Planning

CSE 4308/5360 – Artificial Intelligence I
University of Texas at Arlington

What is Planning

- The goal in artificial intelligence is to emulate intelligent/rational behavior.
- An important part of rational behavior is making plans:
  - Constructing a sequence of actions that achieves a certain goal.

Planning and Search

- The definition of the planning problem (constructing a sequence of actions that achieves a goal) sounds very similar to the definition of the search problem.
- In general, the planning problem is a special case of the search problem.
- However, planning problems often have properties that allow for far more efficient solutions.

Defining a Planning Problem

- To define a planning problem, we need to specify the same elements that define a search problem:
  - States.
  - Actions.
  - Goals.
- In planning, we describe states, actions, and goals using logic.
- We use a language called PDDL (Planning Domain Definition Language).
- PDDL uses a limited version of first-order logic.
  - Limitations allow for efficient inference.

Representing States with PDDL

- A state is a conjunction of “ground, functionless atoms”.
  - To understand this, we need to understand each of the three terms: ground, functionless, atom.
- In PDDL, an atom is an application of a predicate to some arguments. For example:

  At(Plane1, JFK)
  Airport(JFK)
  Airplane(Plane1)
  Have(Milk)

- “Functionless” means that no functions are used.
  - For example: At(Father(George), JFK) is illegal, because it uses function Father.
- “Ground” means that no variables are used.
  - For example: At(x, y) is illegal, because it uses variables x, y.

Practice with State Descriptions

- To determine if a state is legal, we simply have to determine if it is a conjunction of “ground, functionless atoms”.

- Is this state description legal?

  not(Poor(George))

Practice with State Descriptions

- To determine if a state is legal, we simply have to determine if it is a conjunction of “ground, functionless atoms”.

- Is this state description legal?

  not(Poor(George))

- No, it uses a negation. In a conjunction of ground, functionless atoms there is no room for negations.

Practice with State Descriptions

- To determine if a state is legal, we simply have to determine if it is a conjunction of “ground, functionless atoms”.

- Is this state description legal?

  Poor(George) and Rich(Boss(George))

Practice with State Descriptions

- To determine if a state is legal, we simply have to determine if it is a conjunction of “ground, functionless atoms”.

- Is this state description legal?

  Poor(George) and Rich(Boss(George))

- No, it uses a function (Boss).

Practice with State Descriptions

- To determine if a state is legal, we simply have to determine if it is a conjunction of “ground, functionless atoms”.

- Is this state description legal?

  Poor(George) and Rich(Liz)

Practice with State Descriptions

- To determine if a state is legal, we simply have to determine if it is a conjunction of “ground, functionless atoms”.

- Is this state description legal?

  Poor(George) and Rich(Liz)

- Yes, it is a conjunction of ground, functionless atoms.
  - No negations, variables, functions.

Practice with State Descriptions

- To determine if a state is legal, we simply have to determine if it is a conjunction of “ground, functionless atoms”.

- Is this state description legal?

  Poor(George) and Rich(Liz) and At(George, x)

Practice with State Descriptions

- To determine if a state is legal, we simply have to determine if it is a conjunction of “ground, functionless atoms”.

- Is this state description legal?

  Poor(George) and Rich(Liz) and At(George, x)

- No, it uses variable x.

The Closed World Assumption

- PDDL makes two very specific assumptions, when interpreting state descriptions:
- The first such assumption is the **closed world assumption**: Any atom that is not mentioned in the state description is false.
- For example, suppose that we have this state description:

  At(Plane1, JFK)
  Airport(JFK)
  Airplane(Plane1)

- How can we prove that Plane1 is not an airport?

The Closed World Assumption

- PDDL makes two very specific assumptions, when interpreting state descriptions:
- The first such assumption is the **closed world assumption**: Any atom that is not mentioned in the state description is false.
- For example, suppose that we have this state description:

At(Plane1, JFK)
Airport(JFK)
Airplane(Plane1)

- How can we prove that Plane1 is not an airport?
- Since the state description does not mention Airport(Plane1), Airport(Plane1) is false.

The Unique Names Assumption

- PDDL makes also a second assumption in interpreting states: the **unique names assumption**: if two constants have different names, they are not equal to each other.
- We used that assumption implicitly in our previous example:

  At(Plane1, JFK)
  Airport(JFK)
  Airplane(Plane1)

- We said that since Airport(Plane1) is not mentioned, Airport(Plane1) is false.
- Note that Airport(JFK) is mentioned. However, we assume that JFK != Plane1, since these two constants have different names. Thus, Airport(JFK) cannot possibly imply Airport(Plane1).

Representing Actions with PDDL

- An action is defined using this syntax:

Action(Name(var_1, …, var_k),
  PRECOND: $atom_1$ AND … AND $atom_m$,
  EFFECT: $literal_1$ AND … AND $literal_n$)

- In other words:
  - An action has a name.
  - An action is applied to k arguments.
  - An action can only be applied if certain preconditions are met. Symbol m stands for the number of preconditions.
  - An action has certain effects. Symbol n stands for the number of effects.

Preconditions and Effects

- An action is defined using this syntax:

Action(Name(var_1, …, var_k),
PRECOND: $atom_1$ AND … AND $atom_m$,
EFFECT: $literal_1$ AND … AND $literal_n$)

- Preconditions and effects are conjunctions of **functionless literals**.
- Note that here we use term **literals**, whereas for state representations we use the term **atoms**.
- What is a literal?

Preconditions and Effects

- An action is defined using this syntax:

Action(Name(var_1, …, var_k),
  PRECOND: atom_1 AND … AND atom_m,
  EFFECT: literal_1 AND … AND literal_n)

- Preconditions and effects are conjunctions of **functionless literals**.
- Note that here we use term **literals**, whereas for state representations we use the term **atoms**.
- What is a literal? A literal is either an atom or a negation of an atom.
- In short, preconditions and effects are allowed to include negations.

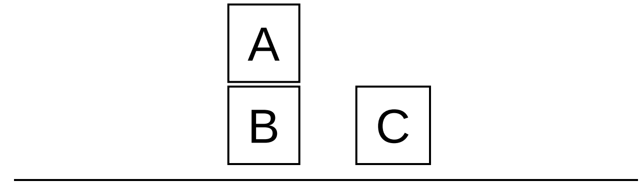
Preconditions and Effects

- An action is defined using this syntax:

Action(Name(var_1, …, var_k),
 PRECOND: $atom_1$ AND … AND $atom_m$,
 EFFECT: $literal_1$ AND … AND $literal_n$)

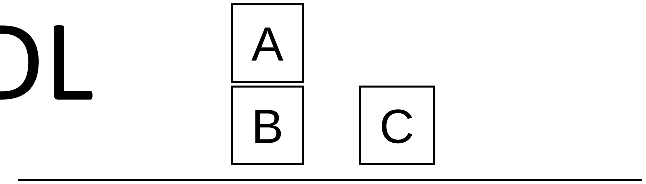
- Preconditions and effects are conjunctions of **functionless literals**.
  - Pretty much, functions are not allowed at all in PDDL.
- However, these literals can include variables.
- They can ONLY include variables var_1, …, var_k, no other variable is allowed.
- In summary, state descriptions must be ground (cannot include variables), but preconditions can include variables.

The Blocks World



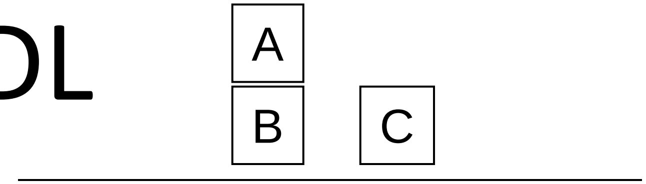
- The blocks world is
  a classic toy problem
  that is used for introducing
  planning concepts.
- We have cubic blocks, called A, B, C, …
  - Often only three blocks are used.
- These blocks can be stacked on top of each other, or just be placed on the table.
- You can move a block only if it is **Clear**, meaning that it has no other block on top of it.
- You can move a block on top of another block only if that other block is also **Clear**.
- You can always place a clear block directly on the table.

The Blocks World in PDDL



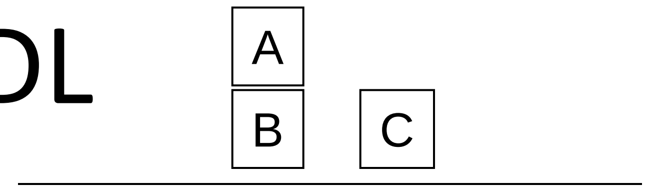
- To represent the blocks world using PDDL, we need to define states and actions.
- To define states and actions, we need to specify constants and predicates.
- What are our constants?

The Blocks World in PDDL



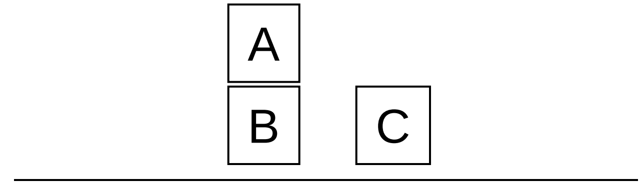
- To represent the blocks world using PDDL, we need to define states and actions.
- To define states and actions, we need to specify constants and predicates.
- What are our constants? A, B, C, Table.
- What are our predicates?

The Blocks World in PDDL



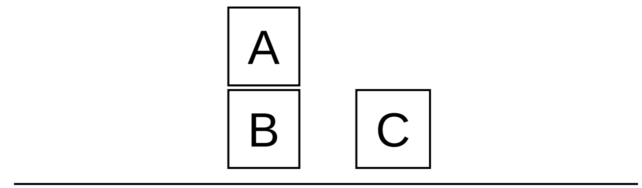
- To represent the blocks world using PDDL, we need to define states and actions.
- To define states and actions, we need to specify constants and predicates.
- What are our constants? A, B, C, Table.
- What are our predicates?
  - On(x, y) is true if block x is on top of y.
  - Clear(x) is true if x is clear (and therefore you can place a block on top of it).

Representing States



- Constants: A, B, C, Table.
- Predicates:
  - On(x, y) is true if block x is on top of y.
  - Clear(x) is true if x is clear (and therefore you can place a block on top of it).
- How can we represent the state that is shown above?

Representing States

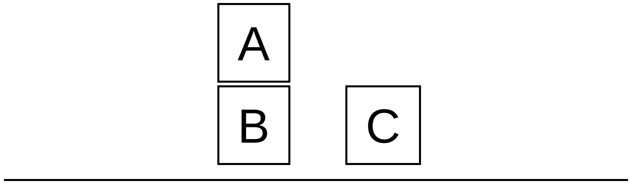


- Constants: A, B, C, Table.
- Predicates:
  - On(x, y) is true if block x is on top of y.
  - Clear(x) is true if x is clear (and therefore you can place a block on top of it).
- How can we represent the state that is shown above?

On(A, B)
On(B, Table)
On(C, Table)
Clear(A)
Clear(C)

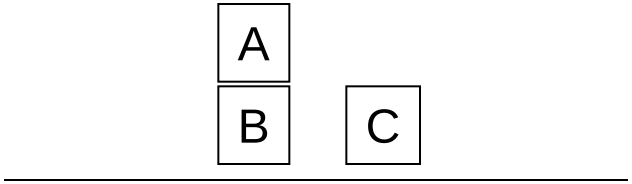
Note: it seems reasonable to also include a statement for Clear(Table), but we will see later that such a statement is not needed.

Representing Actions



- Constants: A, B, C, Table.
- Predicates:
  - On(x, y) is true if block x is on top of y.
  - Clear(x) is true if x is clear (and therefore you can place a block on top of it).
- How can we define actions for this domain?

Representing Actions

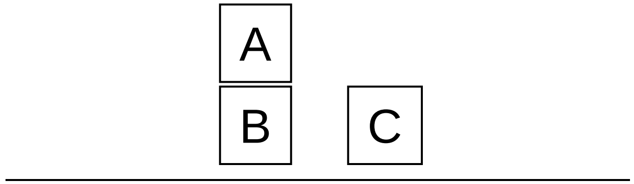


- Constants: A, B, C, Table.
- Predicates:
  - On(x, y) is true if block x is on top of y.
  - Clear(x) is true if x is clear (and therefore you can place a block on top of it).
- How can we define actions for this domain?
- First (incorrect) attempt: define a single action **Move.**

Action(Move(block, from, to),
  PRECOND: On(block, from) AND Clear(block) AND Clear(to)
  EFFECT: On(block, to)

- What is wrong with this?

Representing Actions

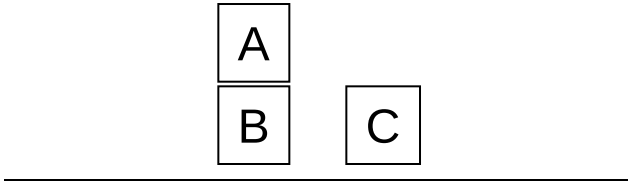


- Constants: A, B, C, Table.
- Predicates:
  - On(x, y) is true if block x is on top of y.
  - Clear(x) is true if x is clear (and therefore you can place a block on top of it).
- How can we define actions for this domain?
- First (incorrect) attempt: define a single action **Move.**

Action(Move(block, from, to),
   PRECOND: On(block, from) AND Clear(block) AND Clear(to)
   EFFECT: On(block, to)

- It fails to mention additional effects, like Clear(from).

Representing Actions

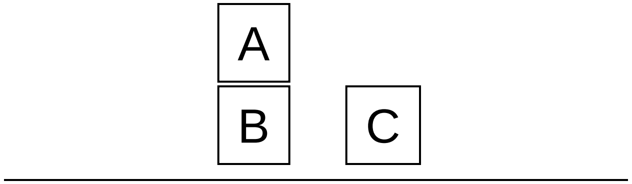


- Constants: A, B, C, Table.
- Predicates:
  - On(x, y) is true if block x is on top of y.
  - Clear(x) is true if x is clear (and therefore you can place a block on top of it).
- Second (incorrect) attempt: define a single action **Move.**

Action(Move(block, from, to),
   PRECOND: On(block, from) AND Clear(block) AND Clear(to)
   EFFECT: On(block, to) AND NOT(On(block, from)) AND
            Clear(from) AND NOT(Clear(to))

- What is wrong with this attempt?

Representing Actions

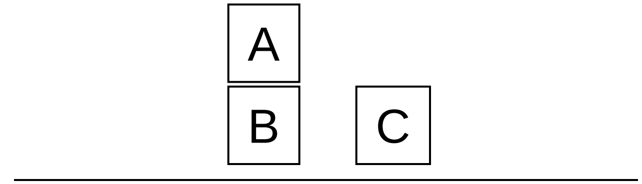


- Constants: A, B, C, Table.
- Predicates:
  - On(x, y) is true if block x is on top of y.
  - Clear(x) is true if x is clear (and therefore you can place a block on top of it).
- Second (incorrect) attempt: define a single action **Move.**

Action(Move(block, from, to),
  PRECOND: On(block, from) AND Clear(block) AND Clear(to)
  EFFECT: On(block, to) AND NOT(On(block, from)) AND
      Clear(from) AND NOT(Clear(to))

- This definition does not capture the fact that the table is always clear (you can always place a block directly on the table).

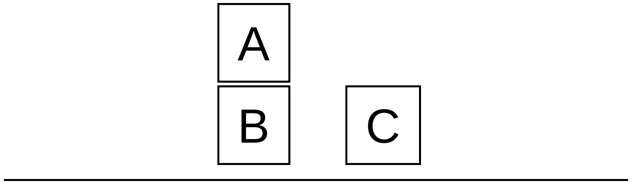
Representing Actions



- Constants: A, B, C, Table.
- Predicates:
  - On(x, y) is true if block x is on top of y.
  - Clear(x) is true if x is clear (and therefore you can place a block on top of it).
- Third (correct) attempt: define a separate action **MoveToTable**.

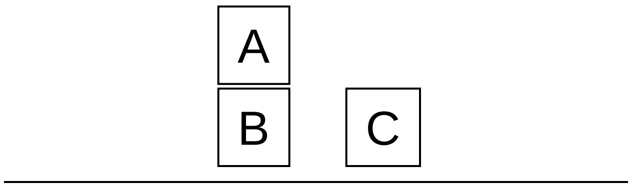
```
Action(Move(block, from, to),
   PRECOND: On(block, from) AND Clear(block) AND Clear(to)
   EFFECT: On(block, to) AND NOT(On(block, from)) AND
           Clear(from) AND NOT(Clear(to))

Action(MoveToTable(block, from),
   PRECOND: On(block, from) AND Clear(block)
   EFFECT: On(block, Table) AND NOT(On(block, from)) AND Clear(from)
```

Blocks World in PDDL



- Suppose we have this state:
- What knowledge base represents this state?
  (We have seen this in previous slides).

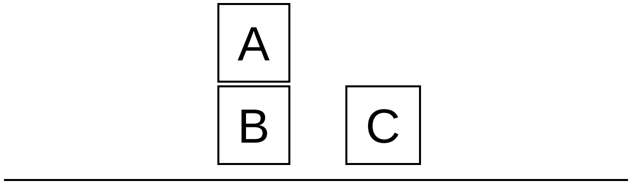
Blocks World in PDDL



- Suppose we have this state:
- What knowledge base represents this state? (We have seen this in previous slides).

