# AI Planning

## Probabilistic AI and Reasoning - Lecture 7

Issa Hanou

Delft University of Technology

September 23, 2024

These lecture slides are inspired by the lectures on AI Planning by Christian Muise (Queen's University)

# Who am I?

- Issa Hanou
- PhD candidate Algorithmics group
- Working on Planning and Scheduling for Railway logistics

Figure: Shunting yard in the Netherlands.

# Outline



PDDL: Planning Domain Definition Language

# Story so far...

- Search Problems
- Logical Reasoning Problems
- Constraint Satisfaction Problems
- Bayesian Networks
- Utility

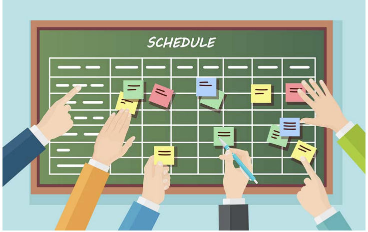
# Story so far...

- Search Problems
- Logical Reasoning Problems
- Constraint Satisfaction Problems
- Bayesian Networks
- Utility
- → Time component
- → Real-World Problems

# What is Planning?

What do you think planning is?

# Examples of Planning

# What is Planning?

*Planning is the art and practice of thinking before acting.*
*–Patrik Haslum*

# Learning Objectives

1. Explain what is planning
2. Explain different approaches to finding plans
3. Read planning problems in the Planning Domain Definition Language (PDDL)
4. Model a problem in PDDL terms (semantically, not syntactically)
5. Reason whether a model or plan is correct and effective

# What is a Plan?

(pickup robot2 bowl)

(putdown robot2 bowl ontable)

(scoop robot1 corn)

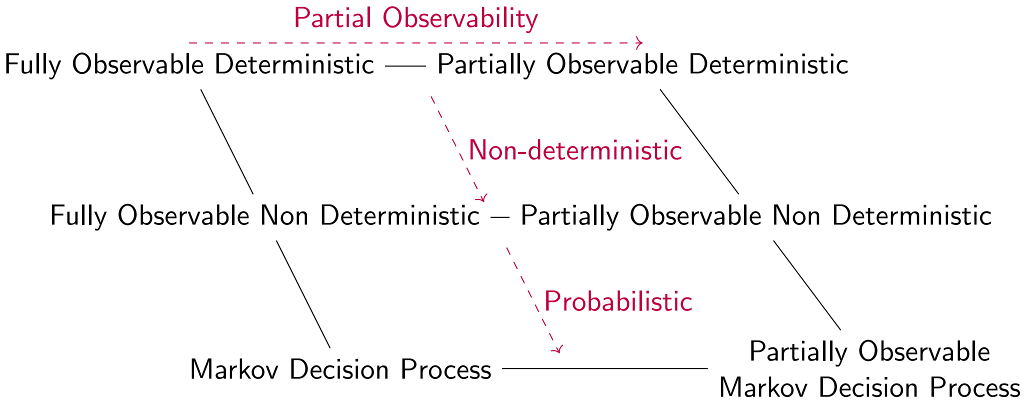
(putdown robot1 corn inbowl)

(pickup robot2 mushrooms)

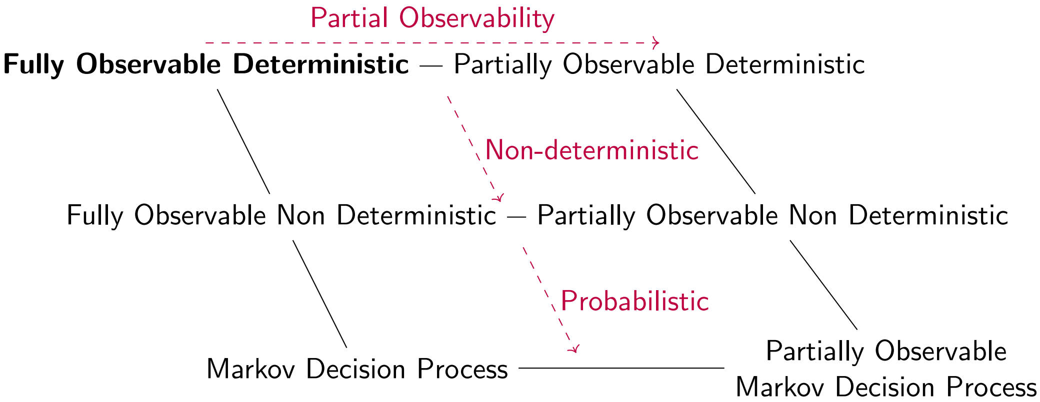
(putdown robot2 mushrooms inbowl)

...

# Types of Planning



# Types of Planning



# Modeling vs Solving vs Executing

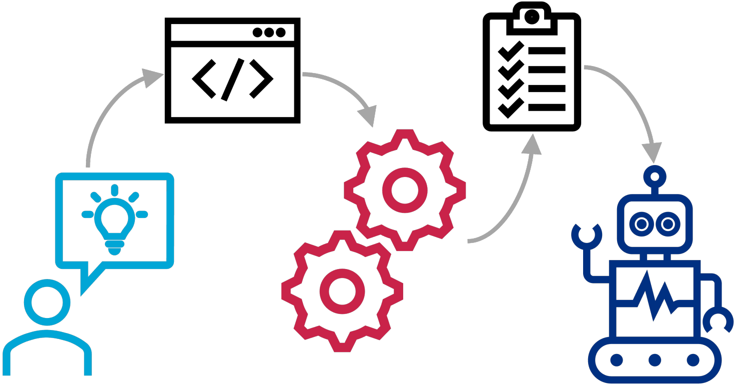


Figure: Planning overview.

# Modeling vs Solving vs Executing

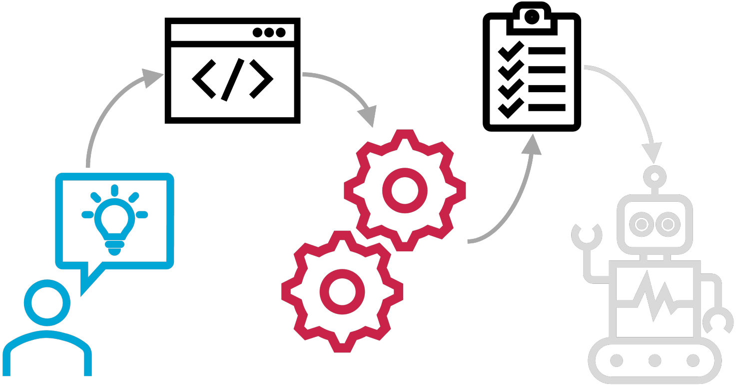


Figure: Planning overview - focus in lecture.

# Domain-Independent Planning

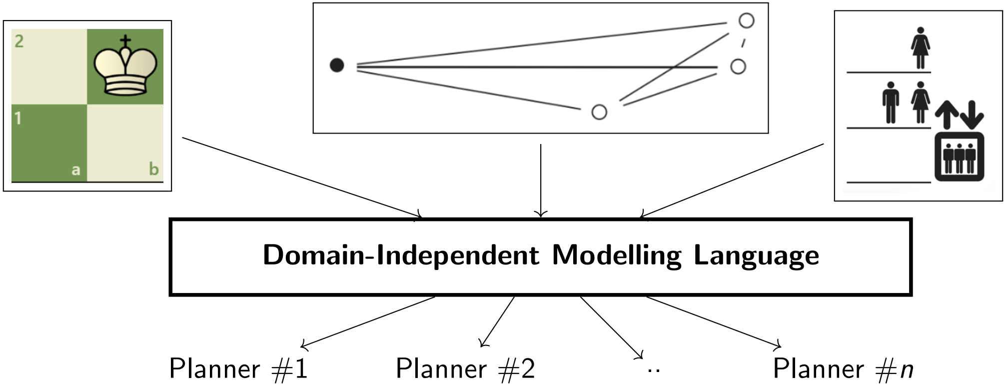
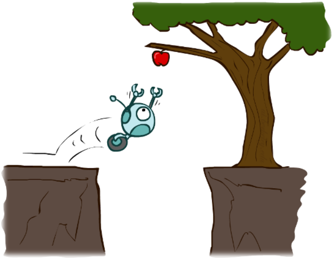


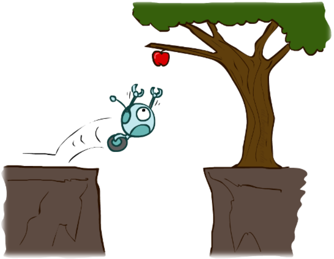

Figure: Domain-Independent Planning.

# Search

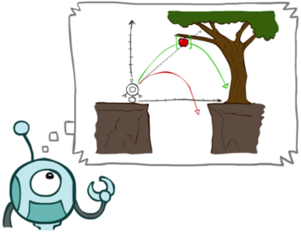


(a) Reflex agent.

# Search

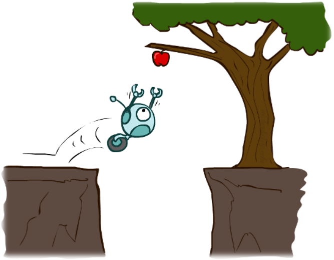


(a) Reflex agent.

