

# Generation and Verification of Plans with Loops

Toby Hu  
Knowledge Representation Group

May 20, 2010

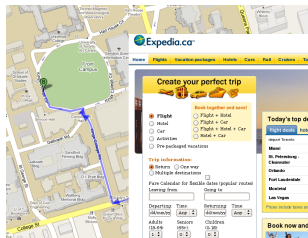
# Towards Intelligent Robots

Building highly intelligent agents is one of the most exciting and challenging tasks of artificial intelligence and computer science.

# Towards Intelligent Robots

Building highly intelligent agents is one of the most exciting and challenging tasks of artificial intelligence and computer science.

► Online



The screenshot displays the Expedia website interface. On the left, a map shows a route starting from a green circular area, moving north, then east, and finally south along a road. The Expedia logo is visible at the top of the interface. The main content area is titled "Create your perfect trip" and includes several sections:

- Book together and save!** with radio button options:
  - Flight
  - Hotel
  - Car
  - AirTrain
  - Pre-packaged vacations
  - Flight + Hotel
  - Flight + Car
  - Flight + Hotel + Car
  - Hotel + Car
- Trip Information:** Includes radio buttons for "Return" and "One way", and a checkbox for "1M+ flight destinations".
- Pairs Calculator for Seattle (popular routes):** A form with "Leaving from" and "Going to" fields.
- Departure Table:**

Departing	Time	Returning	Time
10:00 AM	Dep. 2	10:00 AM	Dep. 2
10:00 AM	Dep. 2	10:00 AM	Dep. 2
10:00 AM	Dep. 2	10:00 AM	Dep. 2
- Today's top deals:** A list of destinations including St. Petersburg - Clearwater, Orlando, Fort Lauderdale, Montreal, and Las Vegas.
- Book now and** with a "Book now" button.

# Towards Intelligent Robots

Building highly intelligent agents is one of the most exciting and challenging tasks of artificial intelligence and computer science.

- ▶ Online
- ▶ Home



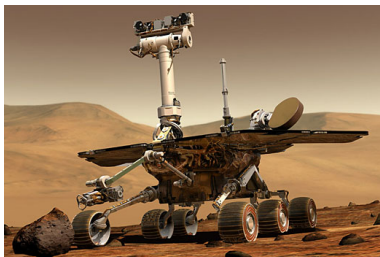




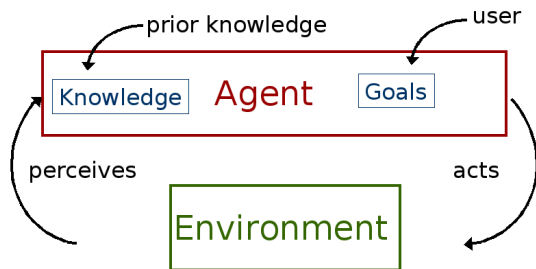
# Towards Intelligent Robots

Building highly intelligent agents is one of the most exciting and challenging tasks of artificial intelligence and computer science.

- ▶ Online
- ▶ Home
- ▶ Community
- ▶ Industrial
- ▶ Scientific
- ▶ ...



# Towards Intelligent Robots

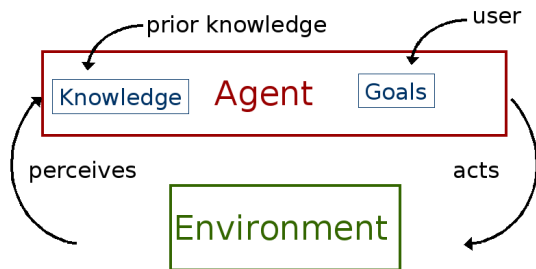


Needed: advancement of all areas of AI

- ▶ Computer Vision
- ▶ Natural Language Processing
- ▶ Knowledge Representation and Reasoning
- ▶ Machine Learning
- ▶ Robotics
- ▶ ...



# Towards Intelligent Robots



Needed: advancement of all areas of AI

- ▶ Computer Vision
- ▶ Natural Language Processing
- ▶ **Knowledge Representation and Reasoning**
- ▶ Machine Learning
- ▶ Robotics
- ▶ ...

