(x) + h(x)$
- Prefer an evaluate-operator to another
  - If it doesn’t have an estimated-cost # get initial values
  - If its total-estimated-cost is less than the others # pursue best
- Final-cost for an operator is when estimated cost is 0
- Create a preference if final-cost($o_1$) < total-estimated-cost($o_2$)
- Complex rules and operators combine estimates from substates
  - Add operators: compute-evaluations, compare-evaluations, compute-best-total-estimate
2: Path: 1.4; Estimated: 2.3; Total 3.7
2: Path: 1.4; Estimated: 2.3; Total 3.7
3: Path: 1; Estimated: 1.4; Total 2.4
2: Path: 1.4; Estimated: 2.3; Total 3.7
3: Path: 1; Estimated: 1.4; Total 2.4
4: Path: 1.4; Estimated: 1.0; Total 2.4
2: Path: 1.4; Estimated: 2.3; Total 3.7
3: Path: 1; Estimated: 1.4; Total 2.4
4: Path: 1.4; Estimated: 1.0; Total 2.4
4: Path: 2.8; Estimated: 1.0; Total 3.8
2: Path: 1.4; Estimated: 2.3; Total 3.7
3: Path: 1; Estimated: 1.4; Total 2.4
4: Path: 2.8; Estimated: 1.0; Total 3.8
3: Path: 3.3; Estimated: 1.0; Total 4.3
1: Path: 1.4; Estimated: 2.3; Total 3.7

2: Path: 3.7; Estimated: 0.0; Final: 3.7

3: Path: 3.3; Estimated: 1.0; Total 4.3

4: Path: 2.8; Estimated: 1.0; Total 3.8
Nuggets and Coal

• Nuggets:
  – Provides task-independent knowledge for controlling deliberate operator evaluation
  – Plays well with chunking

• Coal
  – Requires some knowledge of conventions
  – More advanced methods are pretty complex