Introduction to AI, Spring 2023

Planning Graphs and Graphplan

Planning Graphs and Graphplan

- These two things sound very similar! But they are different!

- **Planning Graphs (or "Plan Graphs")** are abstracted representations of actions/steps and resulting states.
  - Their big benefit is that they provide some nice heuristic estimates for us!

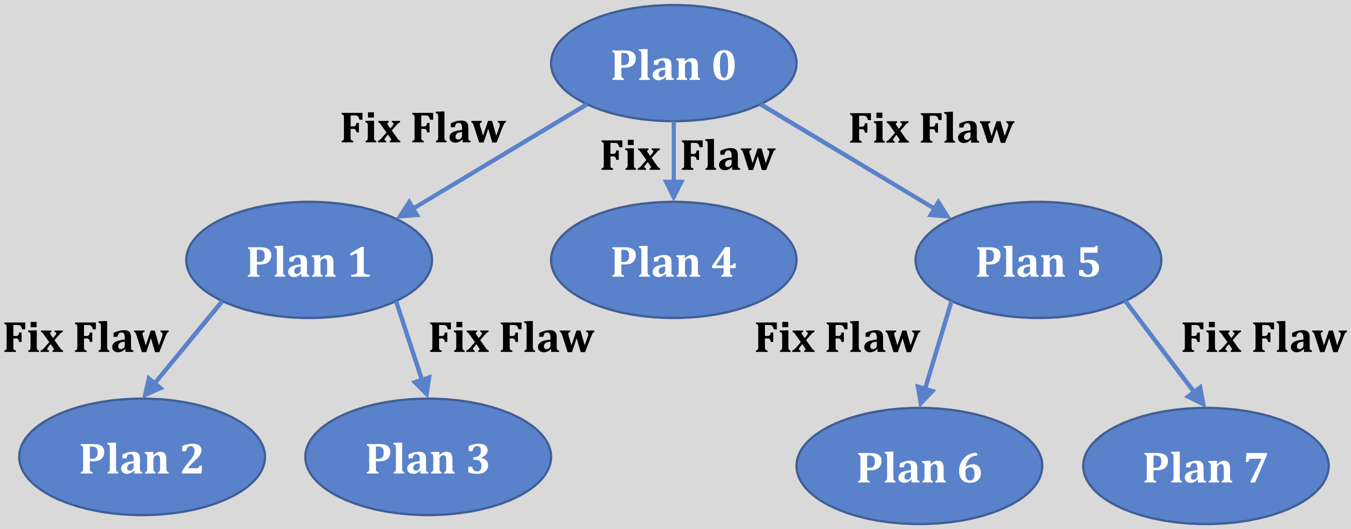
- **Graphplan** is an algorithm that tries to *produce* a plan *via* creating a planning graph.

A Quick Reminder of Planning Issues…

- We’ve seen how search space of planning problems grows quickly!
  - Adding one extra block to Blocks World led to a huge additional branching factor!
- Abstraction helps group many choices into a single choice.
  - For example, with partial order planning/least commitment: the single mandate “S2 < S1” can potentially still accommodate lots of different plans.
- But, that approach required plan-space search, which is a more complicated process than state-space search!
- We’ve promised that state-space search is possible with better heuristics.

Plan-Space Search

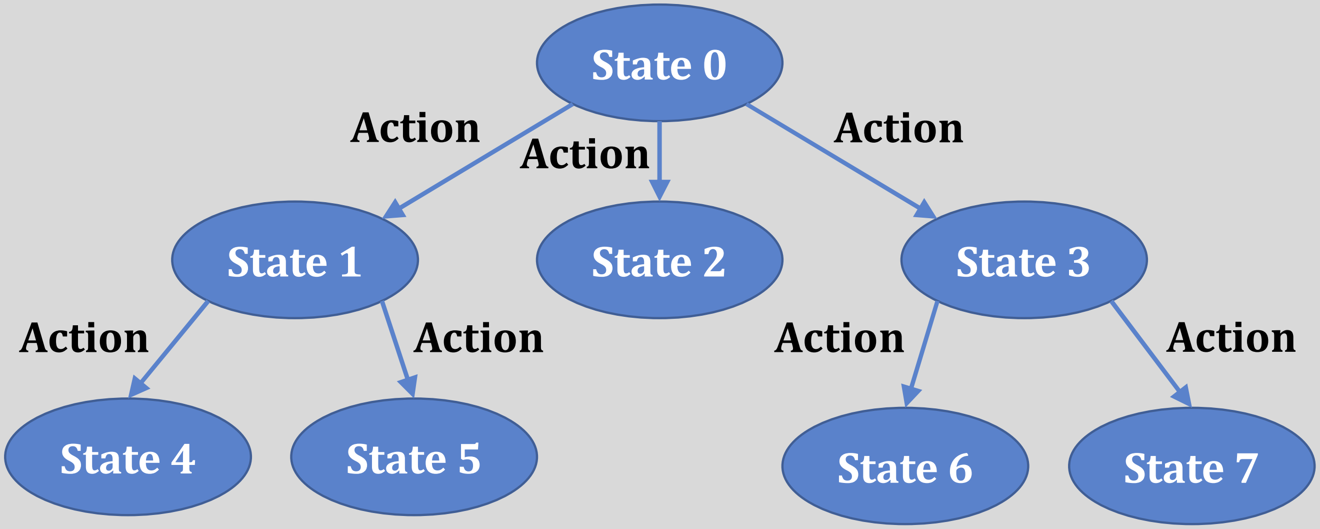
We saw this in "least commitment/partial order planning"



Most plans here are likely incomplete.
Also, remember how I said we got "lucky" with our fixes in the final least commitment example? This graph illustrates the "backtracking" reminiscent of search.

State-Space Search

What likely feels more intuitive, but I keep on saying is hard to do in planning. Also what you need to do for your programming assignment.

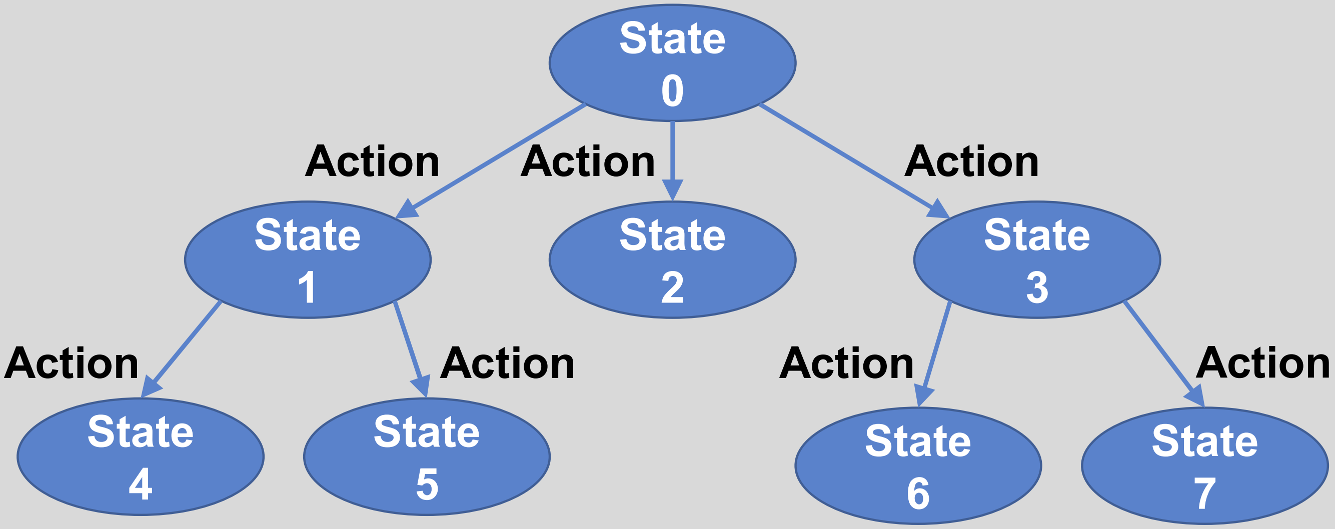


At any given moment, we know what the state of the world is.
Taking an action (i.e., a step in our plan) updates the state of the world by its effects.
And this graph is tiny, but we've seen the branching factor on these trees is actually huge.

