Classical Planning

CS 486/686: Introduction to Artificial Intelligence

Outline

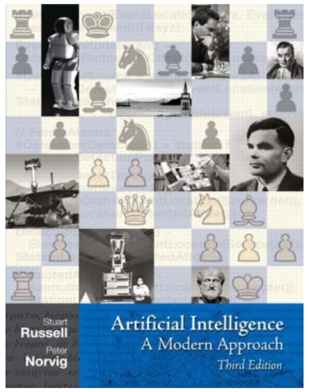
- Planning Problems
- Planning as Logical Reasoning
- STRIPS Language
- Planning Algorithms
- Planning Heuristics

Introduction

- Last class: Logical Inference
  - How to have an agent **understand** its environment using logic.
- This class: Planning
  - How to have an agent **change** its environment, using logic.

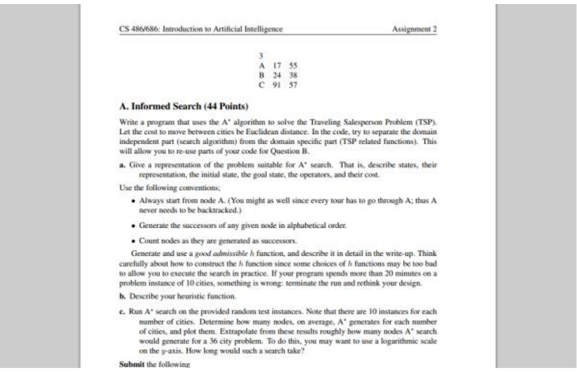
Planning

- A Plan is a collection of actions toward solving a task (or achieving a goal).



```
/**
 * Simple HelloButton() method.
 * @version 1.0
 * @author john doe <doe.j@example.com>
 */
HelloButton()
{
  JButton hello = new JButton( "Hello, wor
  hello.addActionListener( new HelloBtnList

  // use the JFrame type until support for t
  // new component is finished
  JFrame frame = new JFrame( "Hello Button"
  Container pane = frame.getContentPane();
  pane.add( hello );
  frame.pack();
  frame.show();           // display the fra
}
```



CS 486/686: Introduction to Artificial Intelligence — Assignment 2

3		
A	17	55
B	24	38
C	91	57

A. Informed Search (44 Points)

Write a program that uses the A* algorithm to solve the Traveling Salesperson Problem (TSP). Let the cost to move between cities be Euclidean distance. In the code, try to separate the domain independent part (search algorithm) from the domain specific part (TSP related functions). This will allow you to re-use parts of your code for Question B.

a. Give a representation of the problem suitable for A* search. That is, describe states, their representation, the initial state, the goal state, the operators, and their cost.

Use the following conventions:

- Always start from node A. (You might as well since every tour has to go through A; thus A never needs to be backtracked.)
- Generate the successors of any given node in alphabetical order.
- Count nodes as they are generated as successors.

Generate and use a *good admissible* h function, and describe it in detail in the write-up. Think carefully about how to construct the h function since some choices of h functions may be too bad to allow you to execute the search in practice. If your program spends more than 20 minutes on a problem instance of 10 cities, something is wrong: terminate the run and rethink your design.

b. Describe your heuristic function.

c. Run A* search on the provided random test instances. Note that there are 10 instances for each number of cities. Determine how many nodes, on average, A* generates for each number of cities, and plot them. Extrapolate from these results roughly how many nodes A* search would generate for a 36 city problem. To do this, you may want to use a logarithmic scale on the y-axis. How long would such a search take?

Submit the following

Planning

- Properties of (classical) planning:
  - Fully observable
  - Deterministic
  - Finite
  - Static
  - Discrete

Planning Problem

- **Problem:** Find a sequence of actions that moves the world from one state to another state
- The shortest (or fastest) plan is **optimal**
- Need to **reason** about what different actions will do to the world

Planning Problem

- **Goal:** Assignment is written, AND Student has Coffee, AND (John has Assignment OR Kate has Assignment)....
- **Current State:** Assignment is not written, AND Student has no Coffee, AND Coffee_Pot is Empty AND Coffee_Mug is Dirty...
- **To Do:** Clean Coffee_mug AND Place Coffee in Coffee_Pot AND Activate Coffee_Pot AND Write Assignment_Introduction AND...

Outline

- Planning Problems
- Planning as Logical Reasoning
- STRIPS Language
- Planning Algorithms
- Planning Heuristics

Planning as Theorm Proving

1. Represent states as FOL expressions.

2. Represent actions as mappings from state to state (like rules of inference)

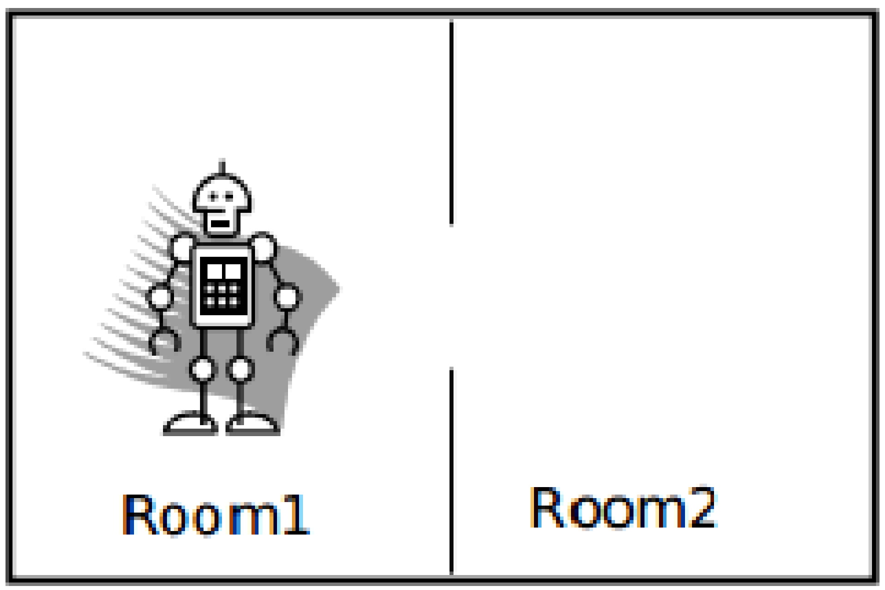
3. Apply theorem provers (search)