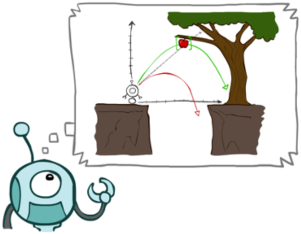


(b) Planning agent.

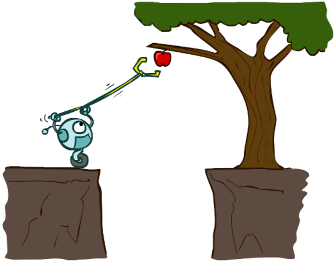
# Search



(a) Reflex agent.



(b) Planning agent.



(c) Agent with a plan.

# Search Problem

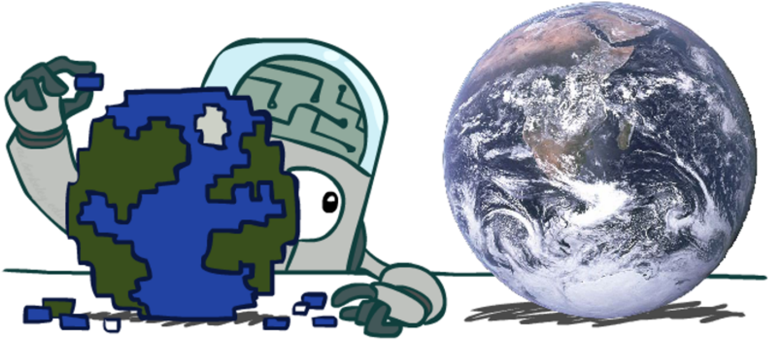
## Definition

A *search problem* consists of:

- A state space
- A successor function
- A start state and goal test

A *solution* is a sequence of actions (a plan) that transforms the start state into a goal state

# Modeling Search Problems



# Example Search Problems

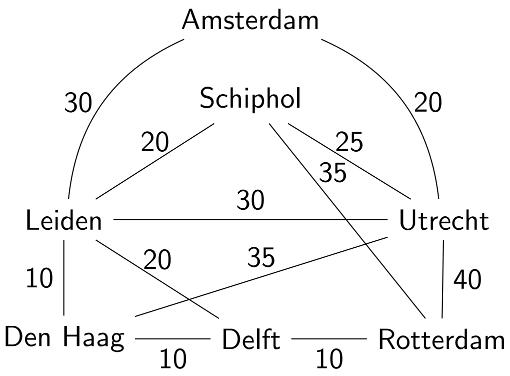



Figure: Partial railway network Netherlands.

Shows time between to travel between two cities

# Example Search Problems

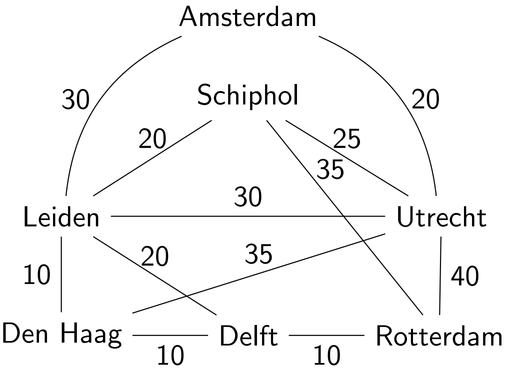



Figure: Partial railway network Netherlands.

Shows time between to travel between two cities

Realistic model?

# Example Search Problems

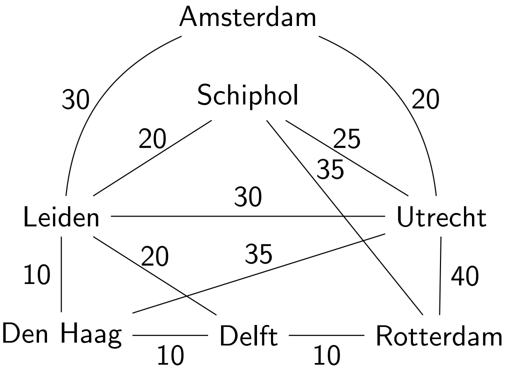



Figure: Partial railway network Netherlands.

Shows time between to travel between two cities

Realistic model?

**Goal-dependent:**

# Example Search Problems

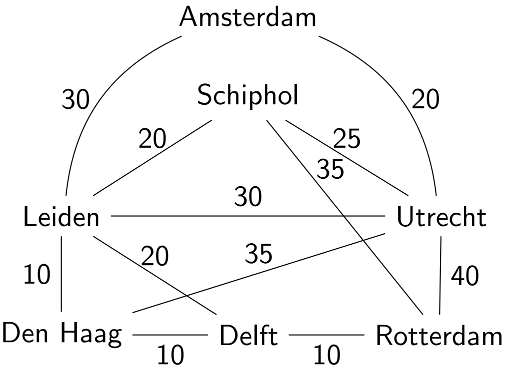



Figure: Partial railway network Netherlands.

Shows time between to travel between two cities

Realistic model?

**Goal-dependent:** Shortest path Delft to Utrecht

**Other factors?**

# Example Search Problems

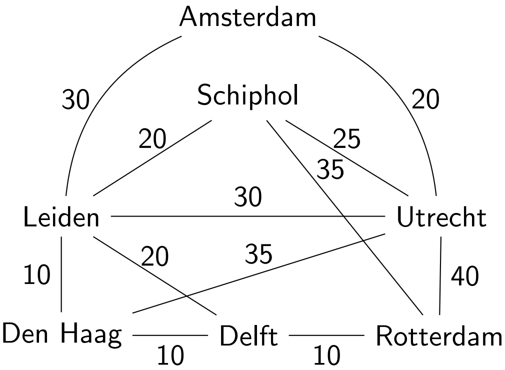



Figure: Partial railway network Netherlands.

Shows time between to travel between two cities

Realistic model?

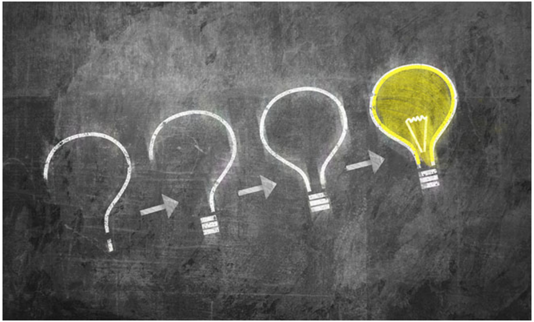
**Goal-dependent:** Shortest path Delft to Utrecht

**Other factors?**

- Transfer time
- Timetables
- (Expected) Train capacity
- ...

# Solving a Planning Problem

## Questions so far?



# State Space Size

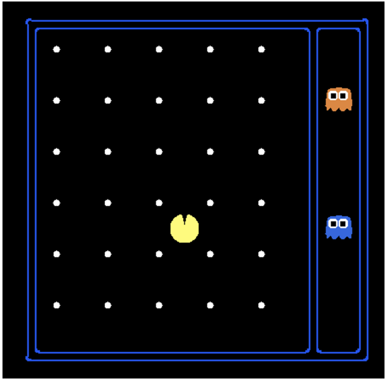


Figure: PacMan problem example with a $12 \times 10$ grid.

**World state**

- Agent positions: 120
- Food count: 30
- Ghost positions: 12
- Agent facing: North, East, South, West

# State Space Size

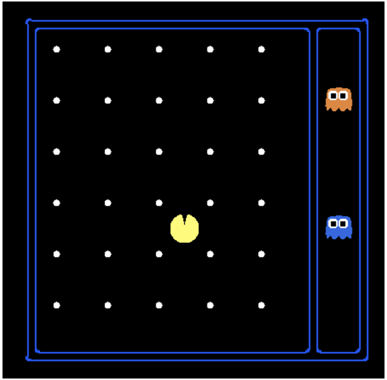


Figure: PacMan problem example with a 12 × 10 grid.

**World state**

- Agent positions: 120
- Food count: 30
- Ghost positions: 12
- Agent facing: North, East, South, West

**How many states?**

# State Space Size

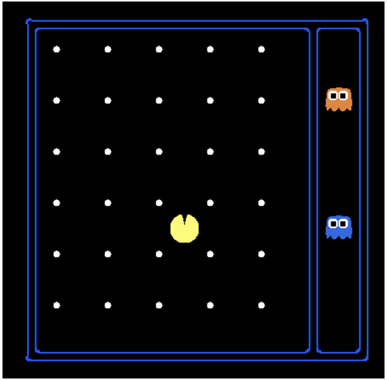


Figure: PacMan problem example with a $12 \times 10$ grid.

**World state**

- Agent positions: 120
- Food count: 30
- Ghost positions: 12
- Agent facing: North, East, South, West

**How many states?**

→ Total number of states:
$120 * (2^{30}) * (12^2) * 4 = 7.4 \cdot 10^{13}$

# State Space Size

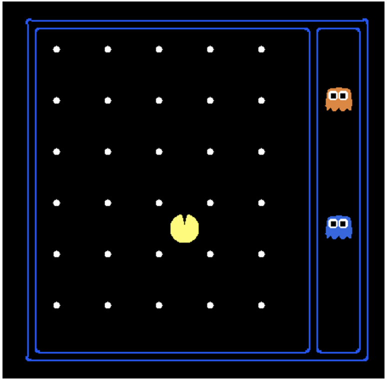


Figure: PacMan problem example with a 12 × 10 grid.