# Knowledge Representation and Reasoning

What is Knowledge Representation and Reasoning

1. Symbolic encoding of believed facts about the environment,
2. Manipulating symbols to produce meaningful inference.

# Knowledge Representation and Reasoning

What is Knowledge Representation and Reasoning

1. Symbolic encoding of believed facts about the environment,
2. Manipulating symbols to produce meaningful inference.

Planning: an instance of KR&R in dynamic environment

- ▶ Knowledge base encodes dynamics of environment:
  - ▶ available actions (precondition/effect)
  - ▶ facts about current state
  - ▶ some goal to be achieved
- ▶ The reasoning task: find actions to achieve the goal!

## Planning: A Simple Example

A cleaning robot is able to perform the following actions:

- ▶  $move(x)$  moves the robot to object  $x$ .
- ▶  $observe(x, r)$  senses the type of object  $x$ , and discovers that the type is  $r$ .
- ▶  $throw(x)$  throws  $x$  out of the room, when  $x$  is garbage.
- ▶  $recycle(x)$  puts  $x$  into a recycle bin, when  $x$  is glass or paper.



The goal is to put all glass and paper into the recycle bin and throw away all garbage.

e.g., Suppose we have four waste objects in the room,  $w_1$  being glass,  $w_2$  being paper,  $w_3$  being garbage, and  $w_4$  being paper.

# Classical Planning

```
(define (domain cleaning-robot)
  (:predicates (at ?x) (garbage ?x) (paper ?x) ... )
  (:action recycle
    (:parameters ?x)
    (:precondition (and (at ?x) (or (glass ?x) (paper ?x))))
    (:effect (in-bin ?x)))
  (:action move ...)
  ... ..)
```

# Classical Planning

```
(define (domain cleaning-robot)
  (:predicates (at ?x) (garbage ?x) (paper ?x) ... )
  (:action recycle
    (:parameters ?x)
    (:precondition (and (at ?x) (or (glass ?x) (paper ?x))))
    (:effect (in-bin ?x)))
  (:action move ...)
  ... ..)

(define (problem cleaning-instance-1)
  (:domain home-robot)
  (:objects w1 w2 w3 w4)
  (:init (glass w1) (paper w2) (garbage w3) (paper w4))
  (:goal (and (in-bin w1) (in-bin w2) (thrown w3) (in-bin w4))))
```

# Classical Planning

```
(define (domain cleaning-robot)
  (:predicates (at ?x) (garbage ?x) (paper ?x) ... )
  (:action recycle
    (:parameters ?x)
    (:precondition (and (at ?x) (or (glass ?x) (paper ?x))))
    (:effect (in-bin ?x)))
  (:action move ...)
  ... ..)
```

```
(define (problem cleaning-instance-1)
  (:domain home-robot)
  (:objects w1 w2 w3 w4)
  (:init (glass w1) (paper w2) (garbage w3) (paper w4))
  (:goal (and (in-bin w1) (in-bin w2) (thrown w3) (in-bin w4))))
```

**A plan:** *move(w<sub>1</sub>), observe(w<sub>1</sub>, glass), recycle(w<sub>1</sub>),  
move(w<sub>2</sub>), observe(w<sub>2</sub>, paper), recycle(w<sub>2</sub>),  
move(w<sub>3</sub>), observe(w<sub>3</sub>, garbage), throw(w<sub>3</sub>),  
move(w<sub>4</sub>), observe(w<sub>4</sub>, paper), recycle(w<sub>4</sub>).*

# Classical Planning

```
(define (domain cleaning-robot)
  (:predicates (at ?x) (garbage ?x) (paper ?x) ... )
  (:action recycle
    (:parameters ?x)
    (:precondition (and (at ?x) (or (glass ?x) (paper ?x))))
    (:effect (in-bin ?x)))
  (:action move ...)
  ... ..)
```

```
(define (problem cleaning-instance-2)
  (:domain home-robot)
  (:objects w1 w2)
  (:init (garbage w1) (glass w2))
  (:goal (and (thrown w1) (in-bin w2))))
```

