IA Planning
Lecture 3: Hierarchical Planning
(HTN) Hierarchical Task Network

Halim Djerroud

LISV
Laboratoire d'ingénierie
des systèmes de Versailles

UVSQ
université PARIS-SACLAY

revision: 1.0

Course Outline

1. Introduction and motivation.
2. Principles of hierarchical decomposition
3. HTN formalization
4. Example

Context: Limitations of Classical Planning

Reminder: Classical Planning (STRIPS/PDDL)

- States defined by predicates
- Actions with preconditions and effects
- Search in state space

Limitations for Complex Problems

- **Combinatorial explosion**: state space too large
- **Lack of abstraction**: all actions at the same level
- **Modeling difficulty**: naturally hierarchical problems
- **Lack of structure**: no solution reuse

Motivating Example: Organizing a Trip

Problem: Organize a trip from Paris to Tokyo

Classical PDDL Approach:

- All actions at the same level
- buy_plane_ticket
- book_hotel
- rent_car
- visit_museum
- ...

⇒ Difficult to manage!

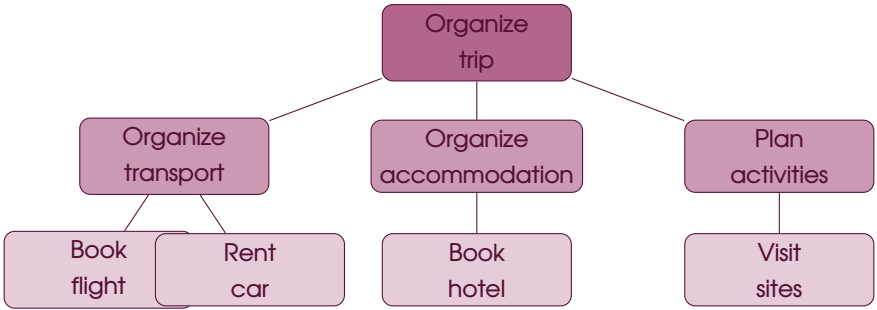
Natural (Human) Approach:

- Organize **transportation**
- Organize **accommodation**
- Plan **activities**

⇒ Hierarchical structure!

Key Idea: Decompose complex tasks into simpler subtasks

The Principle of Abstraction in Planning



- **Upper level**: abstract tasks (strategic goals)
- **Intermediate level**: decomposition into subtasks
- **Lower level**: primitive actions (executable)

Advantages of the Hierarchical Approach

Efficiency

- Reduction of search space
- Structure-guided approach
- Solution reuse

Flexibility

- Multiple decomposition methods
- Context adaptation
- Progressive refinement

Modeling

- More natural for humans
- Captures domain expertise
- Modular and reusable

Expressiveness

- Constraints between tasks
- Partial ordering
- Reasoning at different levels

HTN = *Hierarchical Task Network*

Applications of Hierarchical Planning

Robotics

- Complex navigation
- Object manipulation
- Autonomous missions

Ex: Robot assembling furniture

Logistics

- Supply chain
- Production planning
- Fleet management

Ex: Multi-warehouse delivery

AI for Games

- Agent behaviors
- Complex strategies
- Quest generation

Ex: NPCs in video games

Military Planning

- Tactical operations
- Unit coordination

Intelligent Assistants

- Calendar management
- Task organization

Comparison: Classical PDDL vs HTN

Aspect	Classical PDDL	HTN
Representation	Flat actions	Hierarchical decomposition
Search	State space	Decomposition space
Control	Generic domain	Specific methods
Efficiency	Can be slow	Structure-guided
Modeling	More abstract	Closer to the problem
Expressiveness	Goals to achieve	Tasks to accomplish

Key Takeaway: HTN and PDDL are complementary. HTN excels when the problem structure is known and hierarchical.

Why Hierarchical Planning?