The Idea Behind Plan Graphs

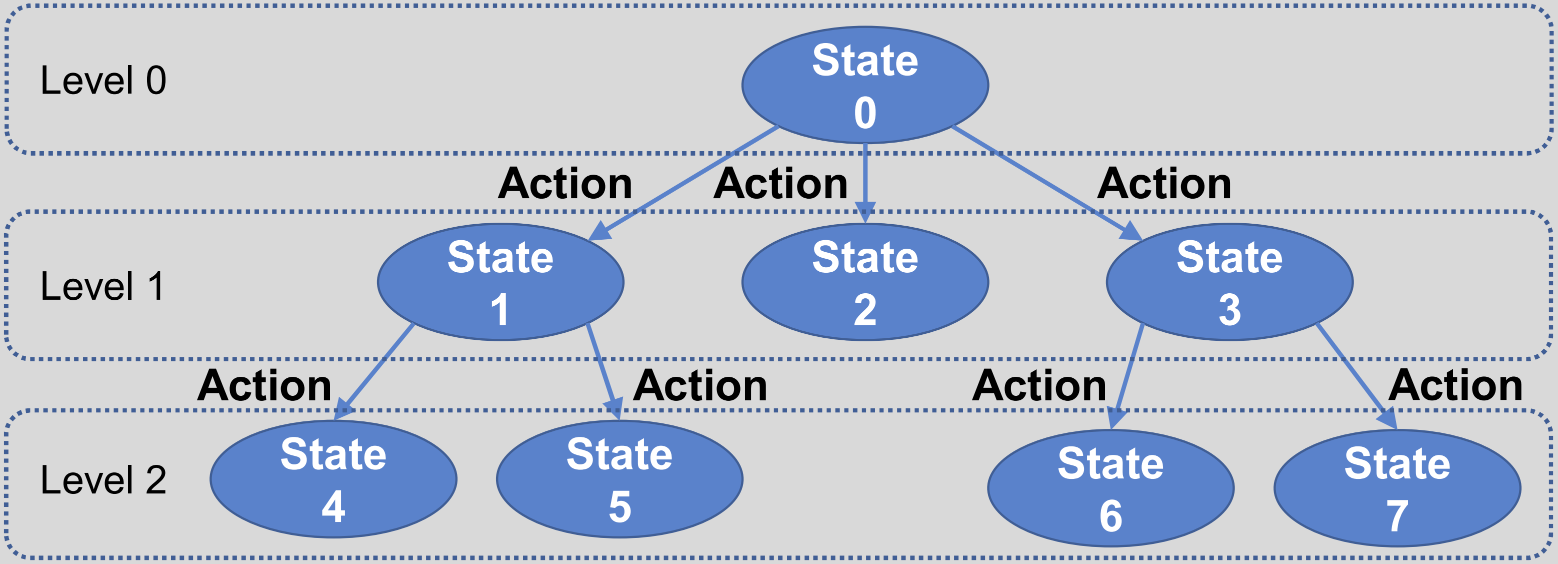
- As previously alluded…
- We are going to use this new thing called a plan graph to abstract the state space.
- We can use it to get accurate estimates (i.e., admissible heuristics) that will eventually be helpful for us…
- It’s also going to be a relatively small data structure that is easier to build and work with.
  - But don’t worry: there will still be some gnarly diagrams coming up!

State Space: Not-Abstracted



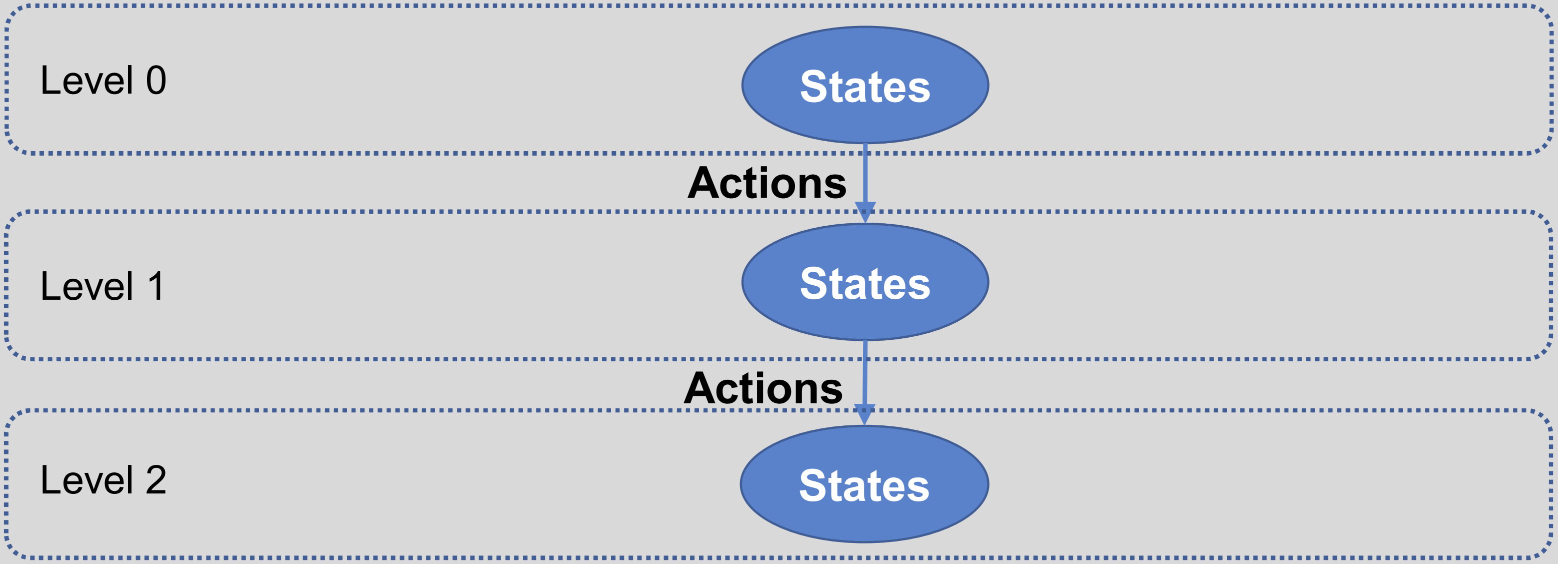
So, this is a tree. We've seen it lots. Each node is a state. Each action takes us from one state to another. Not too scary.

State Space: Not-Abstracted



Let’s divide it up into levels.
So, level 0 are all of the states we can reach after 0 actions…
Level 1 are all of the states we can reach after 1 action… still not too bad?

State Space: Abstracted



Now it’s abstracted!
We basically have condensed everything down…
Note the nodes now say “States”, not “State 0, state 1, state 2, etc.” They each now represent *all* the possible states that might be true after taking 0 actions, 1 action, 2 actions, etc.

