Clojure Tools for Practical Artificial Intelligence

Simon Lynch[1,*] **and Saul Johnson**[1,**]

[1]University of Teesside, Middlesbrough, UK
[*]s.c.lynch@tees.ac.uk
[**]saul.johnson@tees.ac.uk

ABSTRACT

Approaching problems around planning and inference in artificial intelligence can be highly demanding for developers who have never worked in these problem domains before. In this workshop paper, we aim to provide a foundation of techniques and tools to make construction of software systems that utilize planning and inference mechanisms more accessible to developers working in the Clojure programming language. In particular, we investigate how Clojure (in conjunction with a symbolic pattern matching library) can be used to build some key inference engines used in Artificial Intelligence.

1 Introduction

A common theme in Artificial Intelligence is to find a path from some starting position (the start state) to some desired outcome (the goal state). This could be, for example, to reorganize a random configuration of objects into organized piles or to efficiently distribute materials around a warehouse.

The nature of state and the way it is represented depends on the specifics of the problem. For example, if we just need to find a way to get some animated agent from one room to another in a house, we might simply represent the state as:

- the agent's current location
- which rooms connect to which others

This state contains all the information we need to form a plan: where is the agent right now and which rooms must it pass through to get to the desired location? This is a simple case, but in a complex world in which an agent may take multiple actions (each of which may have consequences) the state representation can grow to be necessarily much more complex.

2 Legal Move Generators for a Breadth-First Search

One of the simpler methods for finding a path from a start state to a goal state is to use a breadth-first search. The breadth-first search function used in this paper is publicly available[1]. This search takes 3 arguments: a start state, a goal state and a transformation function which is often called a *legal move generator* (LMG).

An LMG is a simple function which takes a state as its only argument and returns a sequence of *successor states* - states which can be reached by transforming the current state by applying a single operation to it.

2.1 A Maths Problem

Consider a problem in which we need to work out how to get from one number to another using only a small number of simple mathematical operations. For example, we might want to get from $n = 5$ to $n = 35$ by applying only the following operations zero or more times:

- $n = n + 3$
- $n = n \times 2$
- $n = n - 7$

We can trivially write an LMG that captures this set of rules (see Figure 1).

```
(defn lmg [n]
  [(+ n 3) (* n 2) (- n 7)])

user=> (breadth-search 5 35 lmg) ; Get from 5 to 35 using our LMG.
(5 8 16 32 35)

user=> (time (breadth-search 5 1357 lmg)) ; Time how long to get from 5 to 1357.
"Elapsed time: 847.73328 msecs"
(5 8 11 22 44 88 176 169 338 341 682 1364 1357)
```

Figure 1. A very basic legal move generator for our puzzle.

Note that the `breadth-search` function takes 3 arguments: a start state, a goal state and an LMG. The `lmg` function is passed as an argument just like any other data.

2.2 Planning Routes

Since an LMG is a function which, given some state `S` produces a sequence of new states, a Clojure map can be used as long as it is a map of the form {`S => S', S'', S'''...`}. This approach is illustrated in Figure 2. We want to find a route of interconnecting flights from one airport to another given some representation of which airports have direct connecting flights to other airports. We can conveniently specify direct connections using a map, which can then act as an LMG.

```
(def flights
  '{teesside (amsterdam dublin heathrow)
    amsterdam (dublin teesside joburg delhi dubai)
    delhi (calcutta mumbai chennai)
    calcutta (mumbai kathmandu)
    mumbai (chennai delhi dubai)
    chennai (colombo)
    dubai (delhi colombo joburg)})

user=> (flights 'delhi)
(calcutta mumbai chennai)
user=> (flights 'amsterdam)
(dublin teesside joburg delhi dubai)
```

Figure 2. A map of flight connections that can be used as an LMG.

