



Chain of thought

Description

A report by OpenAI on its recent release of ChatGPT o1 asserts: "Similar to how a human may think for a long time before responding to a difficult question, o1 uses a chain of thought when attempting to solve a problem. Through reinforcement learning, o1 learns to hone its chain of thought and refine the strategies it uses. It learns to recognize and correct its mistakes. It learns to break down tricky steps into simpler ones. It learns to try a different approach when the current one isn't working. This process dramatically improves the model's ability to reason." (I put to one side the slide into anthropomorphisms such as "thought," "learning," "recognizing," etc.)

I mentioned in the last post that I asked ChatGPT-o1 to devise logic-based computer code that would carry out the actions of moving a teapot and a jug on boxes on a table from one set of positions to another. I asked for this to be in the programming language Prolog, with which I was familiar from my earliest days researching AI.

After a bit of prompting ChatGPT produced some declarative code, which I was then able to test by running it on the free-to-use Prolog interpreter at <https://swish.swi-prolog.org>. The code worked, giving the correct sequence of actions to move the objects around.

At my request, ChatGPT o1 then translated the Prolog into human-readable English sentences.

So, I am here testing ChatGPT's functionality in generating and debugging working code, translating between regular English and computer code, and offering explanations of its processes. With more time and space I could have interrogated the AI on any of its outcomes.

The advantage of these translations is that the AI could use the logic representation (Prolog) to construct even more complicated worlds of objects and relationships and thereby produce yet more complicated planning processes involving many more objects, and in different domains, such as moving objects around a building construction site. As yet, ChatGPT does not run code, but I believe there are third part apps that attempt to do this.

Bibliography

- Clocksin, W. F. and C. S. Mellish. (1981), *Programming in Prolog*, Berlin: Springer-Verlag.
- Coyne, R. (1988), *Logic Models of Design*, London: Pitman.
- Kowalski, R. A. (1979), *Logic for Problem Solving*, Amsterdam: Elsevier, North Holland. ([PDF](#))

Translation of the Variant PROLOG Code into Human-Readable Text

Domain Definition: Moving Objects with One Hand (Pure Logical Version)

This domain models a scenario where an agent (such as a robot or person) manipulates objects using only one hand. The agent's task is to rearrange objects to achieve a specific goal state while adhering to certain constraints and rules.

Objects and Locations

Objects:

- Jug
- Teapot

These are the items the agent can interact with.

Locations:

- Taller Box
- Shorter Box

These boxes serve as locations where objects can be placed.

Initial State

The starting configuration of the environment is described by a list of facts:

- The **teapot** is on the **taller box**.
 - The **jug** is on the **shorter box**.
 - The agent's hand is **empty**.
-

Goal State

The desired final configuration the agent aims to achieve:

- The **jug** is on the **taller box**.
 - The **teapot** is on top of the **jug**.
 - The agent's hand is **empty**.
-

Action Definitions

The agent can perform several actions to manipulate the objects and reach the goal state.

1. Pick Up an Object

The agent picks up an object from a location.

Conditions:

- The object is on the specified location.
- The agent's hand is empty.

Effects:

- The object is no longer on that location.
- The agent's hand is no longer empty.
- The agent is now holding the object.

2. Put Down an Object

The agent puts down the object it is holding onto a location.

Conditions:

- The agent is holding the object.
- The location can support the object.
- The location does not already have an object if it cannot support both.

Effects:

- The agent is no longer holding the object.
- The object is now on the location.
- The agent's hand becomes empty.

3. Place an Object on Another Object

The agent places the object it is holding onto another object.

Conditions:

- The agent is holding the object.
- The other object can support the held object.
- The other object does not already have another object on it if it cannot support both.

Effects:

- The agent is no longer holding the object.
- The held object is now on the other object.
- The agent's hand becomes empty.

4. Move an Object

A composite action where the agent moves an object from one location to another by picking it up and then putting it down.

Properties and Constraints

These rules define what is possible within the domain regarding object placement and support.

Support Capabilities:

Boxes Can Support Objects:

Both the taller box and the shorter box can support any object (jug or teapot).