Plan Graph

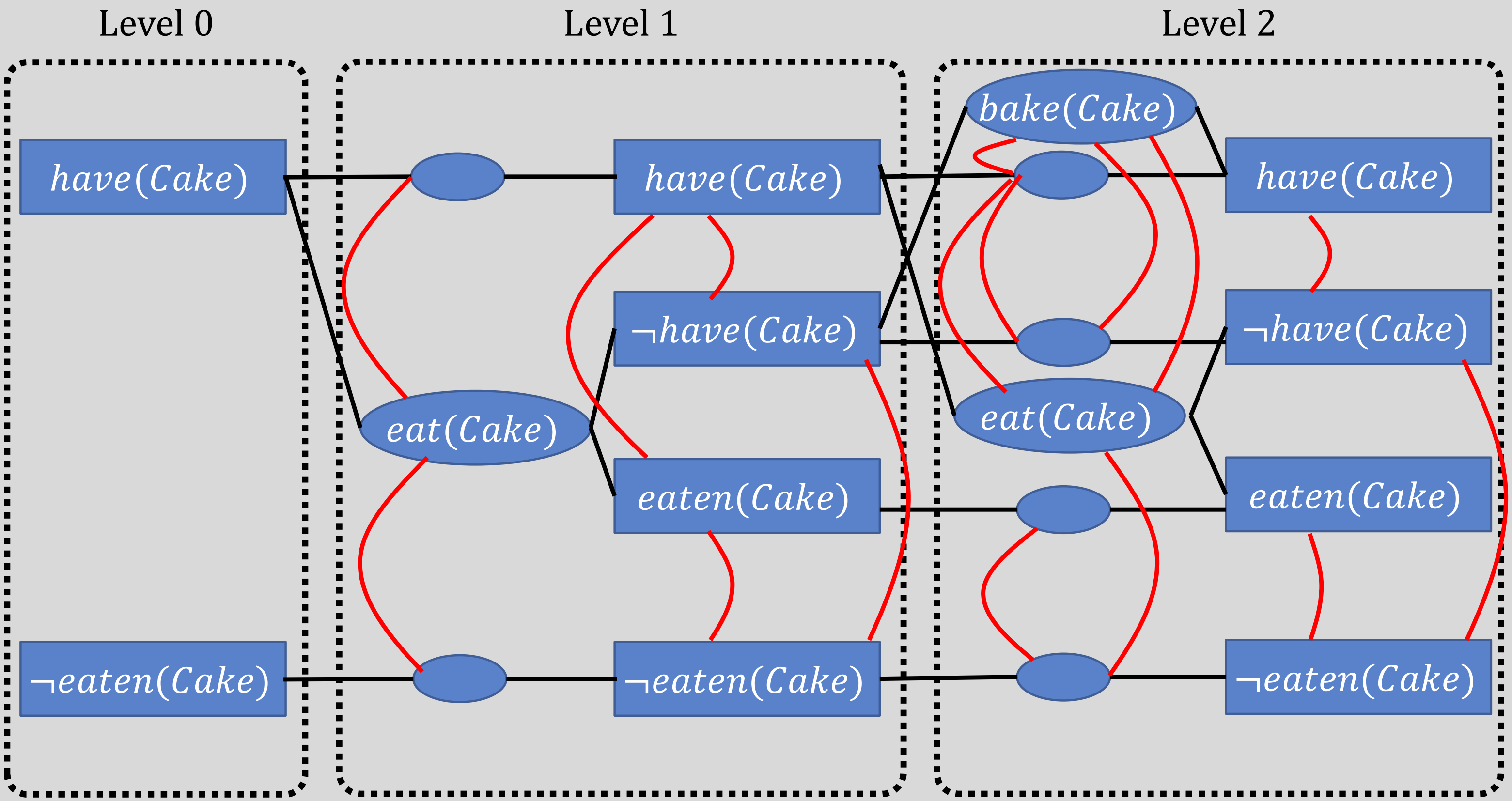
- Examples are coming up! But the basic idea:
- A **Plan Graph** is…
  - a directed graph…
  - divided into levels (i.e., we keep track of “distance from start state”).
  - That has two types of nodes: **Literal Nodes** and **Step Nodes**.

- **Literal Nodes:** Represent a single, ground, predicate literal. i.e., a single fact that may be true or false.
- **Step Nodes**: Represent a ground action.

Let’s see a sneak peak into what we’re going to building towards!

A Glimpse of the Future

This is a Plan Graph! Crazy Huh?!?
But we'll see how we can build this up piece by piece!

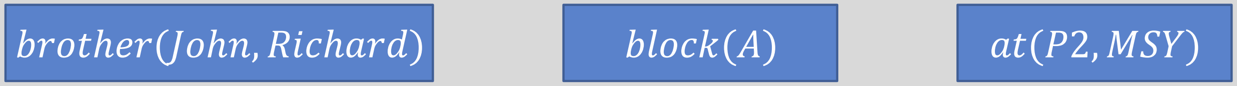


Literal Nodes

- So, each level in our graph will have literal nodes and step nodes.
- Remember: Literal Nodes represent facts about the world.
- What determines if a literal node shows up in a level?
  - A Node for literal L shows up at level 0 if L is true in the initial state (i.e., it is possible for L to be true after taking 0 actions).
  - A Node for Literal L shows up at level $n > 0$ if either:
    - 1.) A Node for L existed at level $n-1$ (i.e., the previous step)
      - (i.e., once it shows up in a level, it will show up in every subsequent level).
    - … OR
    - 2.) There exists a step node at level n whose step has L as an effect.

Literal Nodes

- In the graphs that follow, we are going to represent literal nodes like this:



$brother(John, Richard)$

$block(A)$

$at(P2, MSY)$

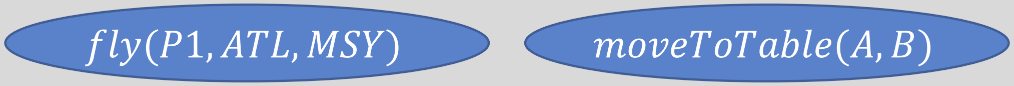
- That is, with rectangular boxes.
- Note: all of these are ground! All have truth values! No functions!

Step Nodes

- So, each level in our graph will have literal nodes and step nodes.
- Remember: Step Nodes represent actions we can take.
- What determines if a step node shows up in a level?
  - No step nodes exist at level 0.
    - Level 0 is the starting state; what life is like before actions are taken.
  - A Node for Step S shows up at level $n > 0$ if, for every precondition P of S, a Literal Node for P exists at the previous level (i.e., at level $n-1$)

Step Nodes

- In the graphs that follow, we are going to represent step nodes like this:



