Robot Design Lab

Task Planning: Representation and Algorithms

Original slides from Dana S. Nau, University of Maryland

Slides updated and extended by Dr. Teena Hassan, University of Bremen, for the purpose of the Robot Design Lab course

Updated 04:04 PM January 6, 2022

Abstract Representation for Task Planning

- Real world is absurdly complex. Therefore, we should approximate the world.
  - **Only represent** what the planner needs to reason about

- S = {abstract states}
    - e.g., states might include a robot's location, but not its position and orientation

- A = {abstract actions}
    - e.g., the action "move robot from location-1 to location-2" may need complex lower-level implementation

- Formal representation defines a language (syntax and semantics) for describing the planning domain, initial state and goal in abstract terms.
  - This makes it easier for the planner to generate plans automatically.

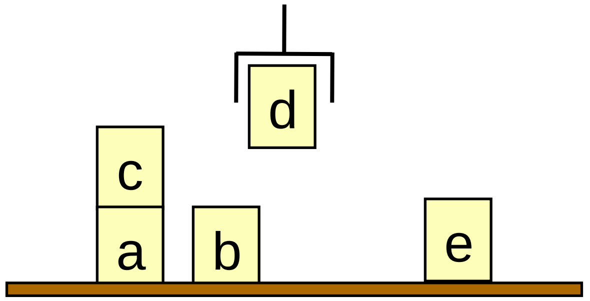
Contents

Example: The Blocks World

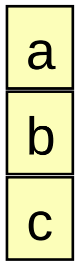
- Infinitely wide table, finite number of children's blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- There's a robot gripper that can hold at most one block

There is one location (table), one gripper (robot hand), and some blocks

- Want to move blocks from one configuration to another
  - e.g.,



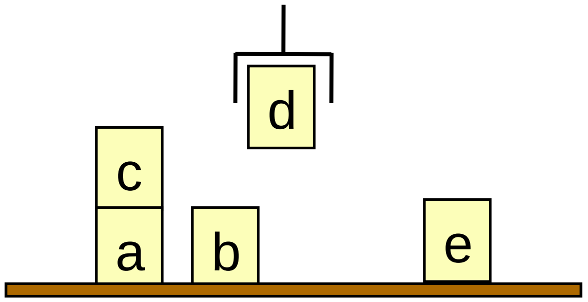
initial state



goal

Classical Representation: Symbols (1/2)

- **Constant symbols** for concrete things:
  - The blocks: a, b, c, d, e

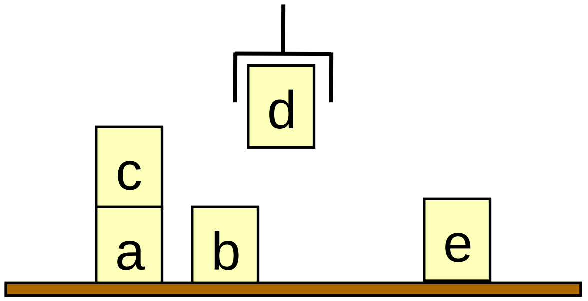


- **Predicates** to represent conditions:
  - ontable(*x*) - block *x* is on the table
  - on(*x*,*y*) - block *x* is on block *y*
  - clear(*x*) - block *x* has nothing on it
  - holding(*x*) - the robot hand is holding block *x*
  - handempty - the robot hand isn't holding anything

- Predicates have the form predicate-symbol(arguments).

Classical Representation: Symbols (2/2)

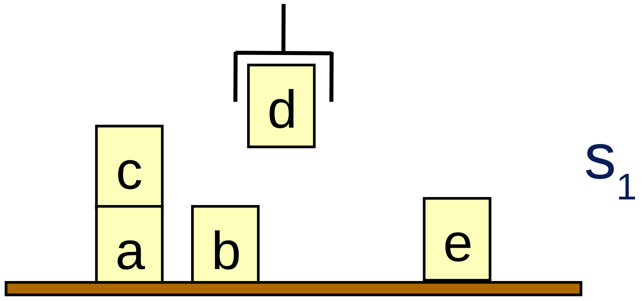
- Predicates have the form predicate-symbol(arguments).
- Arguments in predicates are variables that can be substituted with constant symbols.
  - ontable(a)
  - on(c,a)
  - clear(c)
  - holding(d)
  - ontable(b)
  - ontable(e)



- The process of substituting all variables in a predicate with constant symbols is called **grounding** the predicate.

