stilpo Documentation

Release 0.1.0

Tuukka Turto

Apr 19, 2018

Contents

Contents:

CHAPTER 1

Classic Problem Solver

1.1 API

`depth-first-solver` creates depth first solver.

```
(depth-first-solver :is-goal goal?
                    :operators operators
                    :is-identical identical?)
```

`breadth-first-solver` creates breadth first solver.

```
(breadth-first-solver :is-goal goal?
                      :operators operators
                      :is-identical identical?)
```

`best-first-solver` creates best first solver.

```
(best-first-solver :is-goal goal?
                   :operators operators
                   :is-identical identical?
                   :distance distance-left)
```

`a*-solver` creates a solver that implements a* search.

```
(a*-solver :is-goal goal?
           :distance distance-left
           :distance-between distance-between
           :operators operators
           :is-identical identical?))
```

`goal?` is a function that accepts a single parameter `state` and returns `True` or `False` indicating if the goal has been reached.

`operators` is a function that accepts a single parameter and returns a list of tuples where first element is a function that can create a new state from old one and second element is textual description of the transition.

`identical?` is a function accepting two states and returning `True` or `False` indicating if the states are considered equal. This function is used to detect loops in search path.

`distance` is a function that estimates distance left for given state. It is used to optimize search path in best first and a* search.

`distance-between` is a function that can calculate cost or distance between two different states. It is used to optimize a* search.

1.2 Example

Classic problem solver (CPS for short) is a solver based on simple search and goal detecting routine. Systems starts from an initial state and expands through problem space by trying different operations according to given search criteria.

If we wanted to solve the following problem:

> You have 2 jugs, with capacity of 4 and 5 liters. Your task is to measure 2 liters of water. You have unlimited amount of water in your disposal.

A high level example of typical solver initialization and execution:

```
(require [stilpo.cps [operator]])
(import [stilpo.cps [breadth-first-solver valid-operators]])

(def b-solve (breadth-first-solver :is-goal goal?
                                   :operators operators
                                   :is-identical identical?))
(-> (b-solve initial-state)
    (print-solution))
```

`breadth-first-solver` is a function that will create a solver that uses breadth first search when called. This solver can then be used to solve one or more problems (it doesn't retain state between calls).

We represent state of our problem as a dictionary (any other data structure would work too, stilpo isn't that particular about it). At the beginning, there are two jugs, both empty:

```
(def initial-state {:jug-4 0
                    :jug-5 0})
```

Detecting goal in our case is simple. When ever one of the jugs holds exactly two liters of water, we're done:

```
(defn goal? [state]
  (or (= (:jug-4 state) 2)
      (= (:jug-5 state) 2)))
```

CPS needs to know which operators it can perform to any given state. Operator is just a function that when applied to a state, will return a new state. You are free to structure your code in the way you prefer, but stilpo has an utility functions for building operators and detecting when they can be applied.

`operator` macro is used to define special function that represents an operation that can be done to a `state`:

```
(operator empty-jug-4 "pour 4 liter jug empty"
          (> (:jug-4 state) 0)
          {:jug-4 0
           :jug-5 (:jug-5 state)})
```

First parameter is name of the function being defined, second one is textual description that can be printed out to specify solution to the problem. Third parameter is a form that returns `true` if operator is legal for given state. Rest of the code is used to create a new state that has been modified (4 liter jug poured empty in this example).

Each discrete action is defined as an operator like above and then packed into a function that can check which operators are valid for given state and return their application:

```
(defn operators [state]
  "all valid operators for given state and their descriptions"
  (valid-operators state empty-jug-4 empty-jug-5
                   fill-jug-4 fill-jug-5
                   pour-4-to-5 pour-5-to-4))
```

Final tool we need to define is detection of identical states. This is used by search algorithm to prune possible loops from the solution:

```
(defn identical? [state1 state2]
  (and (= (:jug-4 state1) (:jug-4 state2))
       (= (:jug-5 state1) (:jug-5 state2))))
```

We of course would like to print out our solution, so we define `pretty-print` to do that task for us:

```
(require [hy.extra.anaphoric [ap-each]])

(defn pretty-print [path]
  (when path
    (ap-each path
             (cond [(in :action it)
                    (print (.format "{0} (jugs: {1} and {2})"
                                    (:desc (:action it))
                                    (:jug-4 (:state it))
                                    (:jug-5 (:state it))))]
                   [true (print "starting")]))))
```

Function simple walks the path and prints out textual info of action taken and amount of water held by each jug:

```
starting
fill 4 liter jug with water (jugs: 4 and 0)
pour water from 4 liter jug to 5 liter jug (jugs: 0 and 4)
fill 4 liter jug with water (jugs: 4 and 4)
pour water from 4 liter jug to 5 liter jug (jugs: 3 and 5)
pour 5 liter jug empty (jugs: 3 and 0)
pour water from 4 liter jug to 5 liter jug (jugs: 0 and 3)
fill 4 liter jug with water (jugs: 4 and 3)
pour water from 4 liter jug to 5 liter jug (jugs: 2 and 5)
```

CHAPTER 2

Tiny Rule Engine

2.1 API

`create-tre` creates a tiny rule engine

```
(setv tre (create-tre "Example" :debug False))
```

`assert!` asserts a fact into a given rule engine

```
(assert! tre (hy is lisp))
```

`rule` creates a new rule and inserts it into a rule engine

```
(rule tre (?x is lisp)
      (assert! tre (?x is awesome)))
```

Sometimes you would want to force variable values be unique:

```
(rule tre (?x is on bottom of well)
      (rule (?y is on bottom of well)
            (unique ?x ?y)
            (assert! tre (?x and ?y are on bottom of well))))
```