**A plan:** *move(w<sub>1</sub>), observe(w<sub>1</sub>, garbage), throw(w<sub>1</sub>),  
move(w<sub>2</sub>), observe(w<sub>2</sub>, glass), recycle(w<sub>2</sub>).*



# Classical Planning

```
(define (domain cleaning-robot)
  (:predicates (at ?x) (garbage ?x) (paper ?x) ... )
  (:action recycle
    (:parameters ?x)
    (:precondition (and (at ?x) (or (glass ?x) (paper ?x))))
    (:effect (in-bin ?x)))
  (:action move ...)
  ... ..)

(define (problem cleaning-instance-37251)
  (:domain home-robot)
  (:objects w1 w2 w3 ... w9453)
  (:init (paper w1) (paper w2) (garbage w3) ... (glass w9453))
  (:goal (and (in-bin w1) (in-bin w2) (thrown w3) ... (in-bin w9453))))
```

**A plan:**

```
      move( $w_1$ ), observe( $w_1$ , paper), recycle( $w_1$ ),
      move( $w_2$ ), observe( $w_2$ , paper), recycle( $w_2$ ),
      move( $w_3$ ), observe( $w_3$ , garbage), throw( $w_3$ ),
      .....
      move( $w_{9453}$ ), observe( $w_{9453}$ , glass), recycle( $w_{9453}$ ).
```

# Regularities in Plans

Although the specific instance may vary,

- ▶ all cleaning-robot problems have similar structure, and
- ▶ the solution plans contain regular patterns.

# Regularities in Plans

Although the specific instance may vary,

- ▶ all cleaning-robot problems have similar structure, and
- ▶ the solution plans contain regular patterns.

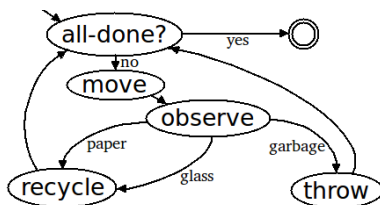
Does there exist a generalized solution that solves all such problems?

## Regularities in Plans

Although the specific instance may vary,

- ▶ all cleaning-robot problems have similar structure, and
- ▶ the solution plans contain regular patterns.

Does there exist a generalized solution that solves all such problems?



No matter how many objects and what objects, the finite-state controller above solves the problem.

## Planning with Loops: Challenges

It is intriguing to automatically generate such Finite-State Automaton-like Plans (FSA plans).

## Planning with Loops: Challenges

It is intriguing to automatically generate such Finite-State Automaton-like Plans (FSA plans).

Unfortunately, with unbounded number of objects, there can be infinitely many initial states, so plan generation and verification can be infinite too.

- ▶ When the cleaning robot wakes up, it maybe given 3 waste objects, or may be 30, or may be 3000, or ...
- ▶ Each object may be glass, paper or garbage.

# Planning with Loops: Our Approach

A simple solution: generate plans that work for finitely many cases, and hopefully it also works for all other cases!

# Planning with Loops: Our Approach

A simple solution: generate plans that work for finitely many cases, and hopefully it also works for all other cases!

In the cleaning robot example, we can generate with the assumption that there are only one or two waste objects to be handled, and verify its correctness only after we get *some* plan.



# Planning with Loops: Our Approach

A simple solution: generate plans that work for finitely many cases, and hopefully it also works for all other cases!

In the cleaning robot example, we can generate with the assumption that there are only one or two waste objects to be handled, and verify its correctness only after we get *some* plan.

The resulting FSAPLANNER alternates between a generation phase and a testing phase:

1. Generation Phase: Generate a plan that works for finitely many cases
2. Testing Phase: Test the plan to see if it also works for other cases:
  - ▶ If it works, then a solution is found;
  - ▶ Otherwise, go to Step 1, and generate a different plan.

# Generation of Plans with Loops

Search in the space of FSA plans

1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

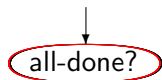
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

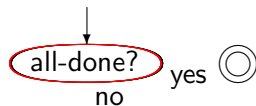
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

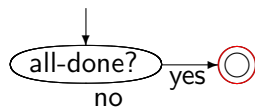
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

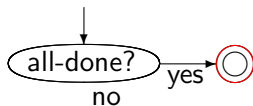
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.

