Automated Planning

Probabilistic AI and Reasoning - Lecture 7

Issa Hanou

Delft University of Technology

September 23, 2024

TUDelft

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

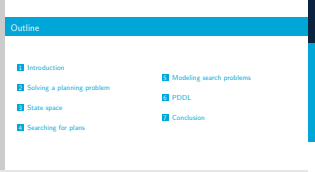
Who am I?
- Issa Hanou
- PhD candidate Algorithmics department
- Working on Planning and Scheduling for Railway logistics

Figure: Shunting yard in the Netherlands.

Who am I?

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

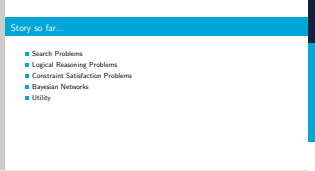
Figure: Shunting yard in the Netherlands.



Outline

└Story so far...

1. Course so far has discussed different types of problems
2. Utility for preferences: model risk of actions
3. Today: Problems so far don't have time, sometimes we need to weigh options or have multiple actions in sequence



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

Story so far...
- Search Problems
- Logical Reasoning Problems
- Constraint Satisfaction Problems
- Bayesian Networks
- Utility
- Time component
- Real-World Problems

Figure: TODO example from previous lecture that illustrates planning

Automated Planning!

Story so far...

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

Figure: TODO example from previous lecture that illustrates planning

Automated Planning!

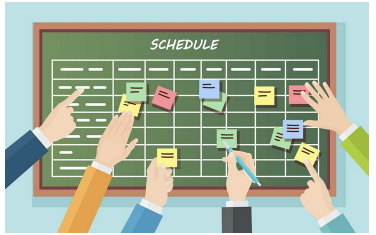
What is Planning?
What do you think planning is?

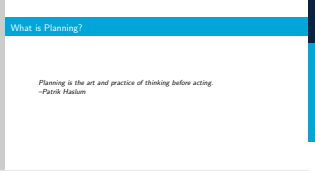
What is Planning?

What do you think planning is?

Examples of Planning

Examples of Planning





Patrik Haslum is an Associate Professor at the Australian National University School of Computing, co-authoring a book on PDDL and planning

What is Planning?

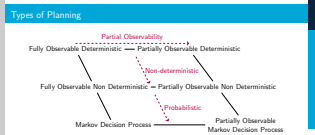
Planning is the art and practice of thinking before acting.
–Patrik Haslum

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)
...

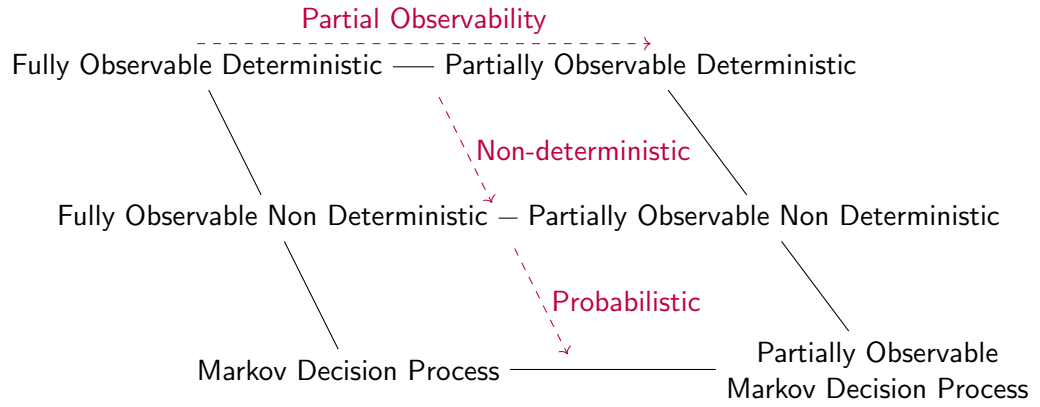
What is a Plan?

(pickup robot2 bowl)
(putdown robot2 bowl ontable)
(scoop robot1 corn)
(putdown robot1 corn inbowl)
(pickup robot2 mushrooms)
(putdown robot2 mushrooms inbowl)
...



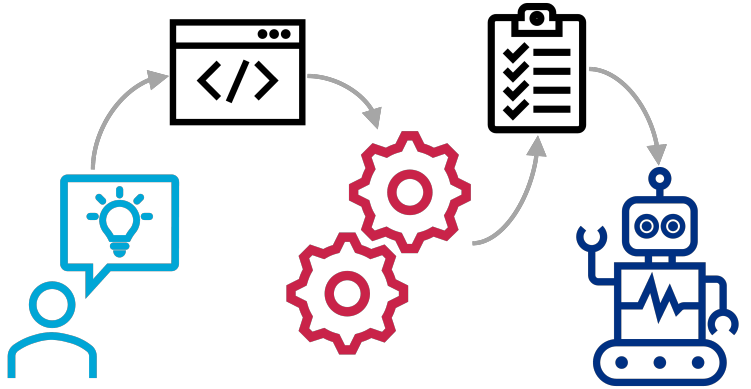
Today we will focus on the Fully observable deterministic setting to consider the automated planning framework, but this can be extended to other types.

Types of Planning



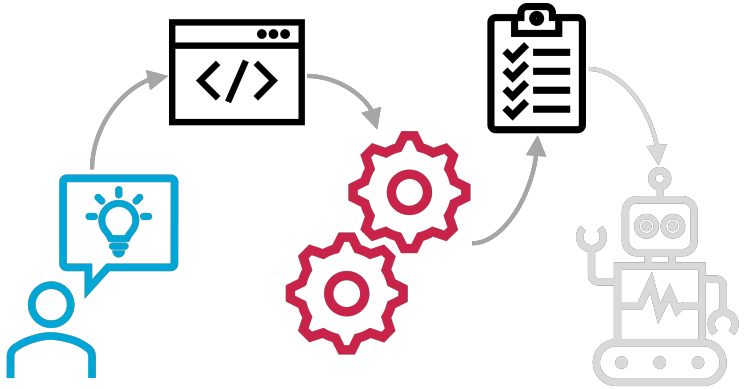
Modeling vs Solving vs Executing

Modeling vs Solving vs Executing



Modeling vs Solving vs Executing

Modeling vs Solving vs Executing



Domain Independent Planning

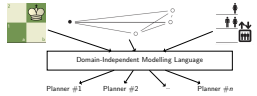


Figure: Domain-Independent Planning.

Domain Independent Planning

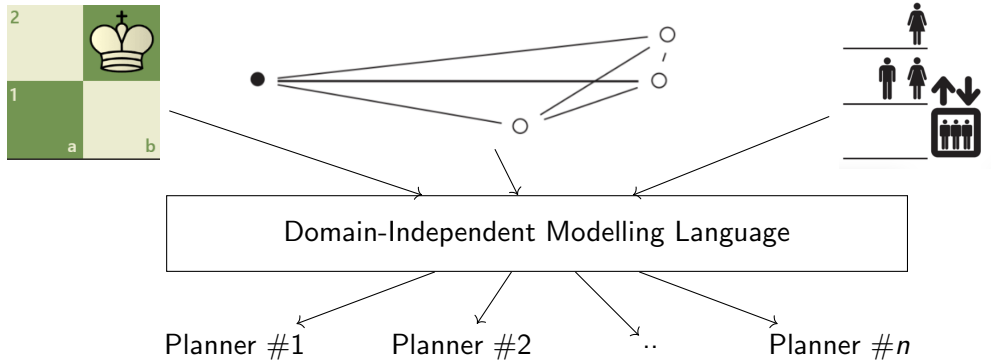
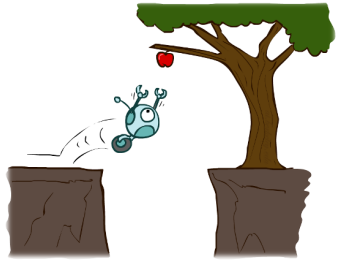


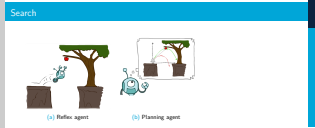
Figure: Domain-Independent Planning.