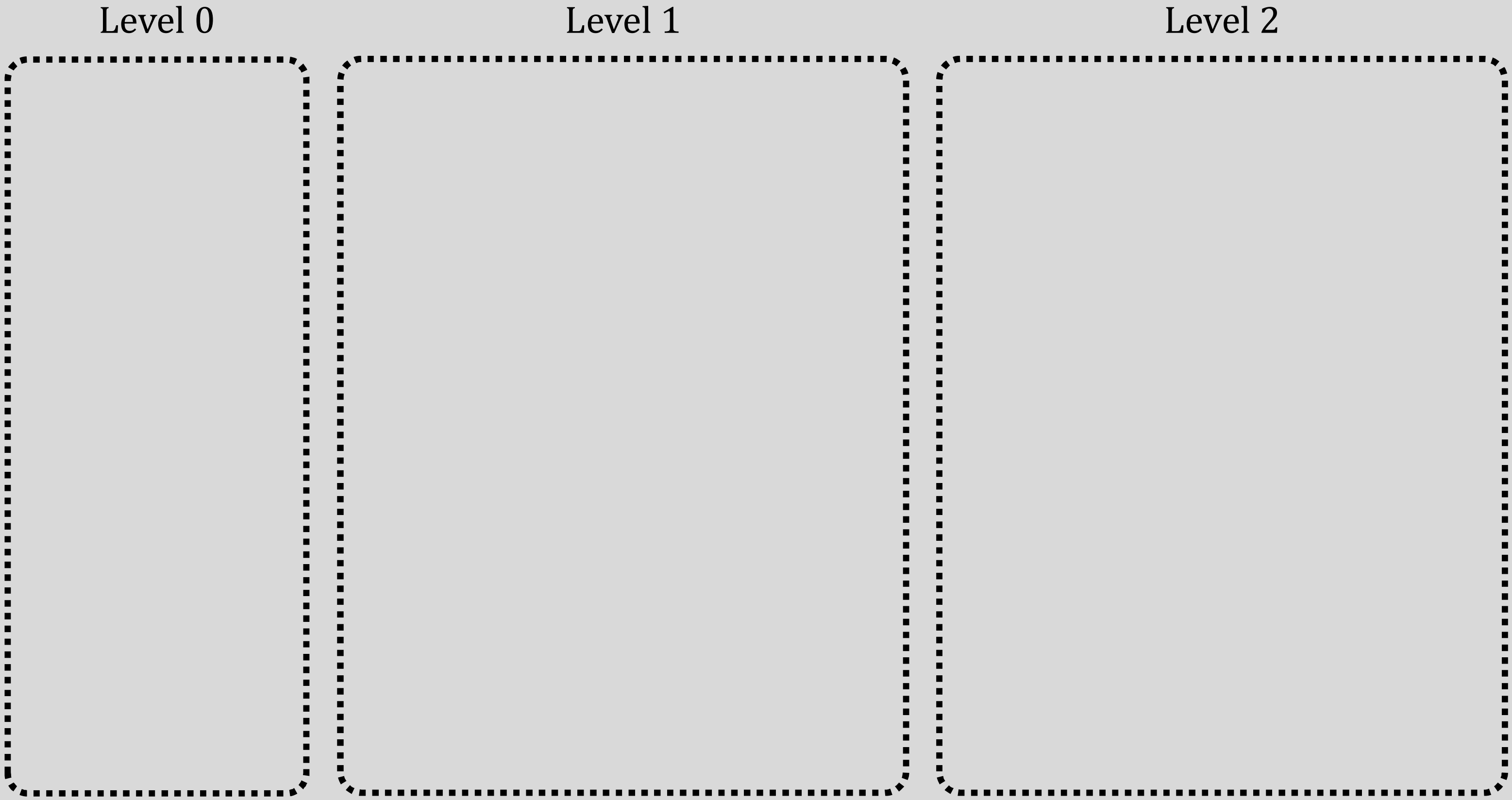
$fly(P1, ATL, MSY)$ $moveToTable(A, B)$

- That is, with circle boxes.
- Note: all of these are ground! No Variables!
- Although…

- We will also see “empty” circles. These are ”special” step nodes. More soon!

Plan Graphs

Once we start constructing plan graphs, they will look like this! Divided up into a bunch of ”Levels”



Level 0

Level 1

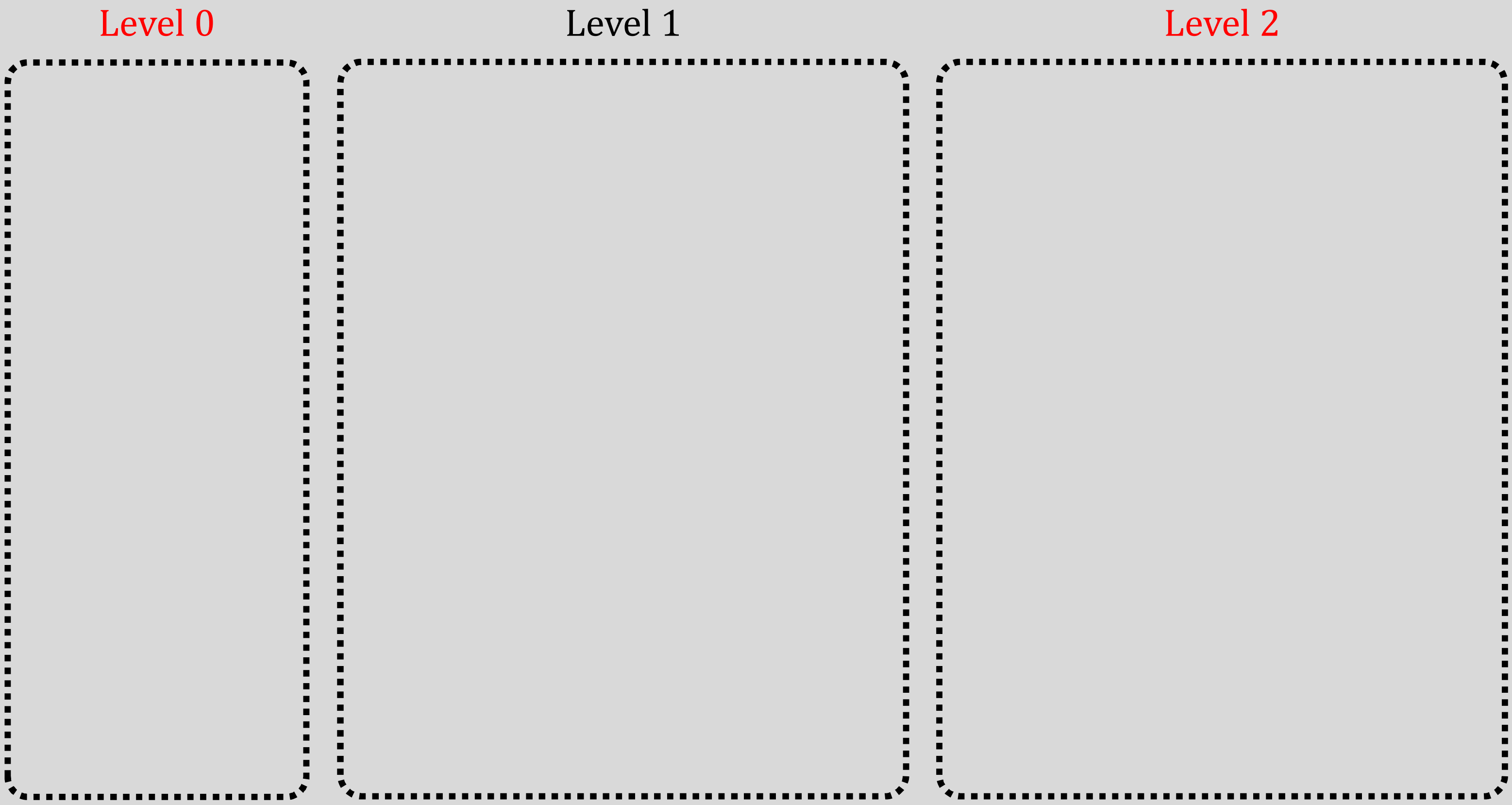
Level 2

Plan Graphs

You can think of the levels as “total number of steps from the start state”

So Level 0 will contain things that are true/possible in the initial state, i.e., after taking 0 steps.

And Level 2 will contain things that *can be* true/possible after taking 2 steps.