On(A, B)
On(B, Table)
On(C, Table)
Clear(A)
Clear(C)

Blocks World in PDDL



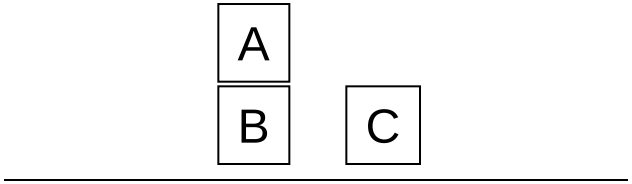
- Suppose we have this state:
- What knowledge base represents this state? (We have seen this in previous slides).

  On(A, B)
  On(B, Table)
  On(C, Table)
  Clear(A)
  Clear(C)

- How can we prove that B is not clear?

Blocks World in PDDL

- Suppose we have this state:

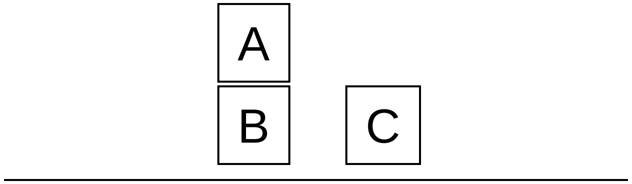


- What knowledge base represents this state?
  (We have seen this in previous slides).

  On(A, B)
  On(B, Table)
  On(C, Table)
  Clear(A)
  Clear(C)

- How can we prove that B is not clear?
- Using the closed-world assumption.
  - The KB does not include Clear(B), therefore B is not clear.

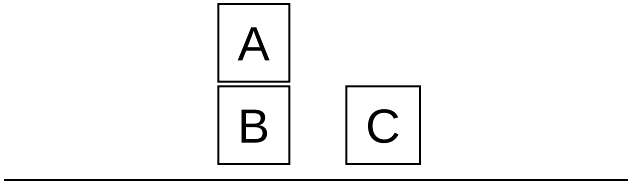
Blocks World in First-Order Logic



- Suppose we have this state:
- What knowledge base represents this state if we use first-order logic?

Blocks World in First-Order Logic

- Suppose we have this state:



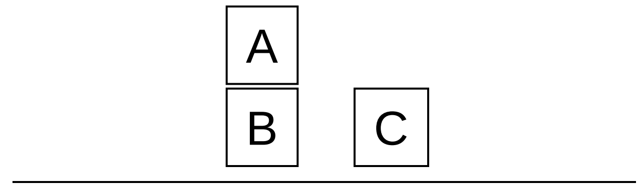
- What knowledge base represents this state if we use first-order logic?

On(A, B)
On(B, Table)
On(C, Table)
Clear(A)
Clear(C)

The knowledge base is identical to the PDDL version.

Blocks World in First-Order Logic

- Suppose we have this state:



- What knowledge base represents this state if we use first-order logic?

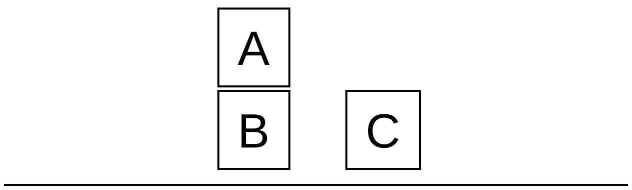
On(A, B)
On(B, Table)
On(C, Table)
Clear(A)
Clear(C)

The knowledge base is identical to the PDDL version.

- How can we prove that B is not clear?

Blocks World in First-Order Logic

- Suppose we have
  this state:



- What knowledge base
  represents this state if we use first-order logic?

On(A, B)
On(B, Table)
On(C, Table)
Clear(A)
Clear(C)

The knowledge base is identical
to the PDDL version.

- How can we prove that B is not clear?
- We can't, without introducing an additional rule in the knowledge base:
  ∀ x, y, On(x, y) => not(Clear(y))

PDDL vs. First-Order Logic

- PDDL is a restricted form of first-order logic.
  - No functions.
  - No universal and existential quantifiers ($\forall, \exists$).
  - States are conjunctions of groundless atoms.
- Disadvantages of PDDL:

PDDL vs. First-Order Logic

- PDDL is a restricted form of first-order logic.
  - No functions.
  - No universal and existential quantifiers (∀, ∃).
  - States are conjunctions of groundless atoms.
- Disadvantages of PDDL:
  - Not using functions makes it impossible to express certain facts, such as properties of integers.
  - Not using quantifiers makes it impossible to express rules (like stating that “when a block X has something on it, then block X is not clear”.

PDDL vs. First-Order Logic

- Advantages of PDDL compared to first-order logic:

PDDL vs. First-Order Logic

- Advantages of PDDL compared to first-order logic:
  - Inference is very fast.
  - How can we prove that an atom is true? For example, how can we prove that On(A, B) is true?

PDDL vs. First-Order Logic

- Advantages of PDDL compared to first-order logic:
  - Inference is very fast.
  - How can we prove that an atom is true? For example, how can we prove that On(A, B) is true?
  - If the knowledge base includes On(A, B), then it is true.

PDDL vs. First-Order Logic

- Advantages of PDDL compared to first-order logic:
  - Inference is very fast.
  - How can we prove that an atom is true? For example, how can we prove that On(A, B) is true?
  - If the knowledge base includes On(A, B), then it is true.
  - How can we prove that an atom is false? For example, how can we prove that On(A, B) is false?

PDDL vs. First-Order Logic

- Advantages of PDDL compared to first-order logic:
  - Inference is very fast.
  - How can we prove that an atom is true? For example, how can we prove that On(A, B) is true?
  - If the knowledge base includes On(A, B), then it is true.
  - How can we prove that an atom is false? For example, how can we prove that On(A, B) is false?
  - If the knowledge base does not include On(A, B), then it is false.
- Suppose that alpha is a conjunction of literals. How can we

Inference in PDDL

- Suppose that alpha is a conjunction of literals.

  alpha = literal_1 AND … AND literal_n

- In PDDL, how can we infer if alpha is true or false in a state?
  - Remember, a state is simply a knowledge base that contains functionless grounded atoms.

Inference in PDDL

- Suppose that alpha is a conjunction of literals.

  alpha = literal_1 AND … AND literal_n

- In PDDL, how can we infer if alpha is true or false in a state?
  - Remember, a state is simply a knowledge base that contains functionless grounded atoms.
- Any literal that is an atom is true if it is included in the knowledge base, false otherwise.
- Any literal that is the negation of an atom is true if it is not included in the knowledge base, false otherwise.
- So, to check if alpha is true we just need to check if each of its literals is true.

Complexity of Inference in PDDL

- Suppose that alpha is a conjunction of literals.

  alpha = literal_1 AND … AND literal_n

- In PDDL, what is the time complexity of inferring if alpha is true or false in a state?

Complexity of Inference in PDDL

- Suppose that alpha is a conjunction of literals.

  alpha = literal_1 AND … AND literal_n

- In PDDL, what is the time complexity of inferring if alpha is true or false in a state?
- We need to check if each literal is true.
- To check each literal, we need to compare it with each of the statements in the knowledge base.
- With n literals in alpha and m statements in the knowledge base, the complexity of a naïve implementation is O(nm).
  - How can this be made even faster?

Complexity of Inference in PDDL

- Suppose that alpha is a conjunction of literals.

  alpha = literal_1 AND … AND literal_n

- In PDDL, what is the time complexity of inferring if alpha is true or false in a state?
- We need to check if each literal is true.
- To check each literal, we need to compare it with each of the statements in the knowledge base.
- With n literals in alpha and m statements in the knowledge base, the complexity of a naïve implementation is O(nm).
  - How can this be made even faster?
  - We can use a hash table for storing the statements of the knowledge base. Then, we can check for every literal if it is true or false in constant time.

Complexity of Inference in PDDL

- Suppose that alpha is a conjunction of literals.

  alpha = literal_1 AND … AND literal_n

- In PDDL, the time complexity of inferring if alpha is true or false in a state is O(nm) or O(n), depending on the implementation.
- If we use first-order logic, what is the corresponding time complexity?

Complexity of Inference in PDDL

- Suppose that alpha is a conjunction of literals.

  alpha = literal_1 AND … AND literal_n

- In PDDL, the time complexity of inferring if alpha is true or false in a state is O(nm) or O(n), depending on the implementation.
- If we use first-order logic, what is the corresponding time complexity?
- In the worst case, infinity!!!
  - Exponential time if the state entails alpha.
  - Infinite time if the state does not entail alpha.
- So, the restrictions of PDDL reduce the time complexity of inference from infinity to linear!!!
  - Now you can see why PDDL is a popular choice for planning.

Planning as Search

- To define a planning problem as a search problem we need to define:
  - An initial state.
  - A state successor function, that defines what actions are applicable at each state.
  - A goal.
- How do we represent an initial state?

Planning as Search

- To define a planning problem as a search problem we need to define:
  - An initial state.
  - A state successor function, that defines what actions are applicable at each state.
  - A goal.
- How do we represent an initial state?
  - We have already covered this, the initial state (like any other state) is a conjunction of atoms in PDDL.

Planning as Search

- How do we represent the state successor function?

Planning as Search

- How do we represent the state successor function?
  - By defining actions as discussed earlier, specifying for each action its arguments, preconditions and effects.

Planning as Search

- How do we represent the state successor function?
  - By defining actions as discussed earlier, specifying for each action its arguments, preconditions and effects.
- The definition of an action is used in two different ways:
- First, to determine, given a state, if an action is applicable in that state.
  - How is that determined?

Planning as Search

- How do we represent the state successor function?
  - By defining actions as discussed earlier, specifying for each action its arguments, preconditions and effects.
- The definition of an action is used in two different ways:
- First, to determine, given a state, if an action is applicable in that state.
  - An action A is applicable in state S if the preconditions of A are true in S.

Planning as Search

- How do we represent the state successor function?
  - By defining actions as discussed earlier, specifying for each action its arguments, preconditions and effects.
- The definition of an action is used in two different ways:
- First, to determine, given a state, if an action is applicable in that state.
  - An action A is applicable in state S if the preconditions of A are true in S.
- Second, to produce the result state S’ that is obtained by applying function A to state S.
  - How do we produce S’?

Planning as Search

- How do we represent the state successor function?
  - By defining actions as discussed earlier, specifying for each action its arguments, preconditions and effects.
- The definition of an action is used in two different ways:
- First, to determine, given a state, if an action is applicable in that state.
  - An action A is applicable in state S if the preconditions of A are true in S.
- Second, to produce the result state S' that is obtained by applying function A to state S.
  - We produce S' by adding to S all the positive effects of A, and removing all the negative effects of A.

Planning as Search

- How do we represent the goal?
- The goal is a conjunction of literals. Example:

  on(A, B) AND on(B, C)

- We have reached the goal if we have reached a state that entails the goal.

Planning as Search

- Since planning can be viewed as a search problem, any of the search algorithms we already know can be used for planning.
  - For example, IDS.
- Problem: standard search algorithms can be horribly slow, even for planning problems that to a human seem trivial.

Example: Ordering 10 Books

- We want to order 10 books from Amazon: book3, book7, book13, book17, book20, book25, book30, book35, book40, book50.
- Initial state:

  has(Amazon, book1)

  has(Amaxon, book2)

  …

  has(Amazon, book1000000)   // Amazon sells lots of book titles…
- Action(buy(person, book, store),
  - PRECOND: has(store, book),
  - EFFECT: owns(person, book))
- Goal:

  owns(me, book3) ∧ owns(me, book7) ∧ owns(me, book13) ∧ owns(me, book17) ∧ owns(me, book20) ∧ owns(me, book25) ∧ owns(me, book30) ∧ owns(me, book35) ∧ owns(me, book40) ∧ owns(me, book50)

Example: Ordering 10 Books

- Solution (one of many):

  buy(me, book3, Amazon)
  buy(me, book7, Amazon)
  buy(me, book13, Amazon)
  buy(me, book17, Amazon)
  buy(me, book20, Amazon)
  buy(me, book25, Amazon)
  buy(me, book30, Amazon)
  buy(me, book35, Amazon)
  buy(me, book40, Amazon)
  buy(me, book50, Amazon)

- Coming up with such a plan is trivial for humans, far from being an intellectually challenging task.

Example: Ordering 10 Books

- Viewed as a traditional search problem, coming up with a plan to order these 10 books is a horrendously challenging task:
  - branching factor: 1,000,000
  - depth of solution: 10
  - would require visiting about $1{,}000{,}000^{10}$ nodes to find a solution.
  - Computationally infeasible!!!
- This example should explain why we are studying planning as a topic of its own in this course.
  - Standard search algorithms can fail even on trivial problems.

Heuristics for Planning

- As we just saw, standard search algorithms can fail even on trivial problems.
- The solution is to use informed search, with appropriate heuristics.
- One can always try to come up with heuristics for a specific planning task.
- However, there are more general techniques, that can be applied to ANY planning task to obtain reasonable heuristics.
- We will study such a general technique, called a **planning graph**.

Towards a Heuristic

- In general, a useful way to come up with heuristics is by relaxing our assumptions, imagining scenarios where illegal actions could actually happen.
- For example:
  - The h_1 heuristic for the 8-puzzle (number of misplaced tiles) is obtained by imagining a scenario where pieces are allowed to move to any position, regardless of whether that position is adjacent or empty.
  - The h_2 heuristic for the 8-puzzle (sum of Manhattan distances) is obtained by imagining a scenario where pieces are allowed to move to any **adjacent** position, regardless of whether that position is empty.
- In planning graphs, we obtain heuristics by imagining a scenario where multiple actions can be taken at the same time.

Planning Graph

- A planning graph is a directed graph, organized into levels.
- The following is an **incomplete** description of how planning graphs are constructed (complete details in a few slides…)
- The initial level is level S_0, and corresponds to the initial state.
  - Level S_0 contains one node for each literal that is true at the initial state.
- The next level is level A_0, corresponding to actions that are applicable to the initial state.
  - Level A_0 contains one node for each action that can be applied to the initial state.

Planning Graph

- The next level is level S_1, that contains one node for every possible literal that could become true by applying an action in A_0.
- The next level is level A_1, that contains one node for every possible action whose preconditions are satisfied by literals in S_1.
- And so on…
  - Level S_i contains one node for every literal that is an effect of an action in A_{i-1}.
  - Level A_i contains one node for every possible action whose preconditions are satisfied by literals in S_i.

Planning Graph Example

Consider the Cake problem:

- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- What is the solution to this problem?

Planning Graph Example

Consider the Cake problem:

- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- What is the solution to this problem? Not that hard:
  - Eat(Cake)
  - Bake(Cake)

Planning Graph Example

Consider the Cake problem:

- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* $\wedge$ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: $\neg$*Have(Cake)* $\wedge$ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: $\neg$*Have(Cake)*
  EFFECT: *Have(Cake))*

- This is a very simple example, that we can use to see how to build planning graphs.

Planning Graph for the Cake Problem

- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- The initial level is level S_0, and corresponds to the initial state.
  - Level S_0 contains one node for each literal that is true at the initial state.
  - What literals are true in the initial state?
  - Note that a literal can also be a **negation** of an atom.

Planning Graph for the Cake Problem

Have(Cake)

$\neg$ *Eaten(Cake)*

- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- The initial level is level S_0, and corresponds to the initial state.
  - Level S_0 contains one node for each literal that is true at the initial state.
  - Above you see the two nodes of level S_0, showing the two literals that are true at the initial state.

Planning Graph for the Cake Problem

Have(Cake)

¬ Eaten(Cake)

- Initial state: *Have(Cake)*
- Goal: *Have(Cake) ∧ Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: *¬Have(Cake) ∧ Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: *¬Have(Cake)*
  EFFECT: *Have(Cake))*

- The next level is level A_0, corresponding to actions that are applicable to the initial state.
  - Level A_0 contains one node for each action that can be applied to the initial state.
  - What actions do we put here?

Planning Graph for the Cake Problem

Have(Cake)

¬ *Eaten(Cake)*

Eat(Cake)

- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- *Bake(Cake)* is not applicable, because ¬*Have(Cake)* is not part of S_0.
- The only action that is applicable is *Eat(Cake)*.

Planning Graph for the Cake Problem

Have(Cake)

¬ *Eaten(Cake)*

P

Eat(Cake)

P

- For each literal C at S_0, we include a "persistence" action, indicated as P.
- The persistence action for literal C has precondition C and effect C.
  - A persistence action just means that we do nothing and thus the literal is preserved.

- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

Planning Graph for the Cake Problem

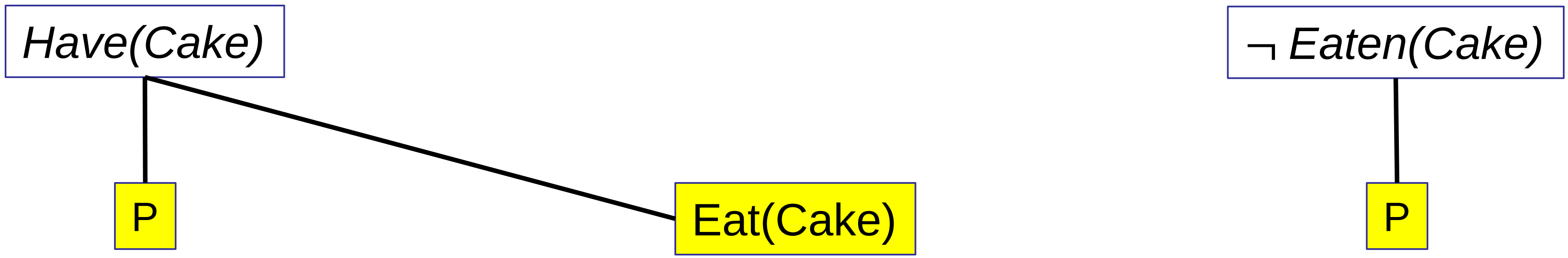
Have(Cake) | $\neg$ *Eaten(Cake)*

P | Eat(Cake) | P

- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* $\wedge$ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: $\neg$*Have(Cake)* $\wedge$ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: $\neg$*Have(Cake)*
  EFFECT: *Have(Cake))*

- Each action at A_0 is linked to its preconditions at S_0.
- What edges do we need to include?

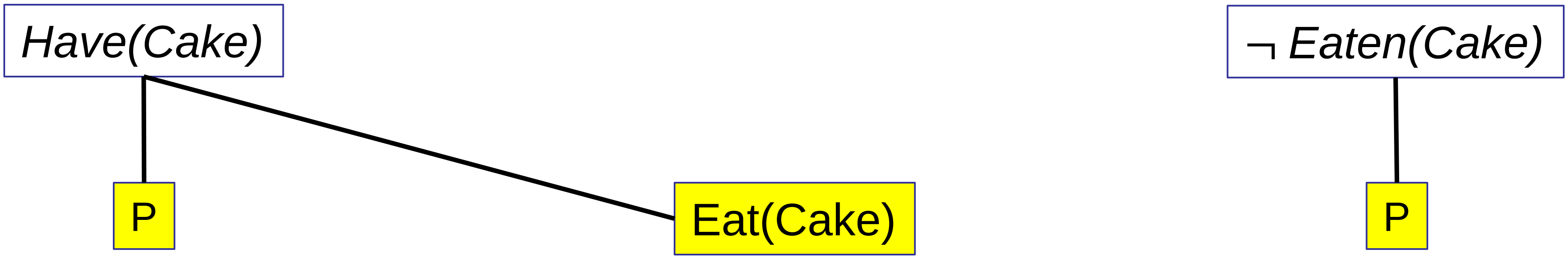
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- Each action at A_0 is linked to its preconditions at S_0.
- These edges are now shown.

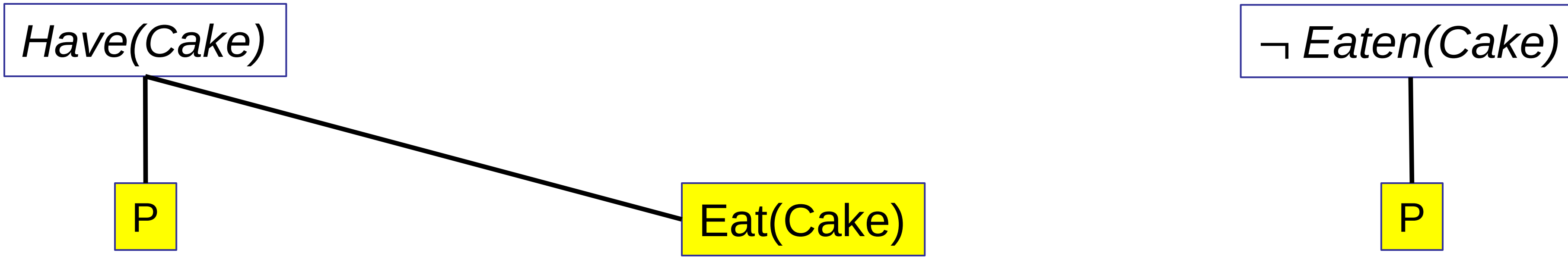
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake) ∧ Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake) ∧ Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- We also need to insert **mutual exclusion** edges (also called **mutex edges**).
- Mutual exclusion edges link actions that cannot happen at the same time.

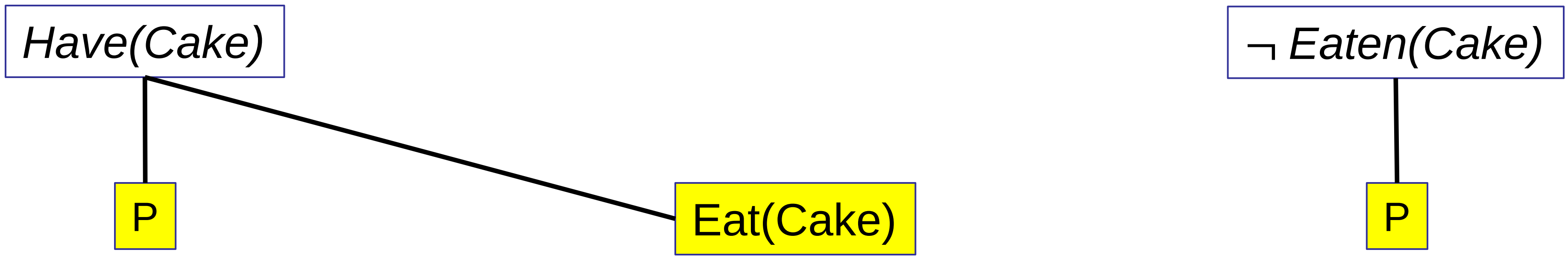
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake) ∧ Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake) ∧ Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- Mutex edges between actions are caused by four things:
- 1: Inconsistent preconditions: one precondition of one action is the negation of a precondition of the other action.
  - Any examples here?

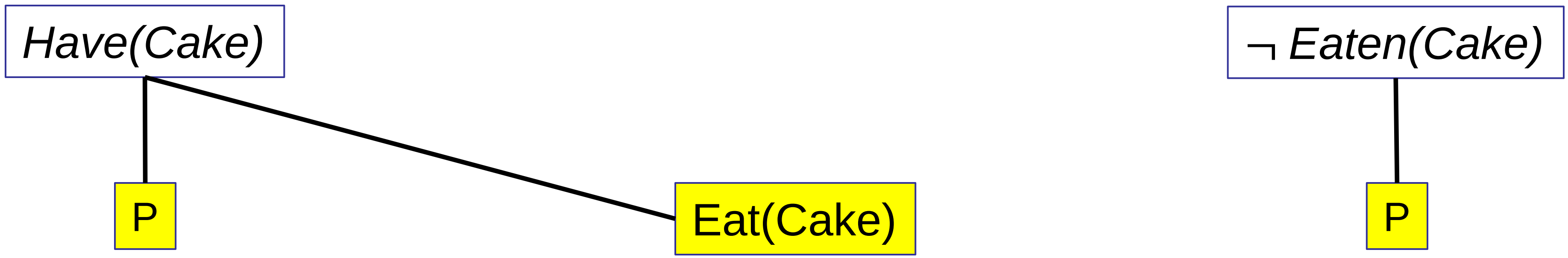
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- Mutex edges between actions are caused by four things:
- 1: Inconsistent preconditions: one precondition of one action is the negation of a precondition of the other action.
  – Any examples here? No

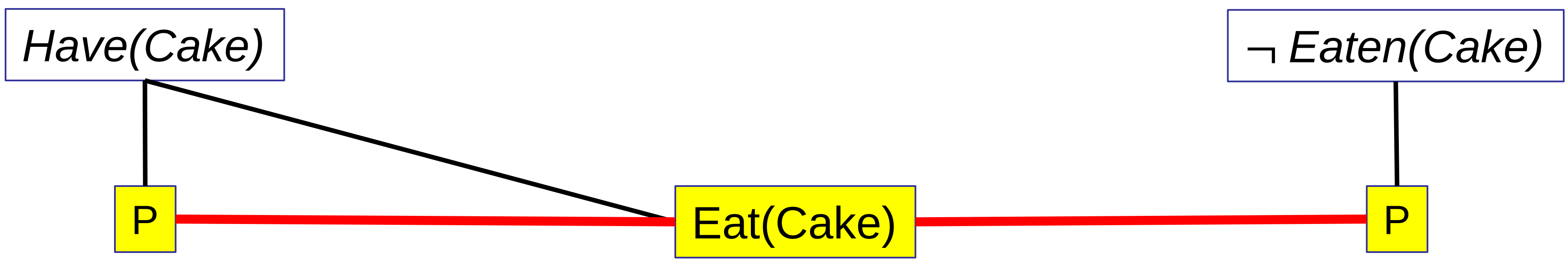
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- Mutex edges between actions are caused by four things:
- 2: Inconsistent effects: one effect of one action negates an effect of the other action.
  - Any examples here?

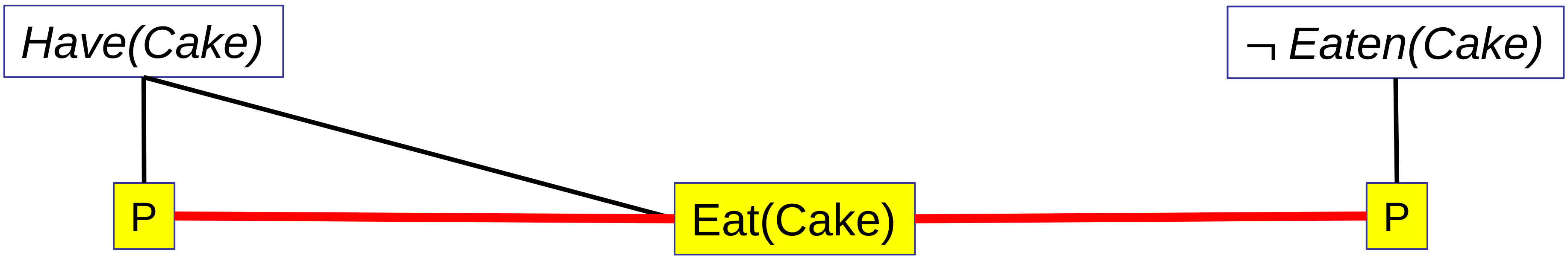
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- Mutex edges between actions are caused by four things:
- 2: Inconsistent effects: one effect of one action negates an effect of the other action.
  - Any examples here?
  - The effects of *Eat(Cake)* negate the effects of both persistence actions.

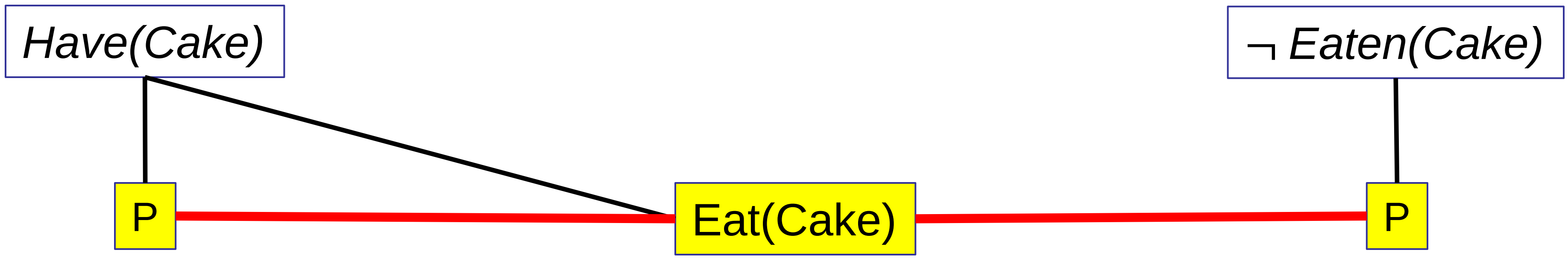
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- Mutex edges between actions are caused by four things:
- 3: Interference: One of the effects of one action is the negation of a precondition of the other action.
  – Any examples here?

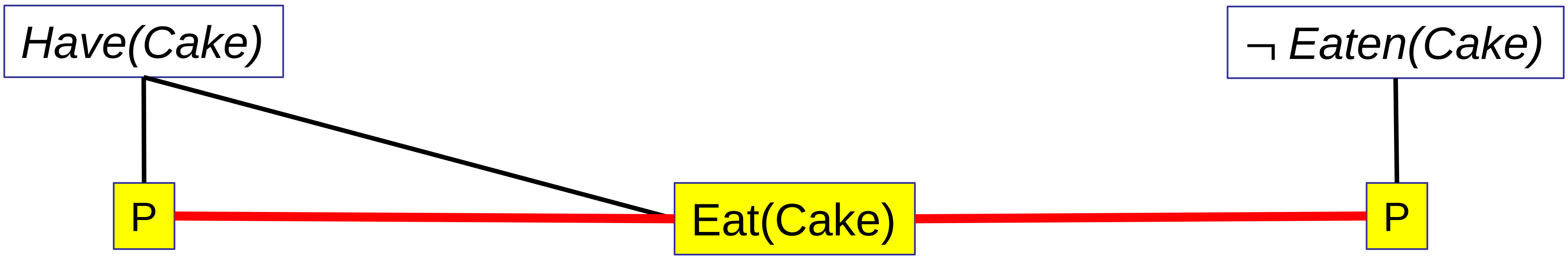
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- Mutex edges between actions are caused by four things:
- 3: Interference: One of the effects of one action is the negation of a precondition of the other action.
  – Any examples here?
  – One effect of *Eat(Cake)* negates the precondition of the persistence action for *Have(Cake)*. Edge already there.

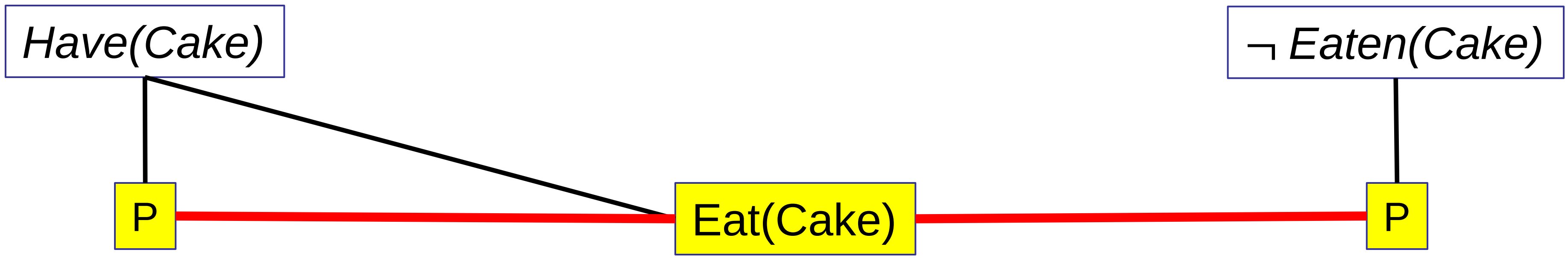
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- Mutex edges between actions are caused by four things:
- 4: Only one "real" action can be performed at a time.
  - Persistence actions are not "real" actions.
  - Any pair of real actions is mutually exclusive.
- Only one real action here, so no such conflict occurs.

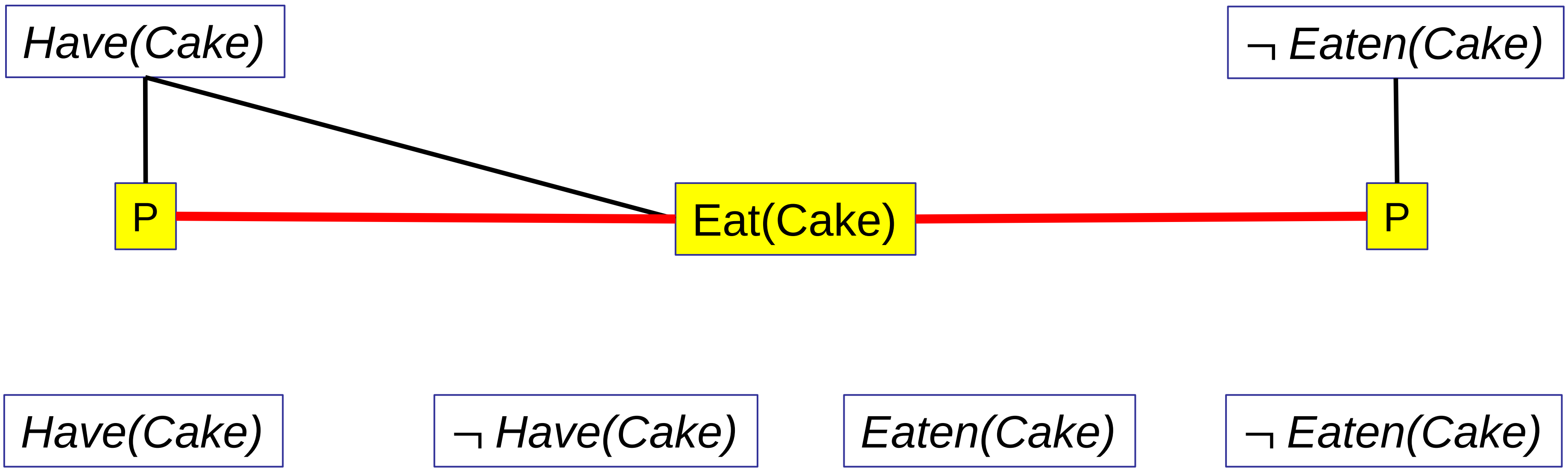
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- The next level is level S_1, that contains one node for every possible literal that could become true by applying an action in A_0.
- What literals do we need to include here?