Search



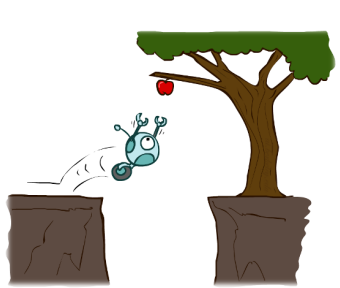
(a) Reflex agent

1. Reflex agent doesn't consider the future consequences of actions
2. Consider how the world currently is, choose an action based on current perception (maybe memory)

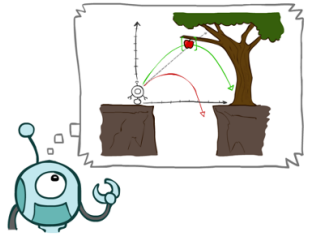


1. Reflex agent doesn't consider the future consequences of actions
2. Consider how the world currently is, choose an action based on current perception (maybe memory)
3. Figure out consequences by planning instead of actually doing, simulate to see consequences of action.
4. Needs a model of the world, and needs a goal formulation

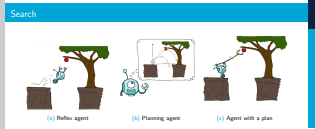
Search



(a) Reflex agent

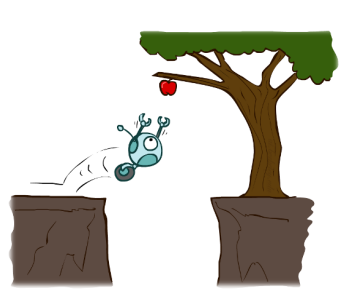


(b) Planning agent

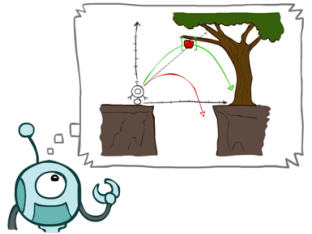


1. Reflex agent doesn't consider the future consequences of actions
2. Consider how the world currently is, choose an action based on current perception (maybe memory)
3. Figure out consequences by planning instead of actually doing, simulate to see consequences of action.
4. Needs a model of the world, and needs a goal formulation

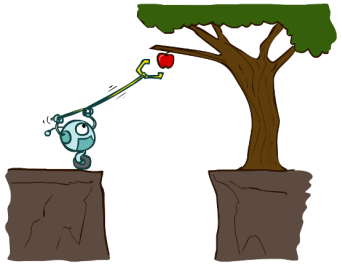
Search



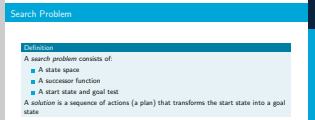
(a) Reflex agent



(b) Planning agent



(c) Agent with a plan



An abstraction of the state space. The goal test is used because there may be more than one state achieving the goal, as this is not a full state specification

Search Problem

Definition

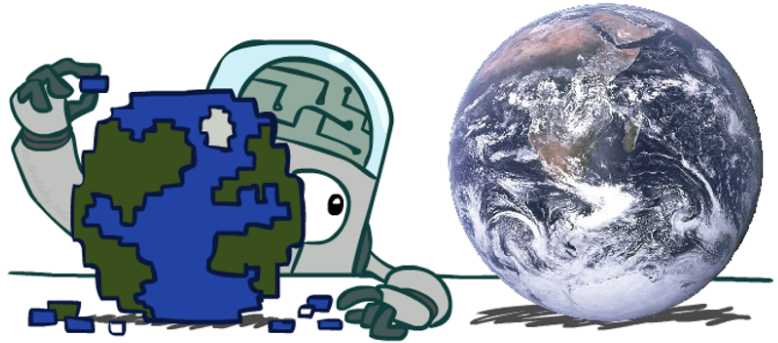
A *search problem* consists of:

- 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

1. A search problem models aspects of the real world. Models aren't perfect
2. Too detailed: can't solve
3. Not detailed enough: doesn't solve the actual problem

Modeling Search Problems



Example Search Problems
Shows time between to travel between two cities
Amsterdam
Schiphol
Leiden
Utrecht
Den Haag
Delft
Rotterdam
Figure: Partial railway network Netherlands.

1. Trying to find a route from Delft to Utrecht.
2. Do we have all the information?

Example Search Problems

Shows time between to travel between two cities

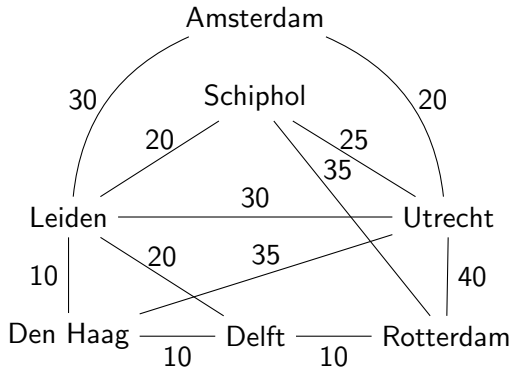


Figure: Partial railway network Netherlands.

Example Search Problems
Shows time between to travel between two cities
Realistic model?
Figure: Partial railway network Netherlands.

Example Search Problems

Shows time between to travel between two cities

Realistic model?

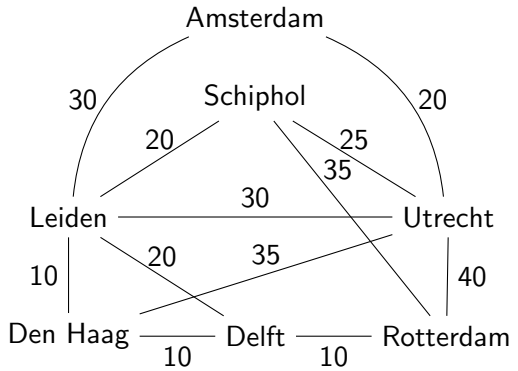


Figure: Partial railway network Netherlands.

Example Search Problems
Amsterdam
Schiphol
Leiden
Utrecht
Den Haag
Delft
Rotterdam
30
20
20
25
30
20
35
10
40
10
10
Shows time between to travel between two cities
Realistic model? Goal-dependent: Delft to Utrecht
Other factors?
Figure: Partial railway network Netherlands.

Example Search Problems

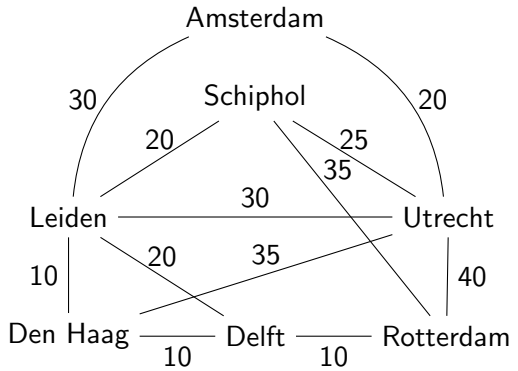


Figure: Partial railway network Netherlands.

Shows time between to travel between two cities
Realistic model? Goal-dependent: Delft to Utrecht

Other factors?

Example Search Problems
Amsterdam
Schiphol
Leiden
Utrecht
Den Haag
Delft
Rotterdam
30
20
20
25
35
30
10
20
35
40
10
10
Shows time between to travel between two cities
Realistic model? Goal-dependent: Delft to Utrecht
Other factors?
Transfer time
Timetables
(Expected) Train capacity
...
Figure: Partial railway network Netherlands.

Example Search Problems

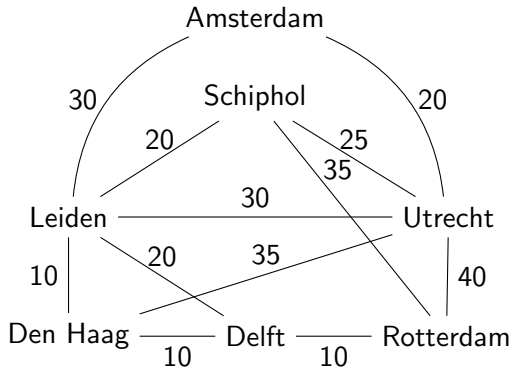


Figure: Partial railway network Netherlands.

Shows time between to travel between two cities
Realistic model? Goal-dependent: Delft to Utrecht

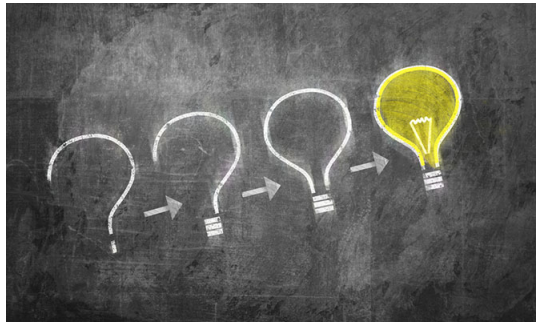
Other factors?

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

Solving a Planning Problem
Questions so far?

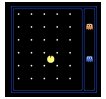
Solving a Planning Problem

Questions so far?



