Generalized Planning: Some Theory, Some Practice

Hector J. Levesque
Dept. of Computer Science
University of Toronto

AAAI Workshop, Summer 2011

Joint work with Toby Hu

Motivation

The motivation for generalized planning is to explore the idea of *automated planning* in a richer context than that of classical planning.

In particular, we are concerned here with efficiently generating plans for a putative robot when some form of *loops* or recursion is required.

In this talk, I will cast this problem within the broad framework of *reasoning about action* within knowledge representation (within artificial intelligence).

In its most general form, this will unfortunately make planning look a lot like automatic programming.

This raises two immediate questions that I will attempt to address:

- Can the theoretical story be realized in practical terms?
- Can a practical story be justified in theoretical terms?

Overview

- planning as knowledge representation
  - classical planning in the situation calculus
  - the adjustments required for sensing
- the `FSAPLANNER` system
  - a planner within a situation calculus reasoner
  - three related planning examples:
    - Two Towers
    - Towers of Hanoi
    - Striped Tower
- finite verifiability
  - some examples and non-examples
- conclusion and future work

In the beginning . . .

In the beginning, there was John McCarthy.

His *Programs with common sense* paper (parts of which date to 1958), was the first paper to talk about automated planning (*inter alia*).

With respect to planning, its main features are

- a formal logical language for expressing knowledge about a dynamic world: *the situation calculus*

  > *"We believe that human intelligence depends essentially on the fact that we can represent in language facts about our situation, our goals, and the effects of the various actions we can perform."*

- the first planning problem: the monkeys and bananas
- planning subsumed as part of commonsense reasoning

The situation calculus

A dialect of classical first-order logic with special sorts for objects called *actions* and objects called *situations*:

- a constant S_0 denoting the initial situation;
- a function do such that the term $do(a, s)$ denotes the situation that results from performing action a in situation s;
- functions and predicates whose values depend on the situation are called *fluents* (and usually take a situation as their last argument).

The [Reiter 01] book shows how to formalize knowledge about a dynamic world in this language, using what is called a *basic action theory*:

$$\begin{aligned}\forall x \forall a.\ LightOn(x, do(a, s)) \quad \equiv \\ a = turnOn(x) \quad \vee \\ a = toggle(x) \ \wedge \ \neg LightOn(x, s) \quad \vee \\ LightOn(x, s) \ \wedge a \neq toggle(x) \ \wedge \ a \neq turnOff(x)\end{aligned}$$

Classical planning (regained)

We can think of planning in these terms:

> Given beliefs about the initial state of the world and how the world changes as the result of the actions it can perform, the agent must find a plan that it believes will achieve the goal.

We can represent the beliefs of the agent as sentences in a KB (a basic action theory), and we need to find a sequence of actions $\vec{a}$ such that

$$\text{KB} \models \mathit{Legal}(do(\vec{a}, S_0)) \wedge \mathit{Goal}(do(\vec{a}, S_0)),$$

where

- $\mathit{Legal}(s)$ means that we got to s by performing actions whose preconditions were satisfied, and
- $\mathit{Goal}(s)$ means the goal we care about is satisfied in s.

But what if there is no sequence of actions that the agent believes will achieve the goal?

When bad things happen to good planners

In some cases, the agent is simply unable to achieve the goal.

> For example, consider a safe whose combination is not known and that can only be opened by dialing the right combination.
>
> Dialing the wrong combination, even once, locks the safe forever.
>
> What to do? *Give up!*

But in other cases, the agent will be able to achieve the goal by acquiring information as it goes.

> For example, consider the safe as above.
>
> Suppose the combination is written on a piece of paper that the agent can read once the paper is picked up.
>
> What to do? *Make a plan!*

Planning reconsidered

We can think of planning in these terms:

> Given beliefs about the initial state of the world and how the world changes as the result of the actions it can perform, the agent must find a plan that it believes will achieve the goal.

However,

- A plan is no longer a simple sequence of actions.

  It must be a structure of some sort that tells the agent unambiguously what to do (without requiring further planning), but it must allow different actions depending on what information is acquired *en route*.

- The beliefs of the agent are no longer what is encoded in the KB.

  The beliefs of the agent also include acquired information (for example, by reading what is on the piece of paper) that is not represented in the KB because it is not known in advance.

Planning reconsidered

We can think of planning in these terms:

> Given beliefs about the initial state of the world and how the world changes as the result of the actions it can perform, the agent must find a plan that it believes will achieve the goal.

Find a plan p (of some sort) such that

$$\text{KB} \models \forall s.\, B(s, S_0) \supset \exists s^*.\, Rdo(p, s, s^*) \wedge Goal(s^*),$$

where

- $B(s', s)$ holds when the agent in situation s believes that for all it knows, it could be located in situation s', and
- $Rdo(p, s, s')$ holds when s' is the end situation that results from doing plan p (whatever that is) legally in situation s.

See [Levesque 96]: *What is planning in the presence of sensing?*

What are plans?