**World state**

- Agent positions: 120
- Food count: 30
- Ghost positions: 12
- Agent facing: North, East, South, West

**How many states if you only want to avoid ghosts?**

# State Space Size

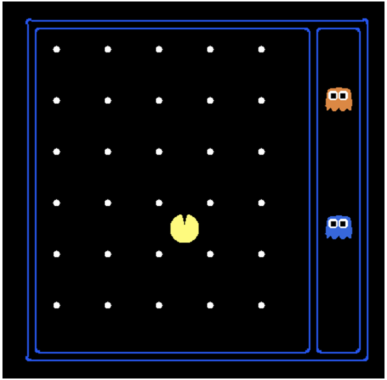


Figure: PacMan problem example with a $12 \times 10$ grid.

**World state**

- Agent positions: 120
- Food count: 30
- Ghost positions: 12
- Agent facing: North, East, South, West

**How many states if you only want to avoid ghosts?**

→ Total number of states: $120 * (12^2) * 4 = 69120$

# State Space Example

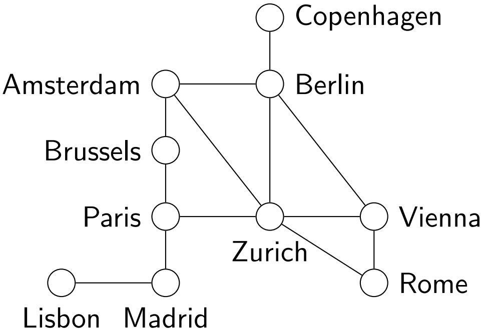



Figure: Logistics problem in Western Europe.

- Transport packages
- Take trains between connected cities
- Can fly longer distances

**What does state space look like?**

# State Space Example Answers

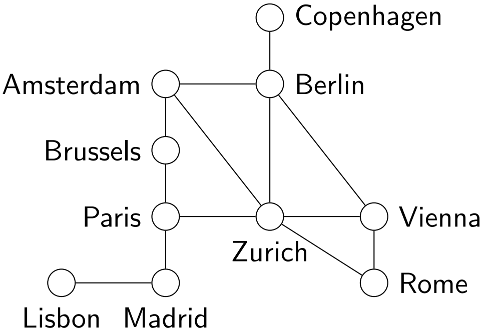



Figure: Logistics problem in Western Europe.

**State space components**

- Connected cities
- Location per city
- Package objects (in locations)
- Set of trains and airplanes

# State Space Example Answers

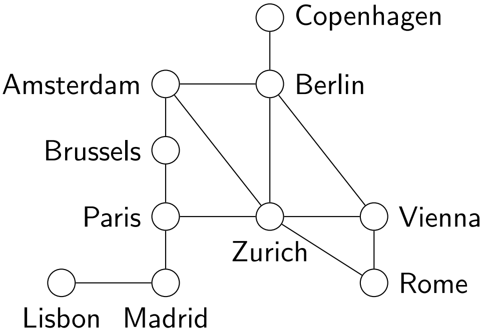



Figure: Logistics problem in Western Europe.

**State space components**

- Connected cities
- Location per city
- Package objects (in locations)
- Set of trains and airplanes

`connected` **in state space?**

# State Space Example Answers

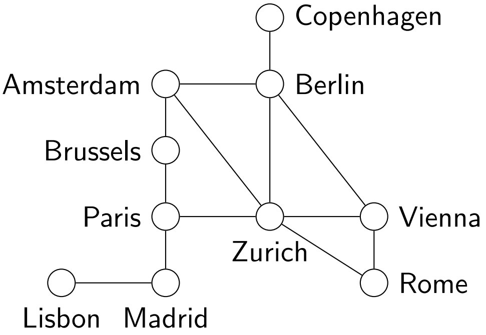



Figure: Logistics problem in Western Europe.

**State space components**

- Connected cities
- Location per city
- Package objects (in locations)
- Set of trains and airplanes

`connected` **in state space?**

→ State space versus successor function

# State Space Graph

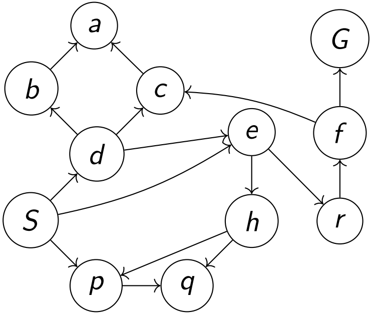



Figure: State space graph for search problem: find path from $S$ to $G$.

# Search Tree

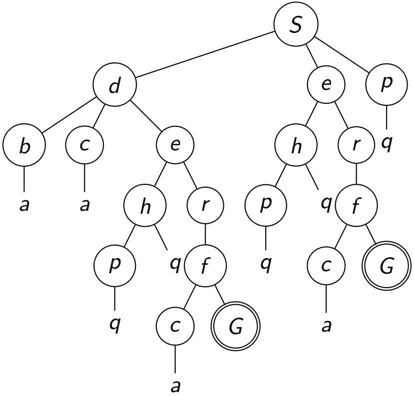
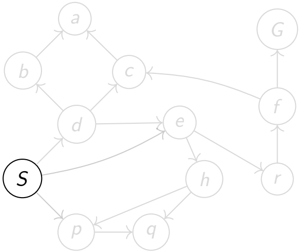
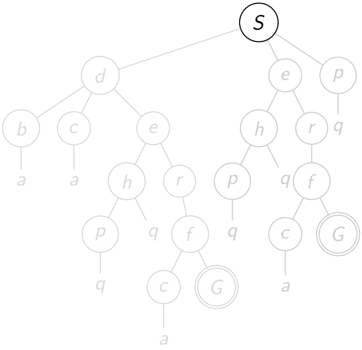



Figure: Search tree for previous search problem.

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \to d$

$S \to d \to b$

$S \to d \to b \to a$

$S \to d \to c \to a$

$S \to d \to e$

$S \to d \to e \to h \to \ldots$

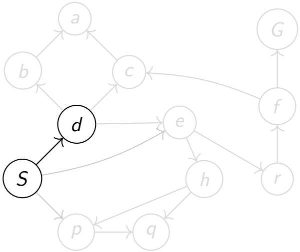
$S \to d \to e \to r$

$S \to d \to e \to r \to f$

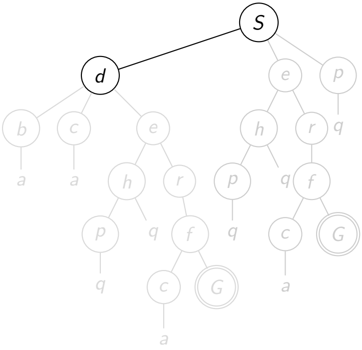
$S \to d \to e \to r \to f \to c$

$S \to d \to e \to r \to f \to G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \rightarrow d$

$S \rightarrow d \rightarrow b$

$S \rightarrow d \rightarrow b \rightarrow a$

$S \rightarrow d \rightarrow c \rightarrow a$

$S \rightarrow d \rightarrow e$

$S \rightarrow d \rightarrow e \rightarrow h \rightarrow \ldots$

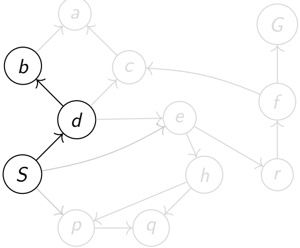
$S \rightarrow d \rightarrow e \rightarrow r$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f$

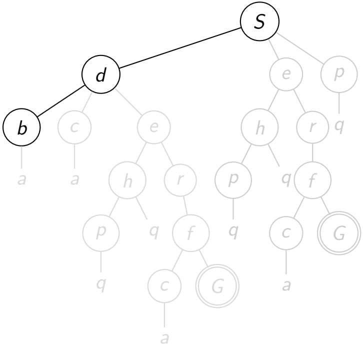
$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow c$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \rightarrow d$

$S \rightarrow d \rightarrow b$

$S \rightarrow d \rightarrow b \rightarrow a$

$S \rightarrow d \rightarrow c \rightarrow a$

$S \rightarrow d \rightarrow e$

$S \rightarrow d \rightarrow e \rightarrow h \rightarrow \ldots$

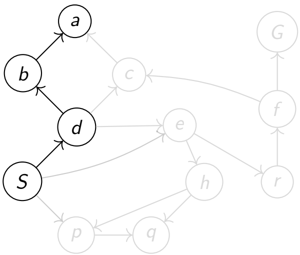
$S \rightarrow d \rightarrow e \rightarrow r$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f$

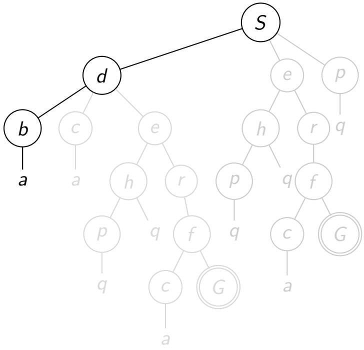
$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow c$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \rightarrow d$

$S \rightarrow d \rightarrow b$

$S \rightarrow d \rightarrow b \rightarrow a$

$S \rightarrow d \rightarrow c \rightarrow a$

$S \rightarrow d \rightarrow e$

$S \rightarrow d \rightarrow e \rightarrow h \rightarrow \ldots$

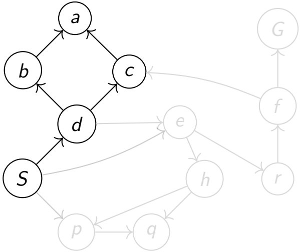
$S \rightarrow d \rightarrow e \rightarrow r$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f$