The call `(flights airport-name)` works like an LMG - it takes one state as input and generates successor states. This means we can use it, for example, to plan a multi-leg journey from Teesside to Colombo airport (Figure 3).

```
user=> (breadth-search 'teesside 'colombo flights)
(teesside amsterdam dubai colombo)
```

Figure 3. Using the flight connections in Figure 2 as an LMG.

3 Tuples

A common approach used in symbolic artificial intelligence is to represent facts as tuples (short structured sequences of symbols). So the fact *"tom is a cat"* could be represented as `[:isa tom cat]`. In particular, note that:

- Here we use the structure `[relation object value]` - this is normal practice.
- We use a Clojure vector as the tuple type (idiomatic but intuitive).
- We use a keyword for the relation (once again, idiomatic but intuitive).

We can then use a fact-based inference engine to inspect and act on collections of facts either to infer new facts or to explicitly identify relationships between existing facts.

3.1 Searching Through Tuples

The pattern matcher used in this paper[2] supports two forms to provide iteration and collection capability, these are called `mfor` and `mfor*`. They iterate over sets of data using one pattern (`mfor`) or multiple patterns (`mfor*`). The following examples use the data in the `food` structure in Figure 4. Note the use of `(? var)` - the `?` macro retrieves a named value from the matcher pseudo-namespace.

```
(def food
 '([isa cherry fruit] [isa cabbage veg]
   [isa chili veg] [isa apple fruit]
   [isa radish veg] [isa leek veg]
   [color leek green] [color chili red]
   [color apple green] [color cherry red]
   [color cabbage green] [color radish red]))

user=> (mfor ['[isa ?f veg] food] (? f))
(cabbage chili radish leek)

user=> (mfor* ['([isa ?f veg] [color ?f red]) food] (? f))
(chili radish)
```

Figure 4. Using the matcher to search through tuples.

Our first example of working with facts considers writing a query mechanism which examines a collection of tuples to identify objects which have some specified properties. For example, we find *"all the red cubes which are on the table"* given in the set of facts in Figure 5.

```
(def blocks
  '([isa b1 cube] [isa b2 cube] [isa b3 cube]
    [isa b4 wedge] [isa b5 wedge] [isa b6 wedge]
    [color b1 red] [color b2 red] [color b3 red]
    [color b4 blue] [color b5 blue] [color b6 blue]
    [on b1 table] [on b2 table] [on b5 table]))

user=> (mfor* ['([isa ?o cube] [on ?o table]) blocks] (? o))
(b1 b2)
```

Figure 5. A set of facts from which we retrieve the names of all red cubes on the table.

The searching and selection forms provided in the matcher pass patterns across collections of data, returning the first match encountered. These matcher forms are called `mfind` (which matches one pattern across a collection of data) and `mfind*` (which consistently matches a group of patterns across a collection of data). The next two examples use the food data (the blocks data above is used for practical exercises in the workshop).

Note that in these example we use vectors in our data. This is perhaps idiomatic but we sometimes prefer wrapping tuples as vectors (rather than as lists) and the matcher deals with either vectors or lists (or maps).

`mfind` takes one pattern while `mfind*` takes multiple patterns (see Figure 6).

```
user=> (mfind ['[isa ?f veg] food] (? f)) ; find one veg.
cabbage

user=> (mfind* ['([isa ?f veg] [color ?f red]) food] (? f)) ; find one red veg.
chili
```

Figure 6. Usage of the `mfind` and `mfind*` functions on the data in Figure 4.

The matcher allows patterns to be created dynamically (i.e. while functions are being evaluated). We can take advantage of this to build a flexible lookup function for tuples (see Figure 7).

```
(defn lookup [reln val tuples]
  (set (mfor [[reln '?x val] tuples] (? x))))

user=> (lookup 'isa 'fruit food)
#{cherry apple}
user=> (lookup 'color 'red food)
#{radish cherry chili}
```

Figure 7. Implementation and usage of a flexible lookup function for tuples.