Level 0

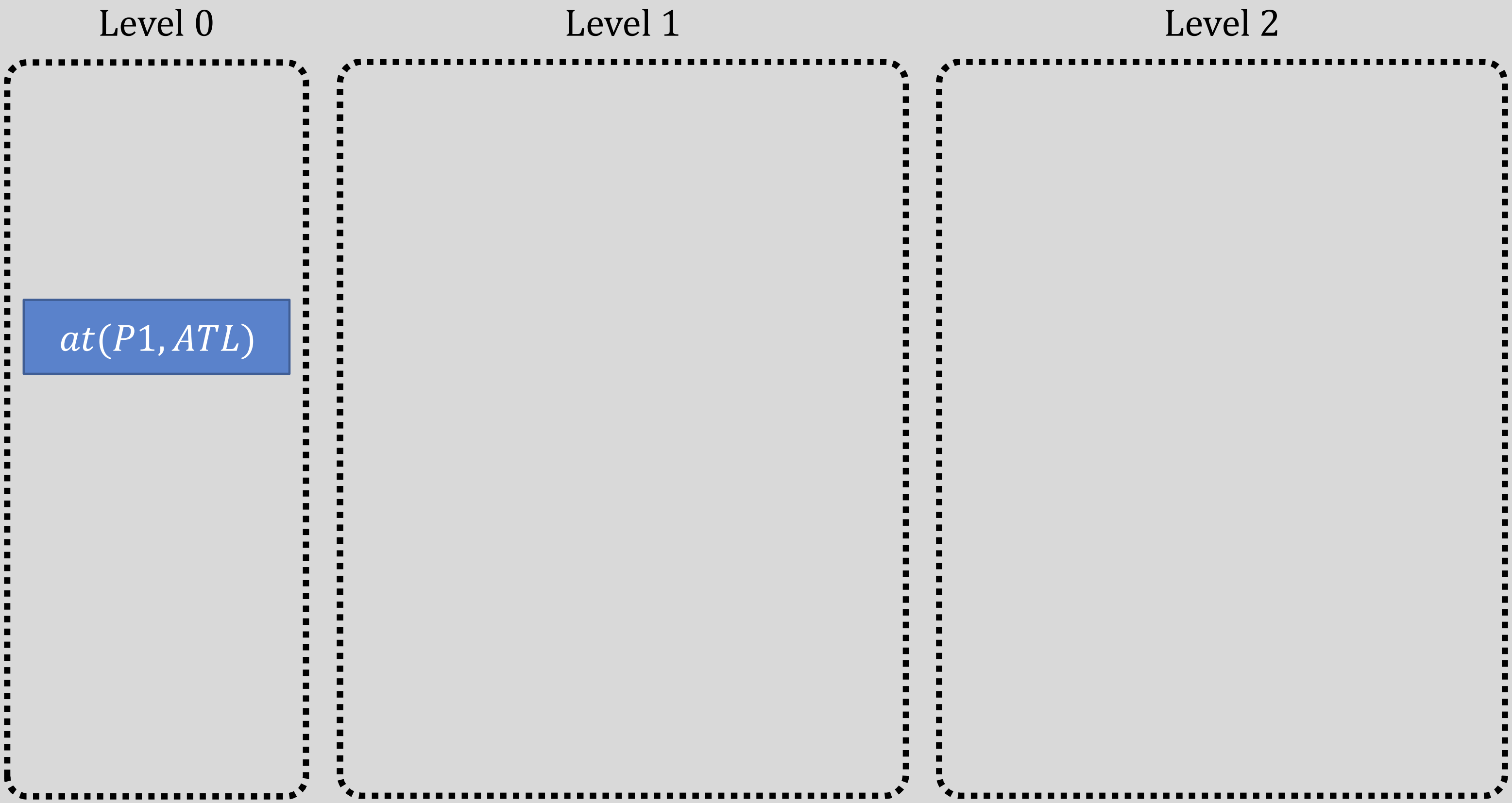
Level 1

Level 2

Plan Graphs

Let’s use the Cargo domain for now.
So, if Plane 1 *starts* at Atlanta, we’d put that literal node at Level 0.

Because we don’t need to take any steps in order to make that true.