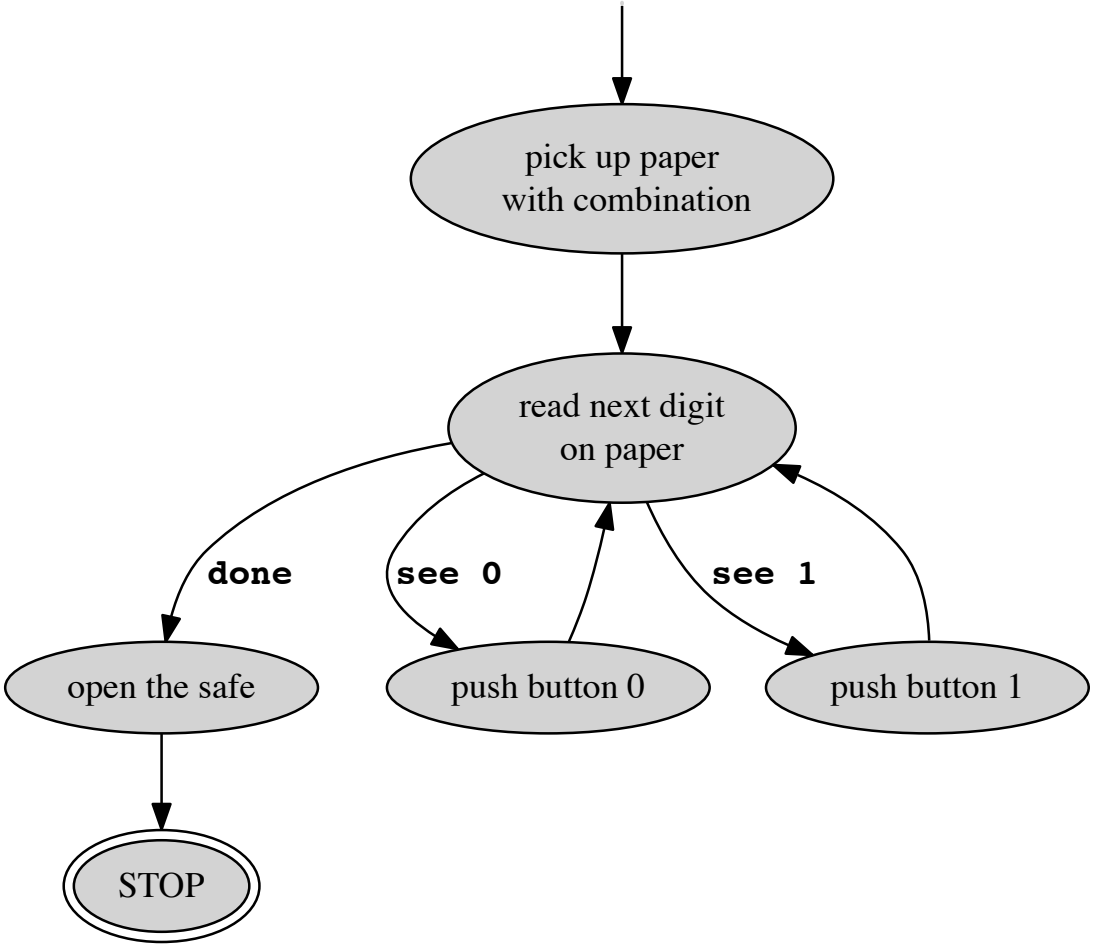
In [Levesque 96], plans were taken to be syntactic structures: robot programs.

However, [Hu & Levesque 09] have shown that a finite-state automaton (a Moore machine) provides a strictly more general notion of a plan:

> For actions $\mathcal{A}$ and sensing results $\mathcal{R}$, an *FSA plan* is a tuple $\langle Q, q_0, q_F, \gamma, \delta \rangle$ where
>
> - Q is a finite set of plan states;
> - $q_0 \in Q$ is the initial state;
> - $q_F \in Q$ is the final state.
> - $\gamma \in [Q \rightarrow \mathcal{A}]$ maps states to actions;
> - $\delta \in [Q \times \mathcal{R} \rightarrow Q]$ is the state transition function;

An FSA-plan can be displayed as a *graph* with actions on the nodes and sensing results on the edges.

Opening a safe using a binary combination



Putting the theoretical story into practice

The definition of planning we have proposed suggests an implementation in terms of *theorem-proving*:

- find a legal plan that satisfies the required logical entailment

  correctness over all initial situations compatible with what is believed

But this is a suggestion that is worth resisting!

Instead, we can put correctness over all initial situations aside temporarily and take our cue from *machine learning*, as in [Srivastava *et al* 08]:

- attempt to induce a plan that works for some set of initial states. (Think of these as training examples.)

Guarantees about the general correctness of what is induced need not be part of the planning process itself, as in [Levesque 05].

A situation calculus reasoner

We have implemented a system called `FSAPLANNER` in Scheme.

It uses a new situation calculus reasoner, which it shares with an upcoming implementation of the `IndiGolog` programming language.

- can use all Scheme objects: numbers, symbols, lists, tables, closures, ...
- fluents can take as value any Scheme object;
- actions can have prerequisites, effects, and sensing results.

Instead of storing a collection of *sentences* representing what is believed about the initial situation, we maintain an explicit *list of initial world states*.

(A world state is an assignment of values to fluents.)

Reasoning is *progressive*, not regressive: we use the equivalent of Reiter successor-state axioms to calculate the changes to a state after an action.

How to find plans

`FSAPLANNER` does a simple depth-first search in the space of legal plans:

- present the initial world states "small" ones first
- go through the initial world states, constructing a plan incrementally, adding transitions to the plan only as new sensing results are seen;
- add a new plan state only if none of the existing plan states can serve as the target for a transition; use a bound on the number of states; (also bound the number of steps required to get to a goal state)
- always stop at dead loops (same plan and world states); optionally also stop at repeated world states;
- for harder problems: optionally randomize the initial world states of each size, and restart the search periodically after a timeout.