4 Simple Rules

Other fact-based inference mechanisms infer new facts from existing facts, we use rules to do this. If, for example, we know *"Sally is Ellen's daughter and Sam is Sally's son"* then we can infer that *"Ellen is one of Sam's grandparents"*. A general-purpose rule such as that in Figure 8 could be used to capture this relationship.

```
[grandparent-rule-2
  [parent ?a ?b] [parent ?b ?c] :=> [grandparent ?a ?c]]
```

Figure 8. A general-purpose rule used to capture the relationship between parents and grandparents.

This rule has 3 parts:

- A name - `grandparent-rule-2`
- A collection of antecedents (facts that must be true for the rule to apply) - `[parent ?a ?b] [parent ?b ?c]`
- A collection of consequents (facts that are inferred if the rule applies) - `[grandparent ?a ?c]`

This style makes use of the matcher to extract the names of people and is therefore general purpose. Later examples will develop rule application mechanisms for this type of rule but first we will consider a simple rule structure.

4.1 Rule Application

Simplified rules are of the form `[pop corn => popcorn]`.

Facts are simple symbols (like `pop` and `corn`) and the rules do not use match expressions (but the matcher is still used in the rule compiler). For the next examples we use the rules in Figure 9.

```
(defn compile-rules [rules]
  (mfor ['[??pre => ??post] rules]
    {:ante (set (? pre)), :consq (set (? post))}))

(def rules
  (compile-rules
    '([tom and jerry => cartoon two-d]
      [shrek => cartoon three-d]
      [cartoon popcorn => movie]
      [pop corn => popcorn]
      [mouse => jerry micky]
      [cat => tom felix]
      [ogre => shrek])))

user=> rules
({:ante #{and jerry tom}, :consq #{two-d cartoon}}
 {:ante #{shrek}, :consq #{three-d cartoon}}
 {:ante #{popcorn cartoon}, :consq #{movie}}) ; Etc.
```

Figure 9. Defining and compiling a collection of rules from which we can infer new facts.

The rule compiler shown in Figure 9 is a simple function which strips apart a symbolic representation of rules, converts antecedent and consequent lists to sets and rebuilds the result into a map. We can then apply the rules to a collection of facts as shown in Figure 10.

```
(defn apply-rule [facts rule]
  (when (subset? (:ante rule) facts)
    (:consq rule)))

; Facts are specified as a set of symbols.
(def facts '#{jerry meets tom with pop and corn})

user=> (apply-rule facts (first rules))
#{two-d cartoon}
```

Figure 10. Inferring by a single rule application that we have a 2D cartoon when Jerry meets Tom with pop and corn.

4.2 Forward Chaining

Forward chaining is a simple inference mechanism which exhaustively applies rules, continuing for as long as it can generate new facts. The function `fwd-chain` in Figure 11 repeatedly applies a set of rules rules keeping track of the number of facts it has and stopping when no new facts are generated. `(reduce union ...)` is used to combine new facts with existing facts.

```
(defn fwd-chain [facts rules]
  (let [fact-count (count facts)
        new-facts (reduce union facts (map #(apply-rule facts %) rules))]
    (if (= fact-count (count new-facts))
      facts
      (recur new-facts rules))))

user=> (def facts '#{mouse meets cat with pop and corn})
#'user/facts
user=> (fwd-chain facts rules)
#{two-d micky felix mouse corn with popcorn meets movie cat
  and cartoon pop jerry tom}
```

Figure 11. Inferring that we have a 2D cartoon movie from pop, corn, a cat and a mouse.

4.3 Backward Chaining

Forward chaining is a *data-driven* inference mechanism, useful when we want to infer new facts from what we already know. If we already have a specific fact in mind that we wish to generate, however, then *goal-driven* backward chaining is more efficient. Using backward chaining, we work backwards from our desired fact to attempt to reach a subset of our initial fact base.

```
(def facts (atom '#{mouse meets cat with pop and corn}))

(defn add-facts [newfs]
  "fact-base updater"
  (swap! facts (fn [fs] (union fs newfs))))

user=> facts
; #<Atom@1d5faf4b: #{mouse corn with meets cat and pop}>
user=> (add-facts '(mango melon banana))
; #{mouse corn mango with meets melon banana cat and pop}
```

Figure 12. Creating a mutable fact base for backward chaining.