1. Pathing: reach goal location.

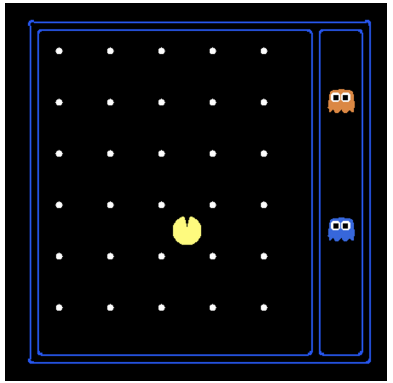
State Space Size



World state
- Agent positions: 120
- Food count: 30
- Ghost positions:12
- Agent facing: NSEW

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

State Space Size

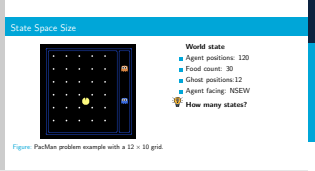


World state

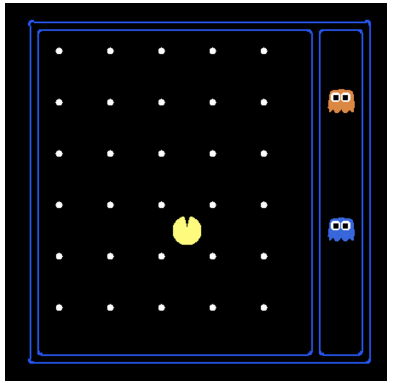
- Agent positions: 120
- Food count: 30
- Ghost positions:12
- Agent facing: NSEW

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

1. Pathing: reach goal location.
2. Total number of states: $120 * (2^{30}) * (12^2) * 4$



State Space Size



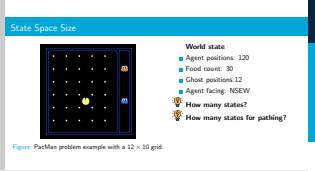
World state

- Agent positions: 120
- Food count: 30
- Ghost positions:12
- Agent facing: NSEW

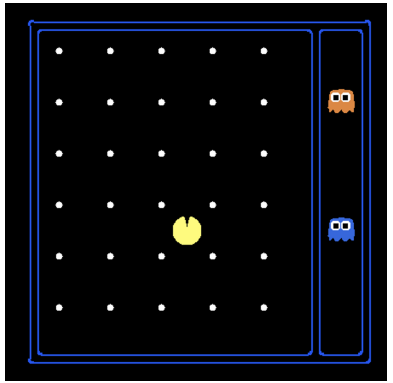
How many states?

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

1. Pathing: reach goal location.
2. Total number of states: $120 * (2^{30}) * (12^2) * 4$
3. 120 states for pathing



State Space Size



World state

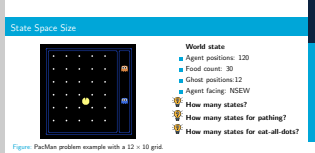
- Agent positions: 120
- Food count: 30
- Ghost positions:12
- Agent facing: NSEW

How many states?

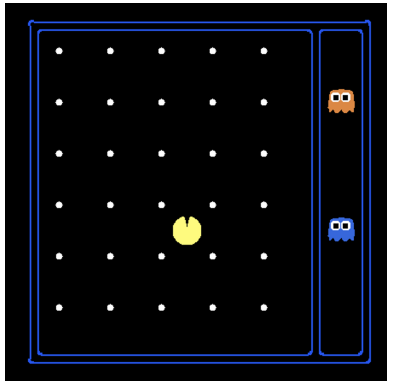
How many states for pathing?

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

1. Pathing: reach goal location.
2. Total number of states: $120 * (2^{30}) * (12^2) * 4$
3. 120 states for pathing
4. $120 * (2^{30})$ states for eat-all-dots



State Space Size



World state

- Agent positions: 120
- Food count: 30
- Ghost positions:12
- Agent facing: NSEW

How many states?

How many states for pathing?

How many states for eat-all-dots?

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

State Space Example

- Transport packages
- Take trains between connected cities
- Can fly longer distances
- **What does state space look like?**

Figure: Logistics problem in Western Europe.

State Space Example

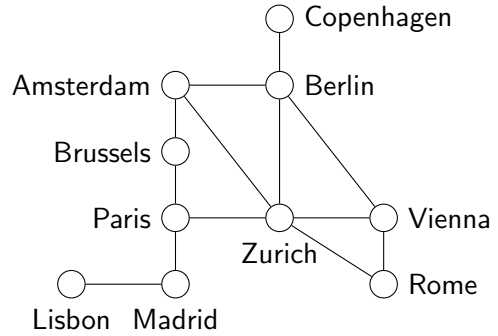


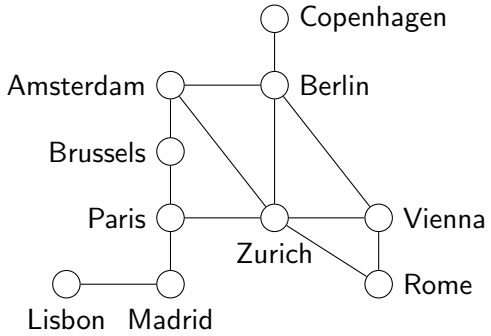
Figure: Logistics problem in Western Europe.

- Transport packages
- Take trains between connected cities
- Can fly longer distances
- **What does state space look like?**

State Space Example Answers

Figure: Logistics in Western Europe with train journeys.

State Space Example Answers



State space components

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

Figure: Logistics in Western Europe with train journeys.

State Space Graph

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

1. Each state occurs only once.
2. Different paths to reach same state.
3. Nodes are abstracted world configurations.
4. Graph is rarely built completely.

State Space Graph

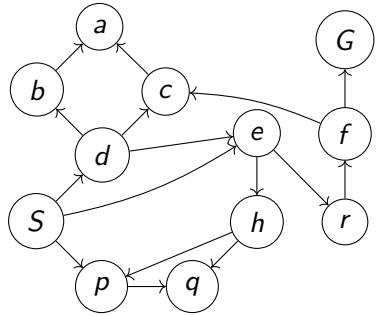


Figure: State space graph for search problem: find path from S to G.

Search Tree

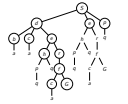


Figure: Search tree for previous search problem.

1. Each node in the search tree is an entire path in the state space graph.
2. Different paths have different nodes, even for achieving the same state.
3. Every plan is included in the tree, most are ignored.
4. Tree is rarely built completely.

Search Tree

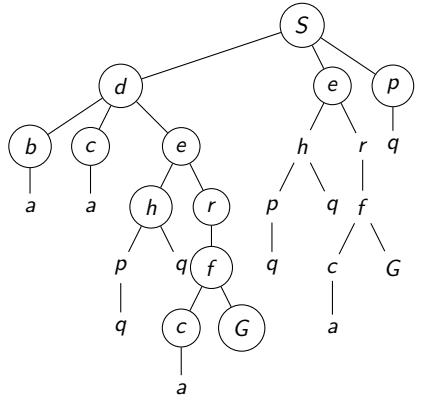
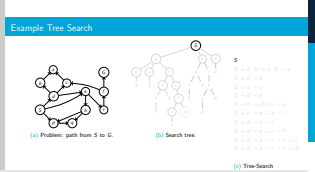
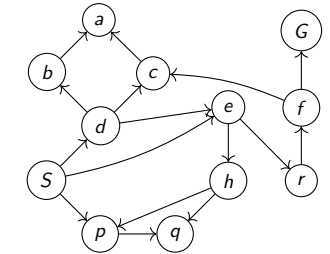


Figure: Search tree for previous search problem.

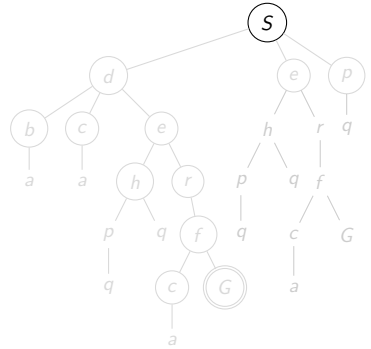


What search is this? Showing breadth-first search

Example Tree Search



(a) Problem: path from S to G.



(b) Search tree.

S

$S \to d,\ S \to e,\ S \to p$

$S \to d \to b$

$S \to d \to c$

$S \to d \to e$

$S \to d \to \{b, c\} \to a$

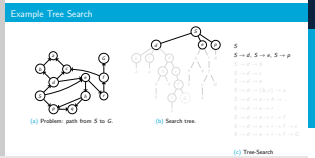
$S \to d \to e \to h \to \ldots$

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

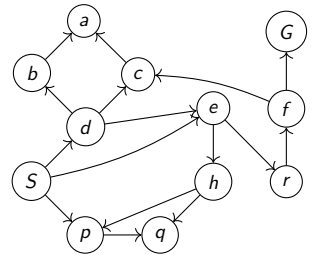
$S \to d \to e \to r \to f$

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

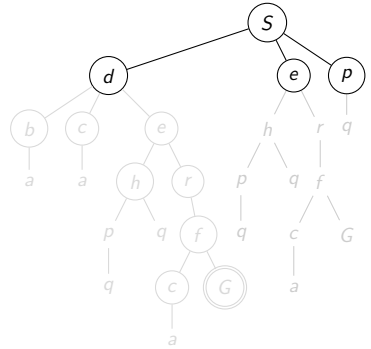
$S \to d \to e \to r \to f \to G$



Example Tree Search



(a) Problem: path from S to G.