States

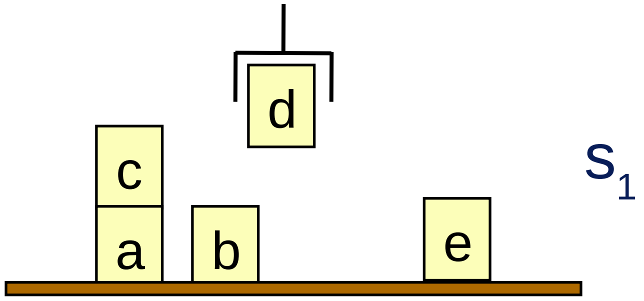
- *State*: a set of ground predicates
  - Represents the conditions or facts that are true in that state.
  - Since we have only a finite no. of predicates and a finite no. of constant symbols, we have only a finite no. of possible states.



s_1 = {ontable(a), ontable(b), ontable(e), on(c,a), clear(c), clear(b), clear(e), holding(d)}

States

- *State*: a set of ground predicates
  - Represents the conditions or facts that are true in that state.
  - Since we have only a finite no. of predicates and a finite no. of constant symbols, we have only a finite no. of possible states.



s_1 = {ontable(a), ontable(b), ontable(e), on(c,a), clear(c), clear(b), clear(e), holding(d)}

- handempty is false, and therefore it is not included in $s_{1.}$
- **Closed world assumption:** What is not known to be true is assumed to be false.

Operators

Operators are generalizations of actions.

Operator: a triple o=(name(o), precond(o), effects(o))

- precond(o): *preconditions*
  - Predicates and their negations that must be true in order to use the operator.
- effects(o): *effects*
  - Predicates and their negations that the operator will make true.
- name(o): a syntactic expression of the form $n(x_1,\ldots,x_k)$
  - n is an *operator symbol* - must be unique for each operator.
  - $(x_1,\ldots,x_k)$ is a list of every variable symbol (parameter) that appears in o.

Operators

Operators are generalizations of actions.

Operator: a triple o=(name(o), precond(o), effects(o))

- precond(o): *preconditions*
- effects(o): *effects*
- name(o): a syntactic expression of the form $n(x_1,\ldots,x_k)$
    - » n is an *operator symbol* - must be unique for each operator
    - » $(x_1,\ldots,x_k)$ is a list of every variable symbol (parameter) that appears in o

Operators

Operators are generalizations of actions.

Operator: a triple o=(name(o), precond(o), effects(o))

- precond(o): *preconditions*
- effects(o): *effects*
- name(o): a syntactic expression of the form $n(x_1,\ldots,x_k)$
  - » n is an *operator symbol* - must be unique for each operator
  - » $(x_1,\ldots,x_k)$ is a list of every variable symbol (parameter) that appears in o

Rather than writing each operator as a triple, we'll usually write like this:

pickup(x)

;; gripper picks up block x from the top of the table

Precond: ontable(x), clear(x), handempty

Effects: ¬ontable(x), ¬clear(x), ¬handempty, holding(x)

¬ indicates negation

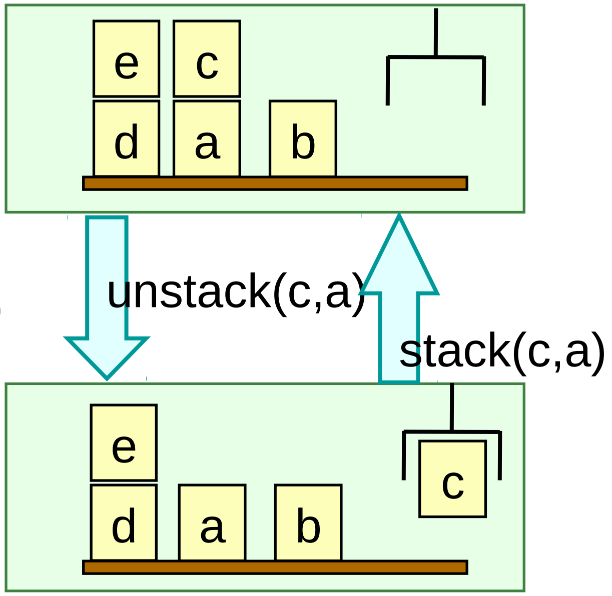
Classical Operators

unstack(x,y)

;; pick block x from the top of block y

Precond: on(x,y), clear(x), handempty

Effects: ¬on(x,y), ¬clear(x), ¬handempty, holding(x), clear(y)



stack(x,y)

;; put block x on top of block y

Precond: holding(x), clear(y)

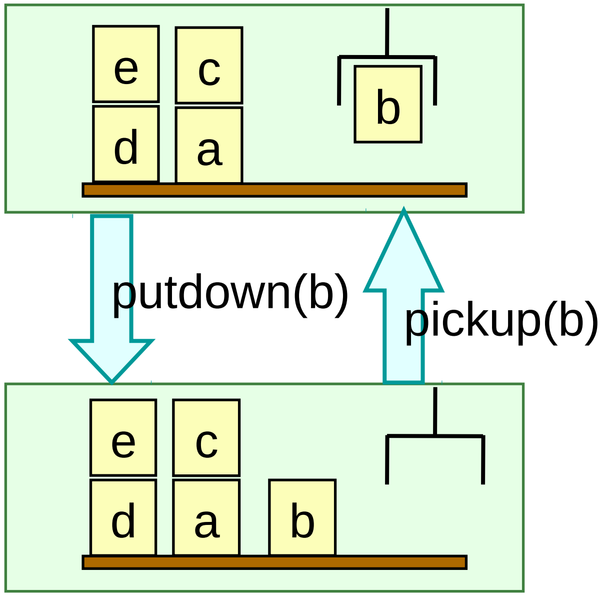
Effects: ¬holding(x), ¬clear(y), on(x,y), clear(x), handempty

pickup(x)