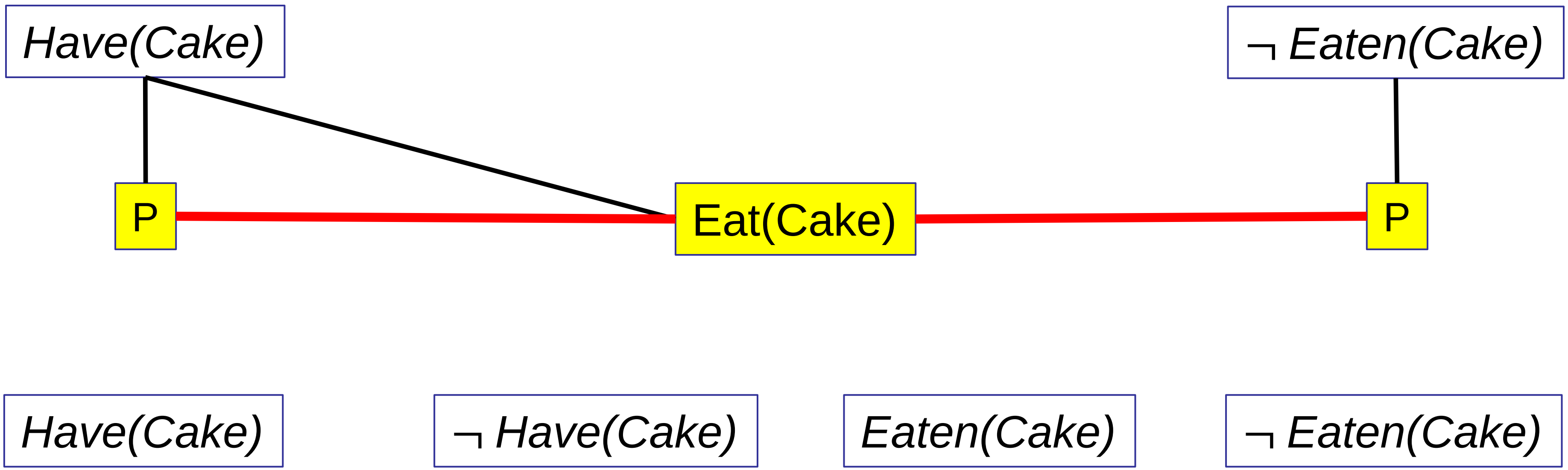
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- The next level is level S_1, that contains one node for every possible literal that could become true by applying an action in A_0.
- What literals do we need to include here?
- Every literal is now possible.

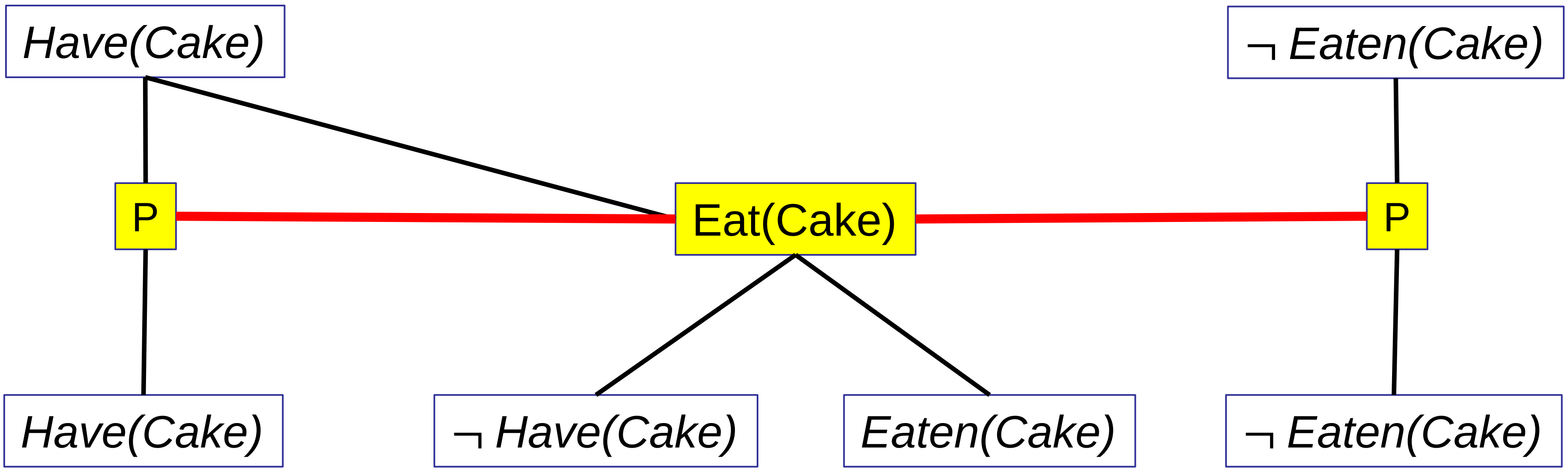
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- We add edges connecting each literal to each action at the previous level that has that literal as an effect.
- What edges do we need to add?

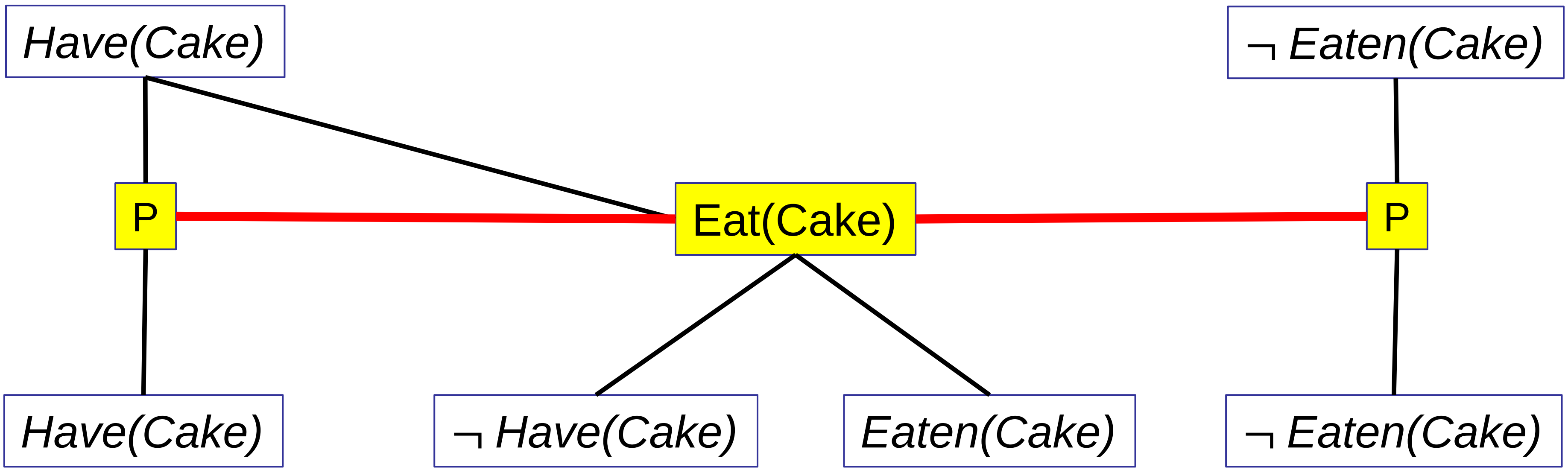
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- We add edges connecting each literal to each action at the previous level that has that literal as an effect.
- What edges do we need to add?
  - The edges are now shown.

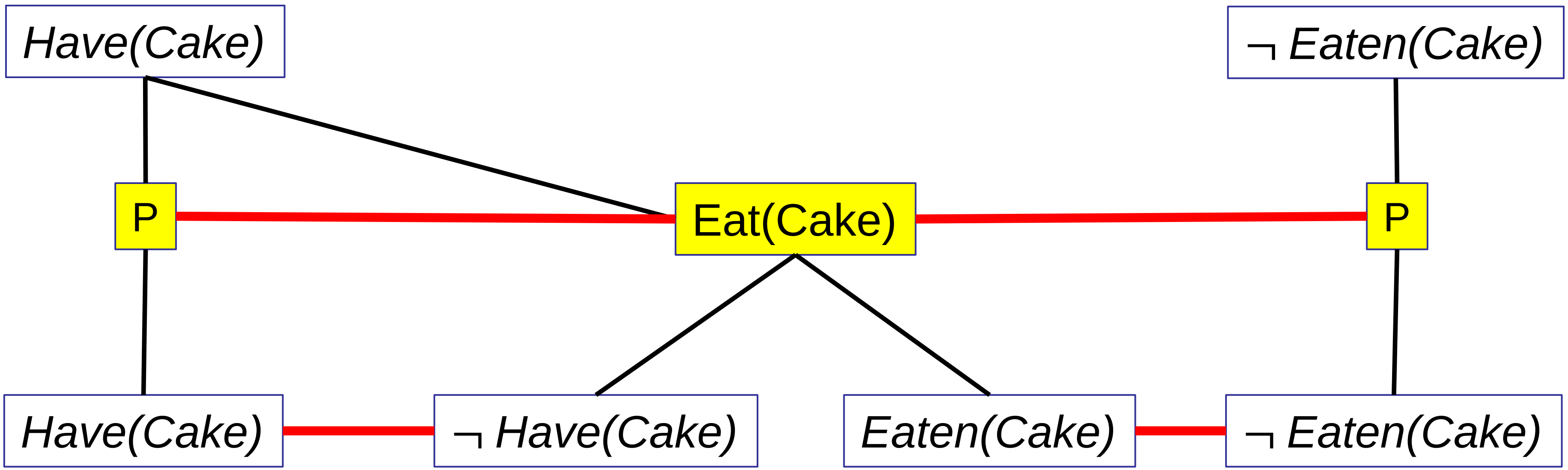
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- We also add mutex edges between literals at the same level, in two cases:
- 1: one literal is the negation of the other literal.
  - What edges do we need to add for this case?

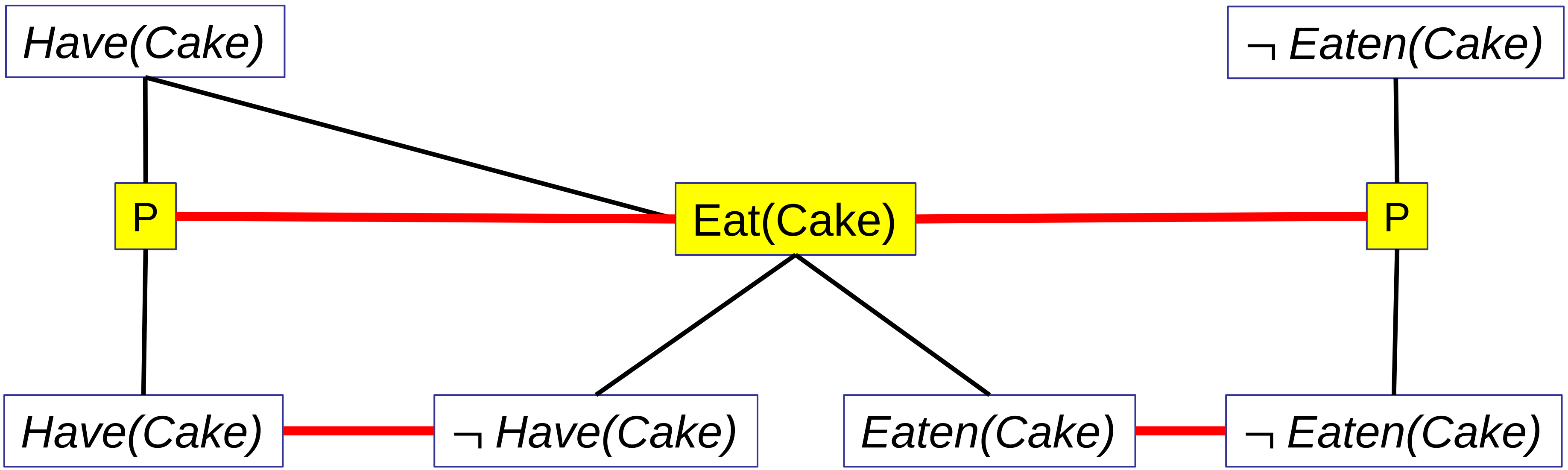
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- We also add mutex edges between literals at the same level, in two cases:
- 1: one literal is the negation of the other literal.
  - What edges do we need to add for this case?
  - The edges are now shown.

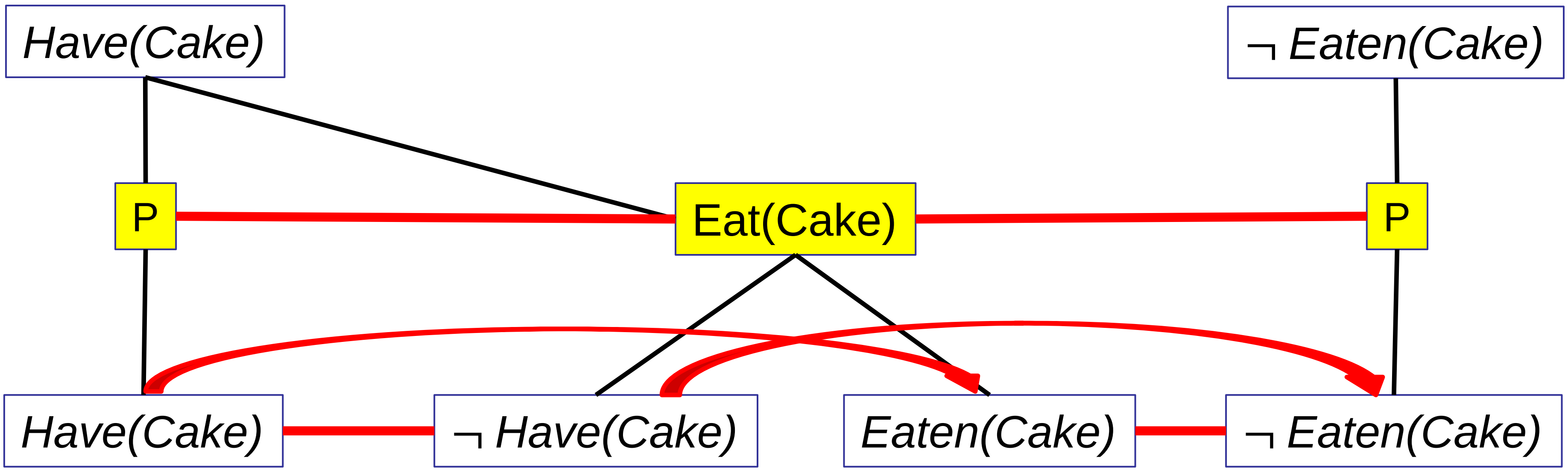
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- We also add mutex edges between literals at the same level, in two cases:
- 2: Each possible pair of actions achieving those two literals is mutually exclusive.
  – Edges for this case?