1. **Complexity Reduction**
   - Recursive decomposition of complex tasks
   - Search space reduced by structure
2. **Natural Modeling**
   - Corresponds to human way of thinking
   - Captures domain expertise
3. **Reusability**
   - Reusable decomposition methods
   - Solution libraries
4. **Practical Applications**
   - Robotics, logistics, games, assistants...
   - Real-world problems with hierarchical structure

Primitive tasks vs compound tasks

Primitive tasks

- **Directly executable**
- Correspond to atomic actions
- Have preconditions and effects
- Modify the world state

Examples:

- `pick(object, location)`
- `move(location1, location2)`
- `put_down(object, location)`

Compound tasks

- **Not directly executable**
- Must be decomposed
- Represent abstract goals
- No direct effect on state

Examples:

- `transport(object, destination)`
- `organize_trip(city)`
- `make_coffee()`

A compound task decomposes into a **network of subtasks** (primitive or compound)

Decomposition methods

Definition

A **method** specifies how to decompose a compound task into subtasks.

Structure of a method:

- **Name**: method identifier
- **Task**: the compound task to decompose
- **Preconditions**: conditions to apply this method
- **Subtasks**: resulting task network
- **Constraints** (optional): ordering, resources, temporal...

Decomposition methods

Method 1: use a robot

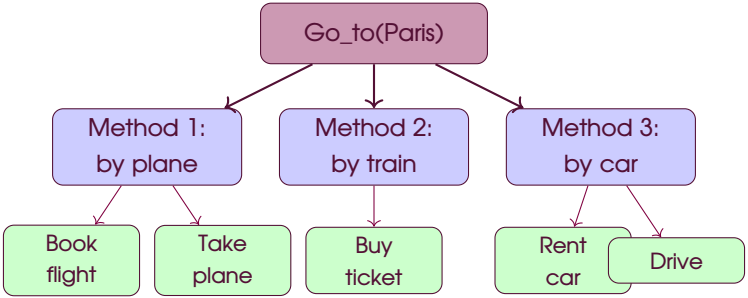
- Subtasks: `go_to(source)`, `pick(object)`, `go_to(destination)`, `put_down(object)`
- Constraint: strict ordering of these actions

Method 2: use a conveyor

- Subtasks: `place_on_conveyor(object)`, `activate_conveyor()`

Multiple methods for the same task

Principle: A compound task can have multiple alternative methods



The planner chooses which method to apply according to:

- Satisfied preconditions
- Preferences or costs
- Resource availability

Task Network

Definition

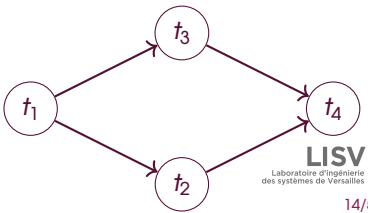
A **task network** is a set of tasks with constraints:

- Ordering constraints (before/after)
- Causal constraints (links between effects and preconditions)
- Temporal constraints (durations, deadlines)
- Resource constraints

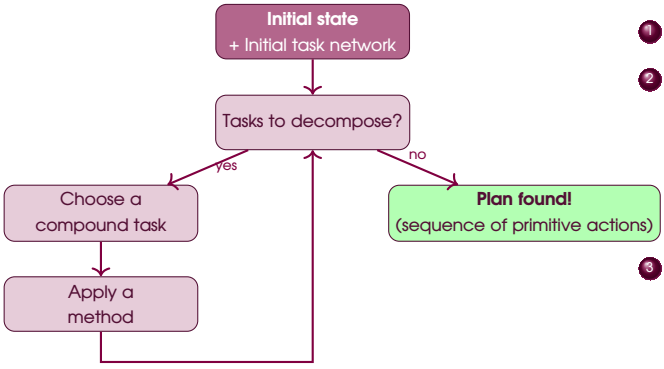
Total ordering:

Strict order: $t_1 \prec t_2 \prec t_3$

Partial ordering:



HTN planning process



1. Start from initial task network.
2. While compound tasks remain:
   - Select a compound task.
   - Choose an applicable method.
   - Replace task with its subtask network.
3. When all tasks are primitive:

   Plan found!