Situation Calculus

- A **situation** is a representation of the state of the world.
- All our predicates and functions should depend on the situation.
  - e.g. crown(John) -> crown(John, **s**)
  - e.g. in(Room1, Robot, 1) -> in(Room1, Robot, **s**)

Situation Calculus

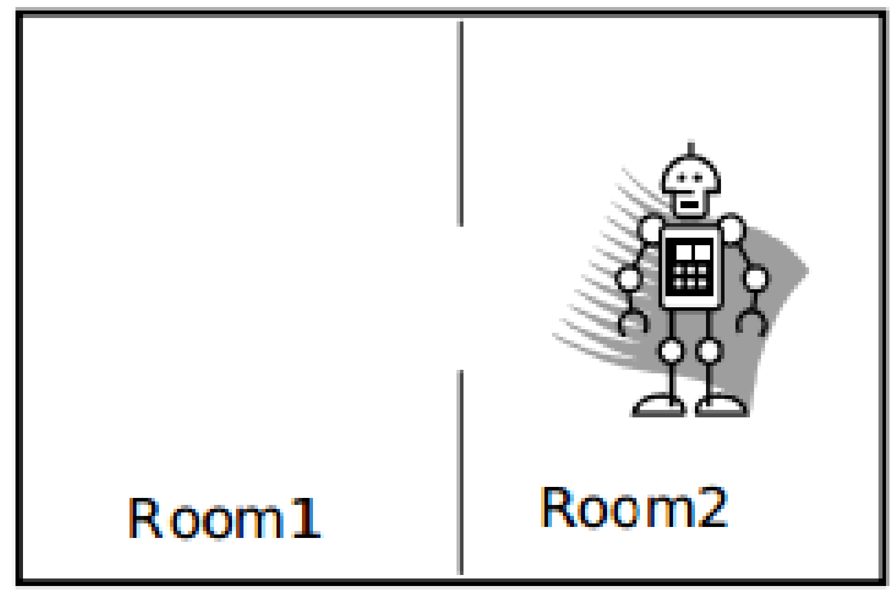
Situation S0



in(robby,room1,s0)

~in(robby,room2,s0)