Jug Can Support Teapot:

The jug can support the teapot, meaning the teapot can be placed on the jug.

Support Limitations:

Boxes Cannot Support Both Objects Directly:

Neither the taller box nor the shorter box can support both the jug and the teapot at the same time directly.

Planning Logic

The planning logic determines the sequence of actions (plan) required to transition from the initial state to the goal state.

Plan Function:

Base Case:

If the current state is equivalent to the goal state, no further actions are needed.

Recursive Case:

If the current state is not the goal state:

Select an applicable action.

Ensure the resulting new state hasn't been visited before to avoid cycles.

Continue planning from the new state.

Helper Functions

These functions assist in the planning process by selecting actions and checking state equivalence.

Action Selection:

Determines which actions can be applied in the current state based on the conditions and effects defined.

State Equivalence Check:

Checks if two states are equivalent, ignoring the order of facts. This ensures that the planning process recognizes when the goal state has been achieved, even if the facts are listed in a different order.

State Membership Check:

Determines if a state has already been visited to prevent the planning process from entering an infinite loop.

Execution of the Plan

The execution process initiates the planning and generates the plan.

Process:

Initialization:

Start from the initial state.

Define the goal state.

Planning:

Use the planning logic to find a sequence of actions that leads from the initial state to the goal state.

Keep track of visited states to avoid redundant paths.

Result:

Obtain the sequence of actions (the plan) that the agent should perform to reach the goal state.

Example of a Possible Plan

Here is how the agent might achieve the goal state:

Pick Up the Teapot from the Taller Box:

The agent picks up the teapot from the taller box.

Place the Teapot on the Jug on the Shorter Box:

The agent places the teapot onto the jug, which is on the shorter box.

Pick Up the Jug (with Teapot) from the Shorter Box:

The agent picks up the jug (with the teapot on it) from the shorter box.

Put Down the Jug (with Teapot) on the Taller Box:

The agent puts down the jug (with the teapot on it) onto the taller box.

Understanding the Logic Flow

The agent manipulates objects while respecting the constraints:

The agent can use only one hand.

Certain locations cannot support both objects simultaneously.

By placing the teapot on the jug, the agent effectively combines them into a single unit that can be moved together.

The plan ensures the agent's hand is empty at the end, fulfilling the goal state's requirements.

Conclusion

This logical model provides a framework for the agent to plan and execute a sequence of actions to achieve a desired configuration. It abstracts the physical actions into logical steps and uses recursive planning to find a valid solution while adhering to the defined constraints and properties of the objects and locations.

Objective:

Rearrange the objects (jug and teapot) to achieve the goal state using a sequence of logical actions.

Constraints:

The agent can use only one hand.

Locations cannot support both objects directly unless specified.

The agent must avoid revisiting the same state to prevent cycles.

Process:

Action Selection:

The agent selects applicable actions based on the current state and the defined conditions.

State Transition:

The agent transitions to new states by performing actions and updating the state accordingly.

Planning:

The agent recursively searches for a sequence of actions that lead to the goal state.

Key Points to Remember:

The agent's actions are governed by predefined conditions and effects.

The planning process ensures that the agent does not get stuck in loops by keeping track of visited states.

The goal is to achieve the desired configuration while respecting all constraints.

End of Human-Readable Text

This translation explains the code in plain English, outlining the key components, actions, and logic used in the domain. It provides a clear understanding of how the agent operates within the defined environment to reach the goal state, without including any code blocks.