The simplest way to specify a backward chaining mechanism is use a global (but still immutable) set of rules and a global and mutable fact base. To allow the fact base to be mutable we use the `atom` reference type (see Figure 12 for an example and the Clojure core documentation for details).

The `bwd-chain` function (see Figure 13) is given a goal-fact to prove or refute. It operates a kind of depth first search using rules to generate successor states. If the goal is present in the initial set of facts then it is proven and the facts are returned. Otherwise it finds any rules which can prove the goal (those which include the goal in their consequents) then sequentially steps through these rules, trying to prove their antecedents. When a rule has its antecedents satisfied, its consequents are added to the fact base. The backward chaining engine fails if cannot find any chain of rule applications which lead to the goal.

```
(defn bwd-chain [goal]
  (if (contains? @facts goal) @facts ; Goal proven.
    ; Else run through suitable rules.
    (let [rules2 (filter #(contains? (:consq %) goal) rules)]
      (some (fn [r] (when (every? bwd-chain (:ante r))
                          (add-facts (:consq r))))
            rules2))))

user=> (def facts (atom '#{mouse meets cat with pop and corn}))
#'user/facts
user=> (bwd-chain 'movie)
#{two-d micky felix mouse corn with popcorn meets movie cat
  and cartoon pop jerry tom}

; Notice that `bwd-chain` doesn't do more work than necessary.
user=> (def facts (atom '#{mouse meets cat with pop and corn}))
# 'user/facts
user=> (bwd-chain 'two-d)
#{two-d micky felix mouse corn with meets cat and cartoon pop jerry tom}
```

Figure 13. Using backward chaining to check whether or not we have a movie, given our `facts` data structure.

5 Matching Rules

Before investigating the forward and backward chaining functions above we noted that general purpose rules contained *matching expressions* to match against facts. Having established the general principles using non-matching rules, we now consider a rule application mechanism which uses matching.

5.1 Application of Matching Rules

Consider the rule and `family` data in Figure 14.

```
(def grandparent-rule
  '[rule 15 [parent ?a ?b] [parent ?b ?c] => [grandparent ?a ?c]])

(def family
  '#{[parent Sarah Tom]
     [parent Steve Joe]
     [parent Sally Sam]
     [parent Ellen Sarah]
     [parent Emma Bill]
     [parent Rob Sally]})
```

Figure 14. A rule capturing the parent-grandparent relationship.

As an initial step these rules are compiled into a map and sets. This time we use a rule compiler for individual rules implemented in Figure 15.

```
(defmatch compile-rule []
  ([rule ?id ??antecedents => ??consequents]
    :=> {:id (? id), :ante (? antecedents) :consq (? consequents)}))
```

Figure 15. An implementation of a rule compiler using the matcher.

A suitable rule application mechanism needs to split the rule into its constituent parts, search for all consistent sets of antecedents, ripple any antecedent variable bindings through to consequents and collect evaluated consequents for each rule every time it is run. In practice these requirements can be met by using a match function to pull a rule apart, `mfor*` to satisfy all possible antecedent combinations and `mout` to bind variables into consequents. This can be specified as in Figure 16.

```
(defn apply-rule [facts rule]
  (set (mfor* [(:ante rule) facts]
         (mout (:consq rule)))))

user=> (apply-rule family (compile-rule grandparent-rule))
#{[grandparent Ellen Tom] [grandparent Rob Sam]}
```

Figure 16. Compiling and applying the rule specified in Figure 14.