;; pick up block x from the top of the table

Precond: ontable(x), clear(x), handempty

Effects: ¬ontable(x), ¬clear(x), ¬handempty, holding(x)



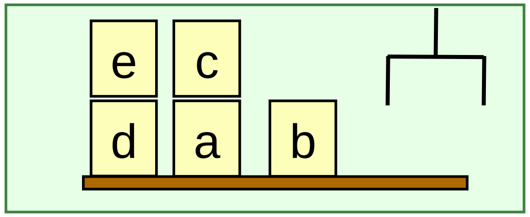
putdown(x)

;; put down block x on top of the table

Precond: holding(x)

Effects: ¬holding(x), ontable(x), clear(x), handempty

Actions



pickup(*x*)

;; gripper picks up block *x*

Precond: ontable(*x*), clear(*x*), handempty

Effects: ¬ontable(*x*), ¬clear(*x*), ¬handempty, holding(*x*)

An *action* is a **ground** instance (via substitution) of an operator

- Suppose we substitute *x* with b.
- Then we get the following action:

pickup(b)

precond: ontable(b), clear(b), handempty

effects: ¬ontable(b), ¬clear(b), ¬handempty, holding(b)

- i.e., The robot's gripper picks up block b off the table.

Notation

Let *a* be an operator or action. Then

- $\text{precond}^+(a)$ = {predicates that appear positively in *a*'s preconditions}
- $\text{precond}^-(a)$ = {predicates that appear negatively in *a*'s preconditions}
- $\text{effects}^+(a)$ = {predicates that appear positively in *a*'s effects}
- $\text{effects}^-(a)$ = {predicates that appear negatively in *a*'s effects}

- Example:

  pickup(b)

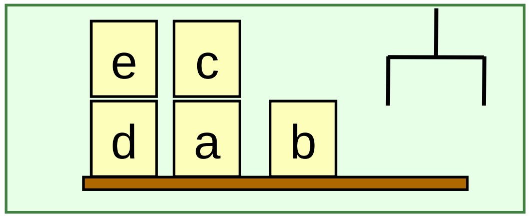
  precond: ontable(b), clear(b), handempty

  effects: ¬ontable(b), ¬clear(b), ¬handempty, holding(b)

- precond^+(pickup(b)) = {ontable(b), clear(b), handempty}
- precond^-(pickup(b)) = { }
- effects^+(pickup(b)) = {holding(b)}
- effects^-(pickup(b)) = {ontable(b), clear(b), handempty)}

Applicability

s_2



Let s be a state and a be an action

a is *applicable* to (or *executable* in) s if s satisfies precond(a)

- $\text{precond}^+(a) \subseteq s$
- $\text{precond}^-(a) \cap s = \varnothing$

An action:

pickup(b)

precond: ontable(b), clear(b), handempty

effects: ¬ontable(b), ¬clear(b), ¬handempty, holding(b)

A state it's applicable to

s_2 = {ontable(d), ontable(a), **ontable(b)**, on(e,d), on(c,a), clear(e), clear(c), **clear(b)**, **handempty**}

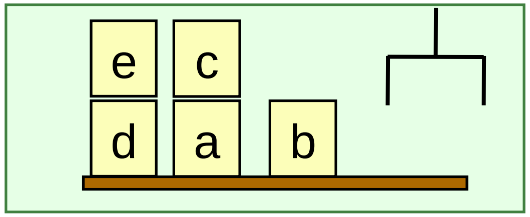
Executing an Applicable Action

Remove *a*'s negative effects from the state,
and add *a*'s positive effects to the state

$\gamma(s,a) = s' = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$

pickup(b)
    precond: ontable(b), clear(b), handempty

    effects: ¬ontable(b), ¬clear(b),
             ¬handempty, holding(b)



s_2

s_2 = {ontable(d), ontable(a), **ontable(b)**, on(e,d), on(c,a), clear(e), clear(c), **clear(b)**, **handempty**}

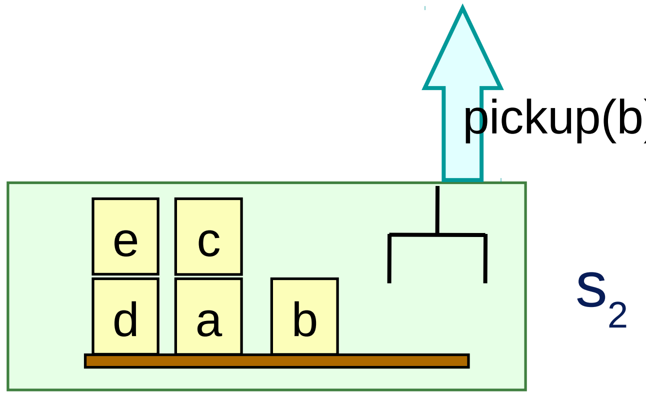
Executing an Applicable Action

Remove *a*'s negative effects from the state,
and add *a*'s positive effects to the state