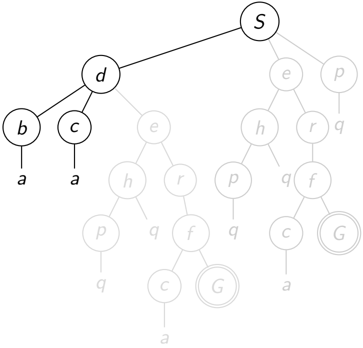
$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow c$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \to d$

$S \to d \to b$

$S \to d \to b \to a$

$S \to d \to c \to a$

$S \to d \to e$

$S \to d \to e \to h \to \ldots$

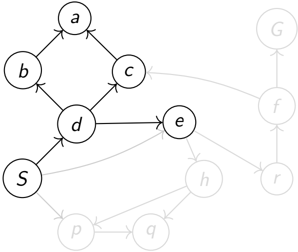
$S \to d \to e \to r$

$S \to d \to e \to r \to f$

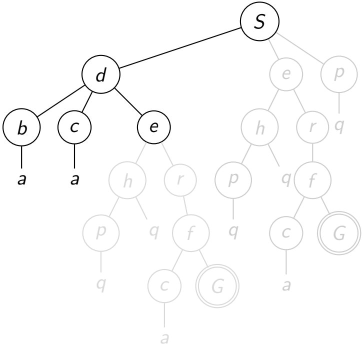
$S \to d \to e \to r \to f \to c$

$S \to d \to e \to r \to f \to G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \to d$

$S \to d \to b$

$S \to d \to b \to a$

$S \to d \to c \to a$

$S \to d \to e$

$S \to d \to e \to h \to \dots$

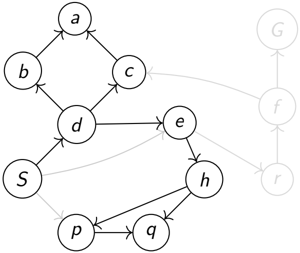
$S \to d \to e \to r$

$S \to d \to e \to r \to f$

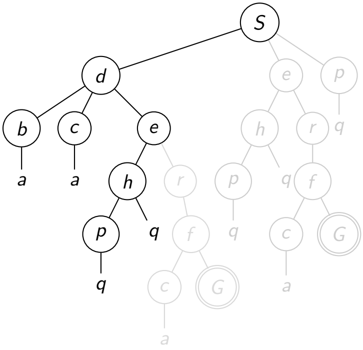
$S \to d \to e \to r \to f \to c$

$S \to d \to e \to r \to f \to G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \rightarrow d$

$S \rightarrow d \rightarrow b$

$S \rightarrow d \rightarrow b \rightarrow a$

$S \rightarrow d \rightarrow c \rightarrow a$

$S \rightarrow d \rightarrow e$

$S \rightarrow d \rightarrow e \rightarrow h \rightarrow \ldots$

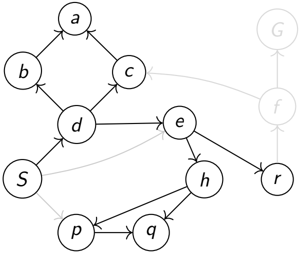
$S \rightarrow d \rightarrow e \rightarrow r$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f$

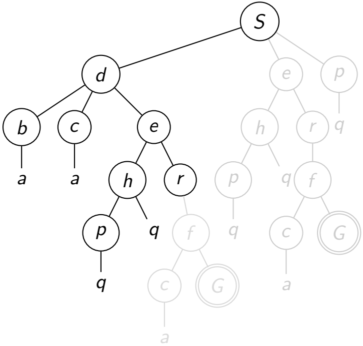
$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow c$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \to d$

$S \to d \to b$

$S \to d \to b \to a$

$S \to d \to c \to a$

$S \to d \to e$

$S \to d \to e \to h \to \ldots$

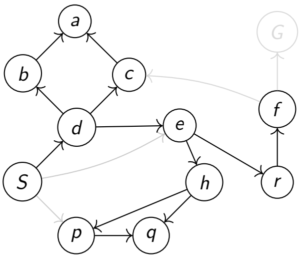
$S \to d \to e \to r$

$S \to d \to e \to r \to f$

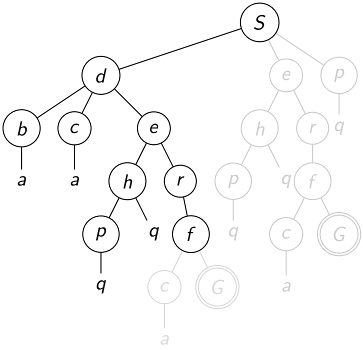
$S \to d \to e \to r \to f \to c$

$S \to d \to e \to r \to f \to G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \rightarrow d$

$S \rightarrow d \rightarrow b$

$S \rightarrow d \rightarrow b \rightarrow a$

$S \rightarrow d \rightarrow c \rightarrow a$

$S \rightarrow d \rightarrow e$

$S \rightarrow d \rightarrow e \rightarrow h \rightarrow \ldots$

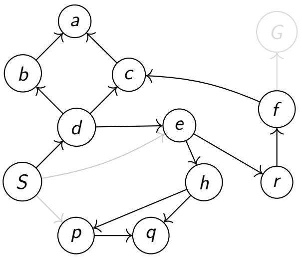
$S \rightarrow d \rightarrow e \rightarrow r$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f$

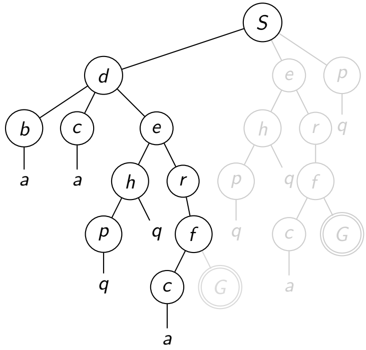
$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow c$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \rightarrow d$

$S \rightarrow d \rightarrow b$

$S \rightarrow d \rightarrow b \rightarrow a$

$S \rightarrow d \rightarrow c \rightarrow a$

$S \rightarrow d \rightarrow e$

$S \rightarrow d \rightarrow e \rightarrow h \rightarrow \ldots$

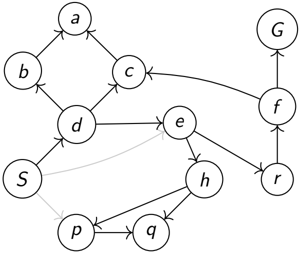
$S \rightarrow d \rightarrow e \rightarrow r$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f$

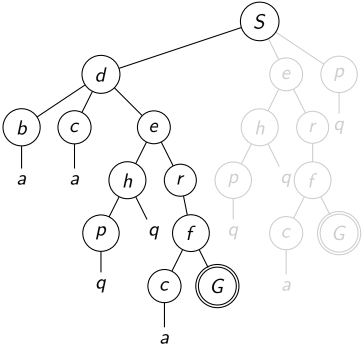
$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow c$

$S \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow G$

# Example Tree Search




(a) State space graph.




(b) Search tree.

$S$

$S \to d$

$S \to d \to b$

$S \to d \to b \to a$

$S \to d \to c \to a$

$S \to d \to e$

$S \to d \to e \to h \to \ldots$

$S \to d \to e \to r$

$S \to d \to e \to r \to f$

$S \to d \to e \to r \to f \to c$

$S \to d \to e \to r \to f \to G$

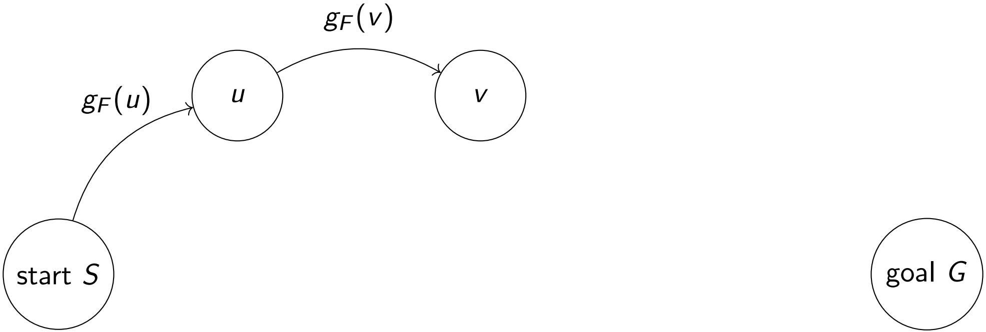
# Forward Search


$g_F(n)$ is cost of best-known path from *start* to $n$.

# Forward Search

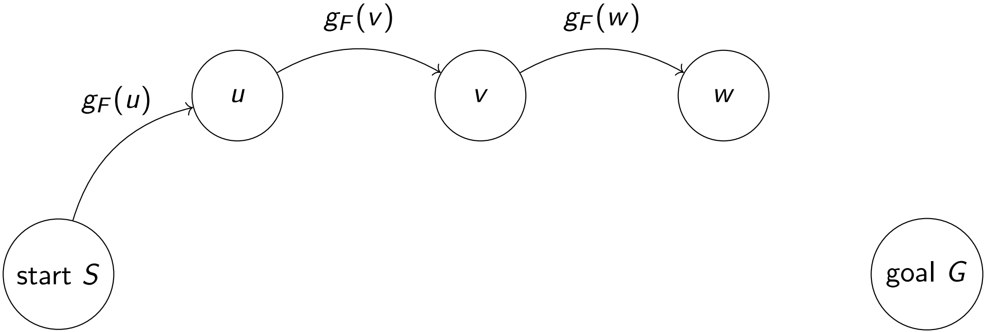

$g_F(n)$ is cost of best-known path from *start* to $n$.