`run` execute tiny rule engine until all rules and assertions are processed

```
(run tre)
```

`show` show assertions associated with given symbol

```
(show tre 'hy)
```

`true?` check if given assertion holds

```
=> (true? tre '(hy is awesome))
True
```

`push-tre` creates a new context where to try things

```
=> (rule tre (?x uses flux-capacitor)
         (assert! tre (?x is from future)))

=> (push-tre tre "Assuming Hy is using flux-capacitor")
=> (assert! tre (hy uses flux-capacitor))
=> (run tre)
=> (true? tre '(hy is from future))
True
```

`frame-title` retrieves name of current frame

```
=> (frame-title tre)
"Assyming Hy is using flux-capacitor"
```

`pop-tre` discards the most recent frame

```
=> (pop-tre tre)
=> (true? tre '(hy is from future))
False
```

`try-in-context` is useful for creating a context and trying out a thing inside of it, before discarding the context automatically.

```
=> (rule tre (?x uses Python)
         (assert! tre (?x is modern system)))
=> (rule tre (?x uses Lisp)
         (assert! tre (?x is timeless system)))
=> (rule tre (?x uses quantum computing)
         (assert! tre (?x is future system)))

=> (try-in-context tre (hy uses quantum computing)
                   (print (true? '(hy is future system))))
True

=> (true? '(hy is future system))
False
```

2.2 Example

Tiny rule engine is pattern directed inference system that operates on symbols and patterns. Essentially, it deduces new assertions based on existing assertions and rules.

For example, we can deduct family relations:

First step is to initialize tiny rule engine and bind a symbol to it:

```
(setv tre (create-tre "family"))
```

Assertions (true statements) are created with `assert!`. Once a truth has been asserted, there is no way to remove it. This is because doing so would have to remove all rules and assertions that it might have created and their results and

so on. Keeping track of web of assertions and rules would have been rather complicated and error prone system, so it was left out.

```
(assert! tre (Alice is parent of Bob))
(assert! tre (Bob is parent of Charlie))
```

Rules are used to create new assertion and rules based on existing ones. They consist of a pattern and body. When tiny rule engine executes a rule, it processed through all assertions, checking if any of them match the pattern. When a match is found, body of the rule is executed. Special notation is used to introduce free variables in the pattern that can then be used in the body:

```
(rule tre (?x is parent of ?y)
      (assert! tre (?y is children of ?x)))

(rule tre (?x is parent of ?y)
      (rule tre (?y is parent of ?z)
            (assert! tre (?x is grand-parent of ?z))))

(rule tre (?x is grand-parent of ?y)
      (assert! tre (?y is grand-children of ?x)))
```

Final step in our example is to execute the engine and review the results, which should show that Alice indeed is grand parent of Charlie:

```
=> (run tre)
=> (show tre 'Alice)
Alice is parent of Bob
Alice is grand-parent of Charlie
Charlie is grand-children of Alice
Bob is children of Alice

=> (true? tre '(Alice is grand-parent of Charlie))
True
```

The order of adding rules and assertions into tiny rule engine doesn't matter. Engine will keep processing rules until no further changes occur in assertions. It is even possible to run tiny rule engine in REPL, working with rules and assertions step by step.

CHAPTER 3

Justification Based Truth Maintenance System

Justification Based Truth Maintenance System or JTMS for short

3.1 API

`create-jtms` create instance of JTMS

`create-node` create node in TMS

`assume-node` make node an assumption and enable it

`enable-assumption` turn assumption `'in`

`retract-assumption` turn assumption `'out`

`justify-node` justify a node with justification

`in-node?` check if node is `'in`

`out-node?` check if node is `'out`

`assumptions-of-node` list all assumptions and premises having effect to this node

`supporting-justification-for-node` which justification is currently supporting a node

CHAPTER 4

Tools

Tools module contain assortment of tools that are generally useful in writing problem solvers.

4.1 unify

`unify` tries to unify an assertion with a pattern, while capturing joker and wildcard values. If unification is successful, a dictionary containing captured variables is returned. If unification failed, `None` is returned instead.

Interface `(unify [assertion pattern bindings unique-symbols])`

- `assertion` s-expression
- `pattern` pattern as s-expression, may contain `?x` and `*x`
- `bindings` dictionary containing predefined values for variables
- `unique-symbols` tuple of tuples defining variables that should be unique

```
=> (unify '(Hello World) '(Hi there, ?x) {} (,))
None

=> (unify '(Hello World) '(Hello ?x) {} (,))
{'?x 'World}

=> (unify '(Hello Charlie Brown) '(Hello *x) {} (,))
{'*x '(Charlie Brown)}

=> (unify '(foo bar baz) '(?x bar ?y) {} (,))
{'?x 'foo '?y baz}

=> (unify '(foo bar foo) '(?x bar ?y) {} (,))
{'?x 'foo '?y foo}

=> (unify '(foo bar foo) '(?x bar ?y) {} (, (, '?x '?y)))
None
```

4.2 fill-assertion

`fill-assertion` replaces variables in assertion with their respective values from supplied bindings.

Interface `(fill-assertion [assertion bindings])`

- `assertion` s-expression containing ?x and *x variables
- `bindings` dictionary containing variables and their values

```
=> (fill-assertion '(Hello ?x) {'?x 'World})
'(Hello World)
```

CHAPTER 5

License

CHAPTER 6

What is stilpo?

Stilpo is a generic ai library for Hy, although at least parts of it should be usable by Python programs too. It is based on ideas explained in excellent Building Problem Solvers by Kenneth D. Forbus and Johan de Kleer and Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp by Peter Norvig.

Stilpo was also a greek philosopher, who was interested in logic and dialetic.

CHAPTER 7

Indices and tables

- genindex
- search