Level 0

Level 1

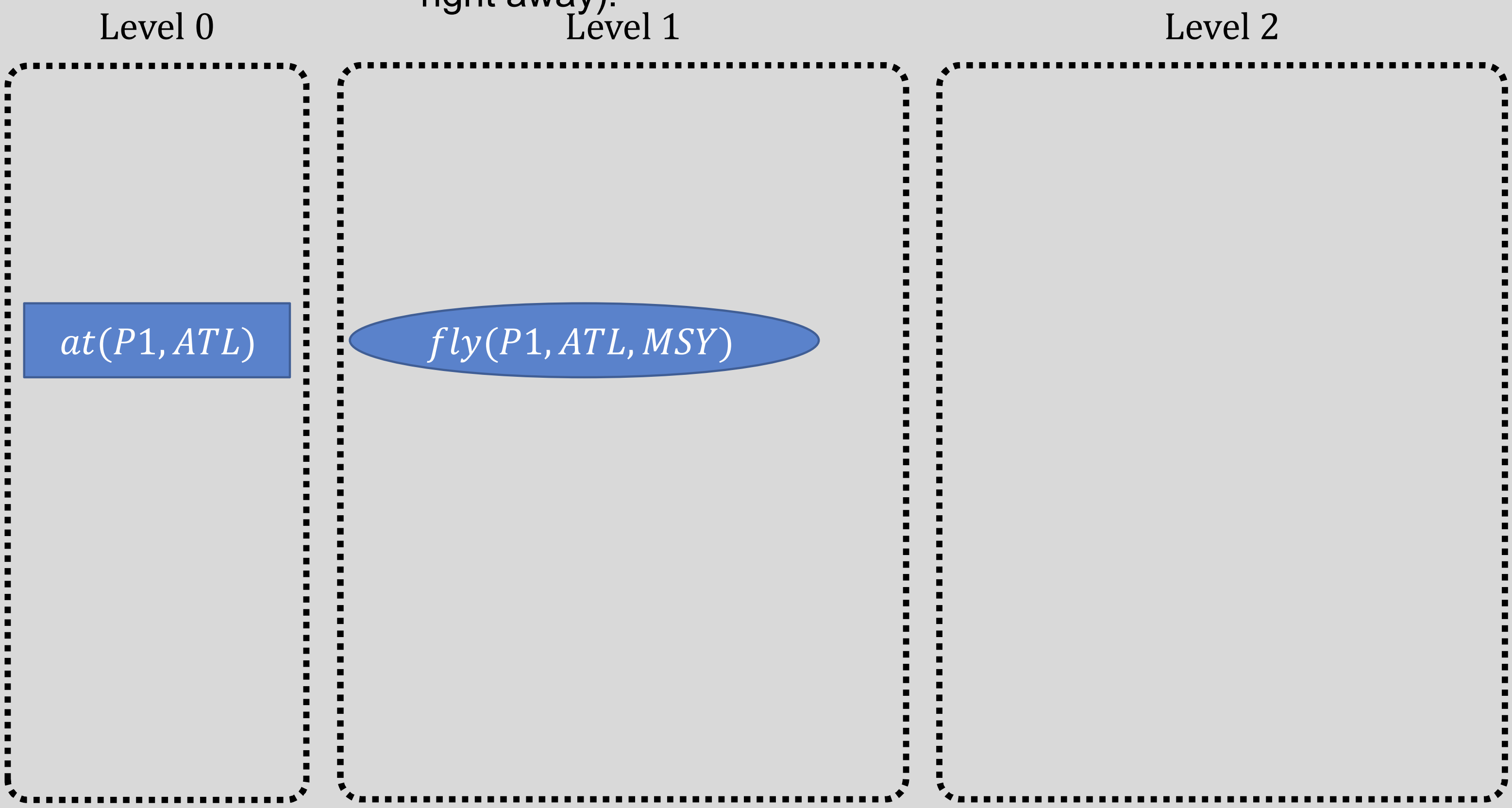
Level 2

$at(P1, ATL)$

Plan Graphs

If we see a step node here at say, Level 1…

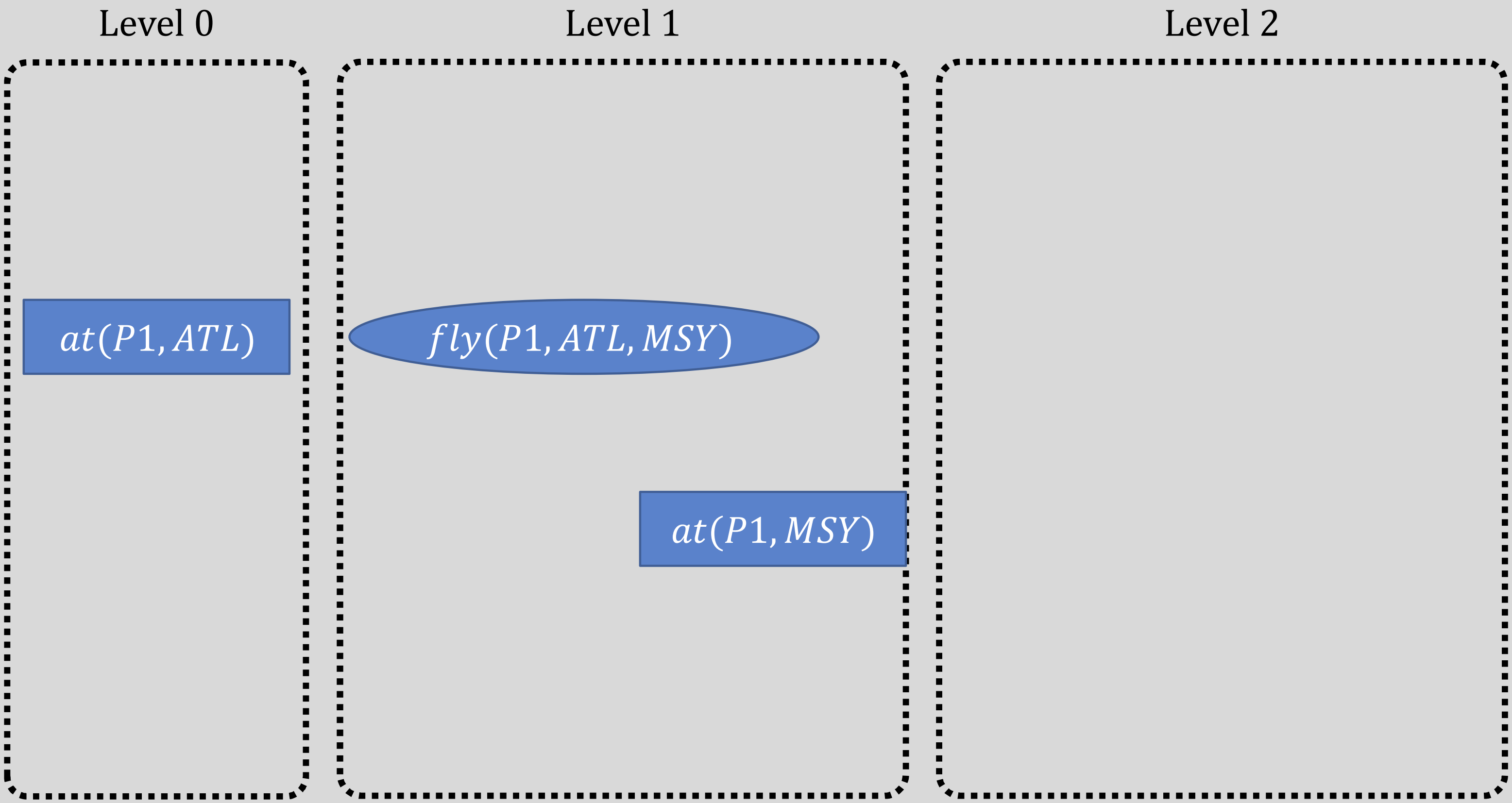
It means that the *soonest* we could possibly take this step is “step 1” (and indeed, since P1 starts at ATL, it’s allowed to fly from there to MSY right away).



Plan Graphs

And levels can have (and will have) step nodes and literal nodes at the same level.

This *at(P1,MSY)* we added to Level 1 means that it will take *at least* one step for P1 to be at MSY. The earliest we can make that predicate true is after 1 step.



Level 0

Level 1

Level 2

$at(P1, ATL)$

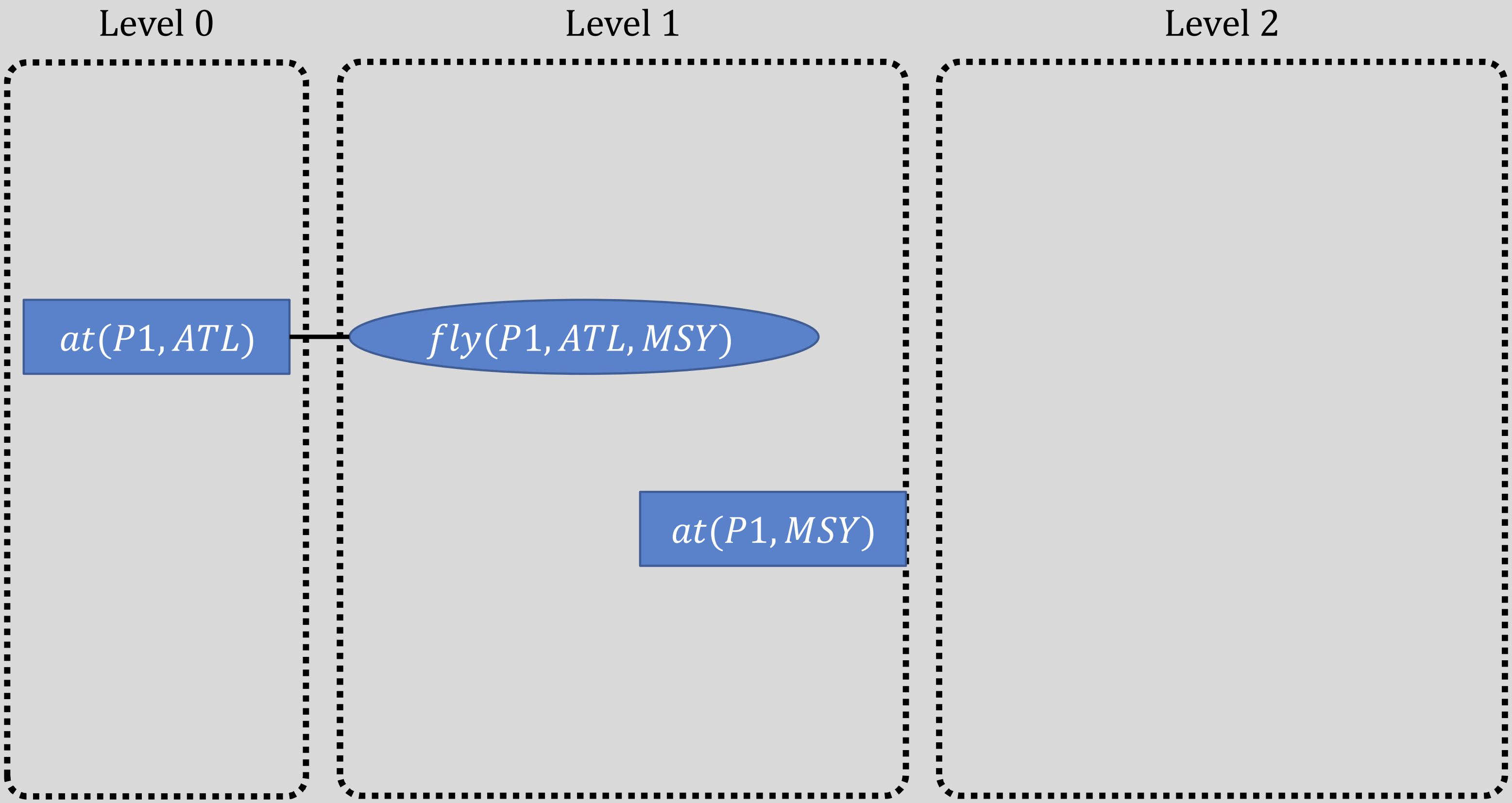
$fly(P1, ATL, MSY)$

$at(P1, MSY)$

Plan Graphs

And we connect the nodes via their preconditions and effects.