Notes

Here is the Prolog code it created, and from which the AI model derived the human-readable plain English procedure:

```
% =====  
% Domain Definition for Moving Objects with One Hand (Pure Logical Version)  
% =====  
  
%   
% Objects and Locations  
%   
  
% Define objects and locations as constants  
object(jug).  
object(teapot).  
location(taller_box).  
location(shorter_box).  
  
% Define initial state as a list of facts  
initial_state([  
    on(teapot, taller_box),  
    on(jug, shorter_box),  
    empty_hand  
]).  
  
% Define the goal state  
goal_state([  
    on(jug, taller_box),  
    on(teapot, jug),  
    empty_hand  
]).  
  
%   
% Action Definitions  
%   
  
% Action: pick_up(Object, From, State, NewState)  
pick_up(Object, From, State, NewState) :-  
    member(on(Object, From), State),
```

```
member(empty_hand, State),  
delete(State, on(Object, From), State1),  
delete(State1, empty_hand, State2),  
NewState = [holds(Object) | State2].
```

```
% Action: put_down(Object, To, State, NewState)  
put_down(Object, To, State, NewState) :-  
    member(holds(Object), State),  
    can_support(To, Object),  
    \+ (member(on(_, To), State), not_supports_both(To)),  
    delete(State, holds(Object), State1),  
    NewState = [on(Object, To), empty_hand | State1].
```

```
% Action: place_on(Object1, Object2, State, NewState)  
place_on(Object1, Object2, State, NewState) :-  
    member(holds(Object1), State),  
    can_support(Object2, Object1),  
    \+ (member(on(_, Object2), State), not_supports_both(Object2)),  
    delete(State, holds(Object1), State1),  
    NewState = [on(Object1, Object2), empty_hand | State1].
```

```
% Composite Action: move(Object, From, To, State, NewState)  
move(Object, From, To, State, NewState) :-  
    pick_up(Object, From, State, TempState),  
    put_down(Object, To, TempState, NewState).
```

```
%  
% Properties and Constraints  
%
```

```
% Locations can support specific objects  
can_support(taller_box, Object) :- object(Object).  
can_support(shorter_box, Object) :- object(Object).  
can_support(jug, teapot).
```

```
% Locations cannot support both the jug and teapot directly  
not_supports_both(taller_box).  
not_supports_both(shorter_box).
```

```
%  
% Planning Predicate  
%
```

```
% plan(InitialState, GoalState, VisitedStates, Plan)  
plan(State, GoalState, _, []) :-  
    equivalent_state(State, GoalState).
```



```
% execute_plan(Plan)  
% Example Query  
% execute_plan(Plan)
```

```
% To find a plan, run:  
% ?- execute_plan(Plan)  
% This will return the sequence of actions in Plan needed to reach the goal state.
```

```
% execute_plan(Plan)  
% End of File  
% execute_plan(Plan)
```

Here's ChatGPT-o1's explanation in real time of the process it went through to generate the human-readable text from the code.

Translating the code. I'm taking a closer look at the user's request to format variant code into human-readable text. This involves converting it into English sentences or pseudocode, effectively translating programming concepts into readable language.

Clarifying code sections. Taking a closer look at the code, I'll ensure human-readable explanations for each section, addressing domain relevance and providing straightforward interpretations.

Translating the model. I'm thinking through a model where an agent, like a robot or person, moves objects with one hand. This human-readable version wraps up the model's functions and the structure of the domain well.

Organizing objects. I'm curious about rearranging two kitchen items—Jug and Teapot—using Taller Box and Shorter Box, based on initial and goal states.

Defining actions. I'm working on outlining the agent's actions: picking up an object if it's present and if the hand is empty, and updating the state to reflect the hand holding the object.

Setting objects down. Agents can place objects on supported locations, clear their hands, and update state accordingly. This involves removing the object's hold status and adding it to the location's attributes.

Transferring objects. I'm exploring how an agent can place an object on another, ensuring the object is held, the surface supports it, and it's not already occupied.

Mapping object movement. I've outlined a move predicate: pick up an object at one point, transport it temporarily, and place it at another, considering the support capabilities of the containers.

Crafting the plan. I'm outlining predicates to devise plans that shift objects to achieve a goal. Actions are evaluated, and cycles prevented.

Initiating the task. Starting with the teapot on the taller box and the jug on the shorter box, the objective is to move the jug to the taller box and the teapot onto the jug, leaving the agent's hand empty.

Crafting action sequences. I'm defining actions for picking up, putting down, placing objects, and moving objects. Considering what objects support each other and ensuring no simultaneous support issues.

- The featured image was generated by WordPress to illustrate this post, and without further prompts.

Category

1. Uncategorized

Date Created

September 28, 2024

Author

rcoyne99

default watermark