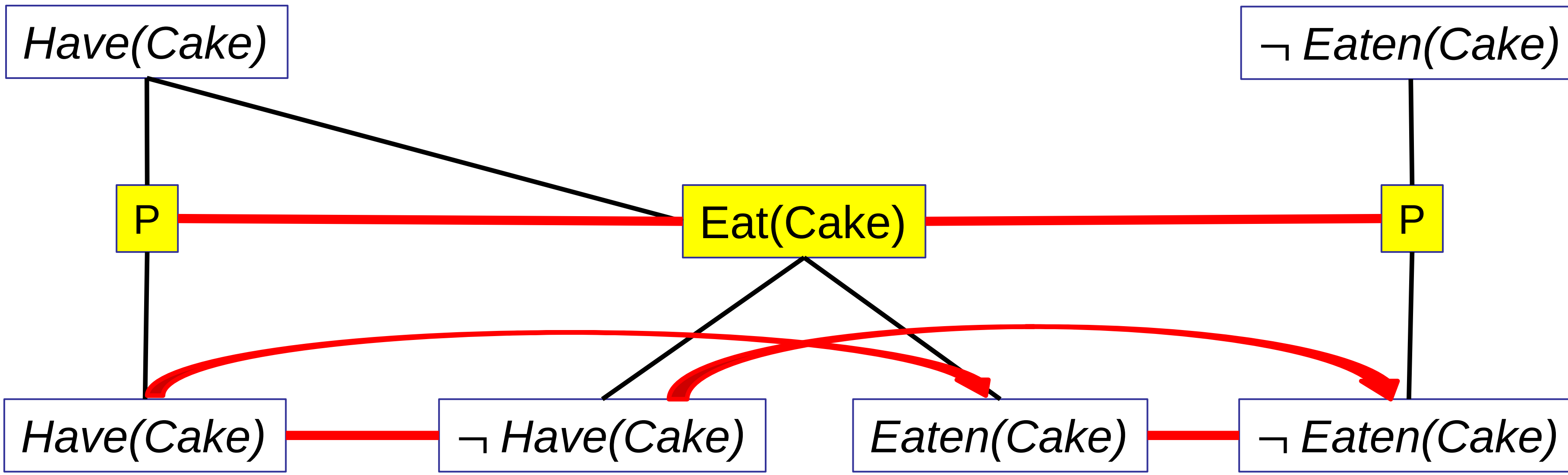
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

- We also add mutex edges between literals at the same level, in two cases:
- 2: Each possible pair of actions achieving those two literals is mutually exclusive.
  Have(Cake) and *Eaten(Cake).*
  ¬ *Have(Cake)* and ¬ *Eaten(Cake).*

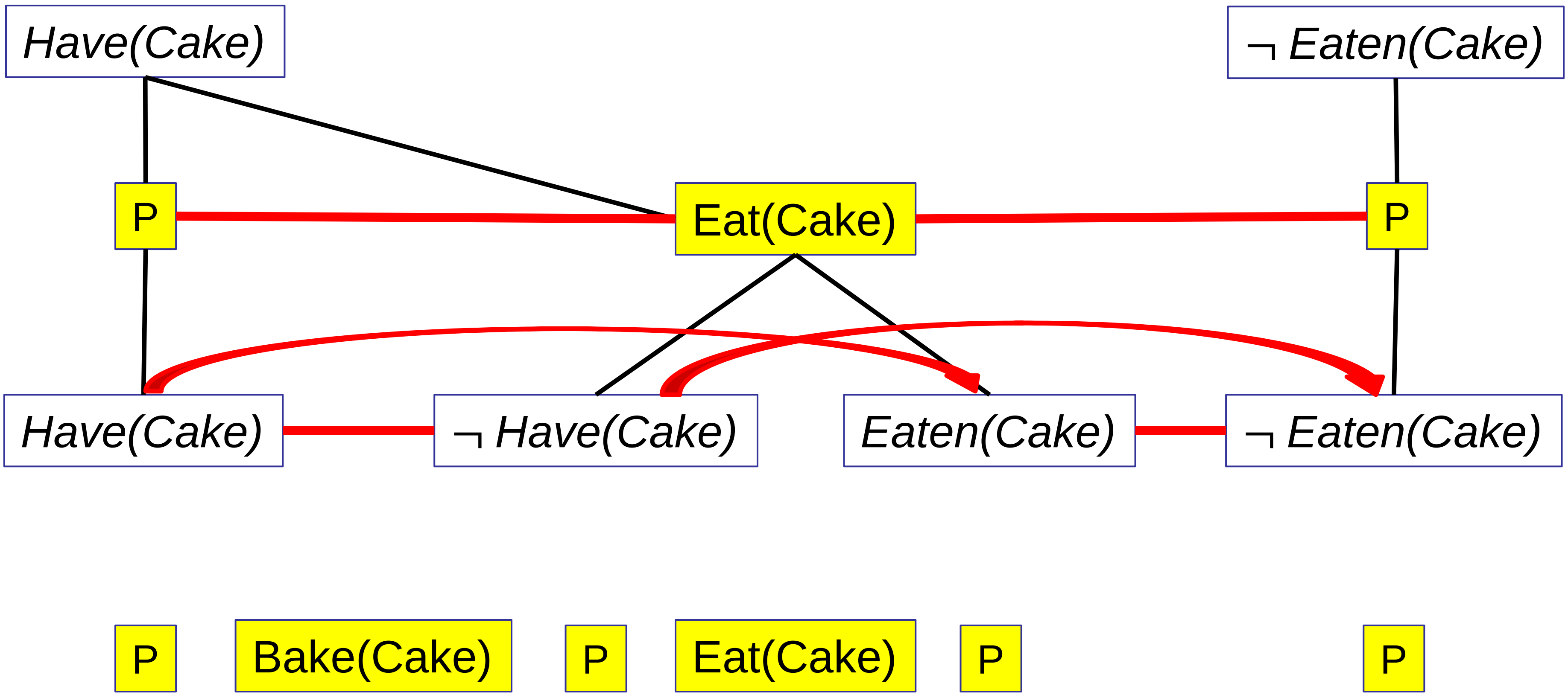
Planning Graph for the Cake Problem



- Initial state: *Have(Cake)*
- Goal: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action(Eat(Cake),*
  PRECOND: *Have(Cake)*
  EFFECT: ¬*Have(Cake)* ∧ *Eaten(Cake))*
- *Action(Bake(Cake),*
  PRECOND: ¬*Have(Cake)*
  EFFECT: *Have(Cake))*

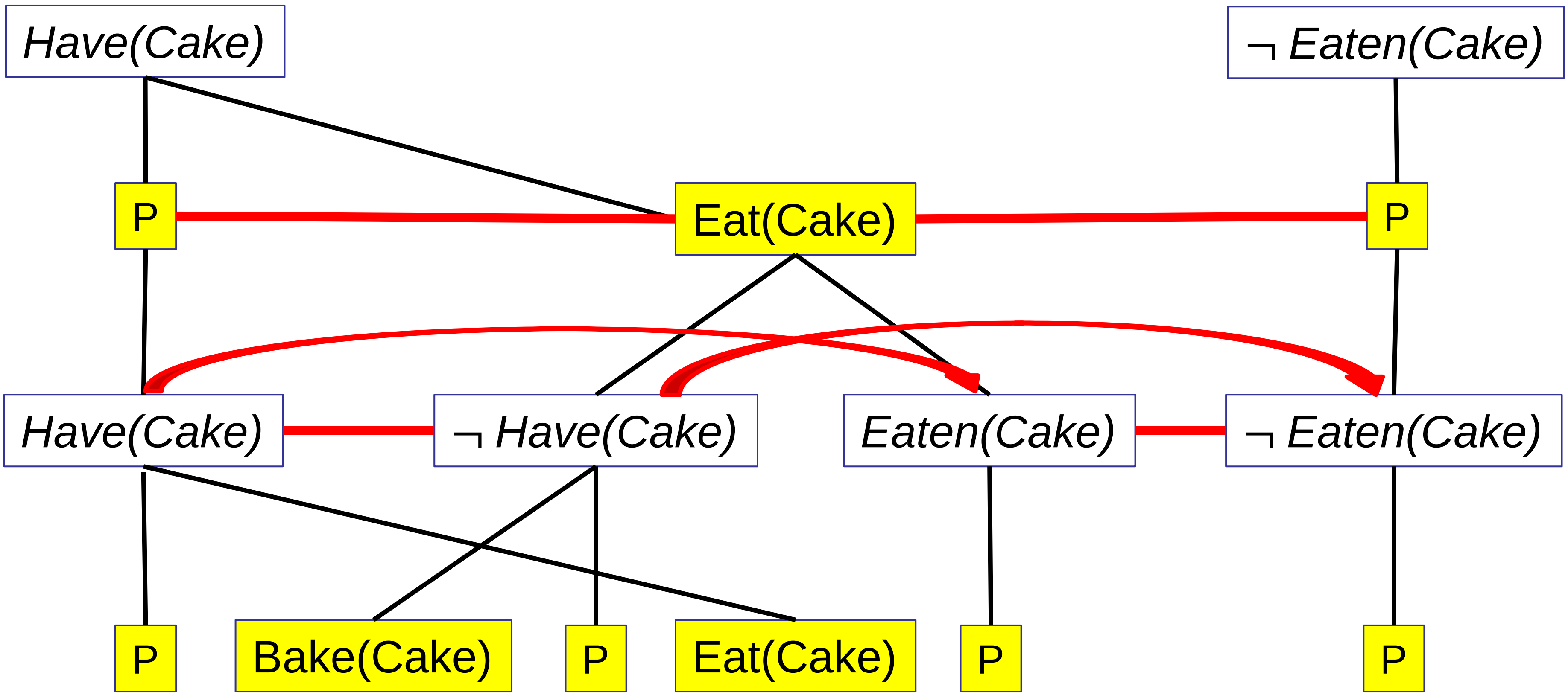
- The next level is level A_1, that contains one node for every possible action whose preconditions are satisfied by literals in S_1.
- What actions do we include in A_1?

Planning Graph for the Cake Problem



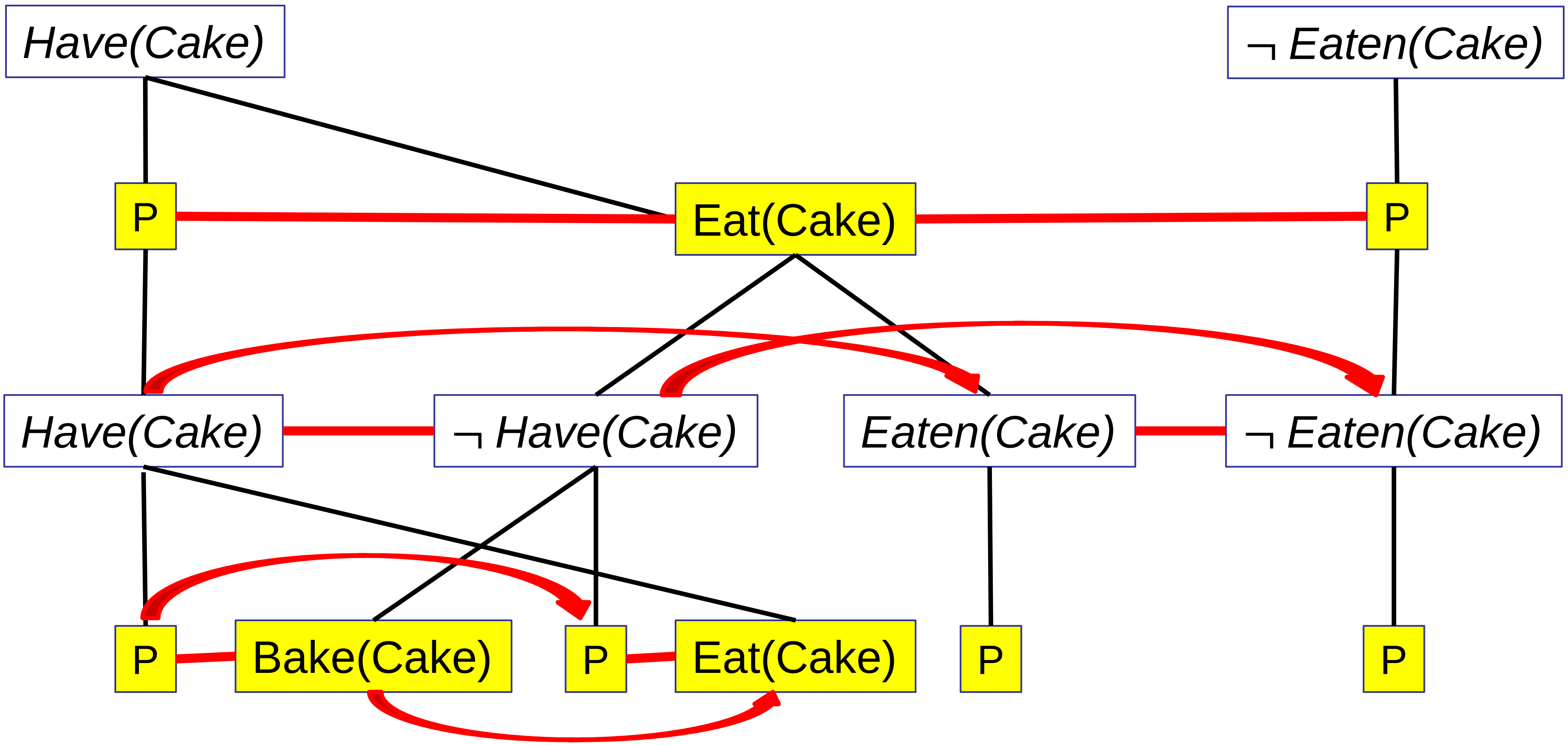
- The next level is level A_1, that contains one node for every possible action whose preconditions are satisfied by literals in S_1.
- What actions do we include in A_1? *Eat(Cake)*, *Bake(Cake)*, and persistence actions.

Planning Graph for the Cake Problem



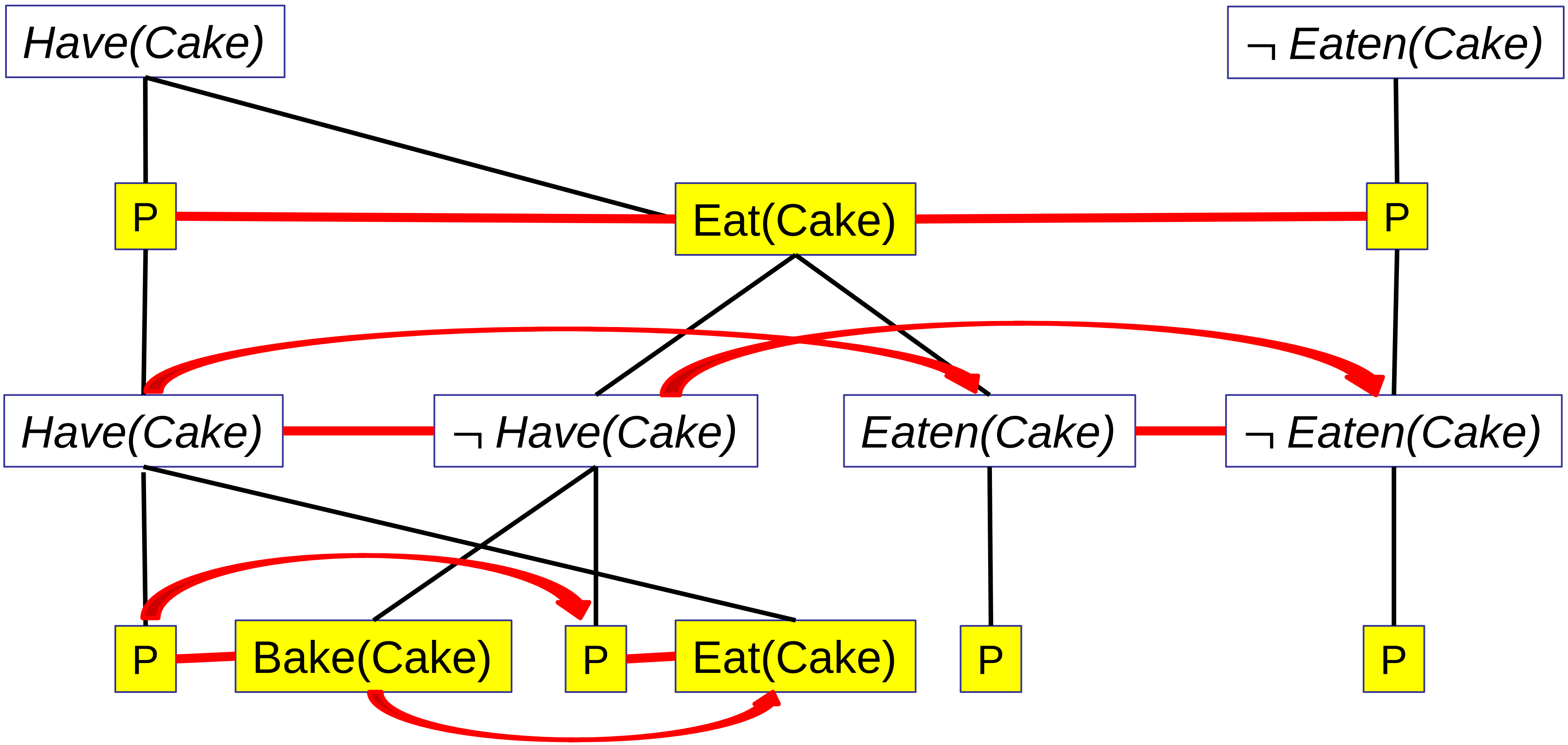
- Mutexes for inconsistent preconditions *Have(Cake)* and ¬ *Have(Cake)*?