# Forward Search



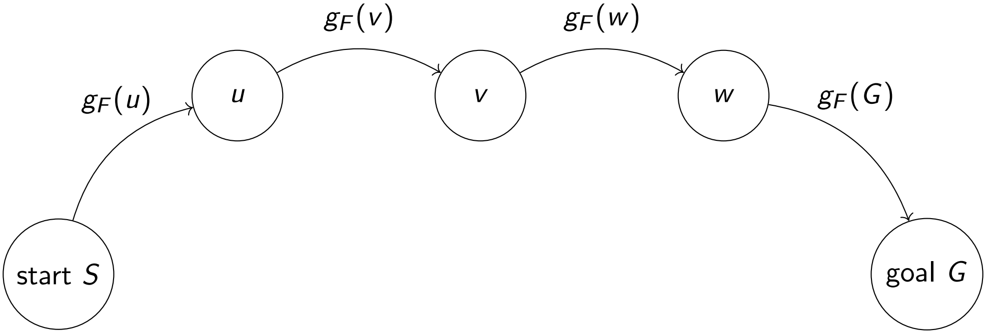

$g_F(n)$ is cost of best-known path from *start* to $n$.

# Forward Search




$g_F(n)$ is cost of best-known path from *start* to $n$.

# Forward Search




$g_F(n)$ is cost of best-known path from *start* to $n$.

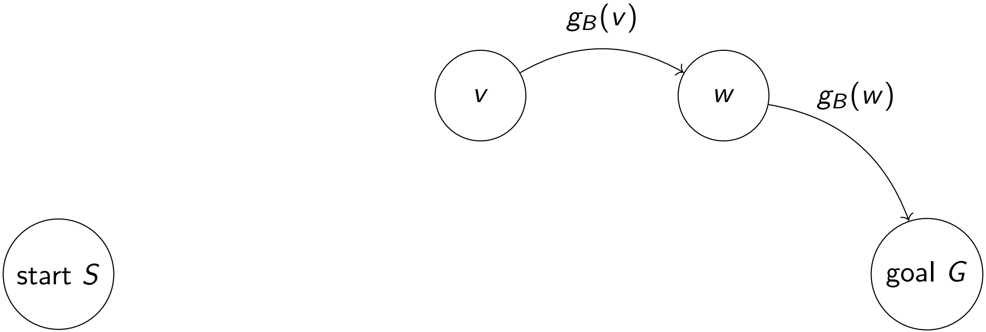
# Backward Search


$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Backward Search

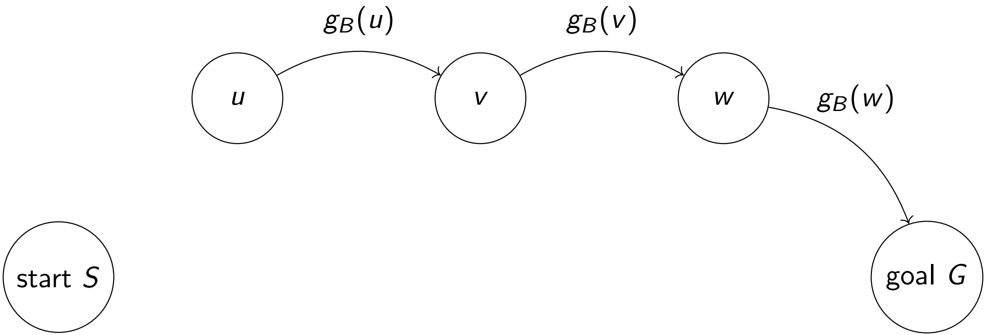

$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Backward Search



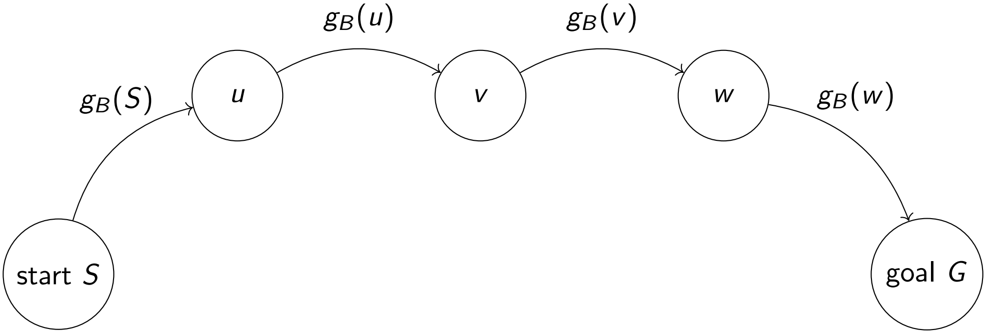

$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Backward Search




$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Backward Search




$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Bidirectional Search


$g_F(n)$ is cost of best-known path from *start* to $n$.
$g_B(n)$ is cost of best-known path from $n$ to *goal*.

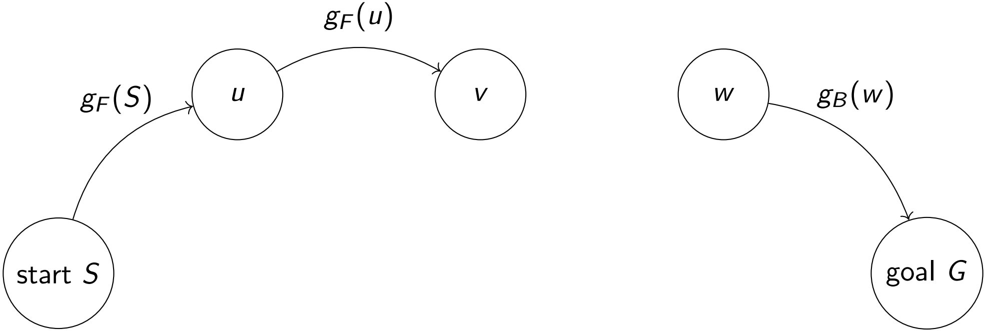
# Bidirectional Search


$g_F(n)$ is cost of best-known path from *start* to $n$.
$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Bidirectional Search

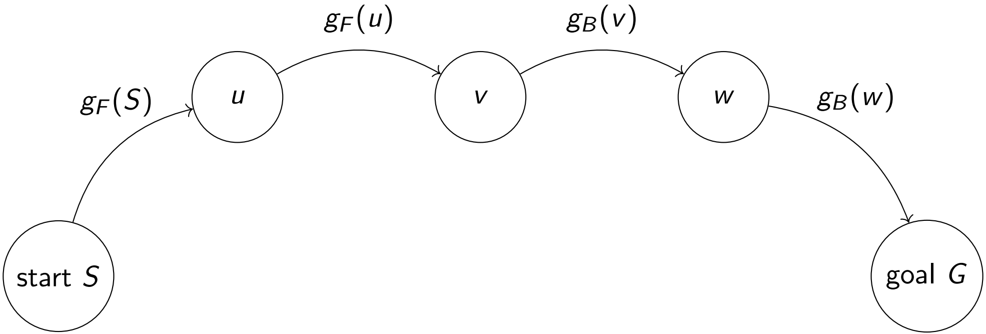

$g_F(n)$ is cost of best-known path from *start* to $n$.
$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Bidirectional Search



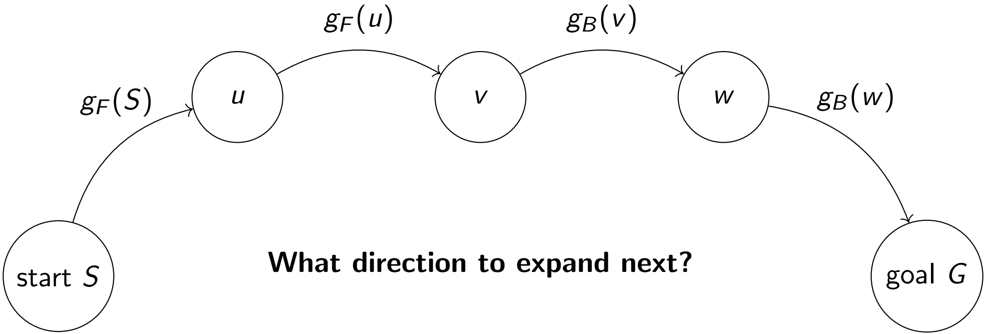

$g_F(n)$ is cost of best-known path from *start* to $n$.
$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Bidirectional Search




$g_F(n)$ is cost of best-known path from *start* to $n$.
$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Bidirectional Search




**What direction to expand next?**

$g_F(n)$ is cost of best-known path from *start* to $n$.
$g_B(n)$ is cost of best-known path from $n$ to *goal*.