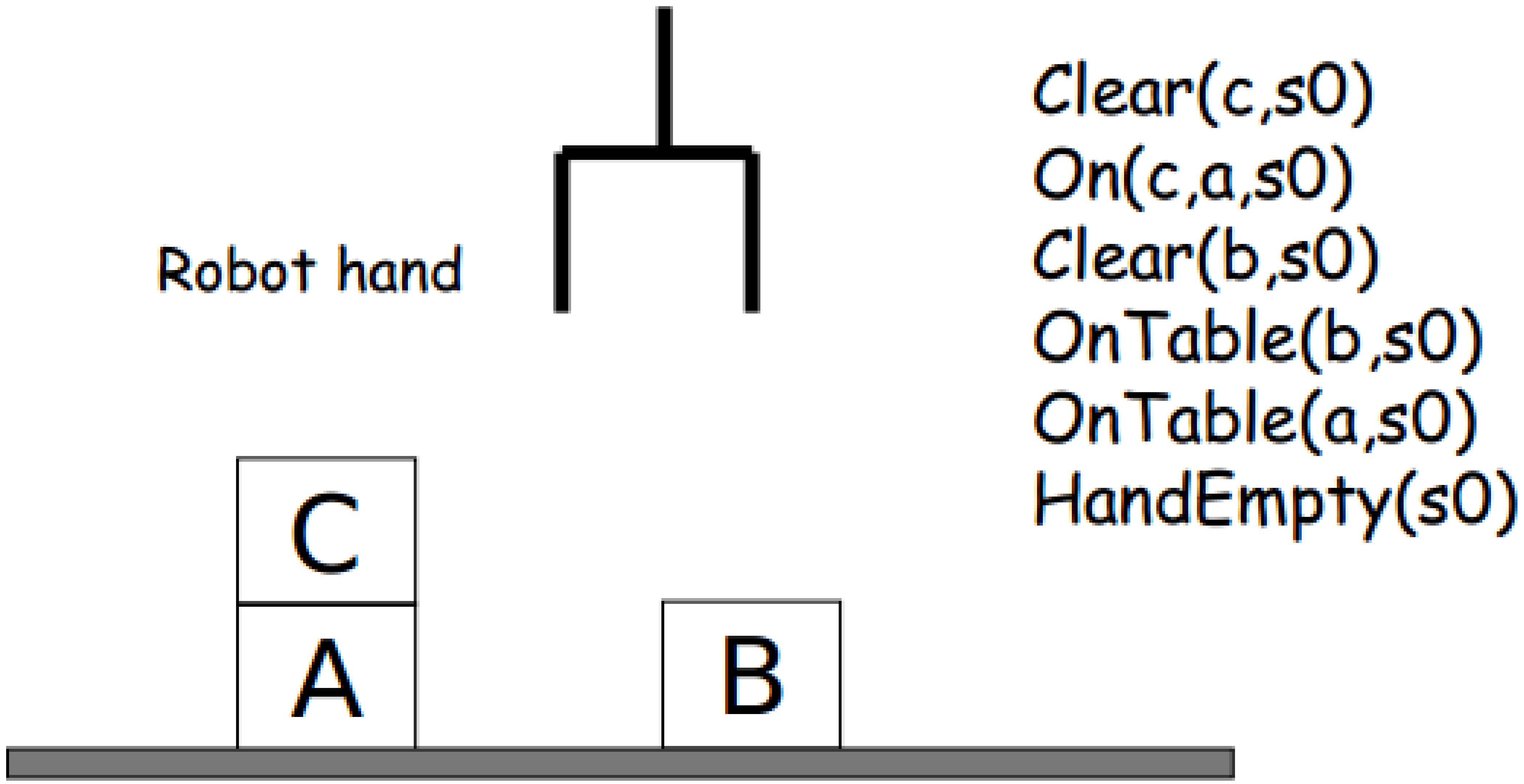
Situation S1



~in(robby,room1,s1)

in(robby,room2,s1)

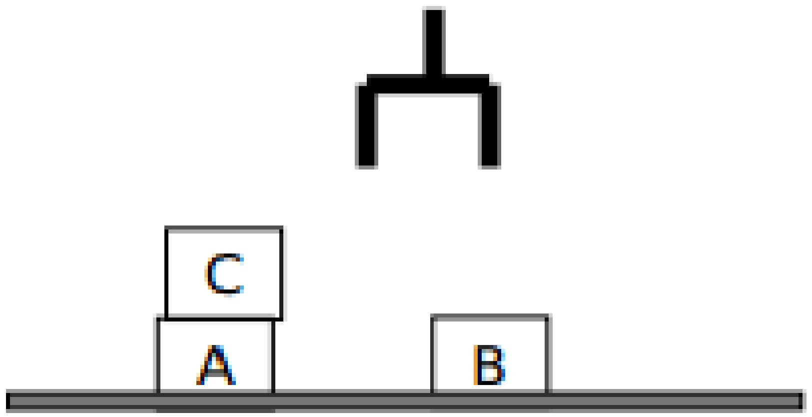
Situation Calculus

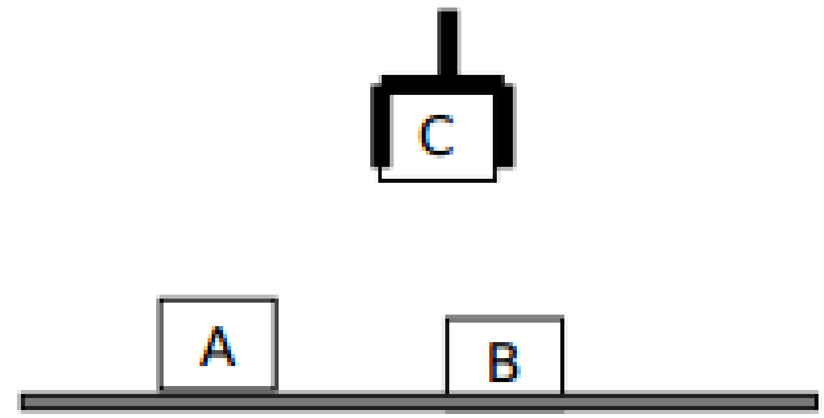


Actions

- **Actions** make atomic changes to the environment
- Allows transitions between situations
  - e.g. result(clean(Coffee_Mug), s0)) is s0 where clean(Coffee_Mug) is now true.

Actions Example





Clear(c,s0)
On(c,a,s0)
Clear(b,s0)
OnTable(b,s0)
OnTable(a,s0)
HandEmpty(s0)

PickUp(c) →

Clear(a,s1)
Clear(b,s1)
OnTable(b,s1)
OnTable(a,s1)
~HandEmpty(s1)

Describing Actions

- Actions are described by a **possibility axiom** and **effect axiom**
- Possibility axiom ~ precondition
- Effect axiom ~ postcondition

Describing Actions

PickUp(*C*)

Preconditions

Clear(*C*,*S*) $\wedge$
HandEmpty(*S*)

Effects

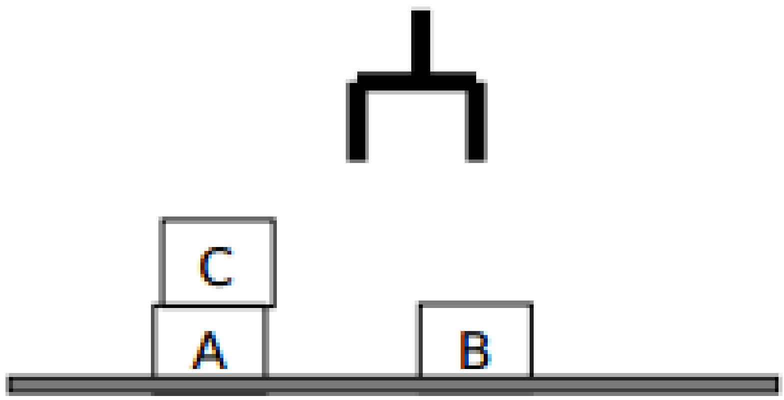
Holding(*C*,Result(PickUp(*C*),*S*))

$\forall$ x ~HandEmpty(Result(PickUp(*C*),*S*))