Planning Graph for the Cake Problem



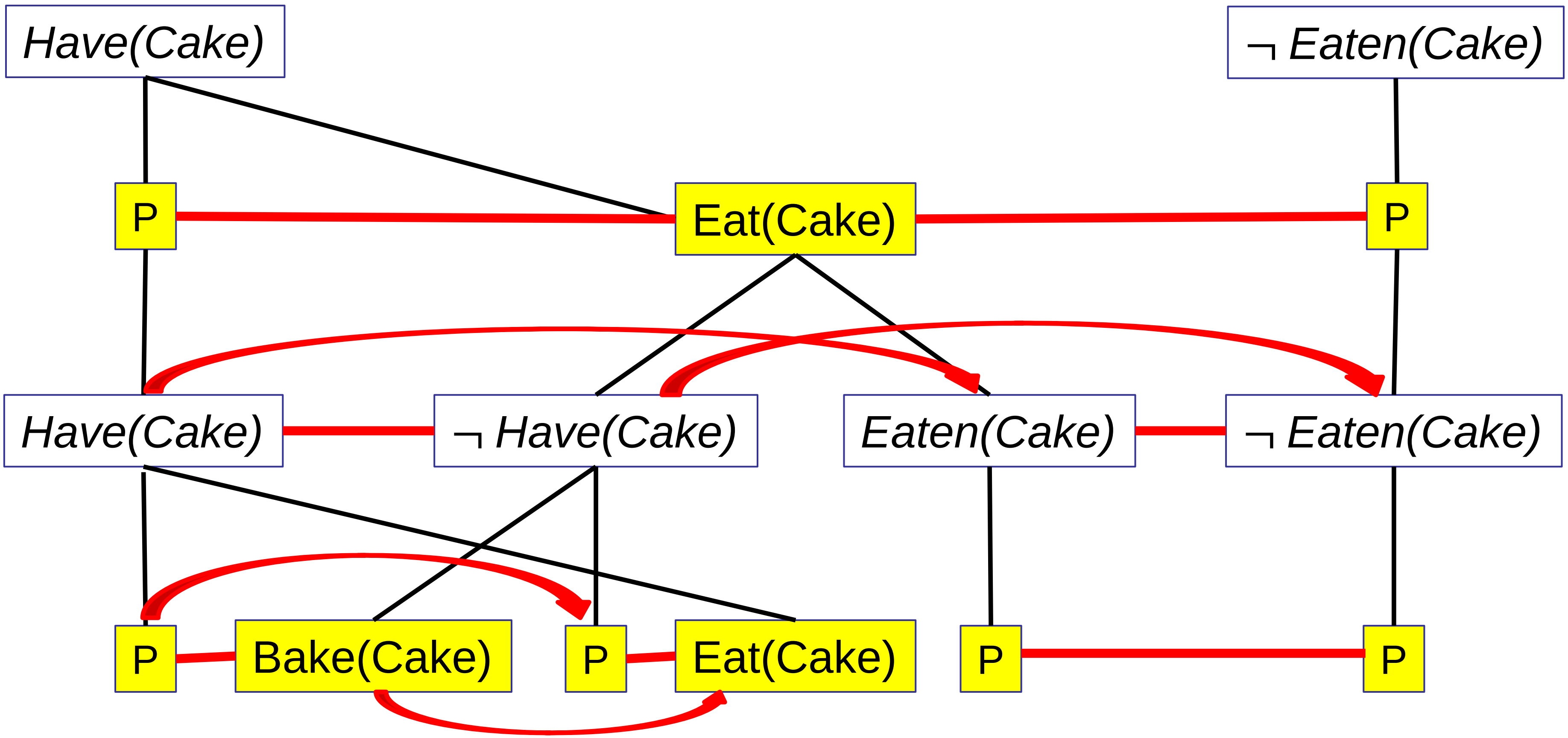
- *Have(Cake)* is a precondition for its persistence and for *Eat(Cake)*.
- ¬ *Have(Cake)* is a precondition for its persistence and for *Bake(Cake).*
- Thus, we need to add four mutex links based on these conflicts.

Planning Graph for the Cake Problem



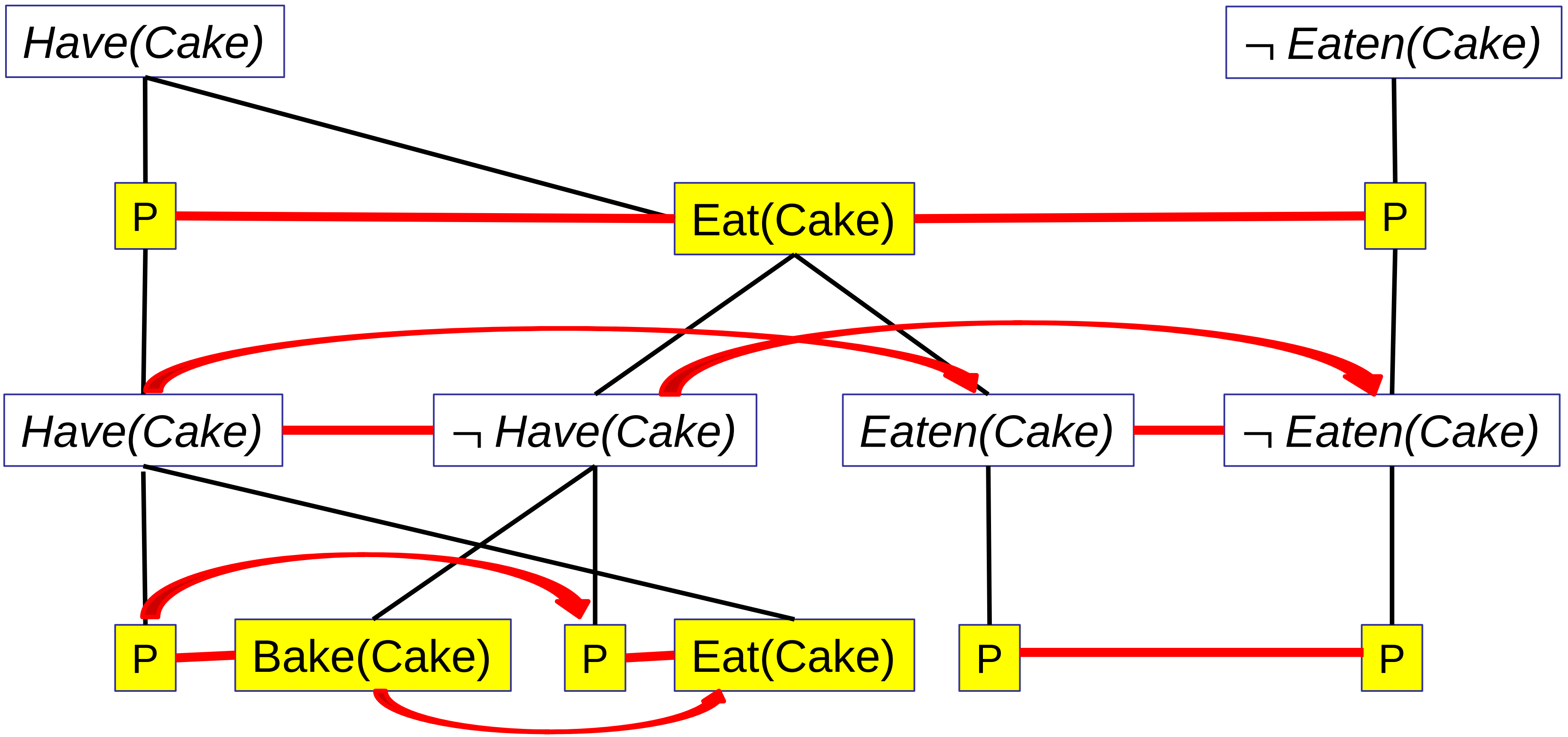
- Mutexes for inconsistent preconditions *Eaten(Cake)* and ¬ *Eaten(Cake)*?

Planning Graph for the Cake Problem



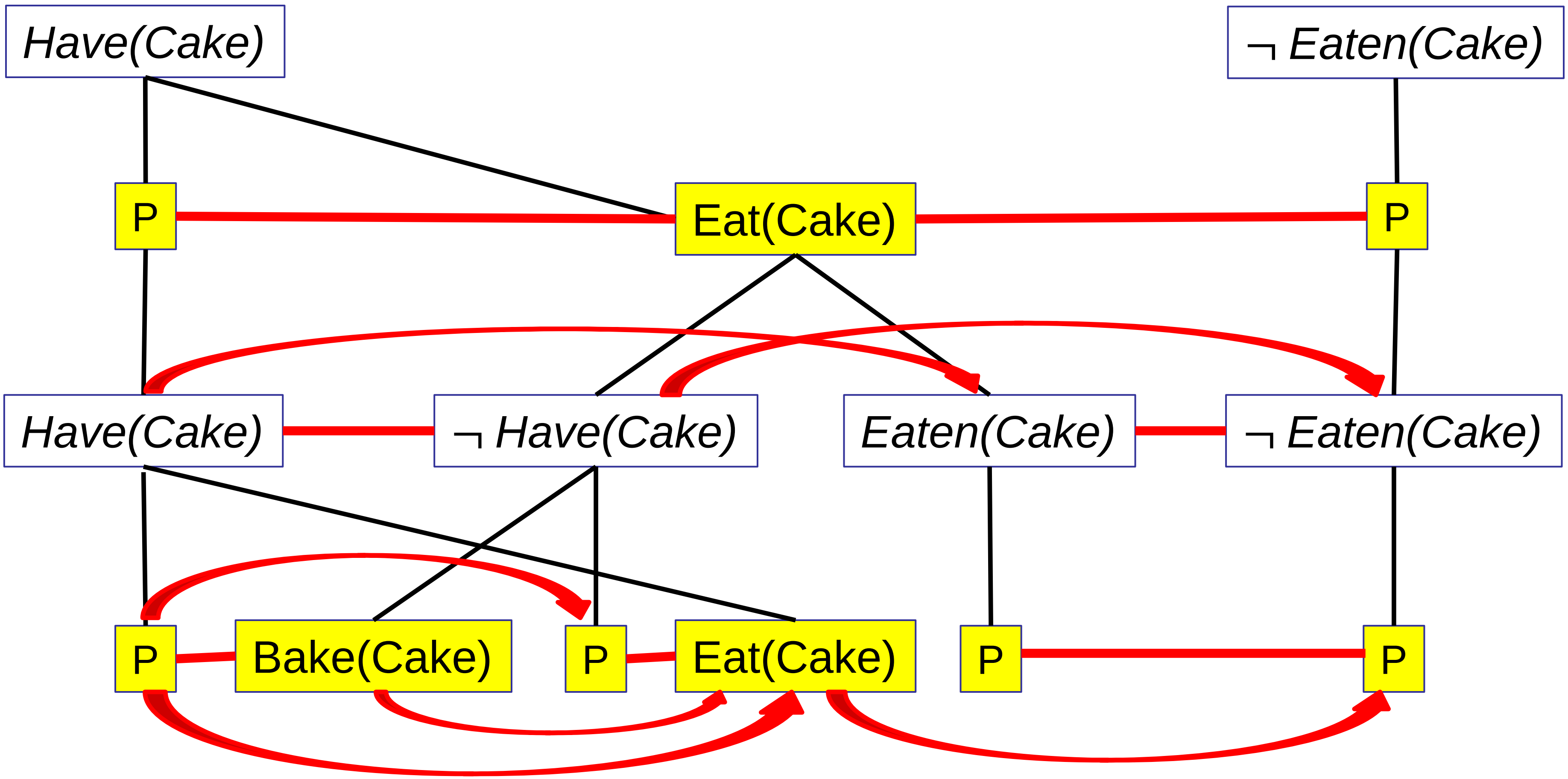
- *Eaten(Cake)* is a precondition for its persistence.
- ¬ *Eaten(Cake)* is a precondition for its persistence.

Planning Graph for the Cake Problem



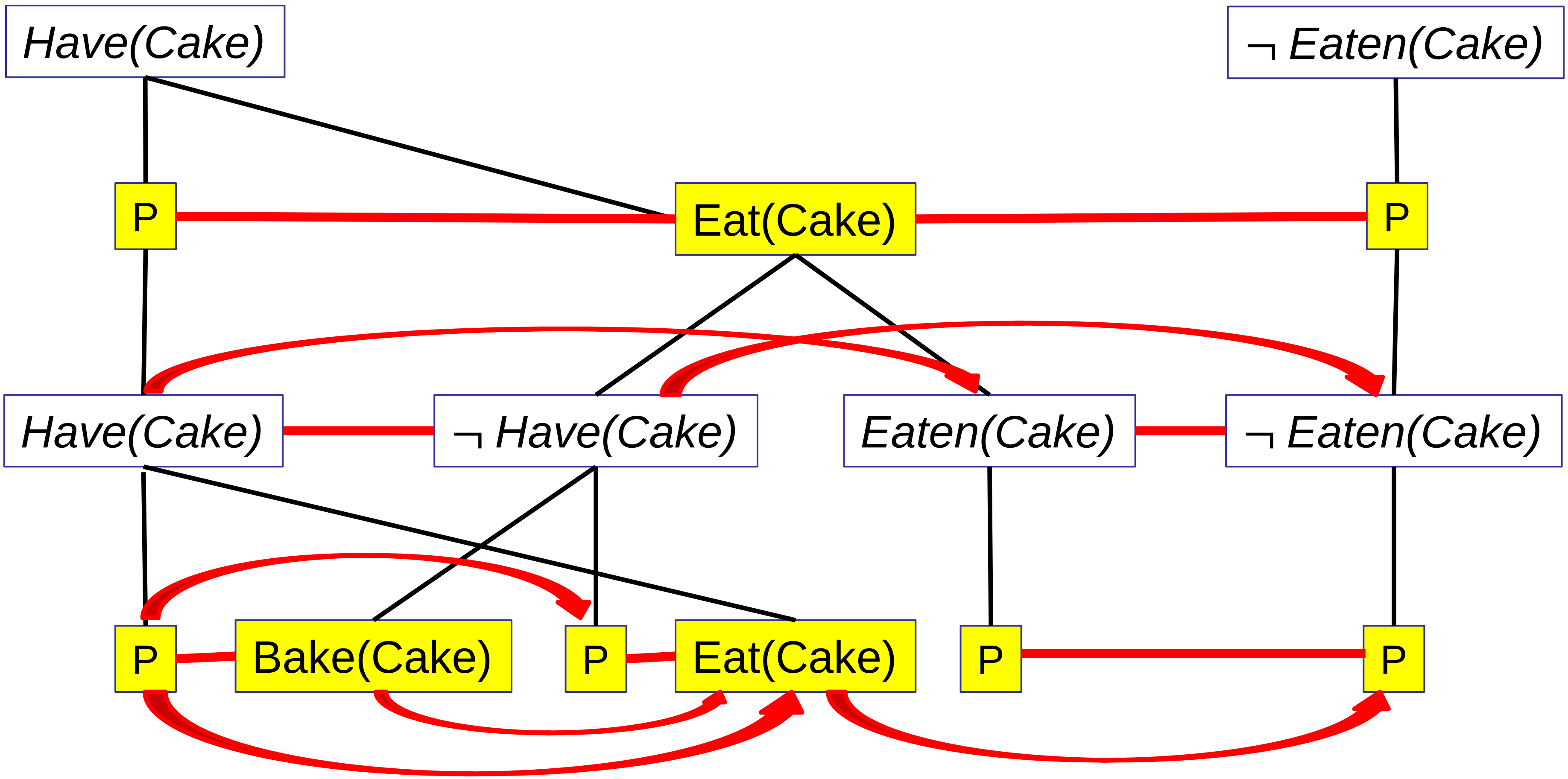
- Additional mutexes for inconsistent effects?

Planning Graph for the Cake Problem



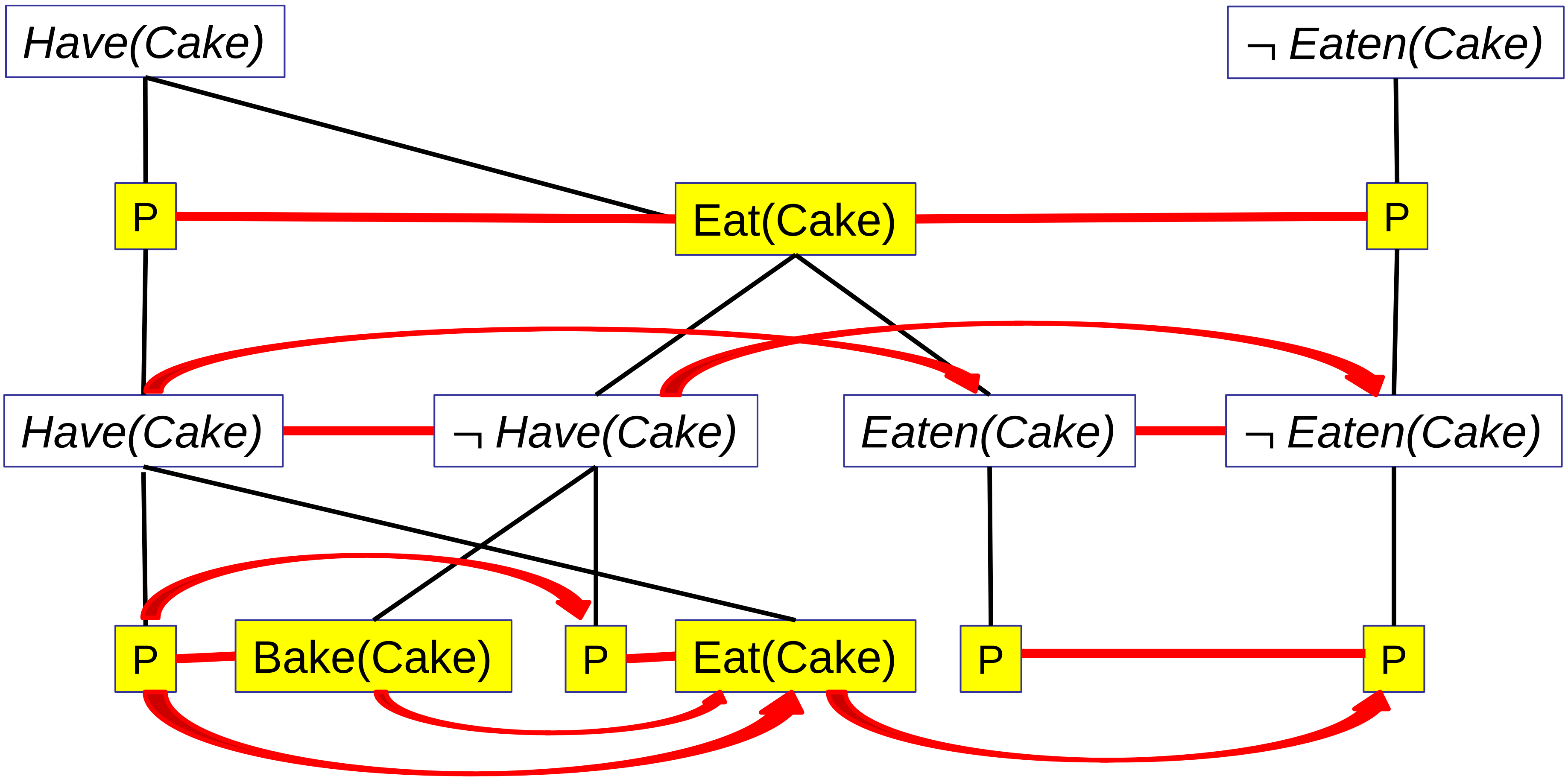
- Additional mutexes for inconsistent effects?
- *Eat(Cake)* negates both *Have(Cake)* and ¬ *Eaten(Cake).*