(b) Search tree.

S

$S \to d,\ S \to e,\ S \to p$

$S \to d \to b$

$S \to d \to c$

$S \to d \to e$

$S \to d \to \{b, c\} \to a$

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

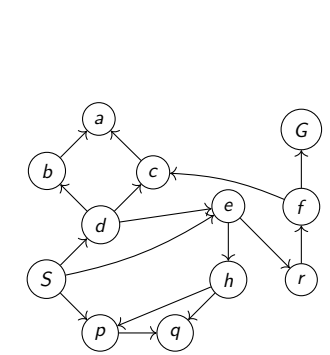
$S \to d \to e \to r$

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

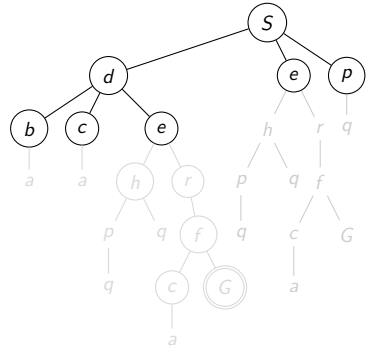
$S \to d \to e \to r \to f \to c$

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

Example Tree Search



(a) Problem: path from S to G.



(b) Search tree.

S

$S \to d,\ S \to e,\ S \to p$

$S \to d \to b$

$S \to d \to c$

$S \to d \to e$

$S \to d \to \{b, c\} \to a$

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

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

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

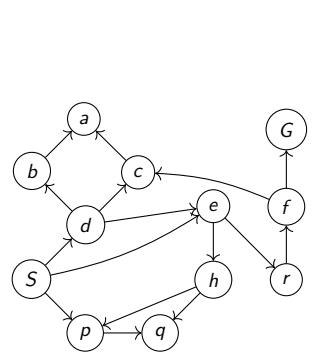
$S \to d \to e \to r \to f \to c$

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

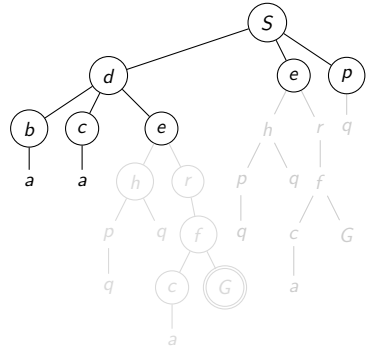
Example Tree Search

(a) Problem: path from S to G.
(b) Search tree.
(c) Tree-Search

Example Tree Search



(a) Problem: path from S to G.



(b) Search tree.

S

$S \to d,\ S \to e,\ S \to p$

$S \to d \to b$

$S \to d \to c$

$S \to d \to e$

$S \to d \to \{b, c\} \to a$

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

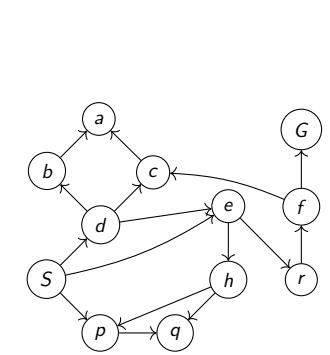
$S \to d \to e \to r$

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

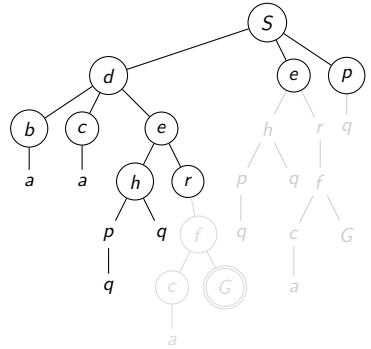
$S \to d \to e \to r \to f \to c$

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

Example Tree Search



(a) Problem: path from S to G.



(b) Search tree.

S

$S \to d,\ S \to e,\ S \to p$

$S \to d \to b$

$S \to d \to c$

$S \to d \to e$

$S \to d \to \{b, c\} \to a$

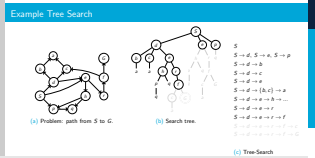
$S \to d \to e \to h \to \ldots$

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

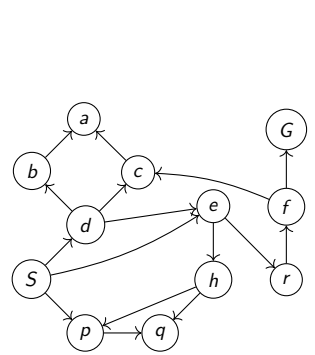
$S \to d \to e \to r \to f$

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

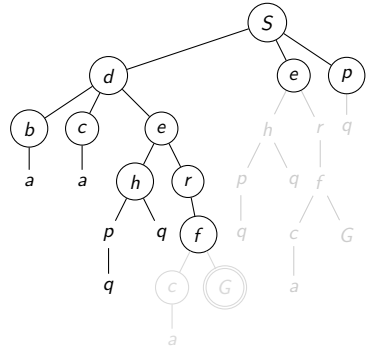
$S \to d \to e \to r \to f \to G$



Example Tree Search



(a) Problem: path from S to G.



(b) Search tree.

S

$S \to d,\ S \to e,\ S \to p$

$S \to d \to b$

$S \to d \to c$

$S \to d \to e$

$S \to d \to \{b, c\} \to a$

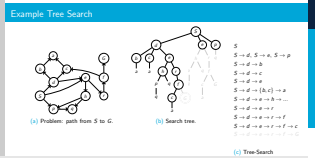
$S \to d \to e \to h \to \ldots$

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

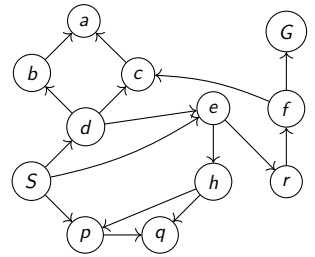
$S \to d \to e \to r \to f$

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

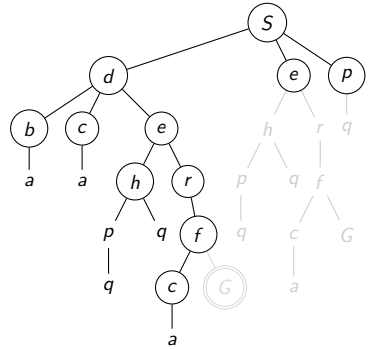
$S \to d \to e \to r \to f \to G$



Example Tree Search



(a) Problem: path from S to G.



(b) Search tree.

S

$S \to d,\ S \to e,\ S \to p$

$S \to d \to b$

$S \to d \to c$

$S \to d \to e$

$S \to d \to \{b, c\} \to a$

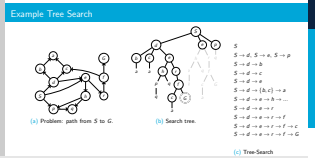
$S \to d \to e \to h \to \ldots$

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

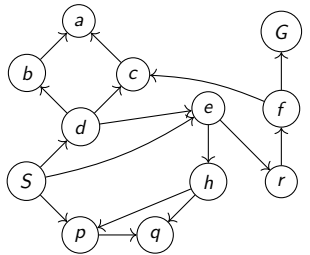
$S \to d \to e \to r \to f$

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

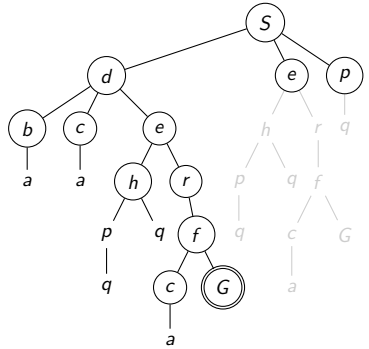
$S \to d \to e \to r \to f \to G$



Example Tree Search



(a) Problem: path from S to G.



(b) Search tree.

S

$S \to d,\ S \to e,\ S \to p$

$S \to d \to b$

$S \to d \to c$

$S \to d \to e$

$S \to d \to \{b, c\} \to a$

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

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

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

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

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

Forward Search
start S
goal G
$g_F(n)$ 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.