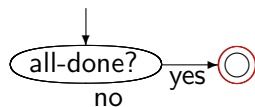




# Generation of Plans with Loops

Search in the space of FSA plans

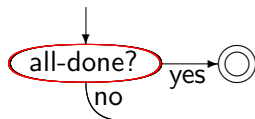
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

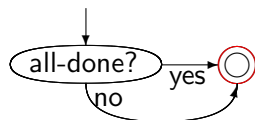
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

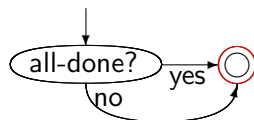
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

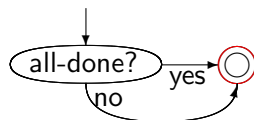
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

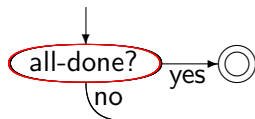
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

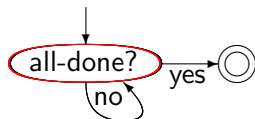
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

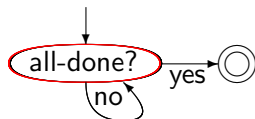
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.

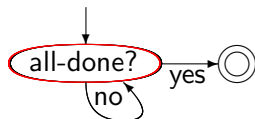




# Generation of Plans with Loops

Search in the space of FSA plans

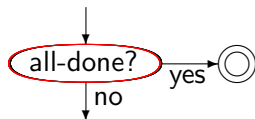
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

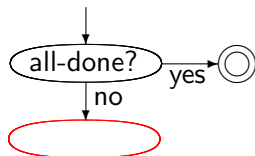
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

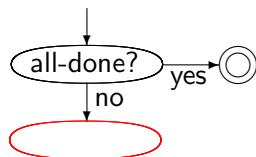
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

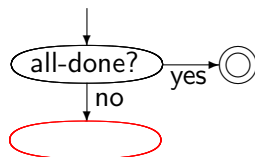
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

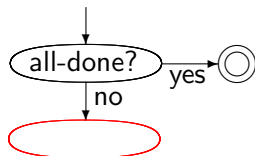
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

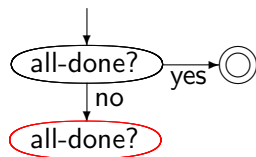
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

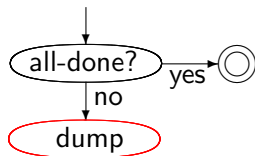
1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



# Generation of Plans with Loops

Search in the space of FSA plans

1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.

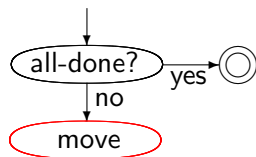




# Generation of Plans with Loops

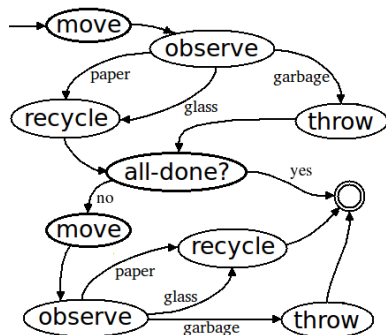
Search in the space of FSA plans

1. Start with the smallest FSA plan with only one non-final state.
2. If the current plan state is final, then the goal must be satisfied.
3. Otherwise, execute the action associated to the current plan state, non-deterministically pick one if none is associated.
4. For each possible sensing result of the action, follow the transition and update the current state. If no transition is associated to the sensing result, non-deterministically pick one for it, add new states as necessary.
5. Repeat from step 2.