Planning Graph for the Cake Problem



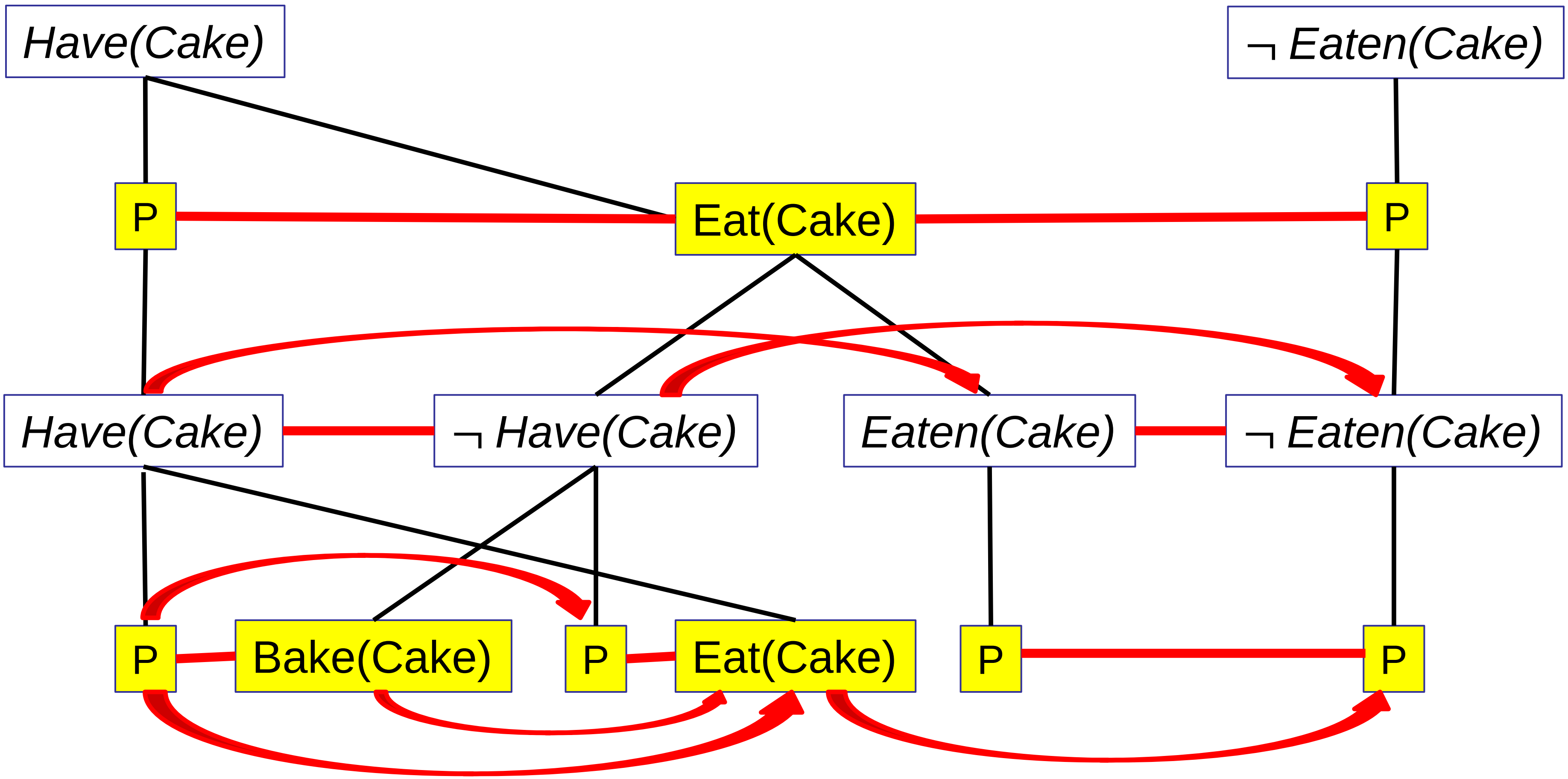
- Additional mutexes for interference?

Planning Graph for the Cake Problem



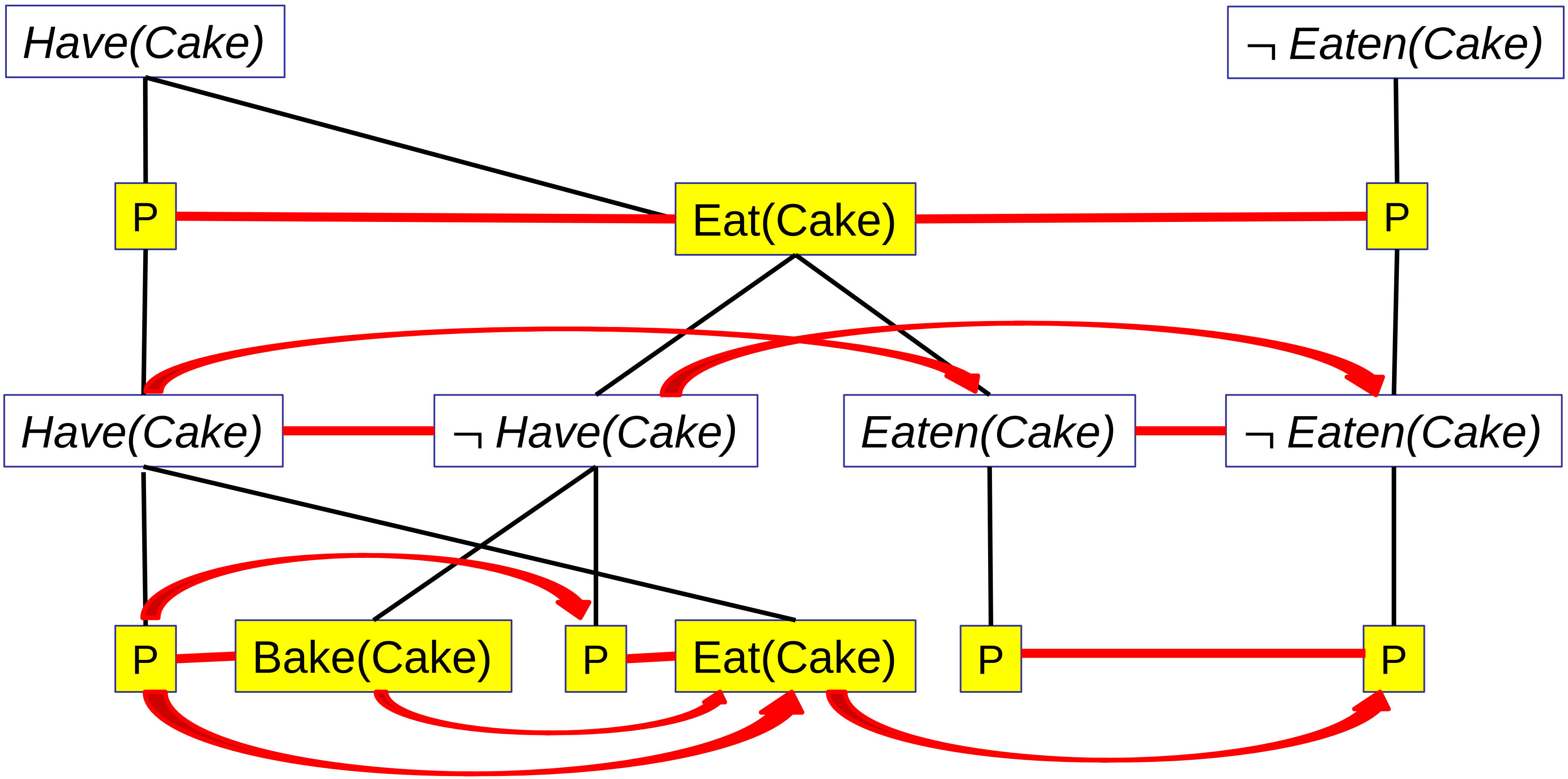
- Also, Bake(Cake) and Eat(Cake) are mutually exclusive because they are both real actions, but they have a mutex edge already, so no new edge is added.

Planning Graph for the Cake Problem



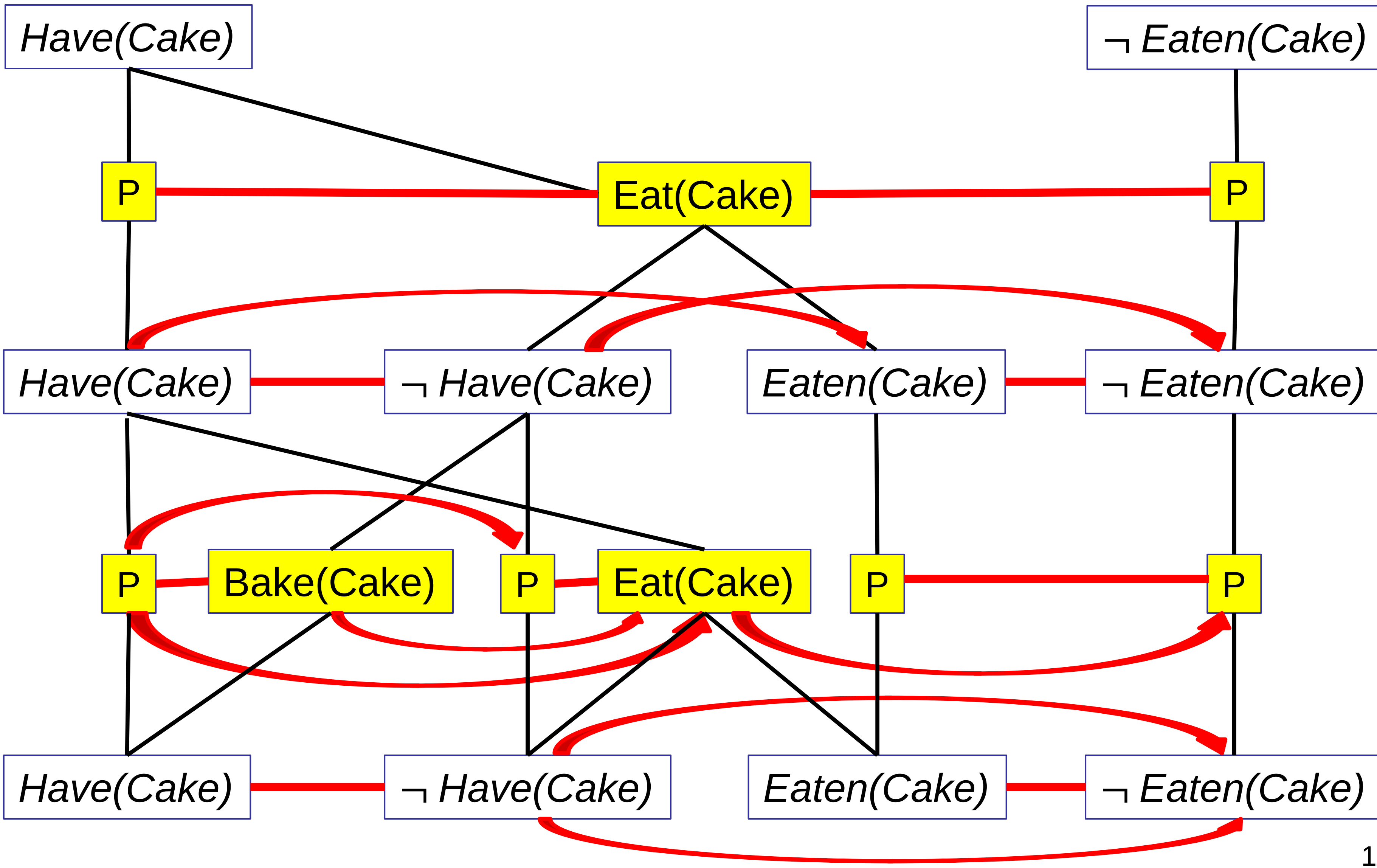
- Additional mutexes for interference? No.

Planning Graph for the Cake Problem



- Next: level S_2. Shown on next slide, with all mutex edges added.

Planning Graph for the Cake Problem



Stopping Criterion

- We can stop the planning graph when we reach a level S_k that satisfies these requirements:
  - S_k includes all the goal literals.
  - There are no mutex edges connecting any pair of goal literals.
- In our Cake example, the goals are *Have(Cake)* and *Eaten(Cake)*.
- Consider the planning graph of the previous slide. Does level S_1 satisfy the requirements?

Stopping Criterion

- We can stop the planning graph when we reach a level S_k that satisfies these requirements:
  - S_k includes all the goal literals.
  - There are no mutex edges connecting any pair of goal literals.
- In our Cake example, the goals are *Have(Cake)* and *Eaten(Cake)*.
- Consider the planning graph of the previous slide. Does level S_1 satisfy the requirements?
- No! *Have(Cake)* and *Eaten(Cake)* are mutually exclusive.

Stopping Criterion

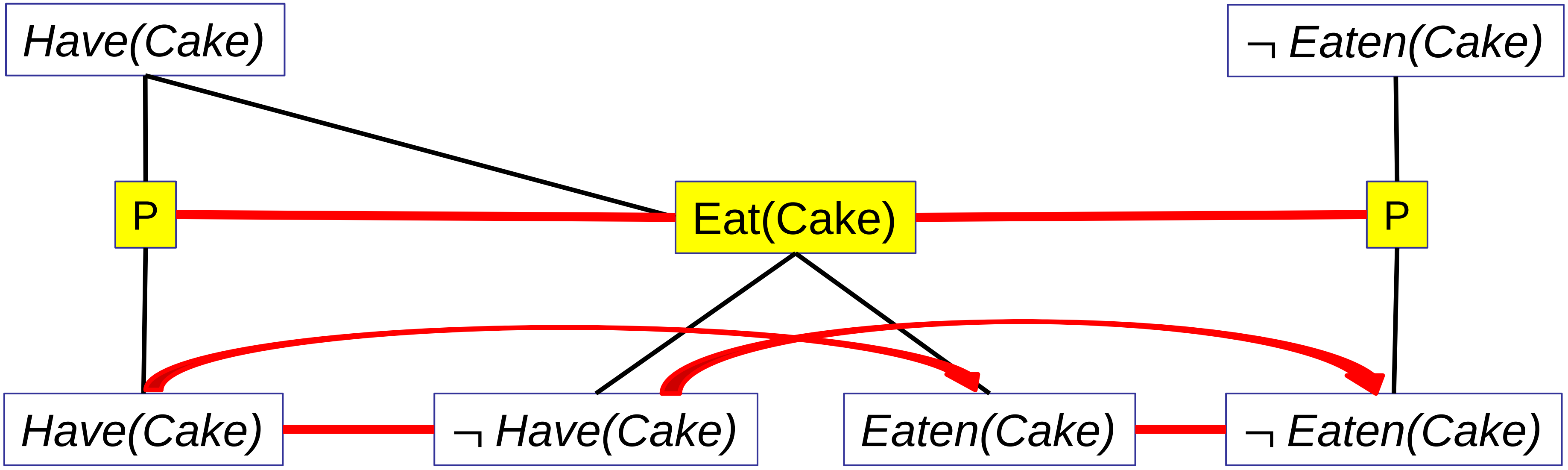
- We can stop the planning graph when we reach a level S_k that satisfies these requirements:
  - S_k includes all the goal literals.
  - There are no mutex edges connecting any pair of goal literals.
- In our Cake example, the goals are *Have(Cake)* and *Eaten(Cake)*.
- Does level S_2 satisfy the requirements?

Stopping Criterion

- We can stop the planning graph when we reach a level S_k that satisfies these requirements:
  - S_k includes all the goal literals.
  - There are no mutex edges connecting any pair of goal literals.
- In our Cake example, the goals are *Have(Cake)* and *Eaten(Cake)*.
- Does level S_2 satisfy the requirements?
- Yes, *Have(Cake)* and *Eaten(Cake)* are both present and NOT mutually exclusive at level S_2.