Note: once all of the relevant sensing results have been seen, it is easy to determine if the resulting plan works for additional initial world states.

Example: A simple robotics world

This is a world where a robot can pick up and put down objects (such as bricks or disks) in three separate stacks, A, B, and C.

We model the actions as *attempts*. For example:

> Picking up from a stack can be performed when the robot hand is empty.
>
> A sensing result will return the value *fail* when the stack is empty.

```
;;;     This is the file generic-pick-put.scm, defining a basic action theory
;;;     consisting of 2 fluents and 2 actions.

;;; The two fluents
;;;     hand:   a robotic hand that holds an object or is empty
;;;     stacks: a table of 3 stacks A, B, C, where objects can be located

;;; The two actions
;;;     pick x: try to pick up an object from stack x
;;;     put x:  try to put down the currently held object onto stack x

;;; Two additional definitions
;;;     (ok-to-put? obj stack)   can the obj be successfully placed on the stack?
;;;     sense-pick-object?       does pick return the object picked or just "ok"?
```

The basic action theory

```
;; the two fluents are initialized by providing a list z of objects for stack A
(define-state (ini-state z)
      hand     'empty                                  ; start holding nothing
      stacks   (hasheq 'A z 'B '() 'C '()))            ; all objects initially on stack A

(define (stack x) (hash-ref stacks x))                 ; for convenience

(define-act (pick x)                                   ; pick an object or sense failure
      prereq   (eq? hand 'empty)
      sensing  (if (null? (stack x)) 'fail
                   (if sense-pick-object? (car (stack x)) 'ok))
      hand     (if (null? (stack x)) 'empty (car (stack x)))
      stacks   (hash-set stacks x (if (null? (stack x)) '() (cdr (stack x)))))

(define-act (put x)                                    ; put an object or sense failure
      prereq   (not (eq? hand 'empty))
      sensing  (if (ok-to-put? hand (stack x)) 'ok 'fail)
      hand     (if (ok-to-put? hand (stack x)) 'empty hand)
      stacks   (hash-set stacks x (if (ok-to-put? hand (stack x))
                                      (cons hand (stack x))        ; change
                                      (stack x))))                 ; no change!
```

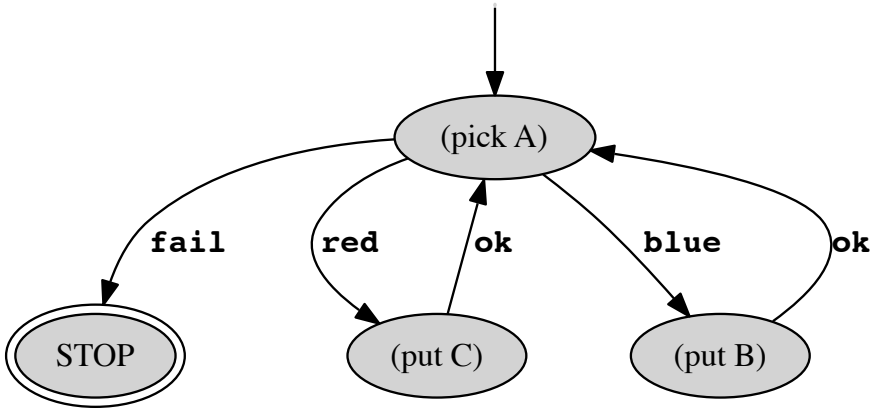
A planning example: building two towers

```
;;;  The stacks, A, B, and C hold blocks of two colours, red and blue.  The robot can
;;;  pick up and put down a block on a stack, one block at a time.  When the robot
;;;  picks up a block, it senses the colour of that block.  It also senses when the
;;;  stack is empty.

;;;  Initially, some number of blocks are on stack A, but B and C are empty.
;;;  The goal is to get the blue blocks onto B and the red ones onto C.

(include "generic-pick-put.scm")         ; load the basic action theory
(define sense-pick-object? #t)           ; picking a block senses its colour
(define (ok-to-put? obj stack) #t)       ; always ok to put a block anywhere

(define-condition goal?
  (and (eq? hand 'empty) (null? (stack 'A))              ; stack A is empty
       (andmap (lambda (b) (eq? b 'blue)) (stack 'B))    ; all blue on stack B
       (andmap (lambda (b) (eq? b 'red)) (stack 'C))))   ; all red on stack C

(define all-actions (list (pick 'A) (put 'B) (put 'C)))  ; 1 pick, 2 put actions

(define all-initial-states                               ; initial states in order
  (map ini-state '(() (red) (blue) (red red blue red) (blue blue blue red red))))

(define (main) (genplan goal? all-actions all-initial-states))
```

What does the plan for Two Towers look like?

When `FSAPLANNER` is run, it produces an FSA plan (defined earlier), which then can be passed as input to the `dot` utility, to display a graph.