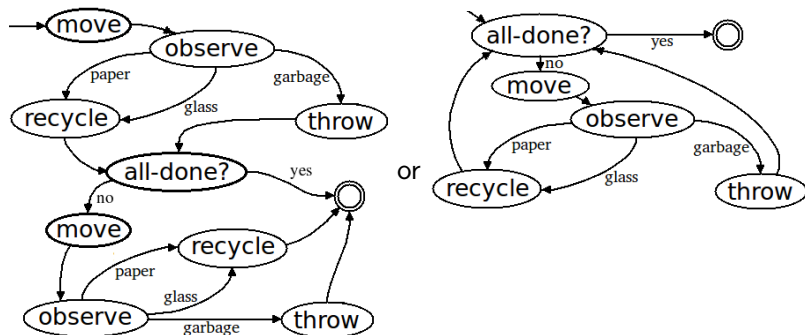
# Generation of Plans with Loops

The generation phase may return as candidate plan



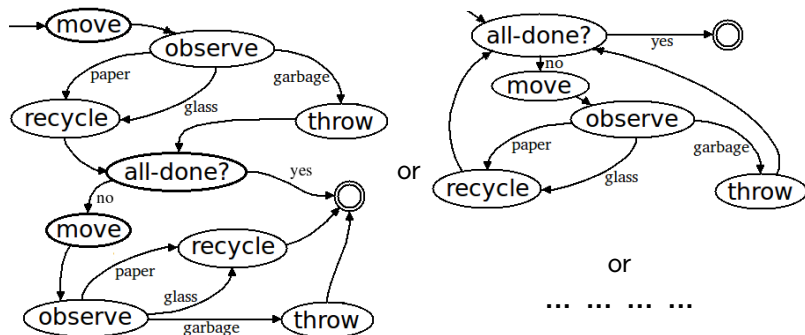
# Generation of Plans with Loops

The generation phase may return as candidate plan



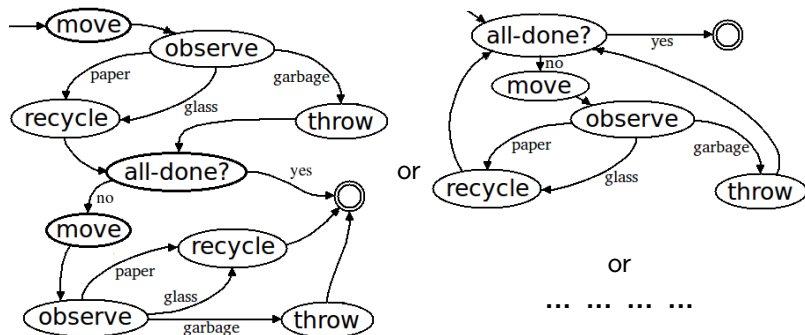
# Generation of Plans with Loops

The generation phase may return as candidate plan



# Generation of Plans with Loops

The generation phase may return as candidate plan



The testing phase is responsible for eliminating the false plans.

## Verification of Plans with Loops

Any candidate plan should be rejected in the testing phase, unless it really works in general.

# Verification of Plans with Loops

Any candidate plan should be rejected in the testing phase, unless it really works in general.

- ▶ Ideally, we need to test against all of the infinitely many possible initial states, but this is impractical.

# Verification of Plans with Loops

Any candidate plan should be rejected in the testing phase, unless it really works in general.

- ▶ Ideally, we need to test against all of the infinitely many possible initial states, but this is impractical.
- ▶ One solution: test the plan against a finite subset of them.
  - ▶ In the cleaning-robot example, we may test if the plan works for 3 and 4 waste objects too.



# Verification of Plans with Loops

Any candidate plan should be rejected in the testing phase, unless it really works in general.

- ▶ Ideally, we need to test against all of the infinitely many possible initial states, but this is impractical.
- ▶ One solution: test the plan against a finite subset of them.
  - ▶ In the cleaning-robot example, we may test if the plan works for 3 and 4 waste objects too.
- ▶ This method appears to work well, but in theory, it is unsound.

# Verification of Plans with Loops

Any candidate plan should be rejected in the testing phase, unless it really works in general.

- ▶ Ideally, we need to test against all of the infinitely many possible initial states, but this is impractical.
- ▶ One solution: test the plan against a finite subset of them.
  - ▶ In the cleaning-robot example, we may test if the plan works for 3 and 4 waste objects too.
- ▶ This method appears to work well, but in theory, it is unsound.

Can we have correctness guarantee with practical computation?

# Verification of Plans with Loops

Any candidate plan should be rejected in the testing phase, unless it really works in general.