Complete example: Making coffee

Initial task: `make_coffee()`

Machine method:

```
Task: make_coffee()
Preconditions:
  available(machine)
Subtasks:
  1. check_water()
  2. insert_pod()
  3. start_machine()
Constraints:
  1 < 2 < 3
```

Manual method:

```
Task: make_coffee()
Preconditions:
  available(coffee_maker)
Subtasks:
  1. grind_beans()
  2. heat_water()
  3. pour_water()
Constraints:
  1 < 3, 2 < 3
```

Decomposition of `check_water()`:

```
Subtasks:
  IF water_level < threshold THEN fill_reservoir()
```

⇒ Recursive decomposition down to atomic primitive actions

Comparison: HTN vs classical planning

Aspect	Classical planning	HTN
Search space	States (world configurations)	Task decompositions
Guidance	State heuristics	Hierarchical structure
Objective	Goal state to reach	Task(s) to accomplish
Actions	All at same level	Multi-level hierarchy
Domain knowledge	Implicit (heuristics)	Explicit (methods)
Flexibility	Free exploration	Method-guided
Optimality	Can be guaranteed (A*)	Depends on methods

Complementarity

HTN and PDDL are not opposed but complementary:

- HTN: when problem structure is known
- PDDL: for freer exploration, automatic generation

Advantages and limitations of HTN

Advantages

- + Drastic reduction of search space
- + Captures domain expertise
- + Natural and intuitive modeling
- + Reuse of method libraries
- + Efficient for structured problems

Limitations

- -- Requires a priori domain knowledge
- -- Less flexible than free search
- -- Quality depends on provided methods
- -- May miss original solutions
- -- Optimality not guaranteed

Rule of thumb: Use HTN when you can encode how to solve the problem, PDDL when you want to discover how to solve it.

Principles of hierarchical decomposition

1. **Two types of tasks**
   - Primitive (executable) vs Compound (to be decomposed)
2. **Decomposition methods**
   - Specify how to transform a task into subtasks
   - Multiple possible methods per task
3. **Task network**
   - Set of tasks with ordering constraints
   - Total or partial ordering
4. **Planning process**
   - Recursive decomposition down to primitives
   - Method choice according to preconditions and context
5. **Strength of HTN**
   - Integrates domain expert knowledge
   - Efficiency through search space reduction

Scenario: Trip from Paris to Tokyo

Problem: A robot must organize a trip from Paris to Tokyo

Problem data

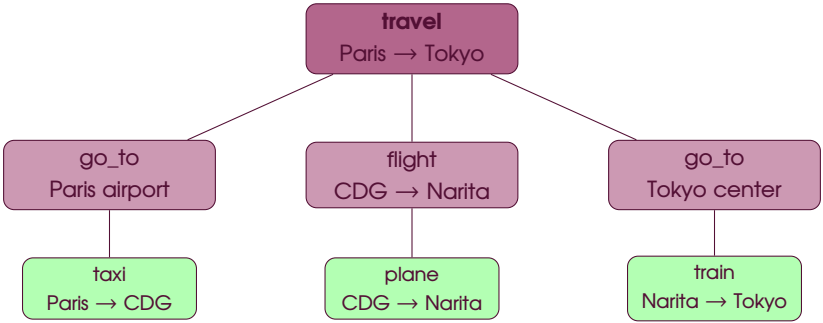
- **Initial position:** Paris
- **Final destination:** Tokyo
- **Available transportation:** taxi, plane, train
- **Intermediate locations:** Paris airport, Tokyo airport

High-level task

```
travel(Paris, Tokyo)
```

Objective: Decompose this task into executable primitive actions

Hierarchical decomposition tree



- **Burgundy nodes**: compound tasks (abstract)
- **Green nodes**: primitive actions (executable)