The result looks like this:

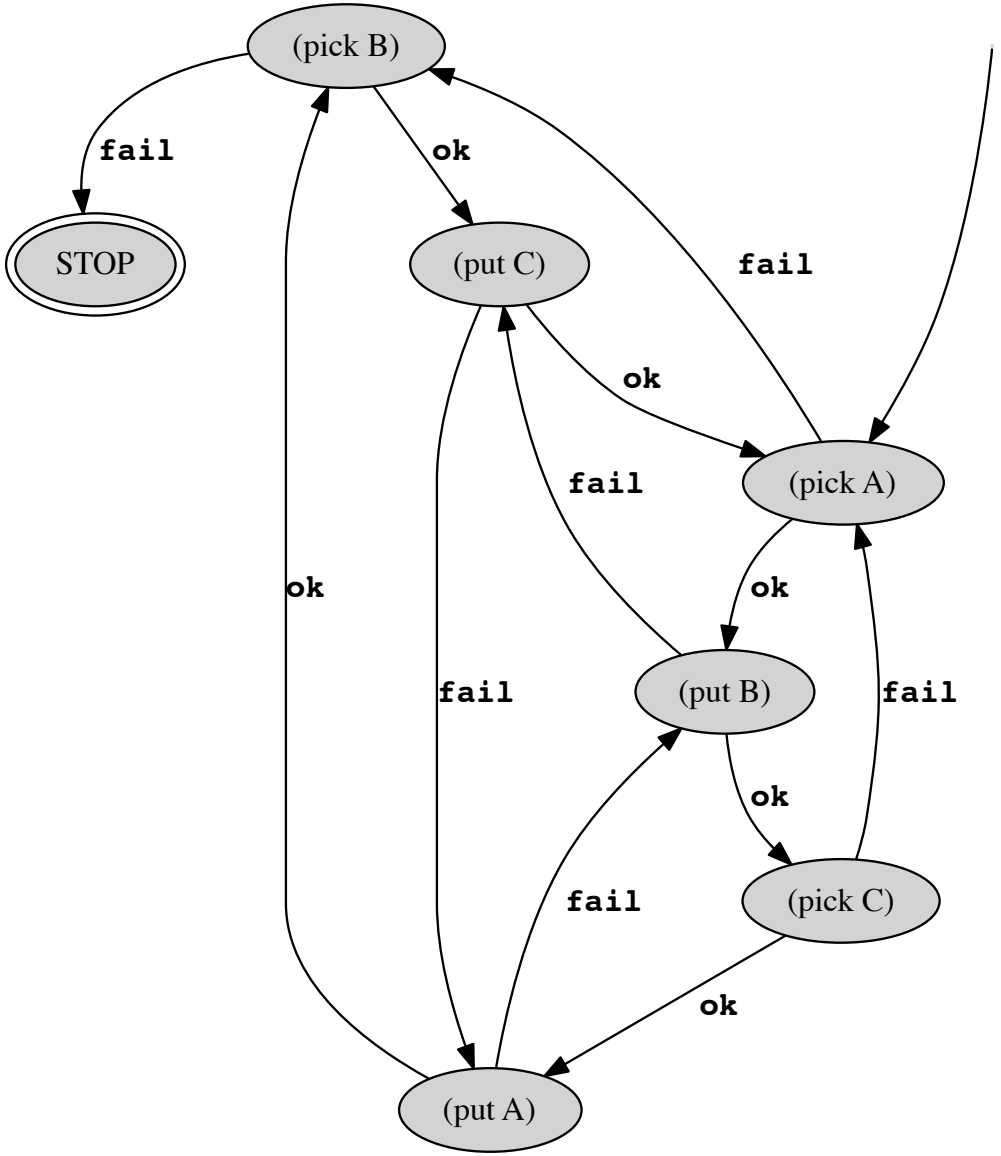


Two Towers.
Generated by `FSAPLANNER`.
Displayed by `dot`.

A much harder example: Towers of Hanoi

```
;;; The stacks are pegs that contain disks but only in ascending order.  The robot
;;; senses the failure of trying to pick up a disk from an empty peg; the robot also
;;; senses the failure of trying to put a disk on a peg whose top disk is smaller.

;;; Initially, all of the disks are on peg A.
;;; The goal is to get them all onto peg C.

(include "generic-pick-put.scm")         ; load the basic action theory
(define sense-pick-object? #f)           ; only sense whether picking is successful
(define (ok-to-put? obj stack)           ; can only put a disk on top of a bigger one
  (or (null? stack) (< obj (car stack))))

(define-condition goal?                  ; get all the disks onto peg C
  (and (eq? hand 'empty) (null? (stack 'A)) (null? (stack 'B))))

(define all-actions (append (map put '(A B C)) (map pick '(A B C))))

(define all-initial-states               ; solve for N = 0,1,2,3 only
  (map (lambda (n) (ini-state (build-list n add1))) '(0 1 2 3)))

(define (main) (genplan goal? all-actions all-initial-states
                        #:steps 60       ; allow for up to 60 steps to the goal
                        #:states 6))     ; but no plans with > 6 states
```

An iterative plan for the Towers of Hanoi



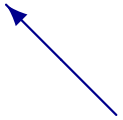
Towers of Hanoi.
Generated by `FSAPLANNER`.
Displayed by `dot`.