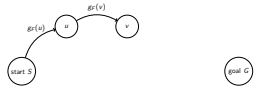
Forward Search

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

Forward Search

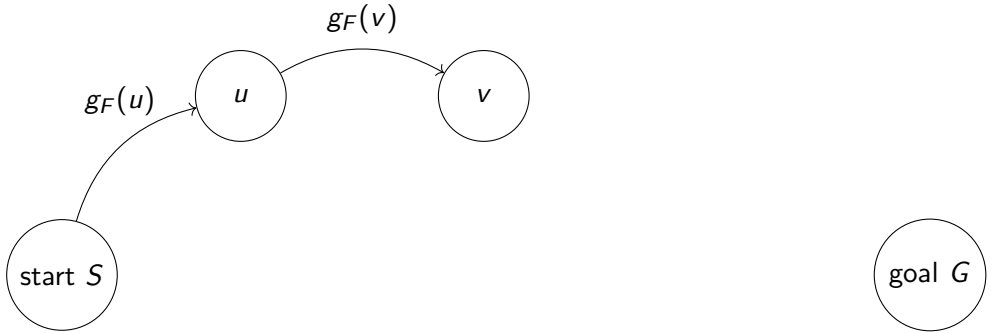
$g_F(n)$ is cost of best-known path from *start* to n.

Forward Search



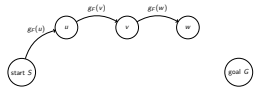
$g_F(n)$ is cost of best-known path from start to n.

Forward Search



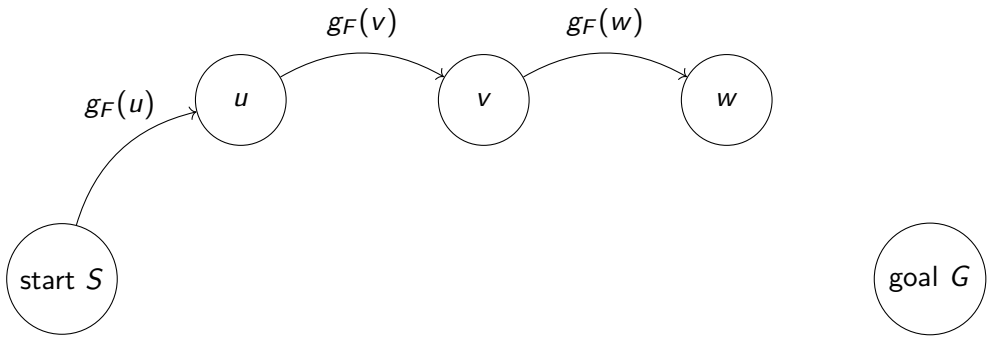
$g_F(n)$ is cost of best-known path from *start* to n.

Forward Search



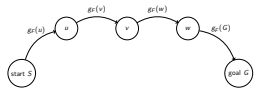
$g_F(n)$ is cost of best-known path from start to n.

Forward Search



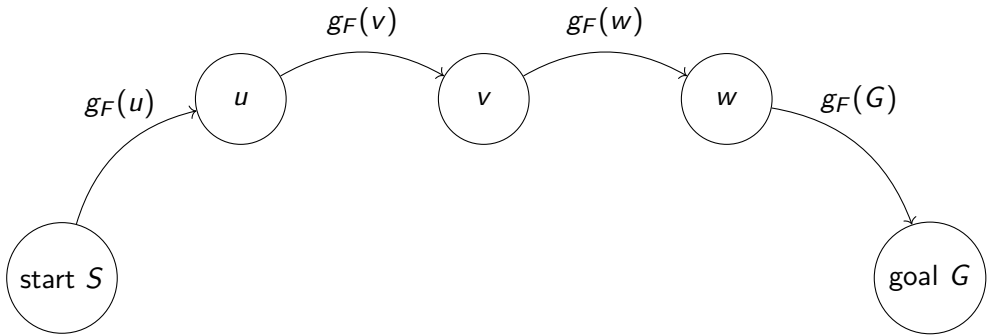
$g_F(n)$ is cost of best-known path from *start* to n.

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 $goal$.

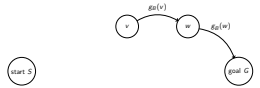
Backward Search

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

Backward Search

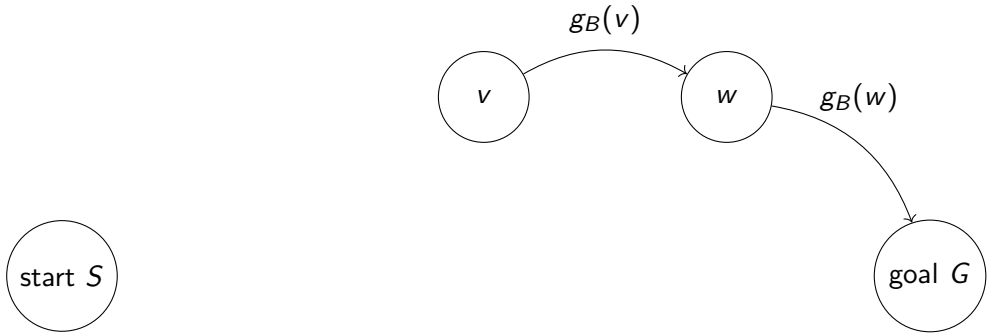
$g_B(n)$ is cost of best-known path from n to $goal$.

Backward Search



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

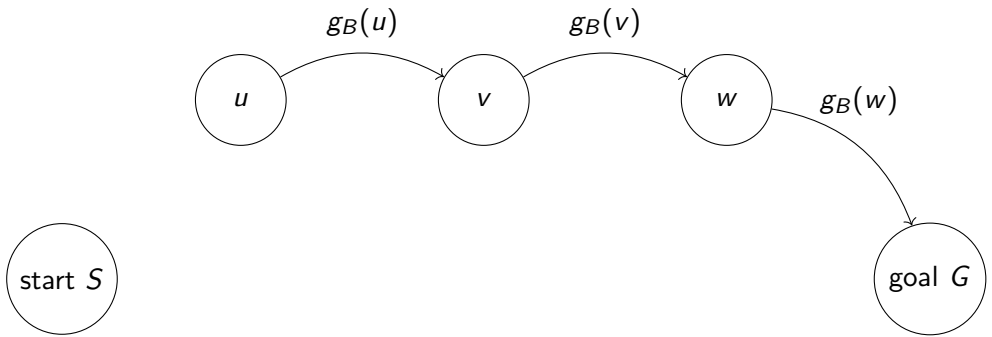
Backward Search



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

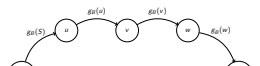
Backward Search
$g_B(u)$ $g_B(v)$ $g_B(w)$
u v w
start S goal G
$g_B(n)$ is cost of best-known path from n to $goal$.

Backward Search



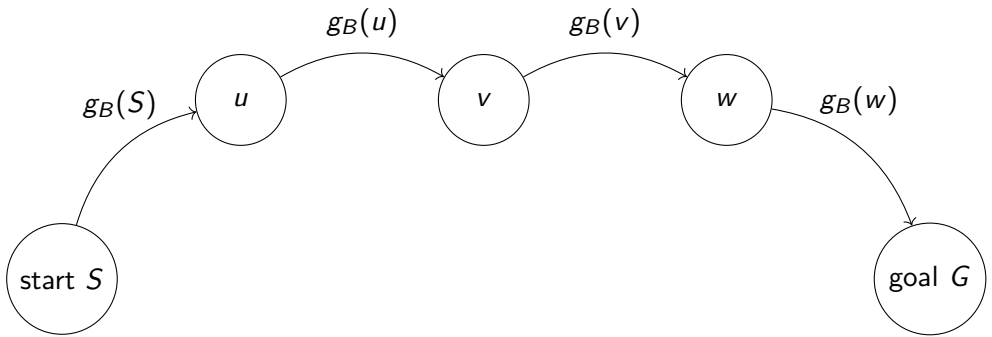
$g_B(n)$ is cost of best-known path from n to $goal$.

Backward Search



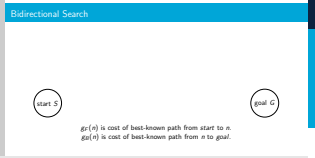
$g_B(n)$ is cost of best-known path from n to $goal$.

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*.