$\gamma(s,a) = s' = (s - \text{effects}^{-}(a)) \cup \text{effects}^{+}(a)$

pickup(b)
    precond: ontable(b), clear(b), handempty

    effects: ¬ontable(b), ¬clear(b),
    ¬handempty, holding(b)



S$_2$

pickup(b)

s_2 = {ontable(d), ontable(a), **ontable(b)**, on(e,d), on(c,a), clear(e), clear(c), **clear(b)**, **handempty**}

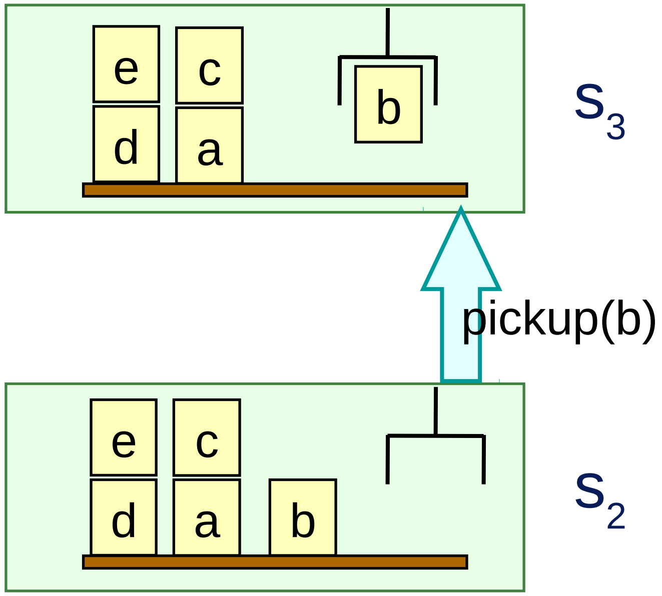
Executing an Applicable Action

Remove *a*'s negative effects from the state,
and add *a*'s positive effects to the state

$\gamma(s,a) = s' = (s - \text{effects}^{-}(a)) \cup \text{effects}^{+}(a)$

pickup(b)
precond: ontable(b), clear(b), handempty

effects: ¬ontable(b), ¬clear(b),
¬handempty, holding(b)



s_3 = {ontable(d), ontable(a), ~~ontable(b)~~,
on(e,d), on(c,a), clear(e), clear(c),
~~clear(b)~~, ~~handempty~~, **holding(b)**}

↑ pickup(b)

s_2 = {ontable(d), ontable(a), **ontable(b)**,
on(e,d), on(c,a), clear(e), clear(c),
clear(b), **handempty**}

Executing an Applicable Action

Remove *a*'s negative effects from the state,
and add *a*'s positive effects to the state

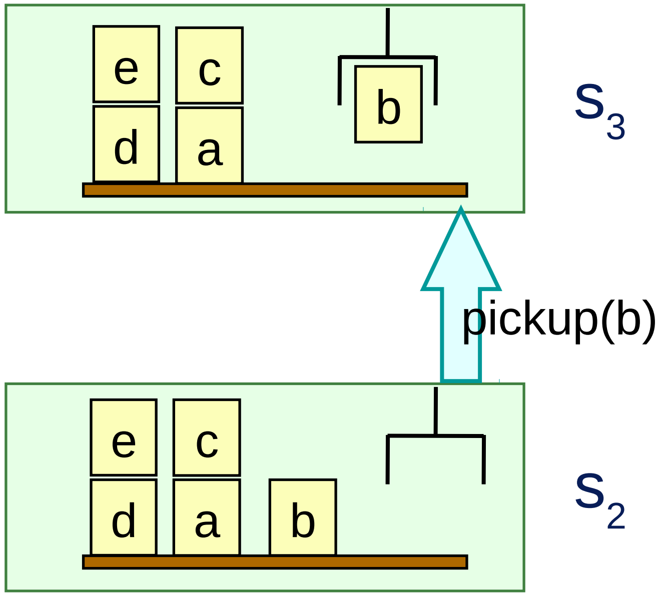
$\gamma(s,a) = s' = (s - \text{effects}^{-}(a)) \cup \text{effects}^{+}(a)$

γ is called the **state transition function.**

$\gamma(s_2,\text{pickup(b)}) = s_3$

pickup(b)
precond: ontable(b), clear(b), handempty

effects: ¬ontable(b), ¬clear(b),
¬handempty, holding(b)



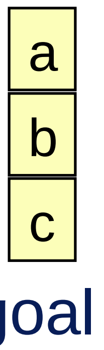
s_3 = {ontable(d), ontable(a), ~~**ontable(b)**~~,
on(e,d), on(c,a), clear(e), clear(c),
~~**clear(b)**~~, ~~**handempty**~~, **holding(b)**}

$\uparrow$ pickup(b)

s_2 = {ontable(d), ontable(a), **ontable(b)**,
on(e,d), on(c,a), clear(e), clear(c),
clear(b), **handempty**}

Goal

- Goal: a set of ground predicates or its negations
  - Represents the conditions that should hold in a goal state.
  - Positive goal conditions should be present in the goal state and negavite goal conditions should be absent in the goal state.



goal = {on(b,c), on(a,b)}

Classical Task Planning Problems

- **Planning domain:**
  - Constant symbols
  - Predicates
  - Operators
- ***Given a planning domain,***
  - The ***statement* of a planning problem**: a triple $P=(O,s_0,g)$
    - » O is the collection of operators
    - » s_0 is the initial state
    - » g is a set of ground predicates or their negations (the goal)
      - Based on g, we can find a set of goal states $\boldsymbol{S_g}$ that satisfy g.