Planning

Making plans

1. Clear(C,s0)
2. On(C,A,s0)
3. Clear(B,s0)
4. OnTable(A,s0)
5. OnTable(B,S0)
6. HandEmpty(s0)



Query the KB about what actions should be performed in order to achieve some goal (expressed as a predicate)

∃ z Holding(B,z)

7. (~Holding(B,Z) v ans(Z))

Resolution

- Convert to CNF

  (possibility axiom) → (effect axiom)

  - OnTable(y,s) AND Clear(y,s) AND HandEmpty(s) → Holding(y, Result(Pickup(y),s)) AND ~HandEmpty(y, Result(Pickup(y),s)....
  - ~OnTable(y,s) OR ~Clear(y,s) OR ~HandEmpty(s) OR Holding(y,Result(Pickup(y),s))
  - ~OnTable(y,s) OR ~Clear(y,s) OR ~HandEmpty(s) OR ~HandEmpty(y,Result(Pickup(y),s))
  - ...

The Answer

1. Ask query: $\exists$ z Holding(B,z)

   7. (~Holding(B,Z) v ans(Z))
2. Use Resolution to find z.
3. z = Result(Pickup(B),s0)

   - A situation where you are holding B is called "Result(Pickup(B),s0)".
   - Name communicates the actions to take to achieve the goal

The Frame Problem

- What about the question:
  - On(C,A,Result(Pickup(B), s0)?
  - Is C still on A after we pick up B?

1. Clear(C,s0)
2. On(C,A,s0)
3. Clear(B,s0)
4. OnTable(A,s0)
5. OnTable(B,S0)
6. HandEmpty(s0)

7.~OnTable(y,s)v~Clear(y,s)v~HandEmpty(s) vHolding(y,Result(PickUp(y),s))

8.~OnTable(y,s)v~Clear(y,s)v~HandEmpty(s)v~HandEmpty(Result(PickUp(y),s))

9.~OnTable(y,s)v~Clear(y,s)~HandEmpty(s)v~OnTable(y,Result(PickUp(y),s))

10.~OnTable(y,s)v~Clear(y,s)v~HandEmpty(s)v~Clear(y,Result(PickUp(y),s)))

11. ~On(C,A,Result(PickUp(B),s0))

The Frame Problem

- What about the question:
  - On(C,A,Result(Pickup(B), s0)?
  - Is C still on A after we pick up B?

1. Clear(C,s0)
2. On(C,A,s0)
3. Clear(B,s0)
4. OnTable(A,s0)
5. OnTable(B,S0)
6. HandEmpty(s0)

7.~OnTable(y,s)v~Clear(y,s)v~HandEmpty(s) vHolding(y,Result(PickUp(y),s))

8.~OnTable(y,s)v~Clear(y,s)v~HandEmpty(s)v~HandEmpty(Result(PickUp(y),s))

9.~OnTable(y,s)v~Clear(y,s)~HandEmpty(s)v~OnTable(y,Result(PickUp(y),s))

10.~OnTable(y,s)v~Clear(y,s)v~HandEmpty(s)v~Clear(y,Result(PickUp(y),s)))

11. ~On(C,A,Result(PickUp(B),s0))

The Frame Problem

- What about the question:
  - On(C,A,Result(PickUp(B), s0)?
  - Is C still on A after we pick up B?

1. Clear(C,s0)
2. On(C,A,s0)
3. Clear(B,s0)
4. OnTable(A,s0)
5. OnTable(B,S0)
6. HandEmpty(s0)

7.~OnTable(y,s)v~Clear(y,s)v~HandEmpty(s) vHolding(y,Result(PickUp(y),s))

8.~OnTable(y,s)v~Clear(y,s)v~HandEmpty(s)v~HandEmpty(Result(PickUp(y),s))

9.~OnTable(y,s)v~Clear(y,s)~HandEmpty(s)v~OnTable(y,Result(PickUp(y),s))

10.~OnTable(y,s)v~Clear(y,s)v~HandEmpty(s)v~Clear(y,Result(PickUp(y),s)))

11. ~On(C,A,Result(PickUp(B),s0))

The Frame Problem

- Resolution computes logical consequences.
- Consequences of PickUp(B) do not specify anything about what happens to On(A,C)
- Recording all non-effects of actions becomes tedious in detailed domains.
  - In some (but not all) worlds after PickUp(B), On(A,C).

A Better Way?

- Planning as theorem proving generally not efficient.
- Can we specialize for the domain?
  - Connect actions and state descriptions
  - Allow adding actions in any order
  - Partition into subproblems
  - **Use a restricted language for describing goals, states and actions**

Outline

- Planning Problems
- Planning as Logical Reasoning
- STRIPS Language
- Planning Algorithms
- Planning Heuristics

Planning Languages

- Planning languages provide a formal, efficient, way to represent problems, using a restricted subset of FOL
- STRIPS used an early Planning Language
- Many important successors based on this language

STRIPS Language

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- **Domain:** Only typed objects allowed (ground terms)
  - Allowed: Coffee_Pot, Shakey_Robot
  - Not Allowed: x, y, father(x)
- **States:** Conjunctions of predicates over objects
  - Allowed: Full(Coffee_Pot) AND On(Robot, Coffee_Pot)
  - Not Allowed: On(x,y) AND Full(x)
- **Closed World Assumption:** Things not explicitly stated are false.