A hardest example: building a striped tower

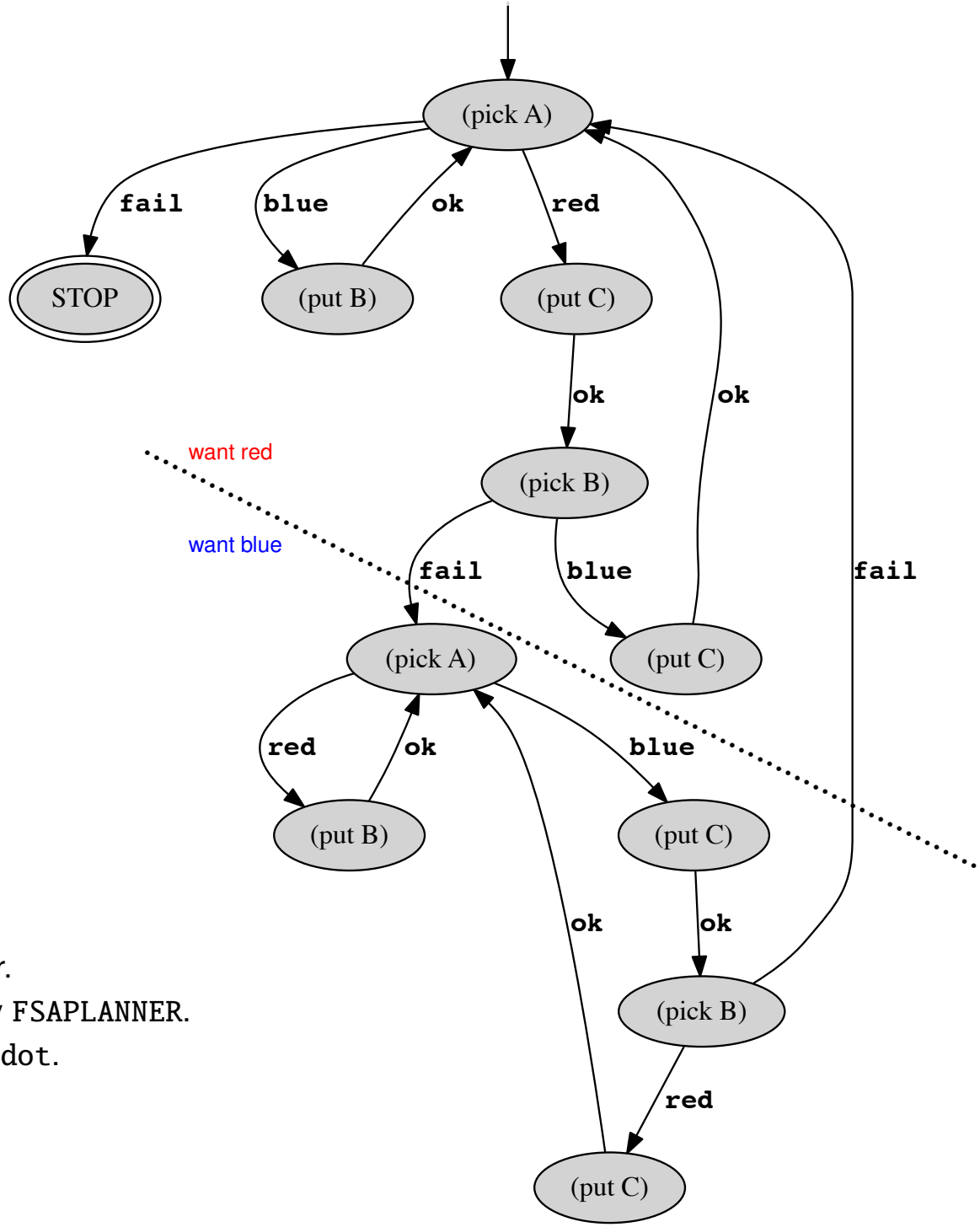
```
;;; This is a more difficult version of a problem from [Srivastava 08].

;;; The setup is similar to the Two Tower problem. Initially, stack A contains an
;;; equal number of red and blue blocks in some order.  The robot can only pick up
;;; from stacks A and B and put onto stacks B and C.

;;; The goal is to have A and B empty and a striped tower on C with blue at the top.

(include "generic-pick-put.scm")          ; load the basic action theory
(define sense-pick-object? #t)            ; picking a block senses its colour
(define (ok-to-put? obj stack) #t)        ; always ok to put a block anywhere

(define-condition goal?
  (and (eq? hand 'empty) (null? (stack 'A)) (null? (stack 'B)) (striped? (stack 'C))))

(define (striped? x)                       ; x is a list (blue red blue red ...)
  (or (null? x) (and (eq? (car x) 'blue) (eq? (cadr x) 'red) (striped? (cddr x)))))

(define all-actions (list (put 'B) (put 'C) (pick 'A) (pick 'B)))   ; note!

(define (all-initial-states n) (map ini-state (even-colours n)))    ; elsewhere

(define (main) (genplan* goal? all-actions (map all-initial-states '(0 1 2 3))
                         #:rand #t        ; randomize within a size group!
                         #:loop #t))      ; detect loops
```

A run of FSAPLANNER on Striped Tower

```
FSA> mzscheme -tm generic-striped.scm
Planning inis:29  max steps:30  max states:10  timeout:0.5  state loop:#t
Randomizing initial states.... Plan found after 2329 ms.
1 (pick A) red 3
1 (pick A) blue 2
1 (pick A) fail 0
2 (put B) ok 1
3 (put C) ok 4
4 (pick B) fail 6
4 (pick B) blue 5
5 (put C) ok 1
6 (pick A) red 9
6 (pick A) blue 7
7 (put C) ok 8
8 (pick B) red 10
8 (pick B) fail 1
9 (put B) ok 6
10 (put C) ok 6
The random seed for this run was 1265801765
```



4 restarts, each after .5 seconds



Striped Tower.
Generated by `FSAPLANNER`.
Displayed by `dot`.