Plans and Solutions

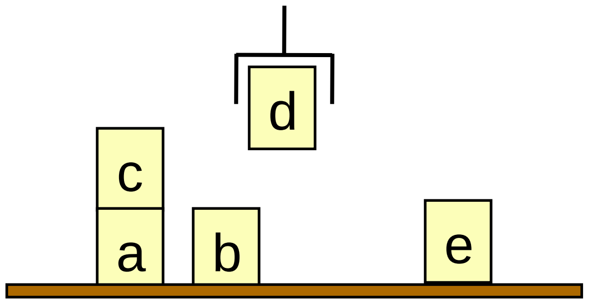
Let $P=(O,s_0,g)$ be a planning problem

Plan: any sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ such that each a_i is a ground instance of an operator in O

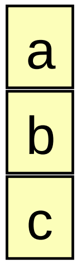
π is a *solution* for $P=(O,s_0,g)$ if it is executable and achieves g

- i.e., if there are states $s_0, s_1, \ldots, s_n$ such that
  - $\gamma(s_0,a_1) = s_1$
  - $\gamma(s_1,a_2) = s_2$
  - …
  - $\gamma(s_{n-1},a_n) = s_n$
  - s_n satisfies g (that is, all positive ground predicates in g are present in s_n and all negative ground predicates in g are absent in s_n)
  - Length of a plan is the no. of actions it contains.

Example: The Blocks World



initial state



goal

- Initial state, s_0:
  - s_0 = {ontable(a), ontable(b), ontable(e), on(c,a), clear(c), clear(b), clear(e), holding(d)}
- Goal
  - g = {on(b,c), on(a,b)}
- Now, we have to come up with a plan (sequence of actions) that will take us from s_0 to any state that satisfies g.
  - π = <putdown(d), unstack(c,a), putdown(c), pickup(b), stack(b,c), pickup(a), stack(a,b)>
  - *Note: Refer Slide11 to see the definitions of all operators.*

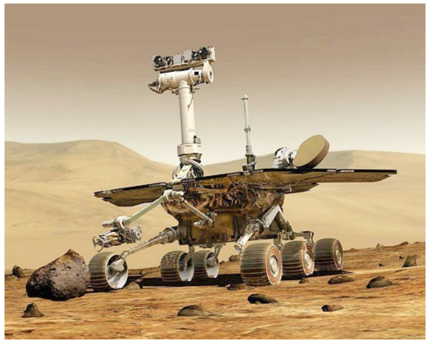
Task Planning Deals with…

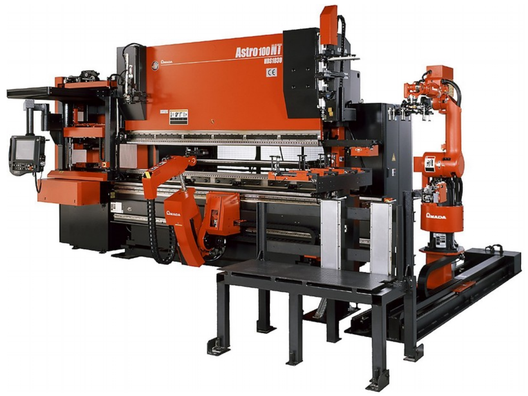
Given a planning domain and a planning problem, how can the robot **autonomously find a plan**, i.e. a sequence of actions, in order to go from the initial state to a state that satisfies the goal?

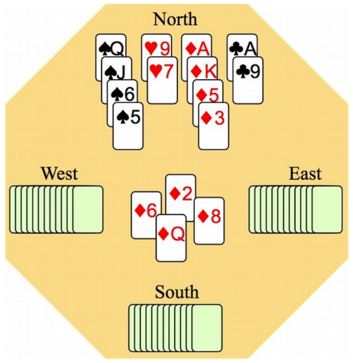
Algorithms that do this are called **planners**.

Three Main Types of Planners

1. Domain-specific planners
   - Made or tuned for a specific planning domain
     - Most successful real-world planning systems work this way (Mars exploration, sheet metal bending, playing bridge, etc.)
   - Won't work well (if at all) in other planning domains
2. Domain-independent planners --> focus of most research
   - In principle, works in any planning domain
   - In practice, need restrictions on what kind of planning domains it can be applied
   - **e.g. Classical task planning**
3. Configurable planners
   - Domain-independent planning engine
   - Input includes info about how to solve problems in some domain







Restrictive Assumptions in Classical Task Planning

A0: Finite system:

- Finite no. of states and actions.

A1: Fully observable:

- The agent always knows the current state of the world.

A2: Deterministic:

- Each action has only one outcome.
- If an action is applied to a state, it will bring the world to a single new state.

A3: Static (no exogenous events):

- Changes to the world state can be caused only through the agent‘s own actions.