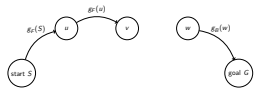
Bidirectional Search

$g_F(n)$ 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

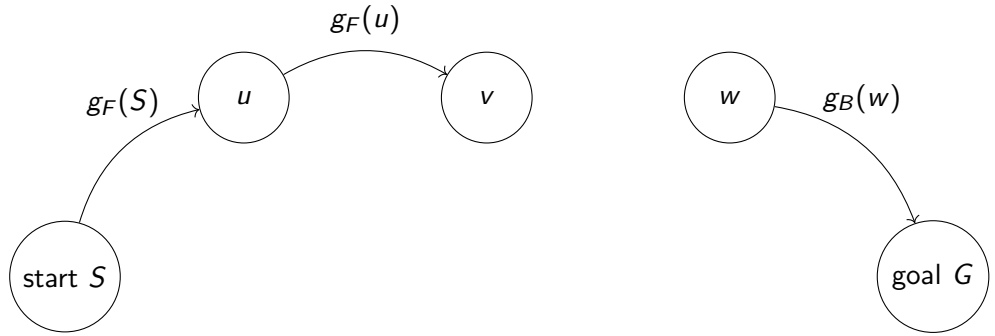
$g_F(n)$ is cost of best-known path from *start* to n.
$g_B(n)$ is cost of best-known path from n to *goal*.

Bidirectional Search



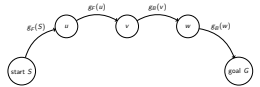
$g_F(n)$ is cost of best-known path from *start* to n.
$g_B(n)$ is cost of best-known path from n to *goal*.

Bidirectional Search



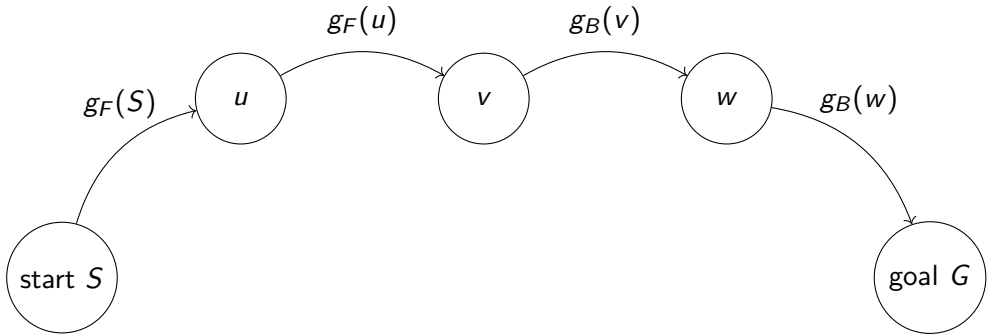
$g_F(n)$ is cost of best-known path from *start* to n.
$g_B(n)$ is cost of best-known path from n to *goal*.

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*.

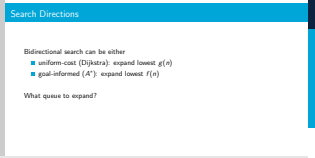
Search Directions
Bidirectional search can be either
- uniform-cost (Dijkstra): expand lowest $g(n)$
- goal-informed (A^*): expand lowest $f(n)$

1. Simply lowest f in A* does not do well as search frontier typically miss each other

Search Directions

Bidirectional search can be either

- uniform-cost (Dijkstra): expand lowest $g(n)$
- goal-informed (A^*): expand lowest $f(n)$


Search Directions

Bidirectional search can be either
- uniform-cost (Dijkstra): expand lowest $g(n)$
- goal-informed (A^*): expand lowest $f(n)$

What queue to expand?

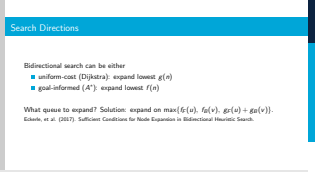
1. Simply lowest f in A* does not do well as search frontier typically miss each other

Search Directions

Bidirectional search can be either

- uniform-cost (Dijkstra): expand lowest $g(n)$
- goal-informed (A^*): expand lowest $f(n)$

What queue to expand?



1. Simply lowest f in A* does not do well as search frontier typically miss each other

Search Directions

Bidirectional search can be either

- uniform-cost (Dijkstra): expand lowest $g(n)$
- goal-informed (A^*): expand lowest $f(n)$

What queue to expand? Solution: expand on $\max\{f_F(u),\ f_B(v),\ g_F(u)+g_B(v)\}$.

Eckerle, et al. (2017). Sufficient Conditions for Node Expansion in Bidirectional Heuristic Search.

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
- Theorem proving

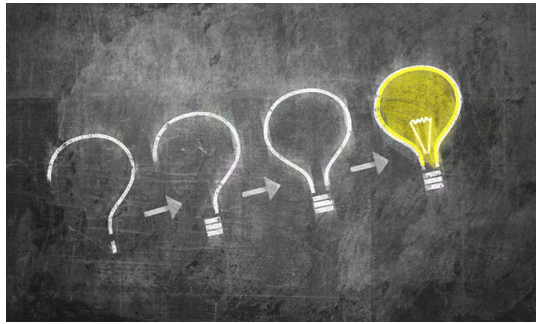
Planning as Constraint Satisfaction Problem

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

Solving Planning Problems
Questions so far?

Solving Planning Problems

Questions so far?

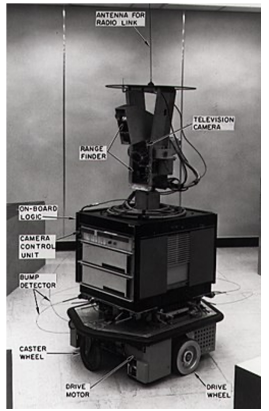


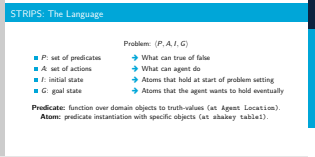
STRIPS
- Stanford Research Institute Problem Solver
- Language + Solver + Search procedure
- Shakey the robot (1971)
- Factored representation of the world

Factored representation: a state is a collection of variables and their values.

STRIPS

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





Atoms are sometimes called fluents (like in Russell & Norvig), but we use the term atom so as not to confuse it with numerical fluents in PDDL.

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

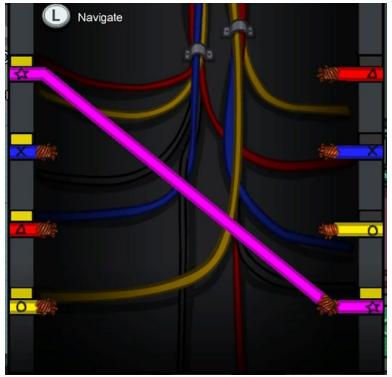
Predicate: function over domain objects to truth-values `(at Agent Location)`.
Atom: predicate instantiation with specific objects `(at shakey table1)`.

Example
Problem: Connect the right wires and then turn on the power.
What are the predicates?
What are the actions?
What is the initial state?
Goal: (power-on)

Example

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

- What are the predicates?
- What are the actions?
- What is the initial state?
  - Goal: `(power-on)`



Example
Problem: Connect the right wires and then turn on the power.
Predicates: (connected Link), (link Link1 Link2), (color Link1 Color), ...
What are the actions?
What is the initial state?
Goal: (power-on)

Example

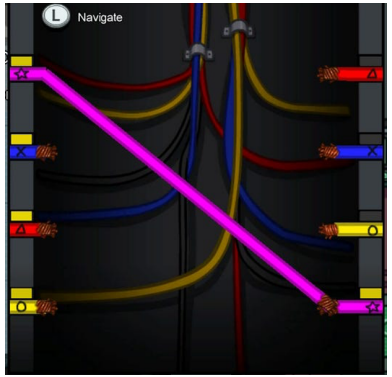
Problem: Connect the right wires and then turn on the power.

- Predicates: `(connected Link)`, `(link Link1 Link2)`, `(color Link1 Color)`, ...

What are the actions?

What is the initial state?

- Goal: `(power-on)`



Example
Problem: Connect the right wires and then turn on the power.
Predicates: (connected Link), (link Link1 Link2), (color Link1 Color), ...
Actions: (connect Link1 Link2), (turn-on-power), ...
What is the initial state?
Goal: (power-on)

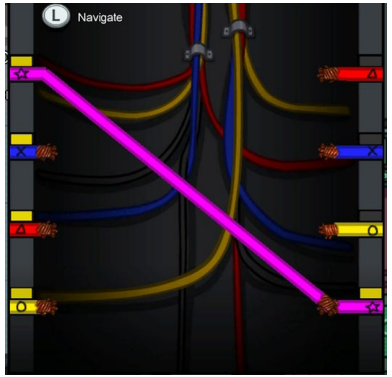
Example

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

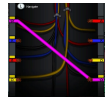
- Predicates: `(connected Link)`, `(link Link1 Link2)`, `(color Link1 Color)`, ...
- Actions: `(connect Link1 Link2)`, `(turn-on-power)`, ...

What is the initial state?