5.2 Forward Chaining with Matching Rules

To investigate this rule deduction example further we use a richer set of facts and rules in Figure 17 (where the consequences of some rules trigger the antecedents of others).

```
(def facts1
  '#{[big elephant] [small mouse]
     [small sparrow] [big whale]
     [on elephant mouse]
     [on elephant sparrow]})

(def rules1
  (map compile-rule
    '([rule 0 [heavy ?x] [small ?y] [on ?x ?y] => [squashed ?y] [sad ?x]]
      [rule 1 [big ?x] => [heavy ?x]]
      [rule 2 [light ?x] => [portable ?x]]
      [rule 3 [small ?x] => [light ?x]])))
```

Figure 17. A richer set of facts in which the consequents of some rules trigger the antecedents of others.

Given these definitions it is possible to develop a function to apply all rules once as in Figure 18.

```
(defn apply-all [facts rules]
  (reduce union
    (map #(apply-rule facts %) rules)))

user=> (apply-all facts1 rules1)
#{[light mouse] [heavy elephant] [light sparrow] [heavy whale]}
```

Figure 18. Applying all rules once to our set of animal facts.

We use this with the `fwd-chain` function we defined in Figure 11 to infer as much as we can from our fact base in Figure 19.

```
user=> (fwd-chain facts1 rules1)
#{(light mouse) (heavy elephant)
  (on elephant sparrow)
  (squashed sparrow)
  (small sparrow) (on elephant mouse)
  (squashed mouse) (small mouse)
  (portable sparrow) (big whale)
  (portable mouse) (big elephant)
  (sad elephant) (light sparrow)
  (heavy whale)}
```

Figure 19. Inferring as much as we can from our fact base.

6 Operators

When we first considered using breadth-first search (the first part of this workshop) we assumed simple (atomic) state representations (number or symbols). Most problems are a little more involved and state representations then become necessarily more complex. One common approach is to represent states as a set of facts (actually we often use 2 sets: one which may change between different problem states, the other which cannot change). There are many good reasons to take this approach but a discussion of these is beyond the scope of this document.

In this example we consider how to apply the kind of state changing operators that are used in some planning systems. Broadly we adapt a representation borrowed from Planning Domain Definition Language (PDDL)[3] for use with a STRIPS[4] style solver. The operators are specified in terms of their preconditions and their effects. We use tuples to capture state information.

Figure 20, for example, describes a simple state in which some animated agent R is at a table is holding nothing and a book is on the table.

```
#{(at R table)
  (on book table)
  (holds R nil)
  (path table bench)
  (manipulable book)
  (agent R)}
```

Figure 20. A state in which some animated agent R is at a table holding nothing, with a book on the table.

In order to generalize an operator (so it can be used with different agents, objects and in various locations) it is necessary to specify it using variables to capture parts of the state information. The matcher is (in part) designed for this type of activity so matcher variables are used for this. An operator which describes a *pickup* activity for an agent and which can be used to produce a new state (new set of tuples) could be described as in Figure 21.

```
{:pre ((agent ?agent)
       (manipulable ?obj)
       (at ?agent ?place)
       (on ?obj ?place)
       (holds ?agent nil))
 :add ((holds ?agent ?obj))
 :del ((on ?obj ?place)
       (holds ?agent nil))}
```

Figure 21. An operator which describes a *pickup* activity for an agent.

The operator is map with three components:

- :pre - a set of preconditions which must be satisfied in order for the operator to be used
- :add - a set of tuples to add to an existing state when producing a new state
- :del - a set of tuples to delete from an existing state

6.1 Application

To apply this kind of operator specification we extract patterns from the operator then use mfind* as in Figure 22.

```
(defn apply-op
  [state {:keys [pre add del]}]
  (mfind* [pre state]
    (union (mout add)
      (difference state (mout del)))))

user=> (apply-op state1 ('pickup ops))
#{(agent R) (holds R book)
  (manipulable book)
  (path table bench) (at R table)}
```

Figure 22. An apply-op function that uses the matcher to apply a STRIPS-style operator.

The patterns used by mfind* are provided dynamically when apply-op is called and furthermore the patterns themselves define the semantics of the operators.