Introduction to PyHop

PyHop: Python Hierarchical Ordered Planner

- Created by **Dana Nau** (University of Maryland) with less than 150 lines of Python code
- Simple and pedagogical
- Operators and methods = Python functions
- States = Python data structures

Installation

`pip install pyhop` git clone https://github.com/oubiwann/pyhop.git cd pyhop sudo python setup.py install

Advantages:

- No special language to learn, ideal for prototyping and teaching
- Easy integration into Python applications

Structure of a PyHop problem

Three main components:

1. State

```
from pyhop import hop

state = hop.State('state0')
state.loc = {'me': 'Paris'}
state.cash = {'me': 500}
```

2. Operators (Actions)

```
def taxi(state, person,
         start, destination):
    if state.loc[person] == start:
        state.loc[person] = destination
        state.cash[person] -= 50
        return state
    return False
```

3. Methods (Decompositions)

```
def travel_plane(state, person,
                 city1, city2):
    airport1 = find_airport(city1)
    airport2 = find_airport(city2)
    return [
        ('go_to', person, airport1),
        ('plane', person,
         airport1, airport2),
        ('go_to', person, city2)
    ]
```

Declaration:

```
hop.declare_operators(taxi, plane)
hop.declare_methods('travel', travel_plane)
```

Definition of initial state

```python
import pyhop
# Create a state
state0 = pyhop.State('initial_state')
# Position of people
state0.loc = {
    'me': 'Paris',
}
# Available money
state0.cash = {
    'me': 1000,  # euros
}
# Distance between locations (for cost calculation)
state0.distance = {
    ('Paris', 'CDG'): 30,
    ('CDG', 'Narita'): 9700,
    ('Narita', 'Tokyo'): 60,
}
print("Initial position:", state0.loc['me'])
print("Budget:", state0.cash['me'], "euros")
```

Definition of operators (1/2)

Operator 1: Take a taxi

```python
def taxi(state, person, start, destination):
    """Operator: move by taxi"""
    # Preconditions
    if state.loc[person] != start:
        return False  # Not in the right place
    if state.cash[person] < 50:
        return False  # Not enough money

    # Effects
    state.loc[person] = destination
    state.cash[person] -= 50

    return state  # Success
```

- **Input:** current state + parameters
- **Output:** new state (or False if impossible)

Definition of operators (2/2)

Operator 2: Take a plane

```python
def plane(state, person, airport1, airport2):
    """Operator: take a flight"""
    # Preconditions
    if state.loc[person] != airport1:
        return False

    distance = state.distance.get((airport1, airport2), 0)
    cost = distance * 0.15  # 0.15 euro/km

    if state.cash[person] < cost:
        return False

    # Effects
    state.loc[person] = airport2
    state.cash[person] -= cost

    return state
```

Operator 3: Take a train (similar)

Definition of methods (1/3)

Method 1: Move locally (same city)

```
def travel_local(state, person, city1, city2):
    """Method: movement within the same city"""
    # Application condition
    if city1 == city2:
        # Already at destination
        return []

    # Decomposition: use a taxi
    return [('taxi', person, city1, city2)]
```

Return:

- Empty list `[]`: task already accomplished
- List of tasks: subtasks to accomplish
- `False`: method not applicable

Definition of methods (2/3)

Method 2: Travel by plane

```python
def travel_plane(state, person, city1, city2):
    """Method: travel by plane between cities"""
    # Find airports
    airports = {
        'Paris': 'CDG',
        'Tokyo': 'Narita',
        'NewYork': 'JFK',
    }
    airport1 = airports.get(city1)
    airport2 = airports.get(city2)

    if not airport1 or not airport2:
        return False  # No airport available
    # Decomposition into subtasks
    return [
        ('travel', person, city1, airport1),  # Go to airport
        ('plane', person, airport1, airport2), # Take plane
        ('travel', person, airport2, city2),   # Go to center
    ]
```

Definition of methods (3/3)