- Goal: `(power-on)`



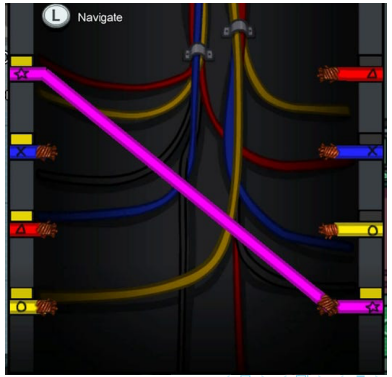
Example
Problem: Connect the right wires and then turn on the power.
- Predicates: (connected Link), (link Link1 Link2), (color Link1 Color), ...
- Actions: (connect Link1 Link2), (turn-on-power), ...
- Initially: (color l3 red), (connected l1 r4), (power-off), ...
- Goal: (power-on)



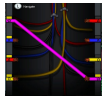
Example

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

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



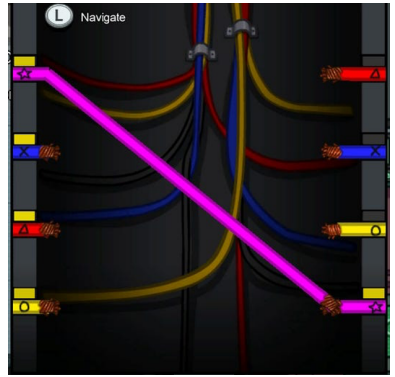
Example
Problem: Connect the right wires and then turn on the power.
Predicates: (connected Link), (link Link1 Link2), (color Link1 Color), ...
Actions: (connect Link1 Link2), (turn-on-power), ...
Initially: (color l3 red), (connected l1 r4), (power-off), ...
What is the goal state?

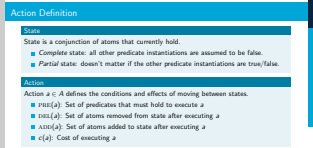


Example

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

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





Action Definition

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.

- $\text{PRE}(a)$: Set of predicates that must hold to execute a
- $\text{DEL}(a)$: Set of atoms removed from state after executing a
- $\text{ADD}(a)$: Set of atoms added to state after executing a
- $c(a)$: Cost of executing a

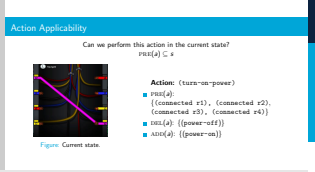


Figure: Current state.

Action Applicability

Can we perform this action in the current state?

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

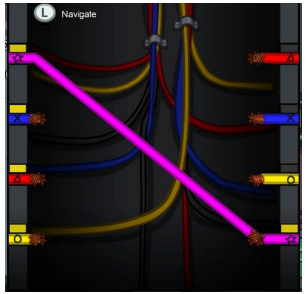


Figure: Current state.

Action: `(turn-on-power)`

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

Action Progression

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

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

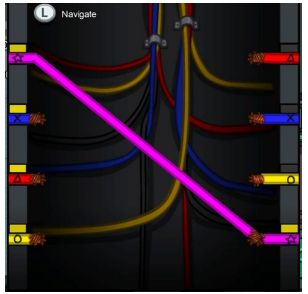


Figure: Current state.

Action: `(connect l2 r2)`

- PRE(a): {`(not (connected l2))`, `(not (connected r2))`, `(= (color l2) (color r2))`}
- 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-power)
PRE(a): {(connected r1), (connected r2), (connected r3), (connected r4)}
DEL(a): {(power-off)}
ADD(a): {(power-on)}

Goal achievement

When are we done?

$G \subseteq s$

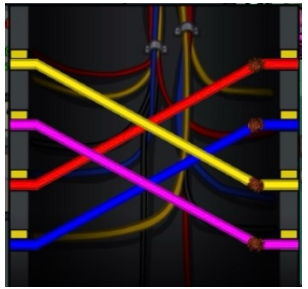


Figure: Current state.

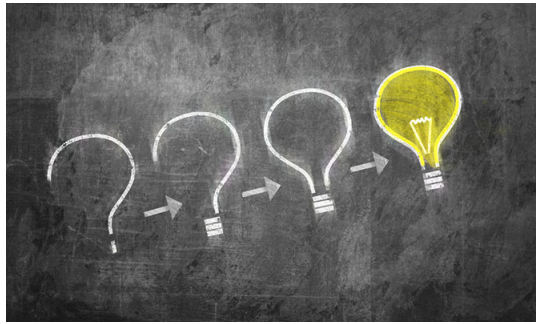
Action: `(turn-on-power)`

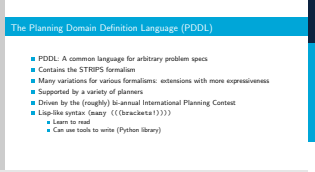
- PRE(a): {`(connected r1)`, `(connected r2)`, `(connected r3)`, `(connected r4)`}
- DEL(a): {`(power-off)`}
- ADD(a): {`(power-on)`}

Modeling Search Problems
Questions so far?

Modeling Search Problems

Questions so far?

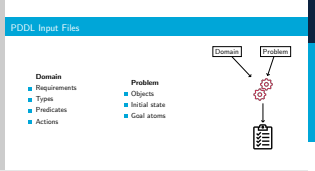




Atoms are sometimes called fluents (like in Russell & Norvig), but we use the term atom, so not to confuse with numerical fluents in PDDL.

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 Contest
- Lisp-like syntax `(many (((brackets!))))`
  - Learn to read
  - Can use tools to write (Python library)



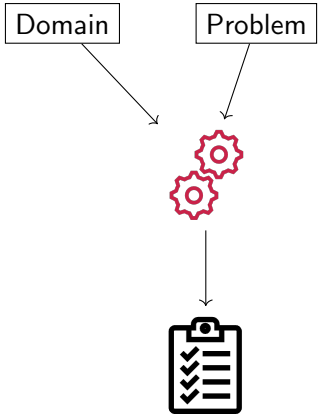
PDDL Input Files

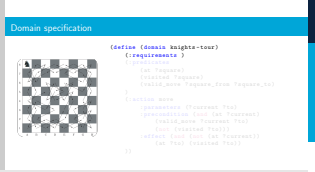
Domain

- Requirements
- Types
- Predicates
- Actions

Problem

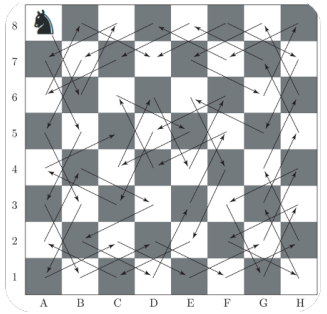
- Objects
- Initial state
- Goal atoms





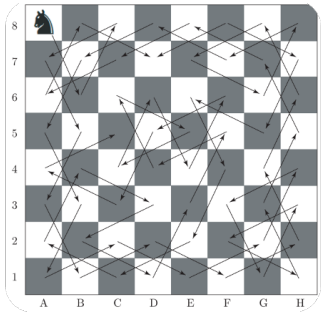
Negative preconditions necessary for the not visited, can also be compiled away.

Domain specification



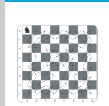

```
(define (domain knights-tour)
    (:requirements )
    (:predicates
        (at ?square)
        (visited ?square)
        (valid_move ?square_from ?square_to)
    )
    (:action move
        :parameters (?current ?to)
        :precondition (and (at ?current)
            (valid_move ?current ?to)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

Domain specification



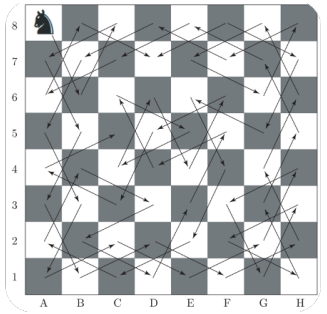

```
(define (domain knights-tour)
    (:requirements )
    (:predicates
        (at ?square)
        (visited ?square)
        (valid_move ?square_from ?square_to)
    )
    (:action move
        :parameters (?current ?to)
        :precondition (and (at ?current)
            (valid_move ?current ?to)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

Domain specification

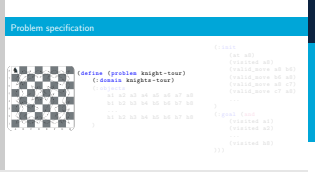


(define (domain knights-tour)
(:requirements)
(:predicates
(at ?square)
(visited ?square)
(valid_move ?square_from ?square_to)
)
(:action move
:parameters (?current ?to)
:precondition (and (at ?current)
(valid_move ?current ?to)
(not (visited ?to)))
:effect (and (not (at ?current))
(at ?to) (visited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