In Figure 23 collections of operators are conveniently held in a map, and ordered sequences of operator applications are formed by chaining apply-op calls.

```
(def ops
  '{pickup {:pre ((agent ?agent)
                  (manipulable ?obj)
                  (at ?agent ?place)
                  (on ?obj ?place)
                  (holds ?agent nil))
            :add ((holds ?agent ?obj))
            :del ((on ?obj ?place)
                  (holds ?agent nil))}
    drop {:pre ((at ?agent ?place)
                (holds ?agent ?obj))
          :add ((holds ?agent nil)
                (on ?obj ?place))
          :del ((holds ?agent ?obj))}
    move {:pre ((agent ?agent)
                (at ?agent ?p1)
                (path ?p1 ?p2))
          :add ((at ?agent ?p2))
          :del ((at ?agent ?p1))}})

user=> (-> state1 (apply-op ('pickup ops))
                  (apply-op ('move ops))
                  (apply-op ('drop ops)))
#{(agent R) (manipulable book)
  (on book bench) (holds R nil)
  (at R bench) (path table bench)}
```

Figure 23. A map of operators applied to a state to modify it.

7 Operator Search

The `ops-search` implementation used in this paper[5] provides a simple, partially optimized implementation of a breadth-first search mechanism for applying simple STRIPS-style operators. The `ops-search` function takes the following arguments:

- `start` - a start state
- `goal` - a minimally specified goal state (see below)
- `operators` - a collection of operators
- `world` - a definition of unchanging world states

A map is returned showing:

- The goal state reached
- The path taken
- Any commands to send to another subsystem
- A textual description of the path

Goals are minimally specified so any state which is a superset of the goal is deemed a goal state. In addition to `:pre`, `:add` and `:del` entries, operators may optionally also specify:

- `:txt` - a textual description of the operator application to aid readability
- `:cmd` - an encoded command for the operator, typically for the benefit of some other subsystem

A more complete set of operators is shown in Figure 24.

The example in Figure 25 uses the `pickup`, `drop` and `move` operators defined in Figure 24 in combination with `ops-search` to plan a set of actions that must be taken to reach the goal state from the start state.

```
(def ops
  '{pickup {:pre ((agent ?agent)
                  (manipulable ?obj)
                  (at ?agent ?place)
                  (on ?obj ?place)
                  (holds ?agent nil))
            :add ((holds ?agent ?obj))
            :del ((on ?obj ?place)
                  (holds ?agent nil))
            :txt (pickup ?obj from ?place)
            :cmd [grasp ?obj]}
    drop {:pre ((at ?agent ?place)
                (holds ?agent ?obj)
                (:guard (? obj))) ; This prevents ?obj matching nil (see docs).
          :add ((holds ?agent nil)
                (on ?obj ?place))
          :del ((holds ?agent ?obj))
          :txt (drop ?obj at ?place)
          :cmd [drop ?obj]}
    move {:pre ((agent ?agent)
                (at ?agent ?p1)
                (connects ?p1 ?p2))
          :add ((at ?agent ?p2))
          :del ((at ?agent ?p1))
          :txt (move ?p1 to ?p2)
          :cmd [move ?p2]}})
```

Figure 24. A set of operators: *pickup*, *drop* and *move*.

```
(def state1
  '#{(at R table)
     (on book table)
     (on spud table)
     (holds R nil)
     (connects table bench)
     (manipulable book)
     (manipulable spud)
     (agent R)})

user=> (ops-search state1 '((on book bench)) ops)
{:state #{(agent R) (manipulable book) (on spud table)
          (on book bench) (holds R nil) (at R bench)
          (manipulable spud) (connects table bench)},
 :path (#{(agent R) (manipulable book) (on spud table)
          (holds R nil) (manipulable spud) (connects table bench)
          (on book table) (at R table)}
        #{(agent R) (holds R book) (manipulable book)
          (on spud table) (manipulable spud)
          (connects table bench) (at R table)}
        #{(agent R) (holds R book) (manipulable book)
          (on spud table) (at R bench)
          (manipulable spud) (connects table bench)}),
:cmds ([grasp book] [move bench] [drop book]),
:txt ((pickup book from table) (move table to bench) (drop book at bench))}
```

Figure 25. Using the pickup, drop and move operators in combination with `ops-search` to create a plan to realize a goal.