Declaration of methods in PyHop

```
# Declare operators
pyhop.declare_operators(taxi, plane, train)
print("Declared operators:", pyhop.operators)

# Declare methods for the 'travel' compound task
pyhop.declare_methods(
    'travel',           # Compound task name
    travel_local,       # Method 1: local movement
    travel_plane,       # Method 2: plane travel
)

print("Declared methods:", pyhop.methods)
```

Important: PyHop tries methods in declaration order

Planner execution

Launch planning

```
# Define the task to accomplish
initial_task = [('travel', 'me', 'Paris', 'Tokyo')]

# Execute the planner
plan = pyhop.pyhop(
    state0,           # Initial state
    initial_task,     # Task(s) to accomplish
    verbose=1         # Display details
)

# Display result
if plan:
    print("\n=== PLAN FOUND ===")
    for i, action in enumerate(plan, 1):
        print(f"{i}. {action}")
else:
    print("No plan found")
```

Planning result (1/2)

Planner output (verbose=1):

```
Searching for a plan for: [('travel', 'me', 'Paris', 'Tokyo')]

Trying method: travel_plane
  Decomposition into: [
    ('travel', 'me', 'Paris', 'CDG'),
    ('plane', 'me', 'CDG', 'Narita'),
    ('travel', 'me', 'Narita', 'Tokyo')
  ]

  Searching for a plan for: ('travel', 'me', 'Paris', 'CDG')
  Trying method: travel_local
    Decomposition into: [('taxi', 'me', 'Paris', 'CDG')]
    Applying operator: taxi
    >>> Success

  Applying operator: plane
  >>> Success
```

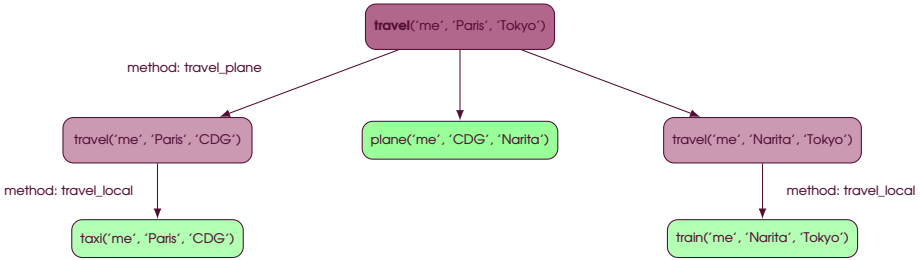
Planning result (2/2)

Planner output (verbose=1):

```
  Searching for a plan for: ('travel', 'me', 'Narita', 'Tokyo')
  Trying method: travel_local
    Decomposition into: [('train', 'me', 'Narita', 'Tokyo')]
    Applying operator: train
    >>> Success

=== FINAL PLAN ===
1. ('taxi', 'me', 'Paris', 'CDG')
2. ('plane', 'me', 'CDG', 'Narita')
3. ('train', 'me', 'Narita', 'Tokyo')
```

Decomposition trace



Process:

1. Recursive decomposition by methods
2. Stop when all tasks are primitive
3. Primitive actions form the final plan

Advantages of PyHop

Strengths

- + Simplicity: pure Python code
- + Flexibility: Python structures
- + Rapid prototyping
- + Easy to learn
- + Easy integration
- + Simple debugging

Limitations

- -- Simple depth-first search
- -- No heuristics
- -- No plan optimization
- -- Total order only
- -- No complete backtracking
- -- Limited performance

Recommended usage: Prototyping, teaching, simple applications. For production, consider SHOP2/3 or PANDA.