STRIPS Language

- **Goals:** Conjunctions of **positive** ground literals
  - Allowed: isHappy(Robot) AND isFull(Coffee_Pot)
  - Not Allowed:
    - ~isHappy(Robot)
    - isHappy(father(Robot))
    - isHappy(Robot) OR isFull(Coffee_Pot)

STRIPS Language

- **Actions: Specified by preconditions and effects**
  - E.g.: Action Fly(p,from,to)
  - Precondition: At(p, from) AND isPlane(p) AND isAirport(from AND isAirport(to)
  - Effect: ~At(p,from) AND At(p,to)

STRIPS Language

- **Actions Scheme:**
  - **Name and parameter list** (e.g. Fly(p,from,to))
  - **Precondition** as a conjunction of function-free **positive** literals
  - **Effect** as a conjunction of function-free literals
  - Variables in the effect must be from the parameter list.

Effects of Actions

- When preconditions are false, actions have no effect.
- When preconditions are true, actions change the world by:
    1. **Deleting** any precondition terms that are now false.
    2. **Adding** any new terms that are now true.
- Example: Fly(p,to,from) first deletes At(p,from), and then adds At(p,to).
- **Order matters:** Delete first

STRIPS Language

- **Solution:** Sequence of actions that, when applied to start state, yield goal state.

Frame Problem?

- No problem here!
- Closed World Assumption: anything unmentioned is implicitly unchanged.
- Reduced language → efficient inference

Pros and Cons

- Pros:
  - Restricted language means fast inference
  - Simple conceptualization: Every action just deletes or adds propositions to KB
- Cons:
  - Assumes actions produce few changes
  - Restricted language means we can't represent every problem

Outline

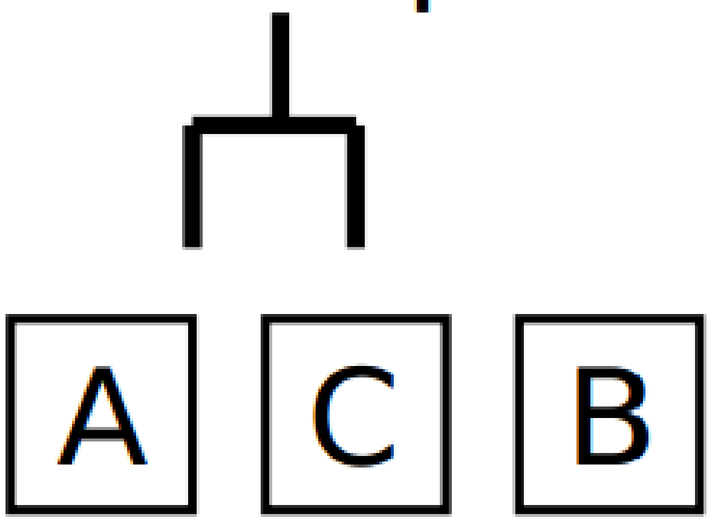
- Planning Problems
- Planning as Logical Reasoning
- STRIPS Language
- Planning Algorithms
- Planning Heuristics

Forward Planning

- Planning as Search
  - **Start State:** Initial state of the world
  - **Goal State:** Goal state of the world
  - **Successors:** Apply every action with a satisfied precondition
  - **Costs:** Usually 1 per action
- Aka "Progressive Planning"

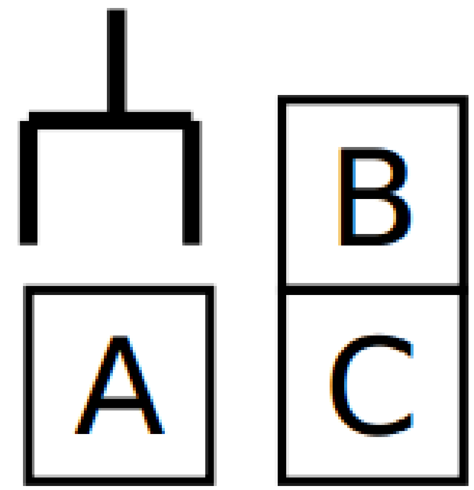
Forward Planning

Example: Block World



Clear(c)
Clear(a)
Clear(b)
OnTable(b)
OnTable(a)
OnTable(c)
HandEmpty()

Goal



Clear(a)
Clear(b)
OnTable(b)
OnTable(a)
HandEmpty()

Pickup(x)
P: OnTable(x), Clear(x), HandEmpty
E: Holding(x), ~OnTable(x), ~HandEmpty

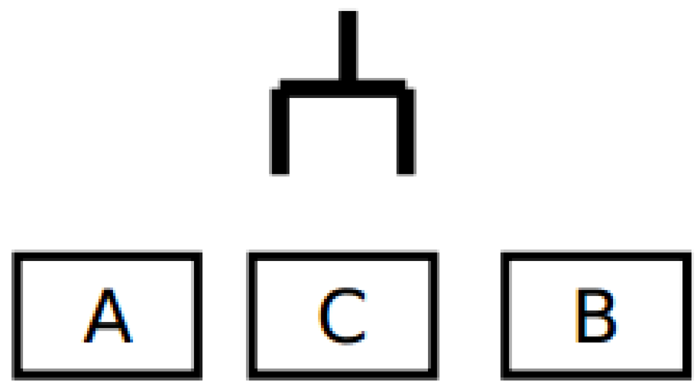
Stack(x,y)
P: Holding(x), Clear(y)
E: On(x,y), Clear(x),
HandEmpty, ~Clear(y),
~Holding(x)

PutDown(x)
P: Holding(x)
E: OnTable(x), Clear(x), HandEmpty,
~Holding(x)

UnStack(x,y)
P: Clear(x), On(x,y), HandEmpty
E: Clear(y), Holding(x),
~Clear(x), ~On(x,y),
~HandEmpty