Domain specification

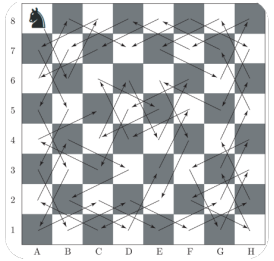



```
(define (domain knights-tour)
    (:requirements )
    (:predicates
        (at ?square)
        (visited ?square)
        (valid_move ?square_from ?square_to)
    )
    (:action move
        :parameters (?current ?to)
        :precondition (and (at ?current)
            (valid_move ?current ?to)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



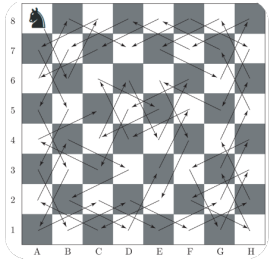
The knight's starting position is arbitrary but always visited

Problem specification




```
(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)
    )))
```

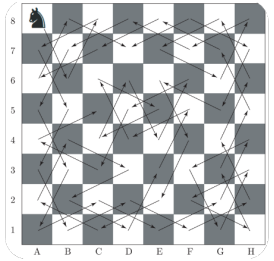
Problem specification




```
(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)
)))
```

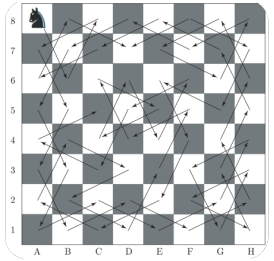
Problem specification




```
(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
    )
```

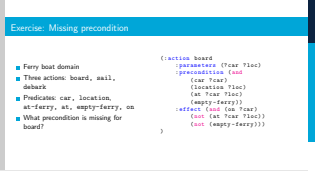
```
(:init
    (at a8)
    (visited a8)
    (valid_move a8 b6)
    (valid_move b6 a8)
    (valid_move a8 c7)
    (valid_move c7 a8)
    ...
)
(:goal (and
    (visited a1)
    (visited a2)
    ...
    (visited h8)
)))
```

Problem specification




```
(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)
)))
```



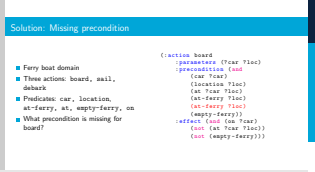
Missing `(at-ferry ?loc)`

Exercise: Missing precondition

- Ferry boat domain
- Three actions: `board`, `sail`, `debark`
- Predicates: `car`, `location`, `at-ferry`, `at`, `empty-ferry`, `on`
- 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 ?car)
        (not (at ?car ?loc))
        (not (empty-ferry)))
)
```

Missing `(at-ferry ?loc)`



Solution: Missing precondition

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

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

Exercise: Spot the mistake

Figure: Towers of Hanoi.

No `(not (clear ?to))` effect.

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 ?to ?disc)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))))
)
```

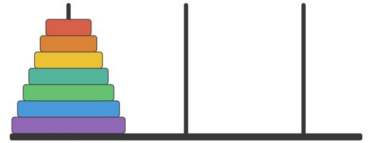
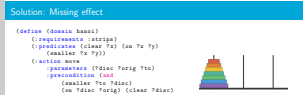


Figure: Towers of Hanoi.

Solution: Missing effect



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 ?to ?disc)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))
        :effect  (and (clear ?orig)
            (on ?disc ?to)
            (not (on ?disc ?orig))
            (not (clear ?to))
        )))
```

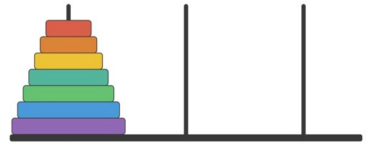
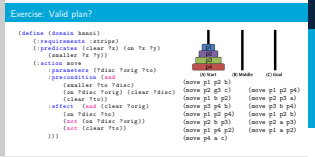


Figure: Towers of Hanoi.

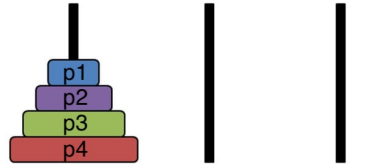
No `(not (clear ?to))` effect.



Commented should be included.

Exercise: Valid plan?

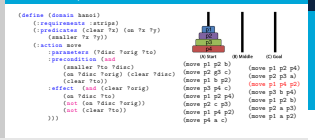
```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates (clear ?x) (on ?x ?y)
        (smaller ?x ?y))
    (:action move
        :parameters (?disc ?orig ?to)
        :precondition (and
            (smaller ?to ?disc)
            (on ?disc ?orig) (clear ?disc)
            (clear ?to))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




```
(move p1 p2 b)
(move p2 g3 c)
(move p1 b p2)
(move p3 p4 b)
(move p1 p2 p4)
(move p2 b p3)
(move p1 p4 p2)
(move p4 a c)
```

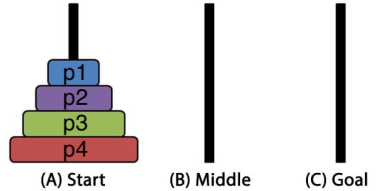
```
(move p1 p2 p4)
(move p2 p3 a)
(move p3 b p4)
(move p1 p2 b)
(move p2 a p3)
(move p1 a p2)
```

Solution: invalid plan



Solution: invalid plan

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




```
(move p1 p2 b)
(move p2 g3 c)
(move p1 b p2)
(move p3 p4 c)
(move p1 p2 p4)
(move p2 c p3)
(move p1 p4 p2)
(move p4 a c)
```

```
(move p1 p2 p4)
(move p2 p3 a)
(move p1 p4 p2)
(move p3 b p4)
(move p1 p2 b)
(move p2 a p3)
(move p1 a p2)
```

PDDL Extension: numeric fluents
Predicates vs fluents
Express numeric properties
Precondition: =, >, <
Effect: increase, decrease
(:functions
(length ?t - track)
(battery)
)
(:init
(length track1 50)
(battery 100)
)
(:effect
(decrease (maxTrips) (length ?t))
)
Figure: Shunting Yard Layout.

PDDL Extension: numeric fluents

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

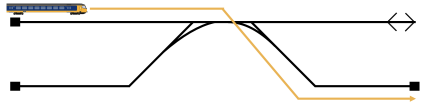
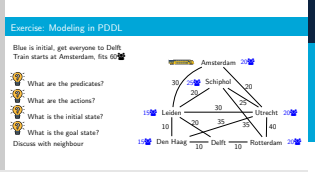


Figure: Shunting Yard Layout.

```
(:functions
    (length ?t - track)
    (battery)
)
(:init
    (length track1 50)
    (battery 100)
)
(:effect
    (decrease (maxTrips) (length ?t))
)
```



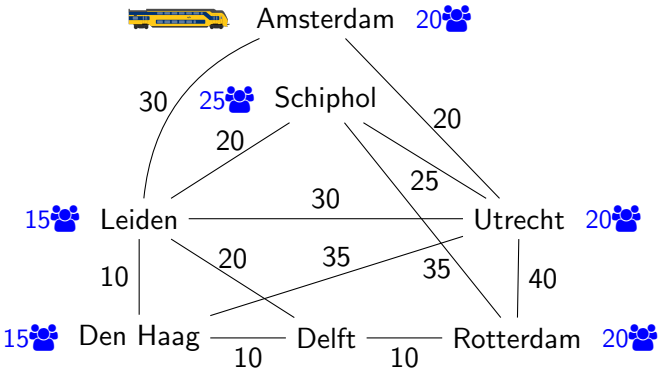
1. Predicates: `(connected ?c1 ?c2)` `(demand ?c1)` `(train-at ?c1)` `(connected ?c1 ?c2)` `(traveltime ?c1 ?c2)`

Exercise: Modeling in PDDL

Blue is initial, get everyone to Delft
Train starts at Amsterdam, fits 60

- What are the predicates?
- What are the actions?
- What is the initial state?
- What is the goal state?

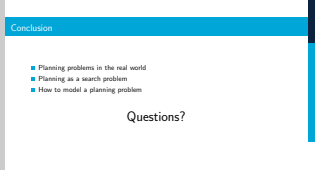
Discuss with neighbour



Relation to Other Lectures
- Search, Inference, Learning, and Optimization
- Effectiveness for solving planning problems

Relation to Other Lectures

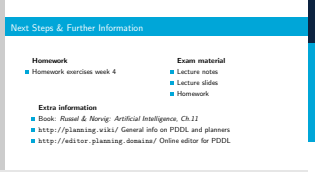
- Search, Inference, Learning, and Optimization
- Effectiveness for solving planning problems



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

More Applications
- Plan Recognition
- Logistics
- Game testing

More Applications

- Plan Recognition
- Logistics
- Game testing