# Planning as Constraint Satisfaction Problem

- State is a black box: arbitrary data structure
- Goal test is a function: set of constraints
- Use general-purpose algorithms

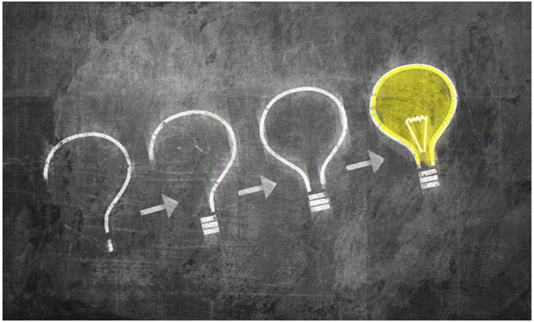
# Planning as Constraint Satisfaction Problem

- State is a black box: arbitrary data structure
- Goal test is a function: set of constraints
- Use general-purpose algorithms

1. Propositionalize initial state
2. $Action^t$ variable
3. Goal check

# Solving Planning Problems

## Questions so far?



# STRIPS

- Stanford Research Institute Problem Solver
- Language + Solver + Search procedure
- Shakey the robot (1971)
- Factored representation of the world

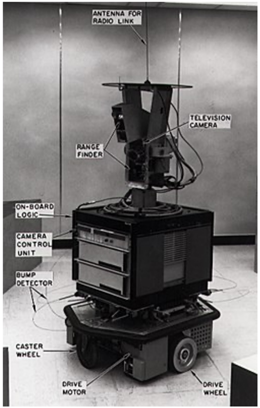



Figure: Shakey the robot.

# STRIPS: The Language

Problem: $\langle P, A, I, G \rangle$

- $P$: set of predicates → What can true of false
- $A$: set of actions → What can agent do
- $I$: initial state → Atoms that hold at start of problem setting
- $G$: goal state → Atoms that the agent wants to hold eventually

# STRIPS: The Language

Problem: $\langle P, A, I, G \rangle$

- $P$: set of predicates → What can true of false
- $A$: set of actions → What can agent do
- $I$: initial state → Atoms that hold at start of problem setting
- $G$: goal state → Atoms that the agent wants to hold eventually

**Predicate:** function over domain objects to truth-values `(at Agent Location)`.

**Atom:** predicate instantiation with specific objects `(at shakey table1)`.

# Example

Problem: Connect the right wires and then turn on the power.

- What are the predicates? `(connected Link)`
- What are the actions?
- What is the initial state?
- What is the goal?

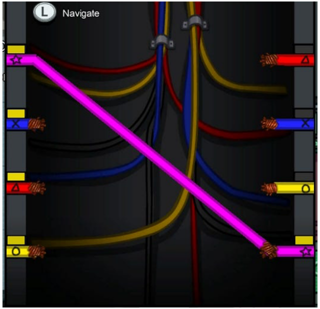



Figure: Wire linking problem.

# Example

Problem: Connect the right wires and then turn on the power.

- Predicates: `(connected Link)`, `(power-on)`, `(link Link1 Link2)`, `(color Link Color)`, `(power-off)`

What are the actions?

What is the initial state?

What is the goal?

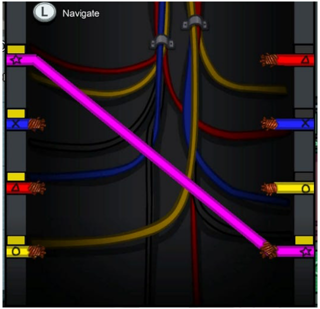



Figure: Wire linking problem.

# Example

Problem: Connect the right wires and then turn on the power.

- Predicates: (connected Link), (power-on), (link Link1 Link2), (color Link Color), (power-off)
- Actions: (connect Link1 Link2), (turn-on), (turn-off), (disconnect Link1 Link2)

What is the initial state?

What is the goal?

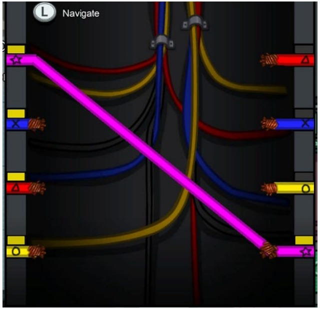



Figure: Wire linking problem.

# Example

Problem: Connect the right wires and then turn on the power.

- Predicates: (connected Link), (power-on), (link Link1 Link2), (color Link Color), (power-off)
- Actions: (connect Link1 Link2), (turn-on), (turn-off), (disconnect Link1 Link2)
- Initially: (connected l1), (link l1 r4), (power-off), (color l3 red), (color r1 red), ...

What is the goal?

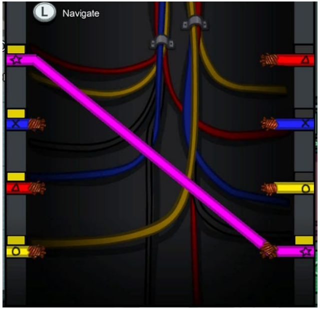



Figure: Wire linking problem.

# Example

Problem: Connect the right wires and then turn on the power.

- Predicates: (connected Link), (power-on), (link Link1 Link2), (color Link Color), (power-off)
- Actions: (connect Link1 Link2), (turn-on), (turn-off), (disconnect Link1 Link2)
- Initially: (connected l1), (link l1 r4), (power-off), (color l3 red), (color r1 red), ...
- Goal: (power-on)

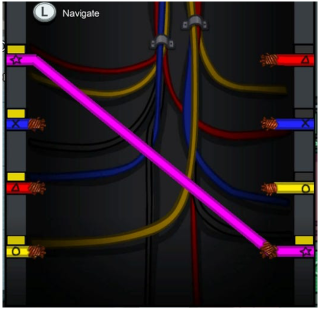



Figure: Wire linking problem.

# States and Actions

## State

State is a conjunction of atoms that currently hold.

- *Complete* state: all other predicate instantiations are assumed to be false.
- *Partial* state: doesn't matter if the other predicate instantiations are true/false.

## Action

Action $a \in A$ defines the conditions and effects of moving between states.

- $\textsc{pre}(a)$: Set of predicates that must hold to execute $a$
- $\textsc{del}(a)$: Set of atoms removed from state after executing $a$
- $\textsc{add}(a)$: Set of atoms added to state after executing $a$

# Action Applicability

Can we perform this action in the current state?

$$\text{PRE}(a) \subseteq s$$

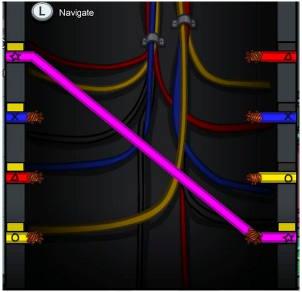



Figure: Current state.

**Action:** `(turn-on)`

- $\text{PRE}(a)$: `{(connected r1), (connected r2), (connected r3), (connected r4)}`
- $\text{DEL}(a)$: `{(power-off)}`
- $\text{ADD}(a)$: `{(power-on)}`

# Action Progression

What happens if we perform this action in the current state?

$$\text{PROGRESS}(s, a) = (s - \text{DEL}(a)) \cup \text{ADD}(a)$$

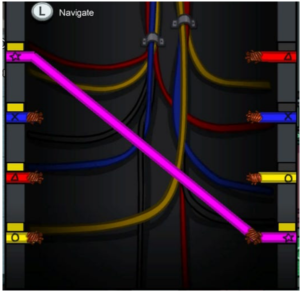



Figure: Current state.

**Action:** `(connect l2 r2)`

- PRE($a$): {`(not (connected l2))`, `(not (connected r2))`, `(color l2 c) (color r2 c)`}
- DEL($a$): {`(not (connected l2))`, `(not (connected r2))`}
- ADD($a$): {`(connected l2) (connected r2)`}

# Goal achievement

When are we done?

$G \subseteq s$

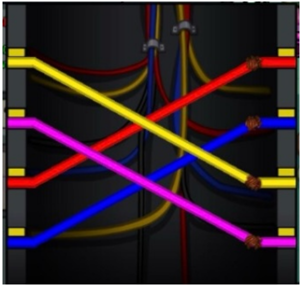


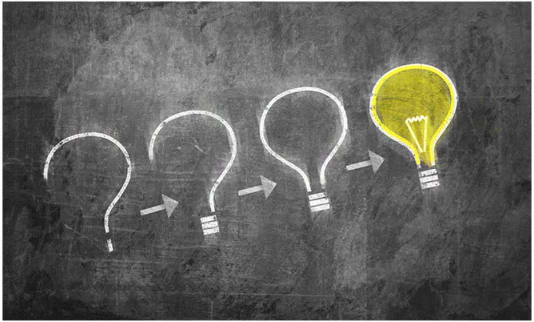
Figure: Current state.

**Action:** (turn-on)

- PRE($a$): {(connected r1), (connected r2), (connected r3), (connected r4)}
- DEL($a$): {(power-off)}
- ADD($a$): {(power-on)}

# Modeling Search Problems

## Questions so far?



# The Planning Domain Definition Language (PDDL)

- PDDL: A common language for arbitrary problem specs
- Contains the STRIPS formalism
- Many variations for various formalisms: extensions with more expressiveness
- Supported by a variety of planners
- Driven by the (roughly) bi-annual International Planning Competition
- Lisp-like syntax (`many (((brackets!))))`
  - Learn to read
  - Can use tools to write (Python library)

# PDDL Input Files

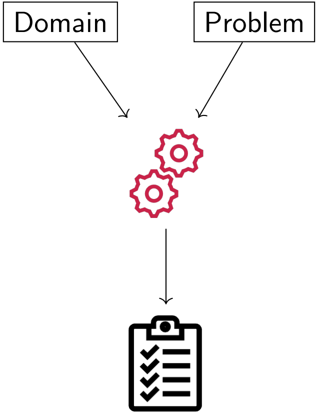



Figure: PDDL structure.