Comparison: PyHop vs HDDL

Aspect	PyHop	HDDL (IPC Standard)
Language	Pure Python	Extended PDDL syntax
States	Python dictionaries	Logical predicates
Operators	Python functions	PDDL declarations
Methods	Python functions	PDDL declarations
Learning curve	Very simple	Medium
Portability	Python only	Multi-planner
Usage	Prototyping, teaching	Competitions, research

Equivalent code example:

PyHop:

```python
def taxi(s, p, a, b):
    if s.loc[p] == a:
        s.loc[p] = b
        return s
    return False
```

HDDL:

```
(:action taxi
  :parameters (?p ?a ?b)
  :precondition (at ?p ?a)
  :effect (and (at ?p ?b)
               (not (at ?p ?a))))
```

Possible extensions

Improvements to the travel domain:

1. **Add constraints**
   - Travel time
   - Hotel reservations
   - Visas and documents
2. **More transportation modes**
   - Bus, subway, rental car
   - Carpooling, bicycle
3. **Optimization**
   - Minimize cost
   - Minimize time
   - User preferences
4. **Error handling**
   - Canceled flights
   - Delays
   - Alternative plans

Scenario: Table preparation by a robot

Problem: A robot must prepare a dining table and then clear it.

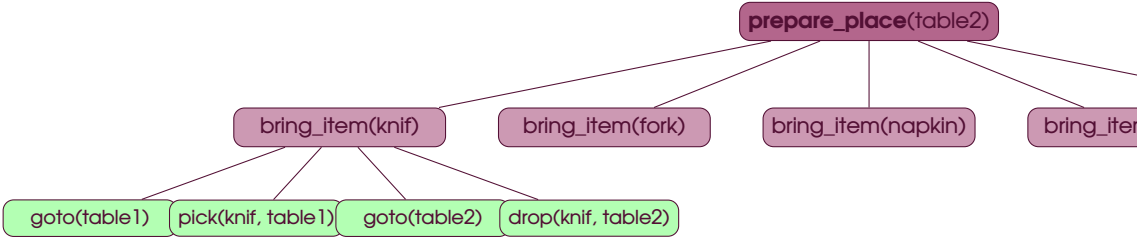
Problem data

- **Tables:** table1 (storage), table2 (to prepare)
- **Objects:** knife, fork, napkin, plate, cup
- **Robot capabilities:**
  - Move between tables
  - Pick and drop objects

High-level task

`prepare_place(table2)` then `clear_place(table2, table1)`

Hierarchical decomposition of the task



- Compound tasks: abstract actions (prepare, bring)
- Primitive actions: executable robot commands

PyHop domain: states and data

State representation

- Tables contain a set of objects with quantities
- Robot has:
  - Position: `robot_at`
  - Gripper status: `gripper_free`
  - Currently held object: `holding`

Initial state

- `robot_at = "table1"`
- `on_table1 = {knif:10, fork:10, ...}`
- `on_table2 = {knif:0, fork:0, ...}`

Definition of primitive operators(1/2)

```python
def goto(state, table):
    state.robot_at = table
    return state

def pick(state, item, table):
    if not state.gripper_free:
        return False
    table_dict = getattr(state, f'on_{table}')
    if state.robot_at == table and table_dict[item] > 0:
        table_dict[item] -= 1
        state.gripper_free = False
        state.holding = item
        return state
    return False
```

Definition of primitive operators(2/2)

```
1
2  def drop(state, item, table):
3      if state.gripper_free:
4          return False
5      table_dict = getattr(state, f'on_{table}')
6      if state.robot_at == table and state.holding == item:
7          table_dict[item] += 1
8          state.gripper_free = True
9          state.holding = None
10         return state
11     return False
```

Declaration of operators

```
1 hop.declare_operators(goto, pick, drop)
2 print(hop.get_operators())
```

Operators represent the robot's physical capabilities

- `goto`: move between tables
- `pick`: take an object
- `drop`: place an object

Compound methods

Method 1: bring an item from one table to another

```
def bring_item(state, item, table_src, table_dest):
    return [
        ('goto', table_src),
        ('pick', item, table_src),
        ('goto', table_dest),
        ('drop', item, table_dest)
    ]
```