Justifying the practical story

The plans shown are guaranteed by the planner to work, but only for those initial states given to the planner.

Is this enough?

For learning, it is often enough to learn from some training examples and come up with a generalization that works for some new test examples.

We can do this as well.

However, we can also go further...

One approach: *finite verifiability*.

> Characterize classes of actions theories and planning problems with the property: for any plan p, if p works for all problems of size N_p or less, then p will work for *all* problems.

A finitely-verifiable action theory

The first example of a simple (but non-trivial) finitely-verifiable action theory was presented in [Hu & Levesque 10].

The *1-dimensional action theories* are like finite action theories except that

- there is a distinguished integer-valued fluent $n(s)$, whose value can only go down, and whose value is 0 at the goal;
- there is a distinguished sequence fluent $f(i, s)$ that is only indexed by the value of n.

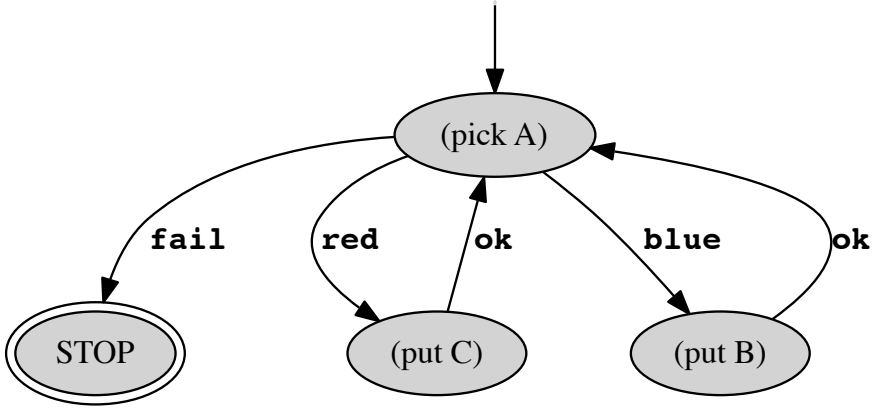
Examples: the Two Towers, the locked safe, recycling [Srivastava 10], delivery [Srivastava 08], (one version of) gripper [Bonet 09], ...

For Two Towers: $n =$ size of stack A; $f(i) = i$-th element of stack A.

Theorem: If a plan is correct for all initial states where $n \leq 2 + k_0 \cdot l^m$, then the plan is correct for all initial states.