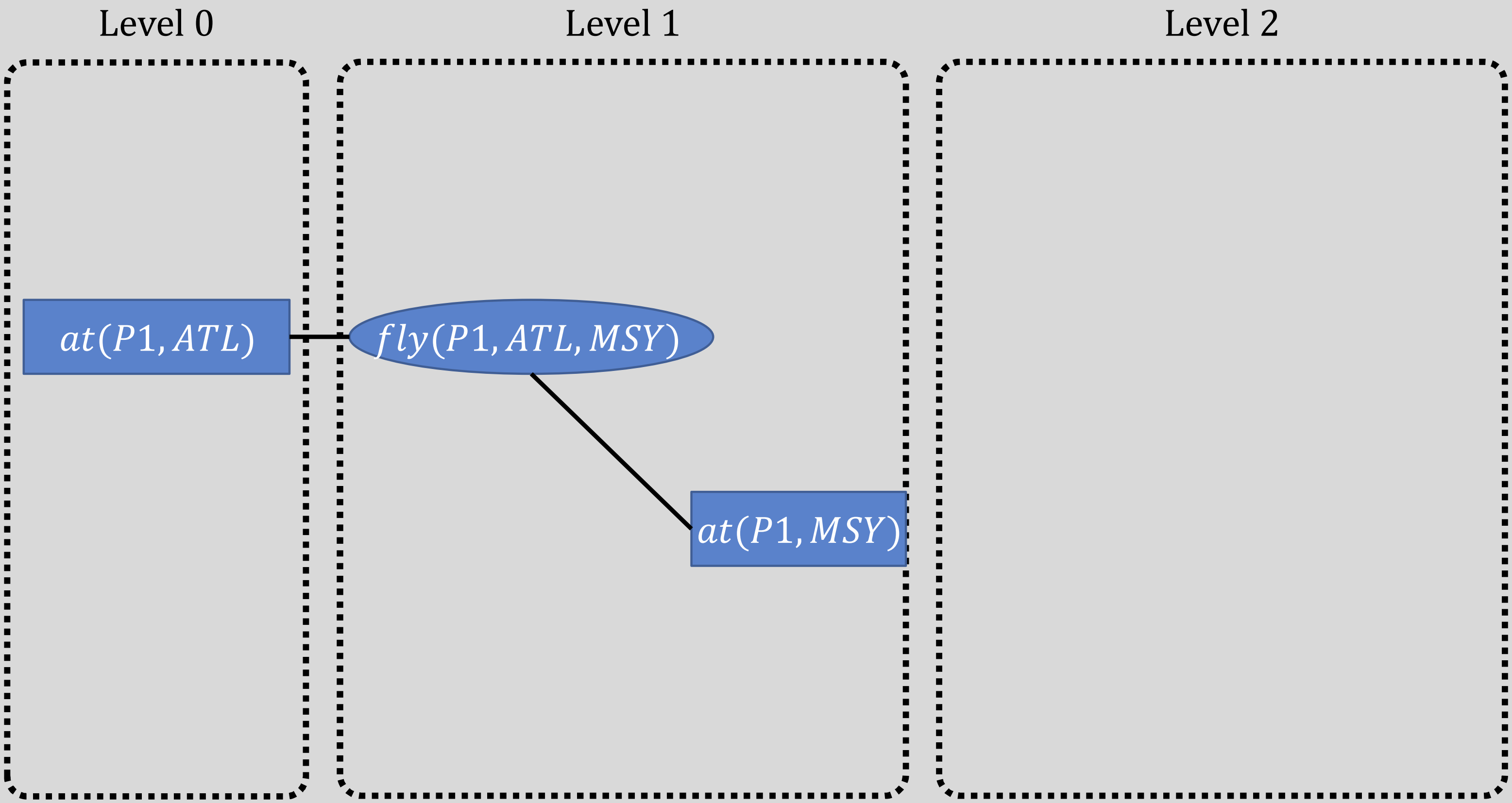
At(P1,ATL) is a precondition for fly(P1,ATL,MSY), so we connect them.



Plan Graphs

And *at(P1,MSY)* is an effect of *fly(P1,ATL,MSY)* so we connect them too.

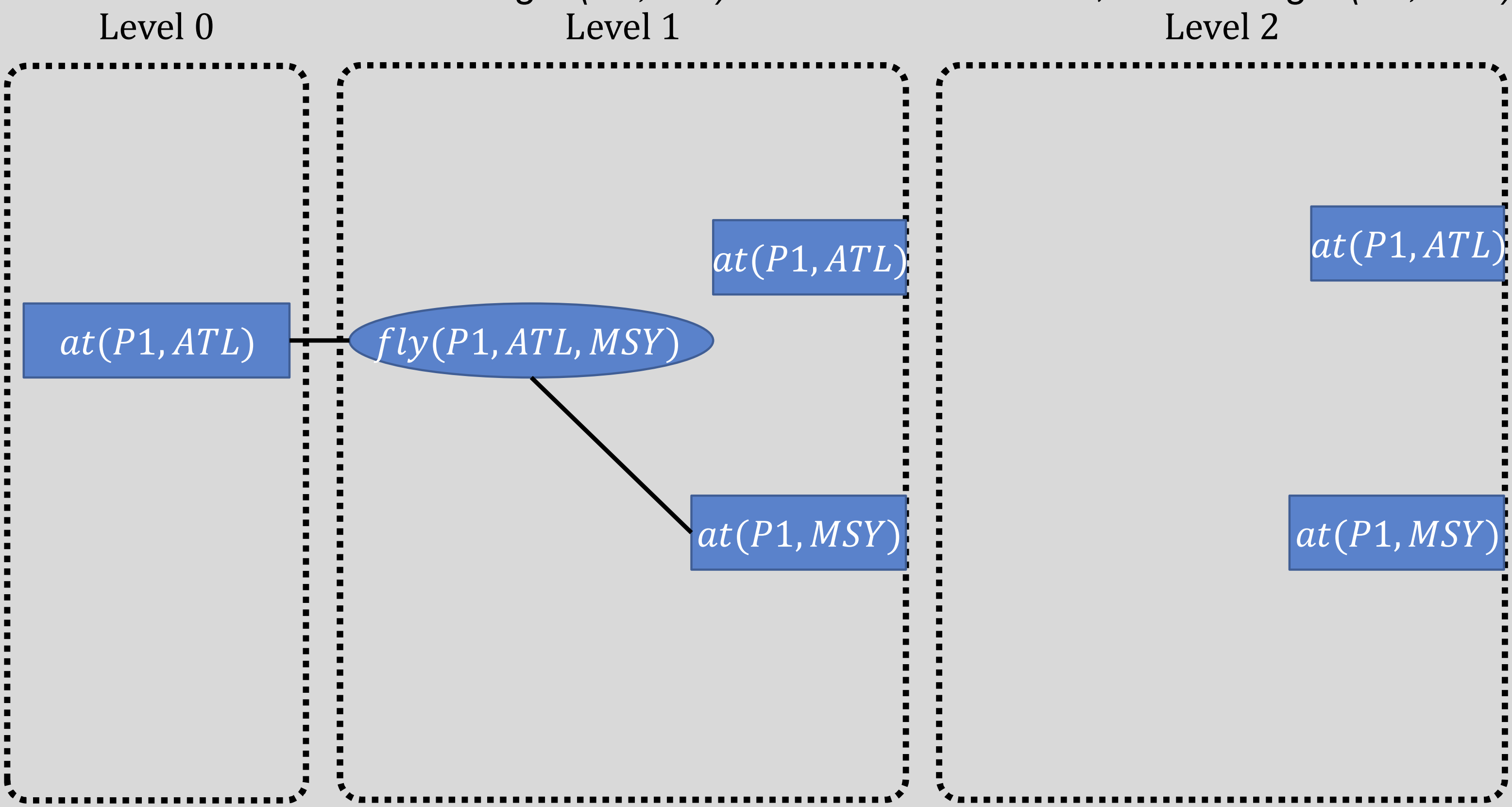
(Right? After flying from ATL to MSY, P1 is now at MSY).



Plan Graphs

And also, don’t forget, once a literal node shows up it will show up in every subsequent Level.

So if we wanted, we could do that for our two literal nodes right now, adding *at(P1,ATL)* from 0 t both 1 and 2, and adding *at(P1,MSY)* to 2 too.



Our First Plan Graph

- Let’s construct a full plan graph together to solve a planning problem!

- We’re going to introduce a new domain, to see if we can finally solve the age old question…

Is it possible to have one’s cake and eat it too?