A4: Attainment goals:

- Goal is specified explicitly and corresponds to a set of goal states $\boldsymbol{S_g}$.

A5: Sequential plans:

- A plan is a linearly ordered sequence of actions (a_1, a_2, … a_n).

A6: Implicit time:

- No time durations; a linear sequence of instantaneous states and actions.

A7: Off-line planning:

- Planner doesn’t know the execution status of the plan while it is generating the plan.

Planning is Searching…

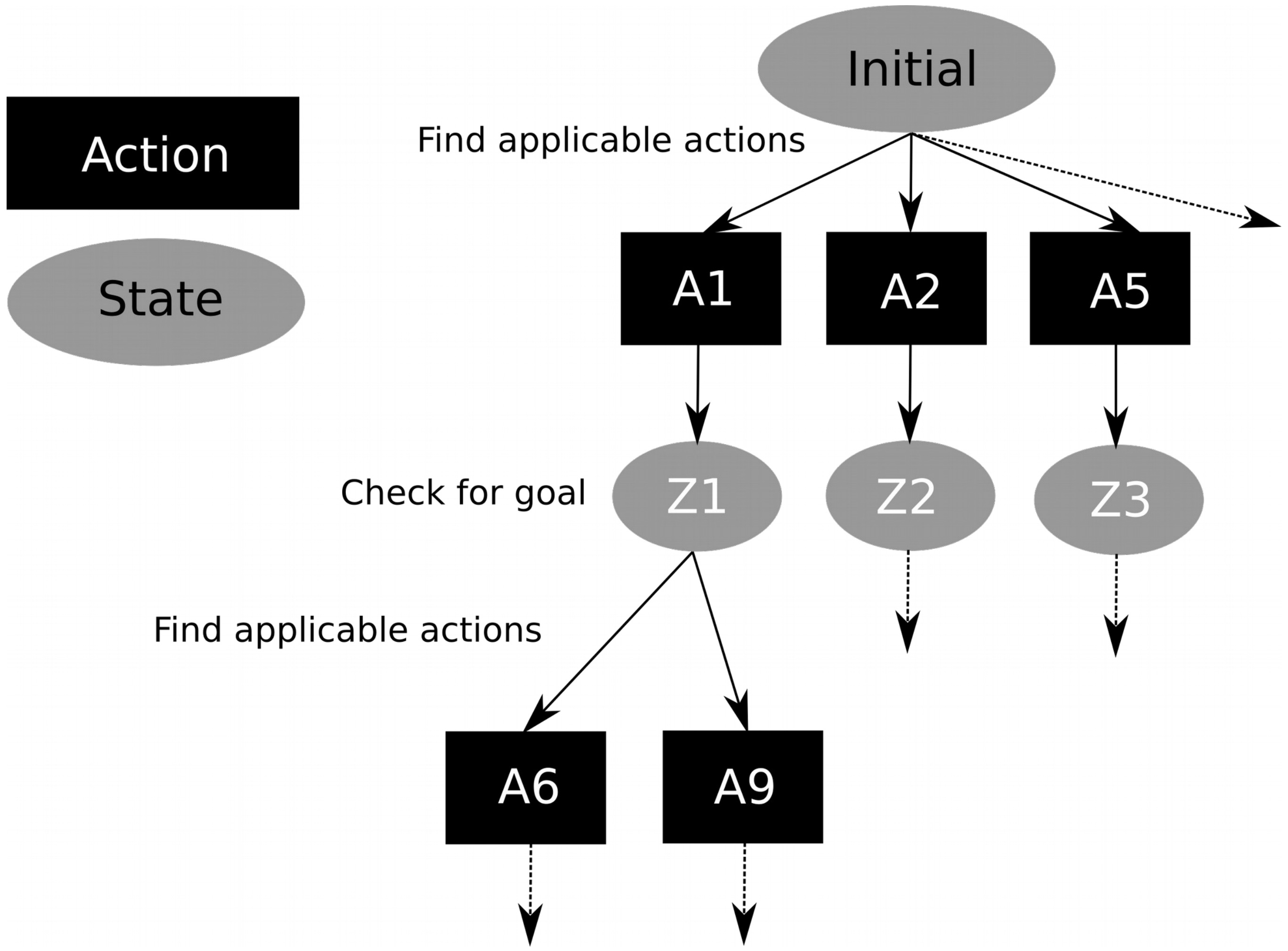
- Nearly all planning procedures are search procedures.
- In this lecture, we will look at ***state-space planning,*** which is used by classical task planners.
  - Similar to path planning algorithms.
  - Here, each node represents a **state** of the world.
  - Finds a path that starts at the initial state and ends at a goal state.

Planning is Searching…

- Nearly all planning procedures are search procedures.
- In this lecture, we will look at ***state-space planning***, which is used by classical task planners.
  - Similar to path planning algorithms.
  - Here, each node represents a **state** of the world.
  - Finds a path that starts at the initial state and ends at a goal state.