**Domain**

- Requirements
- Types
- Predicates
- Actions

**Problem Instance**

- Objects
- Initial state
- Goal atoms

# Domain Specification in PDDL

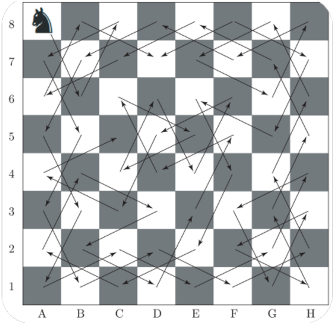



Figure: Knights tour problem.

# Domain Specification in PDDL

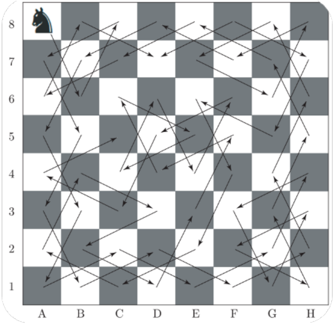



Figure: Knights tour problem.

```
(define (domain knights-tour)
    (:requirements :strips)
    (:predicates
        (at ?square)
        (visited ?square)
        (valid_move ?square_from ?square_to)
    )
    (:action move
        :parameters (?current ?to)
        :precondition (and (at ?current)
            (valid_move ?current ?to)
            (not (visited ?to)))
        :effect (and (not (at ?current))
            (at ?to) (visited ?to))
    ))
```

# Domain Specification in PDDL

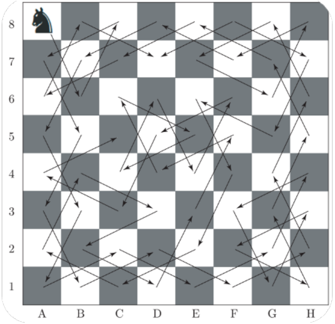



Figure: Knights tour problem.

```
(define (domain knights-tour)
    (:requirements :strips)
    (:predicates
        (at ?square)
        (visited ?square)
        (valid_move ?square_from ?square_to)
    )
    (:action move
        :parameters (?current ?to)
        :precondition (and (at ?current)
            (valid_move ?current ?to)
            (not (visited ?to)))
        :effect (and (not (at ?current))
            (at ?to) (visited ?to))
    ))
```

# Domain Specification in PDDL

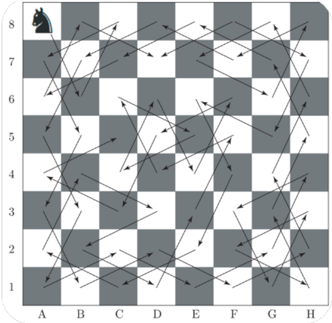
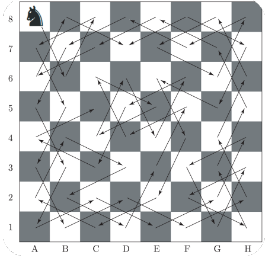



Figure: Knights tour problem.

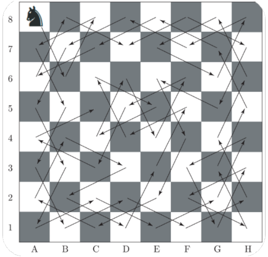
```
(define (domain knights-tour)
    (:requirements :strips)
    (:predicates
        (at ?square)
        (visited ?square)
        (valid_move ?square_from ?square_to)
    )
    (:action move
        :parameters (?current ?to)
        :precondition (and (at ?current)
            (valid_move ?current ?to)
            (not (visited ?to)))
        :effect (and (not (at ?current))
            (at ?to) (visited ?to))
    ))
```

# Problem Instance Specification in PDDL




```
(define (problem knight-tour)
    (:domain knights-tour)
    (:objects
        a1 a2 a3 a4 a5 a6 a7 a8
        b1 b2 b3 b4 b5 b6 b7 b8
        ...
        h1 h2 h3 h4 h5 h6 h7 h8
    )
(:init
    (at a8)
    (visited a8)
    (valid_move a8 b6)
    (valid_move b6 a8)
    (valid_move a8 c7)
    (valid_move c7 a8)
    ...
)
(:goal (and
    (visited a1)
    (visited a2)
    ...
    (visited h8)
)))
```

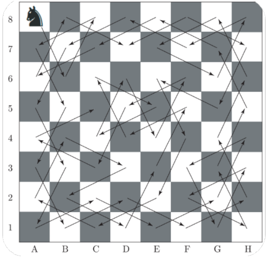
# Problem Instance Specification in PDDL




```
(define (problem knight-tour)
    (:domain knights-tour)
    (:objects
        a1 a2 a3 a4 a5 a6 a7 a8
        b1 b2 b3 b4 b5 b6 b7 b8
        ...
        h1 h2 h3 h4 h5 h6 h7 h8
    )
```

```
(:init
    (at a8)
    (visited a8)
    (valid_move a8 b6)
    (valid_move b6 a8)
    (valid_move a8 c7)
    (valid_move c7 a8)
    ...
)
(:goal (and
    (visited a1)
    (visited a2)
    ...
    (visited h8)
)))
```

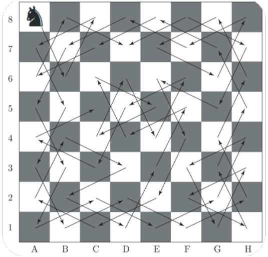
# Problem Instance Specification in PDDL




```
(define (problem knight-tour)
    (:domain knights-tour)
    (:objects
        a1 a2 a3 a4 a5 a6 a7 a8
        b1 b2 b3 b4 b5 b6 b7 b8
        ...
        h1 h2 h3 h4 h5 h6 h7 h8
    )
```

```
(:init
    (at a8)
    (visited a8)
    (valid_move a8 b6)
    (valid_move b6 a8)
    (valid_move a8 c7)
    (valid_move c7 a8)
    ...
)
(:goal (and
    (visited a1)
    (visited a2)
    ...
    (visited h8)
)))
```

# Problem Instance Specification in PDDL




```
(define (problem knight-tour)
    (:domain knights-tour)
    (:objects
        a1 a2 a3 a4 a5 a6 a7 a8
        b1 b2 b3 b4 b5 b6 b7 b8
        ...
        h1 h2 h3 h4 h5 h6 h7 h8
    )
```

```
(:init
    (at a8)
    (visited a8)
    (valid_move a8 b6)
    (valid_move b6 a8)
    (valid_move a8 c7)
    (valid_move c7 a8)
    ...
)
(:goal (and
    (visited a1)
    (visited a2)
    ...
    (visited h8)
)))
```

# Typing

1. **Typing requirement**

2. **Type predicates**

# Typing

## 1. Typing requirement

```
(:requirements typing)
(:types
    vehicle location package
    car truck - vehicle)
(:init
    truck1 - truck
    package1 - package)
(:predicates
    (at ?v - vehicle ?l - location)
    (carry ?t - truck ?p - package)
    (move ?l1 ?l2 - location))
```

## 2. Type predicates

# Typing

## 1. Typing requirement

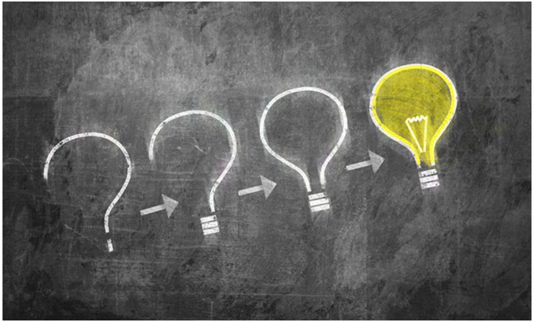
```
(:requirements typing)
(:types
    vehicle location package
    car truck - vehicle)
(:init
    truck1 - truck
    package1 - package)
(:predicates
    (at ?v - vehicle ?l - location)
    (carry ?t - truck ?p - package)
    (move ?l1 ?l2 - location))
```

## 2. Type predicates

```
(:predicates
    (at ?v ?l)
    (carry ?t ?p)
    (package ?p)
    (truck ?c)
    (vehicle ?v))
(:objects truck1 package1 loc1 loc2)
(:init
    (truck truck1)
    (package package1))
(:precondition (and (carry ?t ?p)
    (truck ?t) (package ?p)
))
```

# PDDL

## Questions so far?



# Exercise: Missing precondition

- Ferry boat domain
- Three actions: `(board ?car ?loc)`, `(sail ?loc1 ?loc2)`, `(debark ?car ?loc)`
- Predicates: `(car ?car)`, `(location ?loc)`, `(at-ferry ?loc)`, `(at ?car ?loc)`, `(empty-ferry)`, `(on-ferry ?car)`
- What precondition is missing for board?

```
(:action board
    :parameters (?car ?loc)
    :precondition (and
        (car ?car)
        (location ?loc)
        (at ?car ?loc)
        (empty-ferry))
    :effect (and (on-ferry ?car)
        (not (at ?car ?loc))
        (not (empty-ferry)))
)
```

# Exercise: Missing precondition

- Ferry boat domain
- Three actions: (board ?car ?loc), (sail ?loc1 ?loc2), (debark ?car ?loc)
- Predicates: (car ?car), (location ?loc), (at-ferry ?loc), (at ?car ?loc), (empty-ferry), (on-ferry ?car)
- What precondition is missing for board?