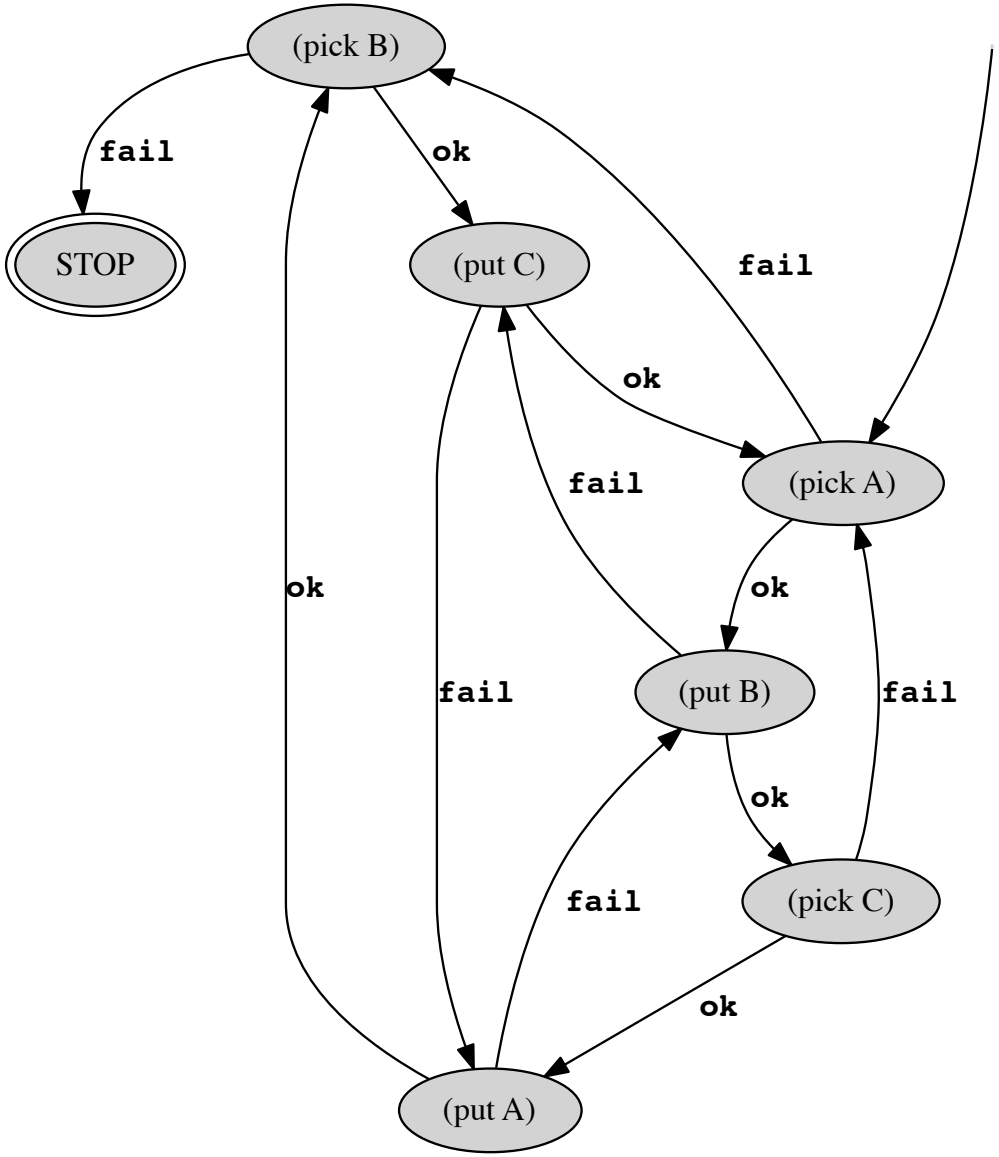
So this plan is correct

Two Towers:

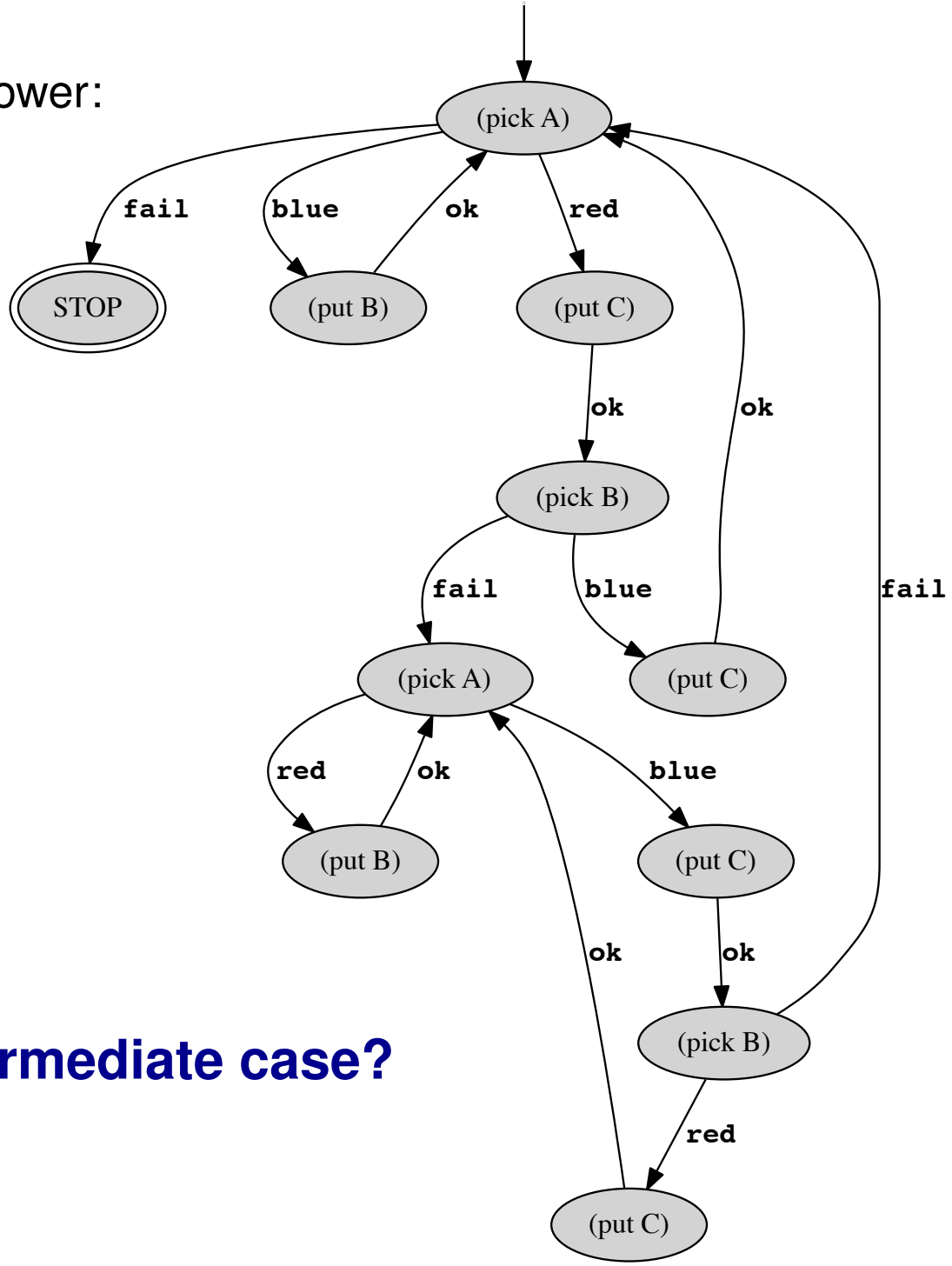


But what about this one?

Towers of Hanoi:



Striped Tower:



An intermediate case?

More finite verifiability

The result for 1-dimensional action theories can be generalized easily to k-dimensional ones.

But what about problems like the Striped Tower where we need an index for stack B that can go *up and down*?

Theorem: Any basic action theory that is 1d as before, except for a second integer-valued fluent $m(s)$ whose value can go up and down and whose value is 0 at the goal, is also finitely verifiable.

Does this cover the Striped Tower problem?