  - Search proceeds as follows:
    1. The algorithm starts the search at the initial state.
    2. If the initial state satisfies the goal, then search ends and we get a plan of length zero.
    3. If not, the algorithm applies all **applicable actions** to the initial state to produce new states.
    4. It checks if any of the new states **satisfies the goal**.
    5. If yes, search ends.
    6. If not, then search continues by applying actions to each of the new states.
    7. Search continues until either all reachable states have been examined or a goal state has been reached.
    8. If a goal state is reached, the algorithm **returns the sequence of actions** that led from the initial state to the found goal state.

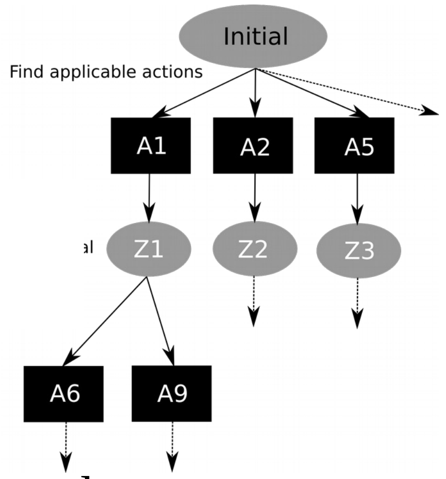
State-Space Planning: Forward Search



Properties

- Forward-search is *sound*
    - Any plan that is returned is guaranteed to be a solution
- Forward-search also is *complete*
    - If a solution exists then Forward-search will return a solution.

Deterministic Implementations

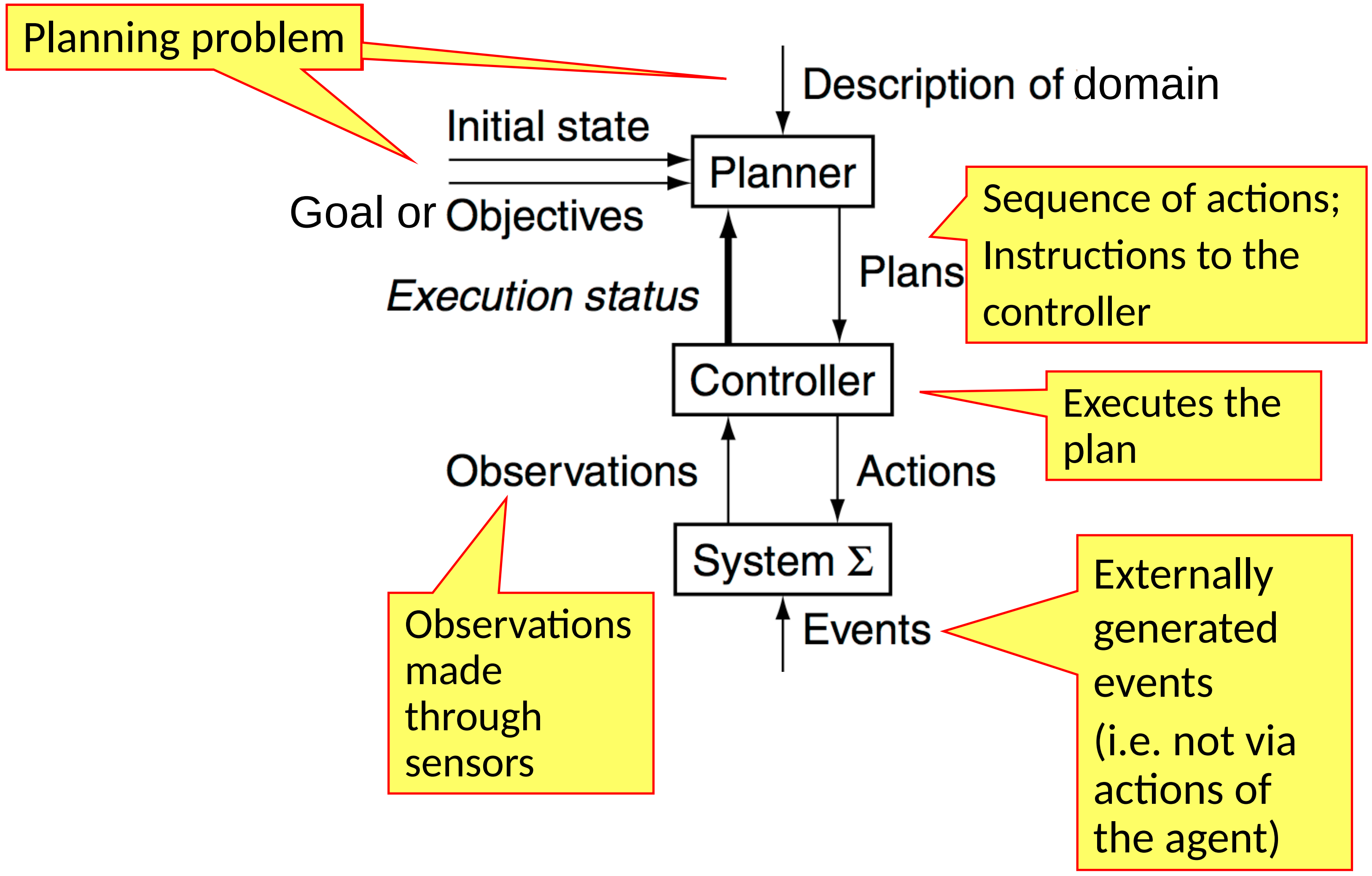


- Some deterministic implementations of forward search:
  - **breadth-first search**
  - **best-first search (e.g., A*)**