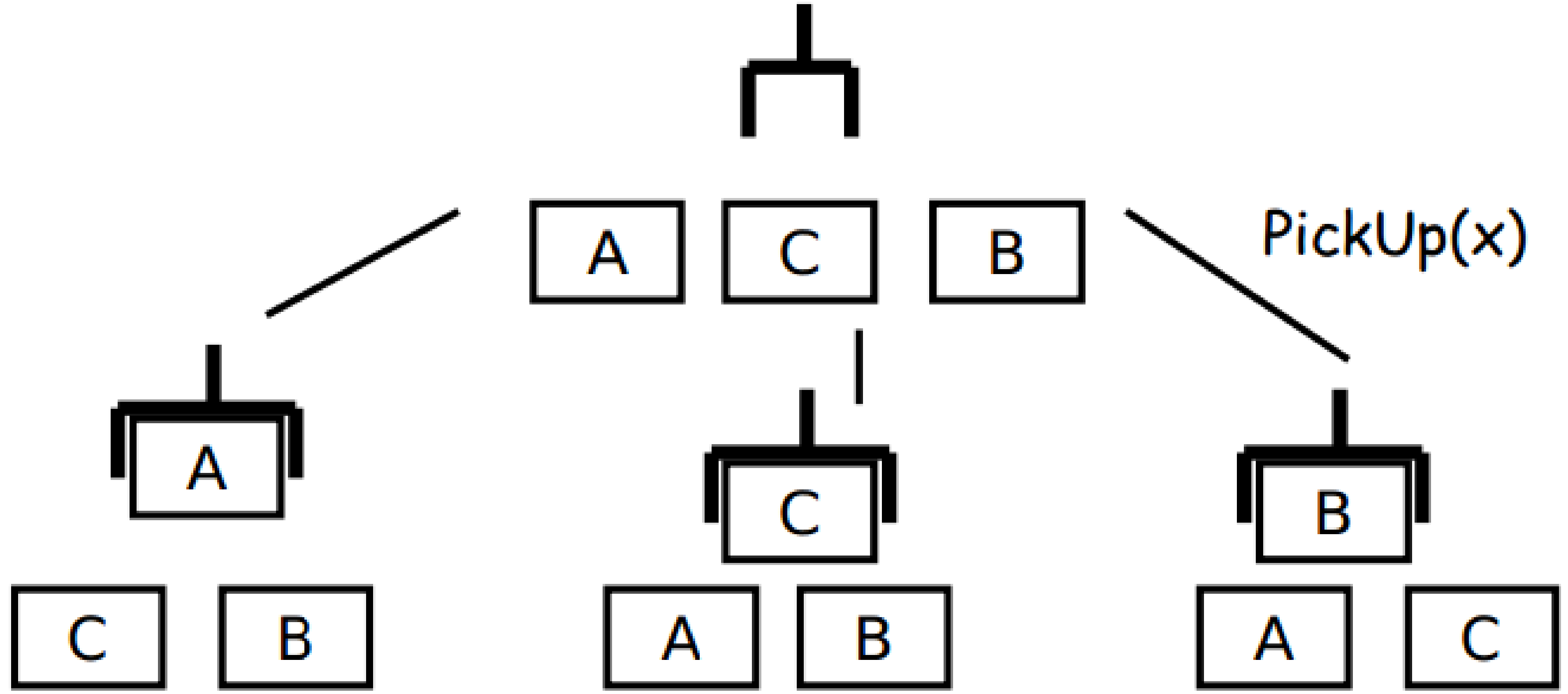
Forward Planning

Example: Progressive Planner

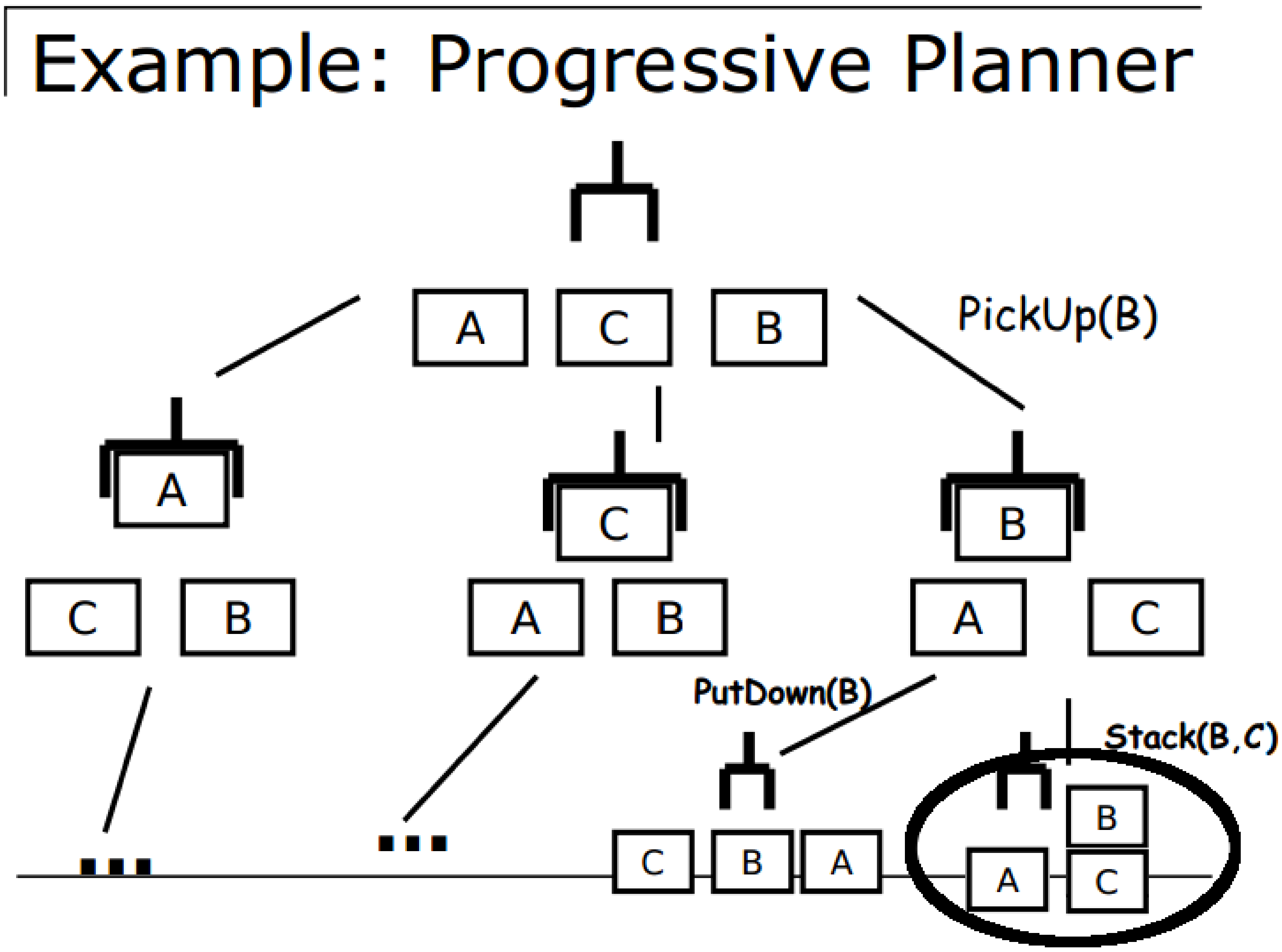


Forward Planning

Example: Progressive Planner



Forward Planning



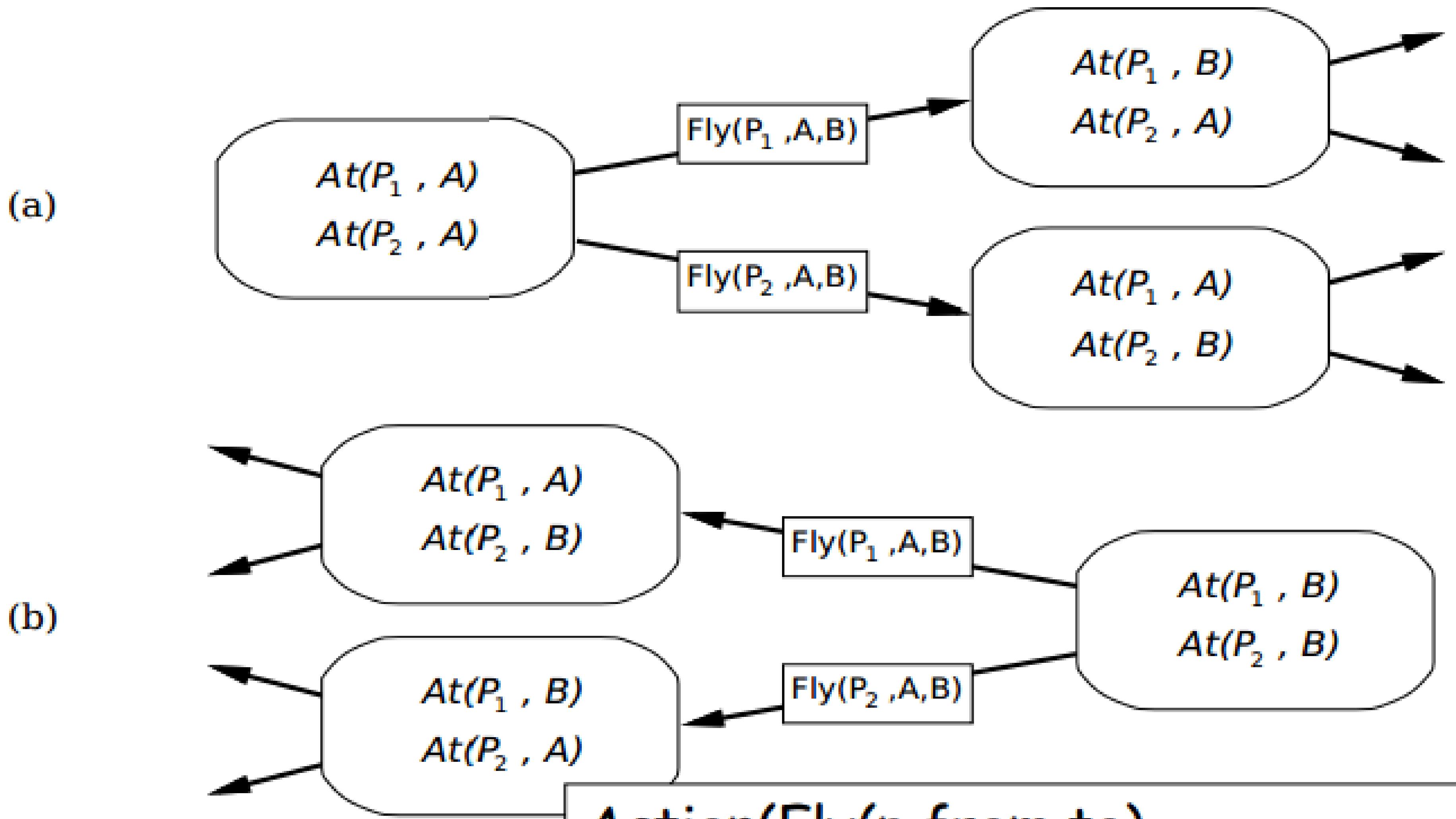
Backward Planning

- Relevant actions
  - Only consider actions that actually satisfy (add) a goal state literal.
- Consistent actions
  - Only consider actions that don't undo (delete) a desired literal