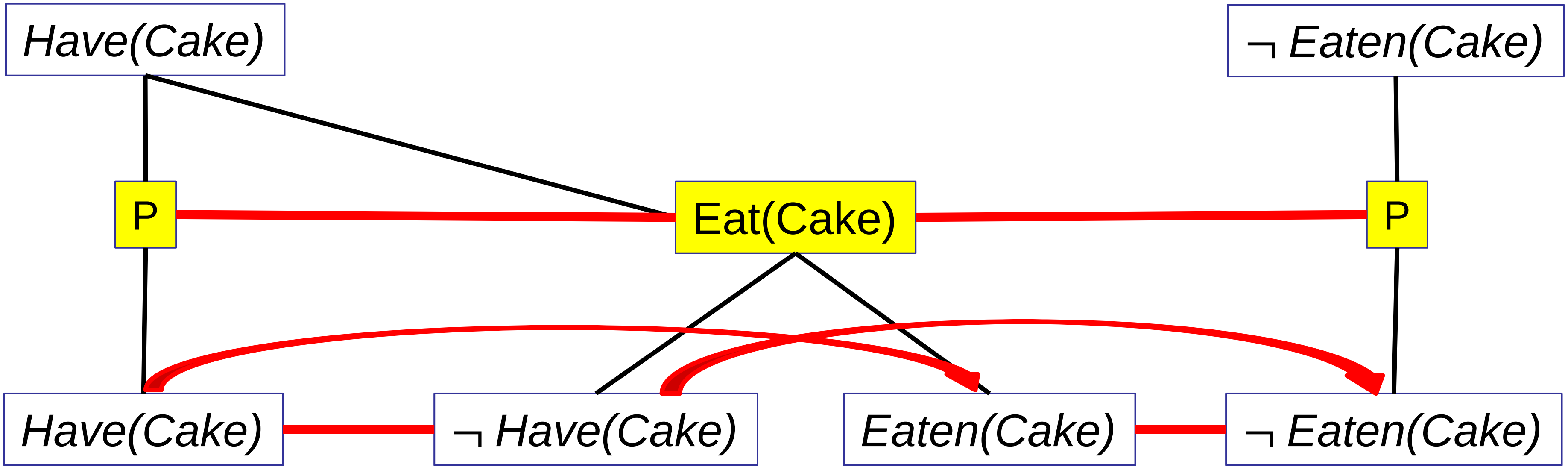
Level Costs

- The level cost of a goal literal g_k is simply the first level where g_k appears in the graph.
- What are the level costs for the four literals in the Cake problem?



Level Costs

- The level cost of a goal literal g_k is simply the first level where g_k appears in the graph.
- What are the level costs for the four literals in the Cake problem?
  - 0 for *Have(Cake)* and ¬ *Eaten(Cake).*
  - 1 for ¬ *Have(Cake)* and *Eaten(Cake).*



Level Costs

- The level cost of a goal literal g_k is simply the first level where g_k appears in the graph.
- What are the level costs for the four literals in the Cake problem?
  - 0 for *Have(Cake)* and $\neg$ *Eaten(Cake).*
  - 1 for $\neg$ *Have(Cake)* and *Eaten(Cake).*
- This is the million dollar question (and the reason we construct graphing plans):
  What does the level cost of g_k tell us about g_k?

Level Costs

- The level cost of a goal literal g_k is simply the first level where g_k appears in the graph.
- What are the level costs for the four literals in the Cake problem?
  - 0 for *Have(Cake)* and $\neg$ *Eaten(Cake).*
  - 1 for $\neg$ *Have(Cake)* and *Eaten(Cake).*
- This is the million dollar question (and the reason we construct graphing plans):
  What does the level cost of g_k tell us about g_k?
  - To achieve g_k we need at least as many actions as the level cost of g_k.

Defining Heuristics

- What heuristics can we define using a planning graph?

The Level Sum Heuristic

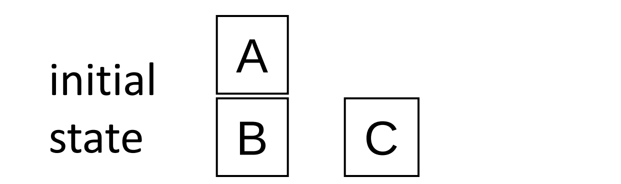
- The **level sum** heuristic is the sum of the level costs of the goal literals.
- Is this admissible?

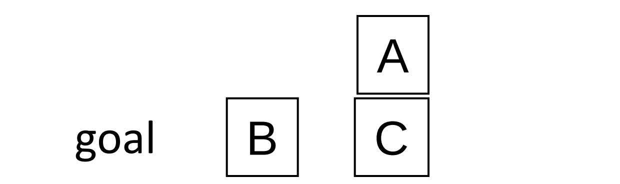
The Level Sum Heuristic

- The **level sum** heuristic is the sum of the level costs of the goal literals.
- Is this admissible?
- No. Here is a counter-example from the block world.
  - The initial state is shown on the left.
  - The goal is *clear(B)* and *on(A, C)*.
  - What is the level cost of the two goal literals?

The Level Sum Heuristic

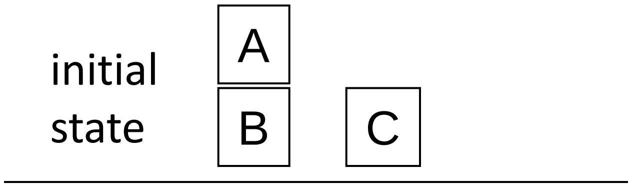
- The **level sum** heuristic is the sum of the level costs of the goal literals.
- Is this admissible?
- No. Here is a counter-example from the block world.
  - The initial state is shown on the left.
  - The goal is *clear(B)* and *on(A, C)*.
  - What is the level cost of the two goal literals?
    - Both *clear(B)* and *on(A, C)* have level cost 1.

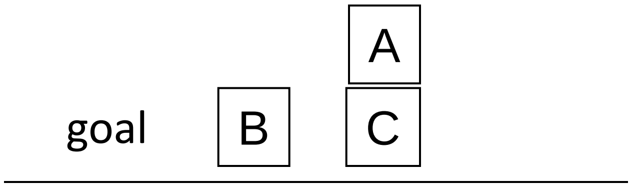




The Level Sum Heuristic

- The **level sum** heuristic is the sum of the level costs of the goal literals.
- Is this admissible?
- No. Here is a counter-example from the block world.
  - The initial state is shown on the left.
  - The goal is *clear(B)* and *on(A, C)*.
  - What is the level cost of the two goal literals?
    - Both *clear(B)* and *on(A, C)* have level cost 1.
  - What is the level sum value?

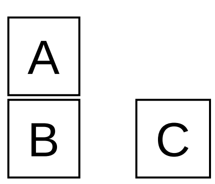




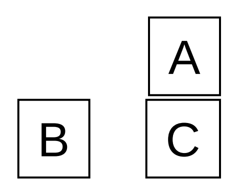
The Level Sum Heuristic

- The **level sum** heuristic is the sum of the level costs of the goal literals.
- Is this admissible?
- No. Here is a counter-example from the block world.
  - The initial state is shown on the left.
  - The goal is *clear(B)* and *on(A, C)*.
  - What is the level cost of the two goal literals?
    - Both *clear(B)* and *on(A, C)* have level cost 1.
  - What is the level sum value? 2.

initial state

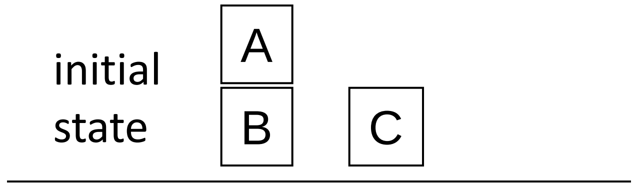


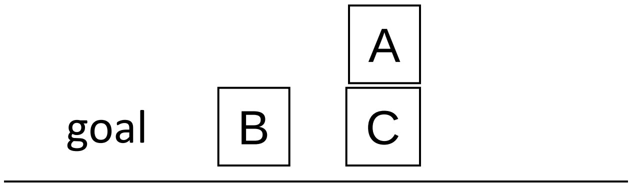
goal



The Level Sum Heuristic

- The **level sum** heuristic is the sum of the level costs of the goal literals.
- Is this admissible?
- No. Here is a counter-example from the block world.
  - The initial state is shown on the left.
  - The goal is *clear(B)* and *on(A, C)*.
  - What is the level cost of the two goal literals?
    - Both *clear(B)* and *on(A, C)* have level cost 1.
  - What is the level sum value? 2.
  - How many actions are needed to solve the problem?

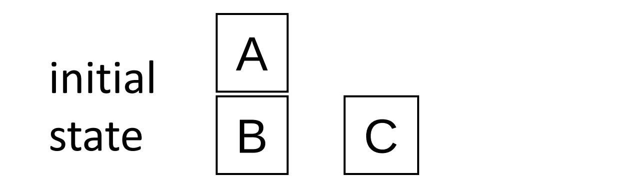




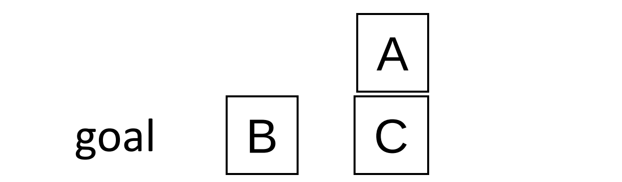
The Level Sum Heuristic

- The **level sum** heuristic is the sum of the level costs of the goal literals.
- Is this admissible?
- No. Here is a counter-example from the block world.
  - The initial state is shown on the left.
  - The goal is *clear(B)* and *on(A, C)*.
  - What is the level cost of the two goal literals?
    - Both *clear(B)* and *on(A, C)* have level cost 1.
  - What is the level sum value? 2
  - How many actions are needed to solve the problem? 1.
- It can still be a useful heuristic, even if not admissible.

initial state



goal



The Max-Level Heuristic

- The **max-level** heuristic is simply the maximum level cost of any of the goal literals.
  - Is this admissible?

The Max-Level Heuristic

- The **max-level** heuristic is simply the maximum level cost of any of the goal literals.
  - Is this admissible?
  - Yes. We need at least max-level actions to achieve the goal literal that has max-level as its level cost.

The Set-Level Heuristic

- The **set-level** heuristic is the first level where:
  - All the goal literals appear.
  - No pair of the goal literals is mutually exclusive.
- Is this admissible?

The Set-Level Heuristic

- The **set-level** heuristic is the first level where:
  - All the goal literals appear.
  - No pair of the goal literals is mutually exclusive.
- Is this admissible?
  - Yes.

The Set-Level Heuristic

- The **set-level** heuristic is the first level where:
  - All the goal literals appear.
  - No pair of the goal literals is mutually exclusive.
- Is this admissible?
  - Yes.
- How does it compare to the max-level heuristic?

The Set-Level Heuristic

- The **set-level** heuristic is the first level where:
  - All the goal literals appear.
  - No pair of the goal literals is mutually exclusive.
- Is this admissible?
  - Yes.
- How does it compare to the max-level heuristic?
  - The set-level heuristic dominates the max-level heuristic.
  - So, the set-level is a better, more accurate heuristic than the max-level heuristic.

POP Planner

- POP stands for partial-order planning.
- It is a different approach to planning than what we have seen so far.
- So far we have seen methods that produce a sequential plan, where actions are explicitly ordered, from first to last.
  - We search through states of the world, looking for actions that take us to a goal state.
- POP, instead, searches through plans.
  - It starts with the empty plan.
  - It keeps adding actions.
  - It stops when it has a plan that achieves the goal.

Example: Ordering 10 Books

- We want to order 10 books from Amazon:
  - book3, book7, book13, book17, book20, book25, book30, book35, book40, book50.
- Facts in the knowledge base:

  has(Amazon, book1)

  has(Amaxon, book2)

  …

  has(Amazon, book1000000) // Amazon sells lots of book titles…
- Action buy(book, store)
  - preconds: has(store, book)

Example: Ordering 10 Books

- POP starts with an empty plan, listing the initial state and the goal literals.
- Red indicates literals that the current plan does not yet achieve. These are called **open preconditions**.

START

has(Amazon, book1), has(Amazon, book2), …, has(Amazon, book1000000),

we_have(book3), we_have(book7), we_have(book13), …, we_have(book50)

FINISH

Example: Ordering 10 Books

- POP picks an action that achieves one of the open preconditions.

START

has(Amazon, book1), has(Amazon, book2), …, has(Amazon, book1000000),

has(Amazon, book13)

buy(book13, Amazon)

we_have(book3), we_have(book7), we_have(book13), …, we_have(book50)

FINISH

Example: Ordering 10 Books

- POP picks another action that achieves one of the open preconditions.

START

has(Amazon, book1), has(Amazon, book2), …, has(Amazon, book1000000),

has(Amazon, book3)

buy(book3, Amazon)

has(Amazon, book13)

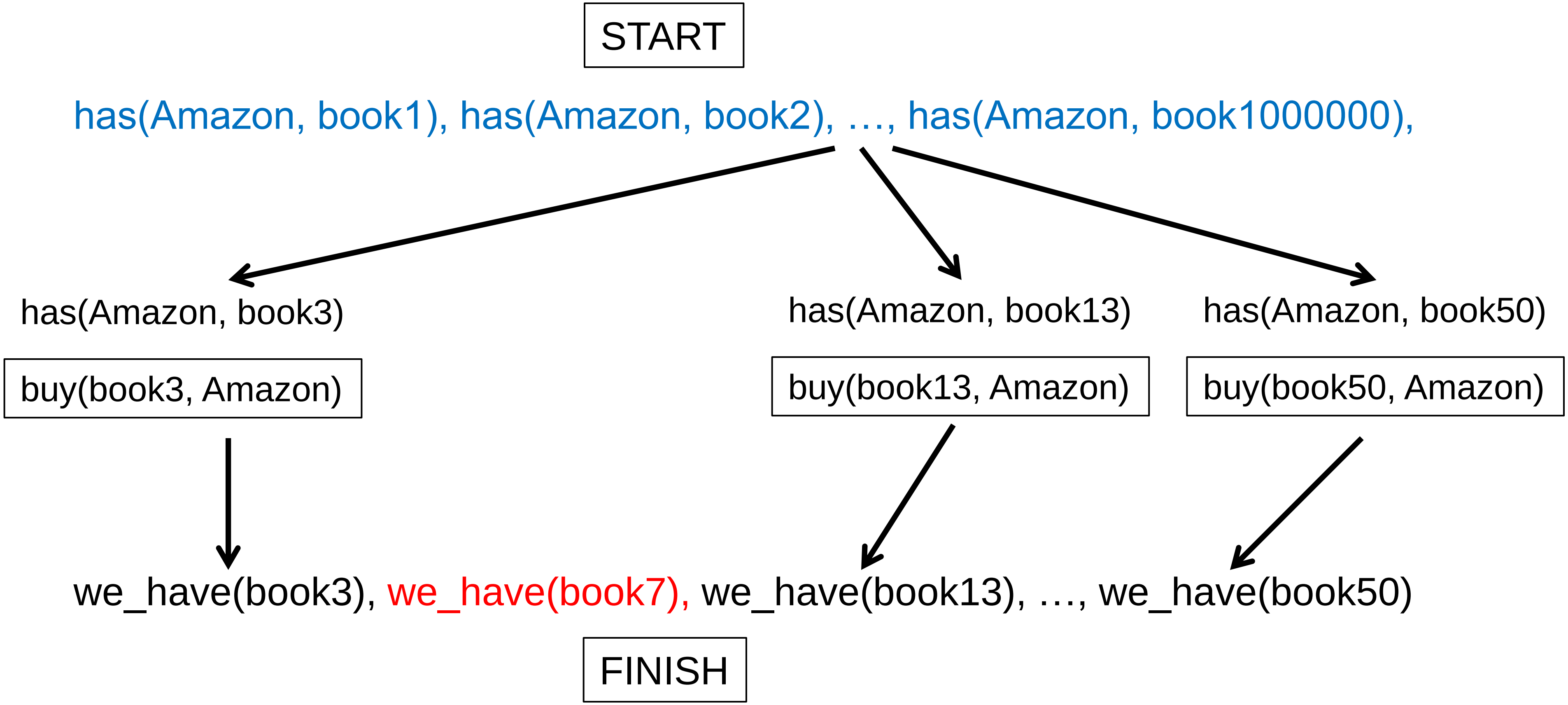
buy(book13, Amazon)

we_have(book3), we_have(book7), we_have(book13), …, we_have(book50)

FINISH

Example: Ordering 10 Books

- POP picks another action that achieves one of the open preconditions.



Example: Ordering 10 Books

- POP picks another action that achieves one of the open preconditions.

START

has(Amazon, book1), has(Amazon, book2), …, has(Amazon, book1000000),

has(Amazon, book3) | has(Amazon, book7) | has(Amazon, book13) | has(Amazon, book50)

buy(book3, Amazon) | buy(book7, Amazon) | buy(book13, Amazon) | buy(book50, Amazon)

we_have(book3), we_have(book7), we_have(book13), …, we_have(book50)

FINISH

POP Planner

- It is still a search algorithm.
- However, there is an important difference from a linear planner: the meaning of a search state:
- Linear planner:
  - A search state is a possible state of the world.
  - The initial search state is the initial state of the world.
  - The goal state is a state that satisfies goal conditions.
- POP planner:
  - A search state is a partial plan.
  - The initial state is the empty plan, with specified initial conditions and goal conditions.
  - The goal state is a complete plan, with no open preconditions.

POP Planner Pitfalls

- In cases like the book-ordering problem, where the goal literals are independent of each other, POP does really well.
  - It actually takes very little time to find the correct solution.
- However, there are more complicated cases, where satisfying one open precondition messes up another one.
  - There are ways for POP to deal with such cases, but we will not cover them in this class.
- Planning overall takes exponential time.
  - We can always find problems where both sequential planners and POP are too slow to be useful.