Method 2: prepare a full table

```
def prepare_place(state, table_dest):
    return [
        ('bring_item', 'knif', 'table1', table_dest),
        ('bring_item', 'fork', 'table1', table_dest),
        ('bring_item', 'napkin', 'table1', table_dest),
        ('bring_item', 'plate', 'table1', table_dest),
        ('bring_item', 'cup', 'table1', table_dest)
    ]
```

Method: clearing the table

```
1  def clear_place(state, table_to_clean, table_dest):
2      lst = []
3      table_dict = getattr(state, f'on_{table_to_clean}')
4      for item in ['knif', 'fork', 'napkin', 'plate', 'cup']:
5          if table_dict[item] > 0:
6              lst.append(('bring_item', item, table_to_clean, table_dest))
7      return lst
8
9  hop.declare_methods('bring_item', bring_item)
10 hop.declare_methods('prepare_place', prepare_place)
11 hop.declare_methods('clear_place', clear_place)
```

Planning and execution

```
plan1 = hop.plan(state1, [('prepare_place', 'table2')],
                 hop.get_operators(), hop.get_methods(),
                 verbose=1)
execute_plan(state1, plan1)
```

Execution function

```
def execute_plan(state, plan):
    for action in plan:
        op_name, *args = action
        operator = hop.get_operators().get(op_name)
        new_state = operator(state, *args)
    return state
```

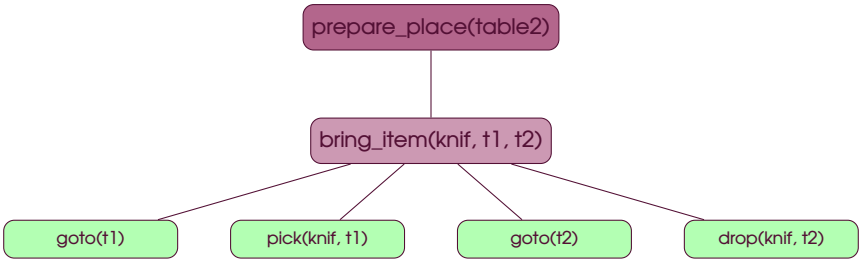
Example of a generated plan

```
Searching for plan for: [('prepare_place', 'table2')]
Trying method: prepare_place
  -> [('bring_item', 'knif', 'table1', 'table2'),
      ('bring_item', 'fork', 'table1', 'table2'), ...]

Decomposing bring_item(knif):
  ('goto', 'table1')
  ('pick', 'knif', 'table1')
  ('goto', 'table2')
  ('drop', 'knif', 'table2')

=== FINAL PLAN ===
1. ('goto', 'table1')
2. ('pick', 'knif', 'table1')
3. ('goto', 'table2')
4. ('drop', 'knif', 'table2')
...
```

Visualization of the decomposition tree



Execution results

After executing the plan:

- Table1 loses one of each item
- Table2 gains one of each item

```
Table1: {'knif': 9, 'fork': 9, 'napkin': 9, 'plate': 9, 'cup': 9}
Table2: {'knif': 1, 'fork': 1, 'napkin': 1, 'plate': 1, 'cup': 1}
```

The same logic is used for the reverse task `clear_place(table2, table1)`.

Summary and discussion

What we have done:

1. Defined a simple robotic domain (tables, objects, robot)
2. Created primitive operators (`goto`, `pick`, `drop`)
3. Defined hierarchical methods:
   - `bring_item`, `prepare_place`, `clear_place`
4. Used PyHop to generate and execute plans

Educational goals:

- Illustrate HTN (Hierarchical Task Network) decomposition
- Show how planning produces executable robot sequences
- Provide a bridge between symbolic AI and robotics

Conclusion

What we have seen:

1. **Hierarchical modeling**
   - Abstract task: `travel(origin, destination)`
   - Decomposition into subtasks
   - Primitive actions: taxi, plane, train

2. **Implementation with PyHop**
   - States = Python structures
   - Operators = functions with preconditions/effects
   - Methods = functions returning subtasks

3. **Planning process**
   - Recursive decomposition
   - Selection of applicable methods
   - Construction of final plan