- ▶ Ideally, we need to test against all of the infinitely many possible initial states, but this is impractical.
- ▶ One solution: test the plan against a finite subset of them.
  - ▶ In the cleaning-robot example, we may test if the plan works for 3 and 4 waste objects too.
- ▶ This method appears to work well, but in theory, it is unsound.

Can we have correctness guarantee with practical computation?

- ▶ Not in general: FSA plans can simulate Turing machines!

# Verification of Plans with Loops

Any candidate plan should be rejected in the testing phase, unless it really works in general.

- ▶ Ideally, we need to test against all of the infinitely many possible initial states, but this is impractical.
- ▶ One solution: test the plan against a finite subset of them.
  - ▶ In the cleaning-robot example, we may test if the plan works for 3 and 4 waste objects too.
- ▶ This method appears to work well, but in theory, it is unsound.

Can we have correctness guarantee with practical computation?

- ▶ Not in general: FSA plans can simulate Turing machines!
- ▶ For restricted problem classes: we can!
  - ▶ e.g. One-Dimensional Problems...

# Finite Verification of One-Dimensional Problems

One-dimensional problems: (roughly speaking)

- ▶ the only unbounded property is a non-negative integer fluent  $p$
- ▶ the only test on  $p$  is whether  $p = 0$
- ▶ the only effect on  $p$  is to increase or decrease it by one

# Finite Verification of One-Dimensional Problems

One-dimensional problems: (roughly speaking)

- ▶ the only unbounded property is a non-negative integer fluent  $p$
- ▶ the only test on  $p$  is whether  $p = 0$
- ▶ the only effect on  $p$  is to increase or decrease it by one

## Theorem (Finite Verifiability of One-Dimensional Problems)

*Given a one-dimensional problem with unbounded integer fluent  $p$ ,  $m$  other fluents taking  $l$  possible values, and a plan with  $k$  nodes,*

*if the plan works for  $p = 0, 1, \dots, k \cdot m^l + 1$ ,  
then it works for all  $p \in \mathbf{N}$ .*

# Finite Verification of One-Dimensional Problems

One-dimensional problems: (roughly speaking)

- ▶ the only unbounded property is a non-negative integer fluent  $p$
- ▶ the only test on  $p$  is whether  $p = 0$
- ▶ the only effect on  $p$  is to increase or decrease it by one

## Theorem (Finite Verifiability of One-Dimensional Problems)

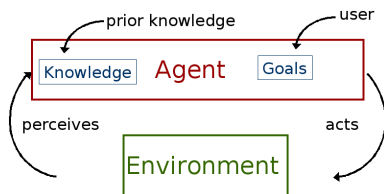
*Given a one-dimensional problem with unbounded integer fluent  $p$ ,  $m$  other fluents taking  $l$  possible values, and a plan with  $k$  nodes,*

*if the plan works for  $p = 0, 1, \dots, k \cdot m^l + 1$ ,  
then it works for all  $p \in \mathbf{N}$ .*

## Proof Sketch.

The proof uses the pigeon hole's principle over configurations in the execution trace, in a somewhat similar flavor to the proof of the pumping lemma. □

# Conclusions



1. Building intelligent agents is a long-lasting goal of AI
2. Planning, or more generally, knowledge representation and reasoning, plays a central role in such agents.
3. (Re)planning for each new problem in a problem class is often inefficient
4. Plans with loops represents a generalized solution to planning domains
5. Generating and verifying plans with loops is undecidable in general, but for restricted classes of problems, efficient algorithms exist.



## Related Research

Generation and verification of plans with loops is not an isolated KR problem. It also relates to

- ▶ program synthesis and verification
- ▶ automata theory and computability
- ▶ inductive reasoning and machine learning

## Related Research

Generation and verification of plans with loops is not an isolated KR problem. It also relates to

- ▶ program synthesis and verification
- ▶ automata theory and computability
- ▶ inductive reasoning and machine learning

Good potential for collaboration and cooperation!

## Related Research

Generation and verification of plans with loops is not an isolated KR problem. It also relates to

- ▶ program synthesis and verification
- ▶ automata theory and computability
- ▶ inductive reasoning and machine learning

Good potential for collaboration and cooperation!

**Thank you for your attention!**

Questions?