- No, not without a sequence fluent for the m! We need to keep track of the contents of stack B since we will be picking up objects from there.
- However, this theorem *does* cover a simpler version of Striped Tower where all the blocks on stack B are required to be of the same colour!

Non-finitely-verifiable action theories

Looking at general cases involving integer-valued fluents whose values can go up and down can easily lead to undecidability results.

Theorem: There is a basic action theory that is finite except for an integer-valued fluent $n(s)$ whose value is 0 at the goal, and a sequence fluent $f(i, s)$ indexed only by n, that is *not* finitely verifiable.

Does this prove that the Striped Tower problem is not finitely verifiable?

- No. The Striped Tower uses Stack B in a very specific way!

What about two unrestricted integer fluents but without sequence fluents?

Theorem: There is a basic action theory that is finite except for two integer-valued fluents whose values are 0 at the goal, whose finite verifiability would confirm the Collatz conjecture.

So . . .

Collatz Conjecture (1937)

One statement:

> Take any positive natural number. If it is even, divide it by 2. If it is odd, multiply it by 3 and add 1. If you repeat this process long enough, you will eventually end up with the number 1.

Or, equivalently: The program below always terminates

```
while N > 1 do
  if N is even
    then N := N/2
    else N := 3*N+1
  end
end
```

Example: $n = 6 \to 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$

Paul Erdös: "Mathematics is not yet ready for such problems!"

Late breaking news! (May 2011) Claim of proof by Gerhard Opfer.

Non-finitely-verifiable action theories

Looking at general cases involving integer-valued fluents whose values can go up and down can easily lead to undecidability results.

Theorem: There is a basic action theory that is finite except for an integer-valued fluent $n(s)$ whose value is 0 at the goal, and a sequence fluent $f(i, s)$ indexed only by n, that is *not* finitely verifiable.

Does this prove that the Striped Tower in not finitely verifiable?

- No. The Striped Tower uses Stack B in a very specific way.

What about two unrestricted integer fluents but no sequence fluents?

Theorem: There is a basic action theory that is finite except for two integer-valued fluents whose values are 0 at the goal, whose finite verifiability would confirm the Collatz conjecture.

So . . . we are not likely to be able to prove its finite verifiability!

Conclusion: our reach should exceed our grasp

Generalized planning provides a nice interaction between theory and practice, with one sometimes far ahead of the other.

- Planning can be understood within the theoretical framework of reasoning about action: fluents and actions, belief and sensing.
- The practical aspects need not be addressed by a theorem-proving program. A framework based on learning can also be productive.
- The basic action theories used for this form of planning can then be analyzed theoretically to determine conditions under which they will produce plans that are correct in general.

We should not be dissuaded when our theory appears to be too hard to put into practice, or when our practical implementations appear to be too hard to analyze theoretically.

Future work: exogenous actions

Generalized planning should be ready for *exogenous actions*.

- Imagine a version of the Two Tower problem where new blocks can arrive exogenously at the end of stack A (= queue A).

  Intuitively, the previous plan is still the correct behaviour!

  Need to be clear about what "correct" means in this context.

- Physical attempts should also be characterized exogenously, as in:
  - trying to sink a basketball (I shoot the ball; sinking may happen)
  - trying to get to the kitchen (I head off; arriving may happen)

  Perhaps planning in this context is best done by assuming that attempts are followed by their *expected* exogenous outcomes.

However, planning, even sophisticated planning, can only go so far towards the intelligent control of robotic behaviour . . .

The Two Towers and Golog

```
(define (control1)              ; deterministic control (no testing of goal needed!)
  (:begin (:act (pick 'A))
          (:until (eq? hand 'empty)
               (:if (eq? hand 'blue) (:act (put 'B)) (:act (put 'C)))
               (:act (pick 'A)))))

(define (control2)              ; non-deterministic choice of put actions
  (:begin (:act (pick 'A))
          (:until (eq? hand 'empty)
               (:choose (:act (put 'B)) (:act (put 'C)))
               (:act (pick 'A)))
          (:test (goal? current-state))))

(define (control3)              ; non-deterministic choice of put and # of iterations
  (:begin (:act (pick 'A))
          (:star (:choose (:act (put 'B)) (:act (put 'C))) (:act (pick 'A)))
          (:test (goal? current-state))))

(define (solve n)               ; generic search for a plan of n steps or fewer
  (:for-some x all-actions
      (:act x)
      (:choose (:test (goal? current-state))
               (:begin (:test (> n 0)) (solve (- n 1))))))
```

References

- [Bonet 09]: Blai Bonet, Héctor Palacios, Hector Geffner. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In ICAPS 2009.
- [Hu & Levesque 09]: Planning with loops: some new results. In ICAPS 2009 Workshop.
- [Hu & Levesque 10]: A correctness result for reasoning about one-dimensional planning problems. In KR 2010.
- [Levesque 96]: What is planning in the presence of sensing? In AAAI 1996.
- [Levesque 05]: Planning with loops. In IJCAI 2005.
- [Reiter 01]: Ray Reiter. *Knowledge in Action*, MIT Press, 2001.
- [Srivastava 08]: Siddharth Srivastava, Neil Immerman, Shlomo Zilberstein. Learning generalized plans using abstract counting. In AAAI 2008.
- [Srivastava 10]: Siddharth Srivastava, Neil Immerman, Shlomo Zilberstein. Computing applicability conditions for plans with loops. In ICAPS 2010.