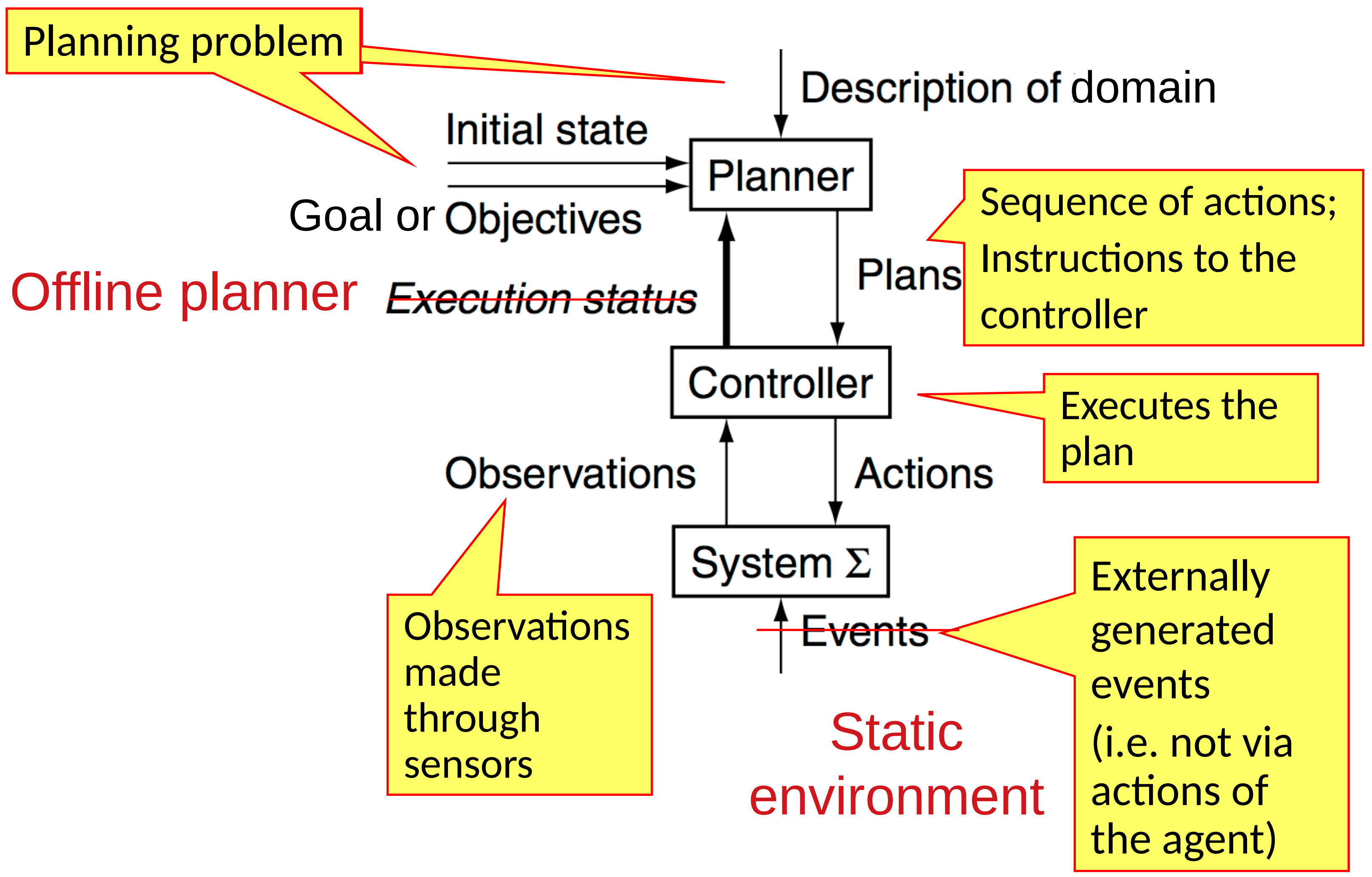
- Breadth-first and best-first search are sound and complete.
  - But they usually aren't practical because they require too much memory.
  - Memory requirement is exponential in the length of the solution.

- Forward search can have a very large branching factor.
  - E.g., many applicable actions that don't progress toward goal.
  - Why this is bad:
    - » Deterministic implementations can waste time trying lots of irrelevant actions.
  - Need a good heuristic function and/or pruning procedure.

Conceptual Model of Planning and Acting



Classical Planning and Acting



Summary

- An example: Blocks world
- Classical representation of the planning domain (constant symbols, predicates, operators), states and actions.
- Applicability of an action to a state
- State transition function
- Assumptions needed for classical task planning
- Statement of a classical task planning problem
- Three types of planners: domain-dependent, domain-independent, configurable
- State-space planning: Forward search
- Conceptual model of planning and acting (plan execution).

Next Part:
Behaviour Trees and Plan Execution