7.1 Using World Knowledge

We can use `:world` definitions to hold unchanging state relations. Separating static/world relations from those that may change improves the efficiency of the search (see Figure 26).

```
(def world
  '#{(connects table bench)
     (manipulable book)
     (manipulable spud)
     (agent R)})

(def state2
  '#{(at R table)
     (on book table)
     (on spud table)
     (holds R nil)})

user=> (ops-search state2 '((on book bench)) ops :world world)
{:state #{(on spud table) (on book bench) (holds R nil) (at R bench)},
 :path (#{(on spud table) (holds R nil) (on book table) (at R table)}
        #{(holds R book) (on spud table) (at R table)}
        #{(holds R book) (on spud table) (at R bench)}),
 :cmds ([grasp book] [move bench] [drop book]),
 :txt ((pickup book from table) (move table to bench) (drop book at bench))}
```

Figure 26. Using the pickup, drop and move operators in combination with `ops-search` to create a plan to realize a goal.

7.2 Using Compound Goals

We are not limited to one fact in our goal state, however. We can specify a list of facts that our goal state must contain as a *compound goal* as in Figure 27.

```
(def world2
  '#{(connects table bench)
     (connects bench table)
     (connects bench sink)
     (connects sink bench)
     (manipulable book)
     (manipulable spud)
     (agent R)})

(def state3
  '#{(at R table)
     (on book table)
     (on spud table)
     (holds R nil)})

user=> (ops-search state3 '((on book bench)(on spud sink)) ops :world world2)
{:state #{(at R sink) (on book bench) (holds R nil) (on spud sink)},
 :path (#{(on spud table) (holds R nil) (on book table) (at R table)}
        #{(holds R book) (on spud table) (at R table)}
        #{(holds R book) (on spud table) (at R bench)}
        #{(on spud table) (on book bench) (holds R nil) (at R bench)}
        #{(on spud table) (on book bench) (holds R nil) (at R table)}
        #{(on book bench) (holds R spud) (at R table)}
        #{(on book bench) (at R bench) (holds R spud)}
        #{(at R sink) (on book bench) (holds R spud)}),
 :cmds ([grasp book] [move bench] [drop book]
        [move table] [grasp spud] [move bench] [move sink] [drop spud]),
 :txt ((pickup book from table) (move table to bench) (drop book at bench)
       (move bench to table) (pickup spud from table) (move table to bench)
       (move bench to sink) (drop spud at sink))}
```

Figure 27. Specifying a compound goal state, which consists of many facts.

Note that goals can be specified as matcher patterns so the call in Figure 28 is valid. An unspecified object `x` must be on the bench and another unspecified object `y` must be on the sink.

```
(ops-search state3 '((on ?x bench) (on ?y sink)) ops :world world2)
```

Figure 28. Specifying a compound goal state using matcher patterns.

8 Summary

This paper has introduced some useful tools[2] [5] and key techniques for constructing planning and inference mechanisms in Clojure. It assumed no prior knowledge of either symbolic computation or artificial intelligence concepts. Legal move generators, a basic breadth-first search based planning mechanism and forward and backward chaining in the context of inference have all been covered.

References

1. Cognesence. cognesence/breadth-search (2017). URL https://github.com/cognesence/breadth-search.

2. Cognesence. cognesence/matcher (2017). URL https://github.com/cognesence/matcher.
3. McDermott, D. *et al.* Pddl-the planning domain definition language. (1998).
4. Fikes, R. E. & Nilsson, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artif. intelligence* **2**, 189–208 (1971).
5. Cognesence. cognesence/ops-search (2017). URL https://github.com/cognesence/ops-search.