Backward Planning

- Backward Search
  - Start at the Goal state G
  - Pick a consistent, relevant action A
  - Delete whatever part of G is satisfied by A
  - Add A's precondition to G (except duplicates)
  - Repeat with updated G
- Aka "regression planning"

Backward Planning



Action(Fly(p,from,to),

PRECOND: At(p,from)∧ Plane(p)∧ Airport(from)∧ Airport(to)

EFFECT: ~At(p,from)∧ At(p,to))

Outline

- Planning Problems
- Planning as Logical Reasoning
- STRIPS Language
- Planning Algorithms
- Planning Heuristics

Planning Heuristics

- State space can be very (very) large
- Many **domain independent** heuristics

Planning Heuristics

- Generally based on relaxation
  - ignore effects undoing part of the goal state
  - ignore prerequisites when picking actions
  - assume sub-problems never interact

Planning Heuristics

- Better heuristics represent some co-dependecies between goals as a graph
- The algorithm **GraphPlan** can reason over this graph directly
  - This is a very fast approach in practice.

Summary

- Planning is another form of Search
- Planning is usually done in specialized representation languages
- Like CSPs, we can exploit the problem structure to get general heuristics

Outline

- Planning Problems
- Planning as Logical Reasoning
- STRIPS Language
- Planning Algorithms
- Planning Heuristics
- The Sussman Anomaly

STRIPS Algorithm

- Uses a Regression Planner
- Stores current state of the world
- Stores a stack of goals and actions

STRIPS Algorithm

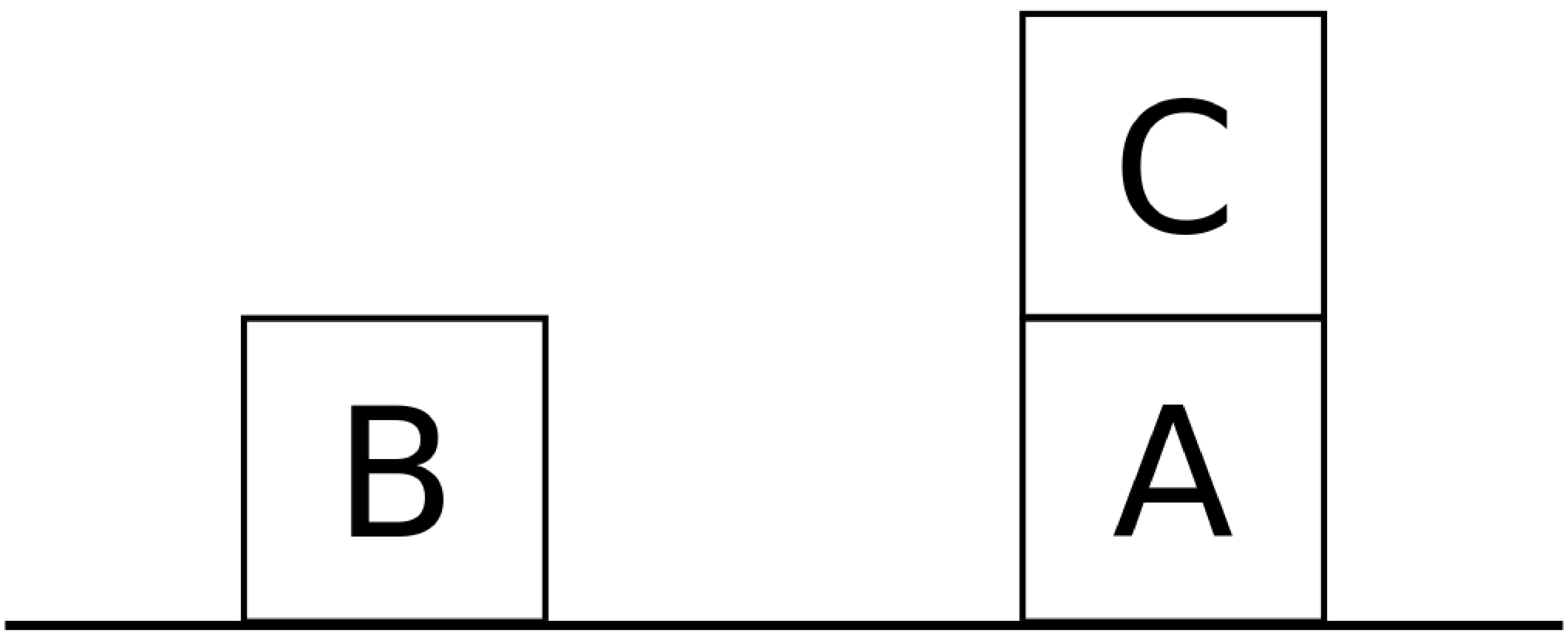
- Push initial goals in any order.
- If stack top is a goal:
  - Push relevant action, and then its prerequisites (new goals).
    - Or just pop if it's already true in the current state.
- If stack top is an action:
  - If prereqs all satisfied, alter state.
    - Push prereqs again if some are unsatisfied.

Sussman Anomaly

- STRIPS seems like a good planning algorithm
  - Simple
  - Representation can model many problems
- ... but STRIPS cannot always find a plan

Sussman Anomaly

The impossible problem:
Stack A on B, and B on C



Sussman Anomaly

- A problem with all approaches that naively split problems into subgoals
- STRIPS is incomplete.