➔ 2 min for yourself, then 1 min discuss with your neighbor

```
(:action board
    :parameters (?car ?loc)
    :precondition (and
        (car ?car)
        (location ?loc)
        (at ?car ?loc)
        (empty-ferry))
    :effect (and (on-ferry ?car)
        (not (at ?car ?loc))
        (not (empty-ferry)))
)
```

# Solution: Missing precondition

- Ferry boat domain
- Three actions: (board ?car ?loc), (sail ?loc1 ?loc2), (debark ?car ?loc)
- Predicates: (car ?car), (location ?loc), (at-ferry ?loc), (at ?car ?loc), (empty-ferry), (on-ferry ?car)
- **Missing precondition** (at-ferry ?loc)

```
(:action board
    :parameters (?car ?loc)
    :precondition (and
        (car ?car)
        (location ?loc)
        (at ?car ?loc)
        (at-ferry ?loc)
        (empty-ferry))
    :effect (and (on-ferry ?car)
        (not (at ?car ?loc))
        (not (empty-ferry)))
)
```

# Exercise: Spot the mistake

Towers of Hanoi

- Discs stacked on pegs `(on ?disc ?peg)`
- Discs can only be on top of larger discs `(smaller ?top ?bottom)`
- Move one disc at a time
- Only keep track of disc position: on other disc or on peg `(on ?d1 ?d2)`
- Clear discs that have no disc on top `(clear ?d)`
- Goal: have the same stack on the final pole `(on ?smallest ?small)`

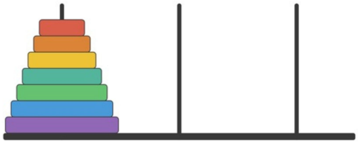


Figure: Towers of Hanoi.

# Exercise: Spot the mistake

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))

    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))))
)
```

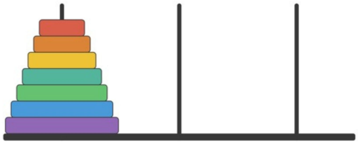


Figure: Towers of Hanoi.

# Exercise: Spot the mistake

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))

    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))))
)
```

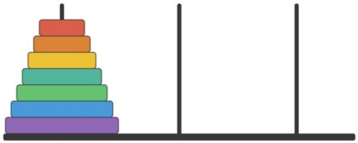


Figure: Towers of Hanoi.

➔ 2 min for yourself, then 1 min discuss with your neighbor

# Solution: Missing effect

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
        )))
```

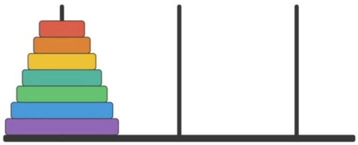
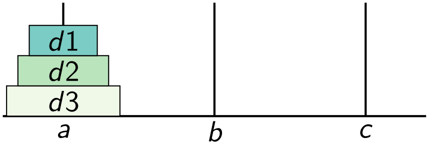


Figure: Towers of Hanoi.

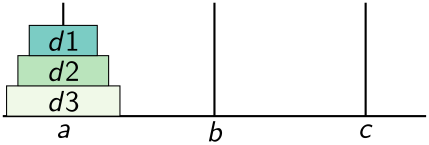
# Tutorial: Valid plan

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
        )))
```



# Tutorial: Valid plan

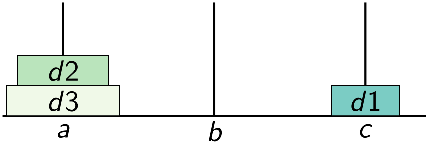
```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
        )))
```




Next: (move d1 d2 c)

# Tutorial: Valid plan

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
    )))
```

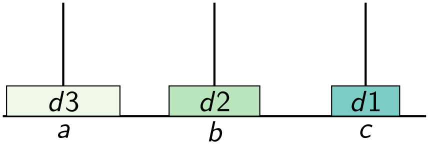



Next: (move d1 d2 c)

Next: (move d2 d3 b)

# Tutorial: Valid plan

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
        )))
```



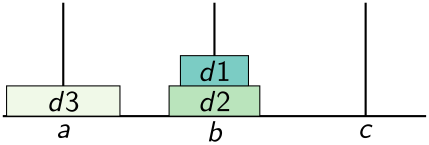

Next: (move d1 d2 c)

Next: (move d2 d3 b)

Next: (move d1 c d2)

# Tutorial: Valid plan

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
        )))
```




Next: (move d1 d2 c)

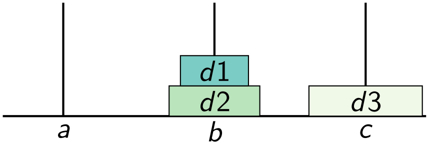
Next: (move d2 d3 b)

Next: (move d1 c d2)

Next: (move d3 a c)

# Tutorial: Valid plan

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
        )))
```




Next: (move d1 d2 c)

Next: (move d2 d3 b)

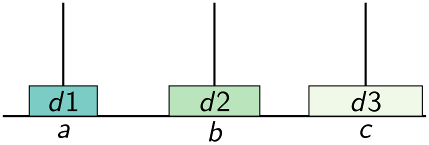
Next: (move d1 c d2)

Next: (move d3 a c)

Next: (move d1 d2 a)

# Tutorial: Valid plan

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
    )))
```




Next: `(move d1 d2 c)`

Next: `(move d2 d3 b)`

Next: `(move d1 c d2)`

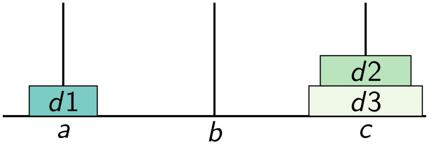
Next: `(move d3 a c)`

Next: `(move d1 d2 a)`

Next: `(move d2 b d3)`

# Tutorial: Valid plan

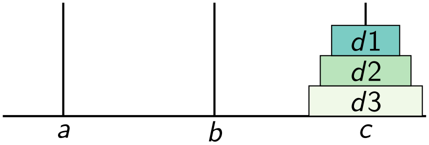
```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
        )))
```




Next: (move d1 d2 c)
Next: (move d2 d3 b)
Next: (move d1 c d2)
Next: (move d3 a c)
Next: (move d1 d2 a)
Next: (move d2 b d3)
Next: (move d1 a d2)

# Tutorial: Valid plan

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?disc ?to)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
        )))
```




```
(move d1 d2 c)
(move d2 d3 b)
(move d1 c d2)
(move d3 a c)
(move d1 d2 a)
(move d2 b d3)
(move d1 a d2)
```

# PDDL Extension: numeric fluents

- Predicates vs fluents
- Express numeric properties
- Precondition: =, >, <
- Effect: `increase`, `decrease`

Example: package delivery domain - partial domain

```
(:predicates (at ?loc) ... )
(:functions
    (distance ?loc1 ?loc2)
    (battery))
(:init
    (distance l1 l2 50)
    (battery 100))
(:precondition
    (> (battery) 0)
    (at ?l2))
(:effect
    (decrease (battery) (distance ?l1 ?l2)
   )
)
```

# Exercise: Modeling in PDDL

Goal: get everyone to Delft

Initial: Train starts at Amsterdam, train capacity 60, initial demand in blue

What are the predicates?

What are the actions?

What is the initial state?

What is the goal?

Discuss with neighbour (10 min)

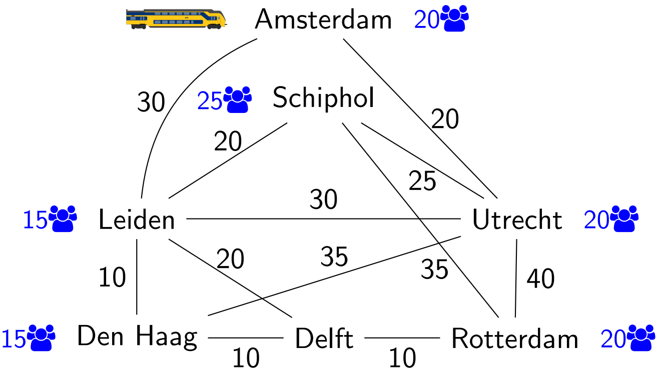



Figure: Problem setting.

# Relation to Other Lectures

- Search, Inference, Learning, and Optimization
- Effectiveness for solving planning problems

# Learning Objectives

1. Explain what planning is
2. Explain different approaches to finding plans
3. Read planning problems in PDDL
4. Model a problem in PDDL terms (semantically, not syntactically)
5. Reason whether a model or plan is correct and effective

# Conclusion

- Planning problems in the real world
- Planning as a search problem
- How to model a planning problem

Questions?

# Next Steps & Further Information

**Homework**

- Homework exercises week 4

**Exam material**

- Lecture notes
- Lecture slides
- Homework

**Extra information**

- Book: *Russel & Norvig: Artificial Intelligence, Ch.11*
- `http://planning.wiki/` General info on PDDL and planners
- `http://editor.planning.domains/` Online editor for PDDL
- `https://unified-planning.readthedocs.io/en/latest/` Python library for PDDL writing and solving

**Questions?** i.k.hanou@tudelft.nl