The Cake Domain and Planning Problem

Initial state: $have(Cake)$,

Action: $eat(Cake)$

Precondition: $have(Cake)$

Effect: $eaten(Cake) \land \neg have(Cake)$

Action: $bake(Cake)$

Precondition: $\neg have(Cake)$

Effect: $have(Cake)$

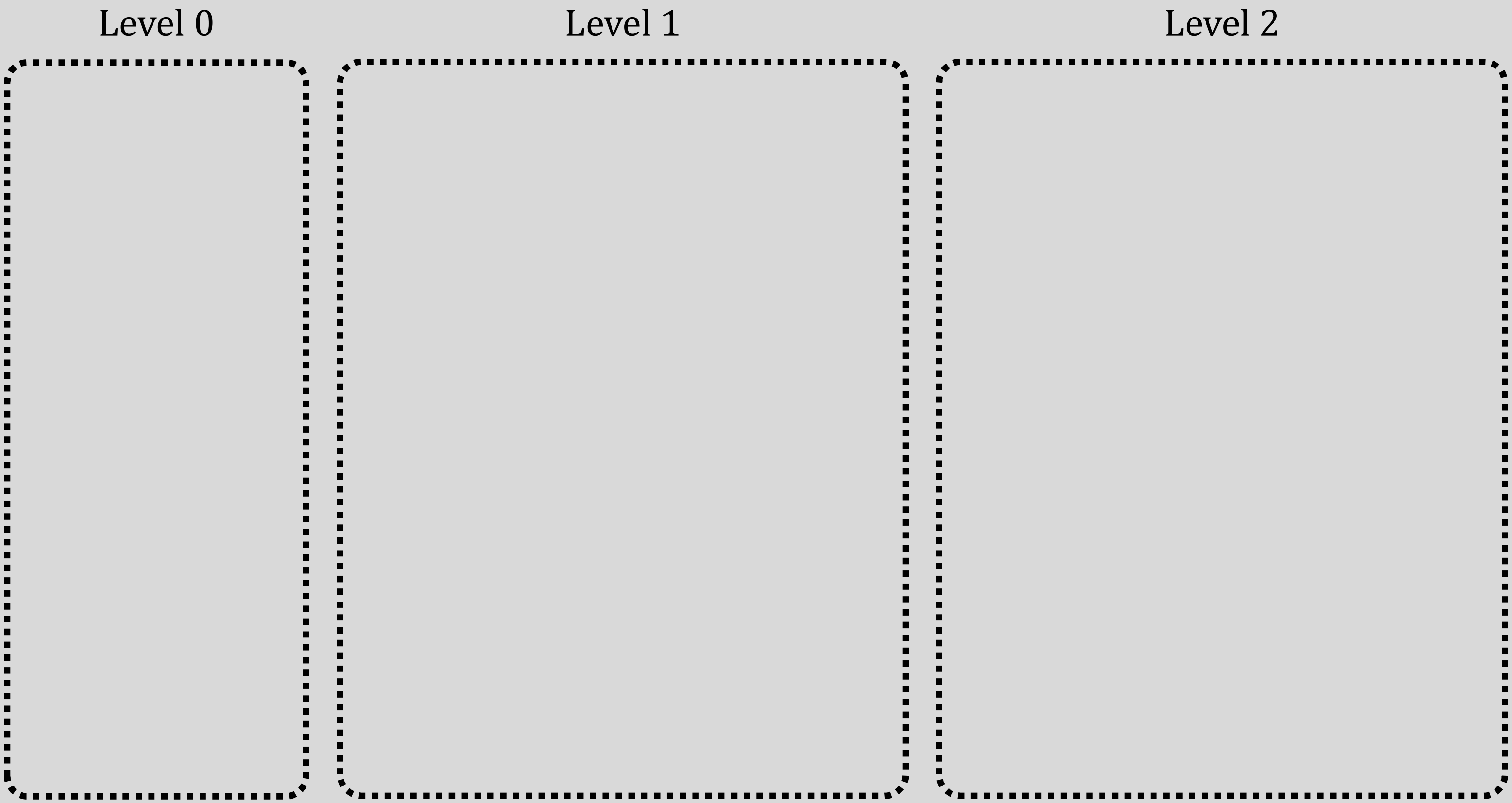
Goal: $have(Cake) \land eaten(Cake)$

Our First Plan Graph

There’s gonna be a lot here eventually, but we’ll take it slow!

A Plan Graph

We talked about how it's divided into levels, with Level 0 being the "Start State." We'll make space for several levels.



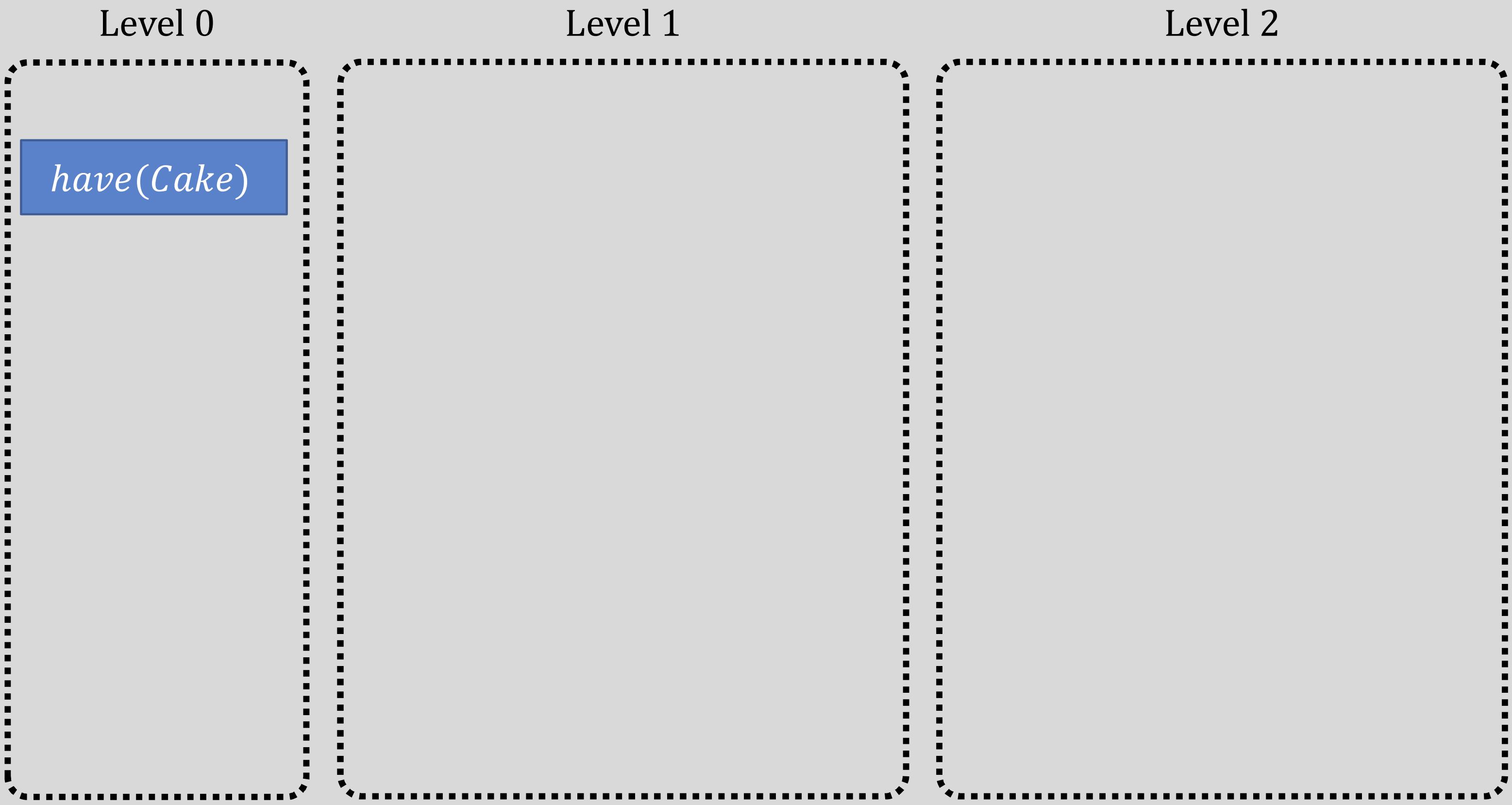
Level 0

Level 1

Level 2

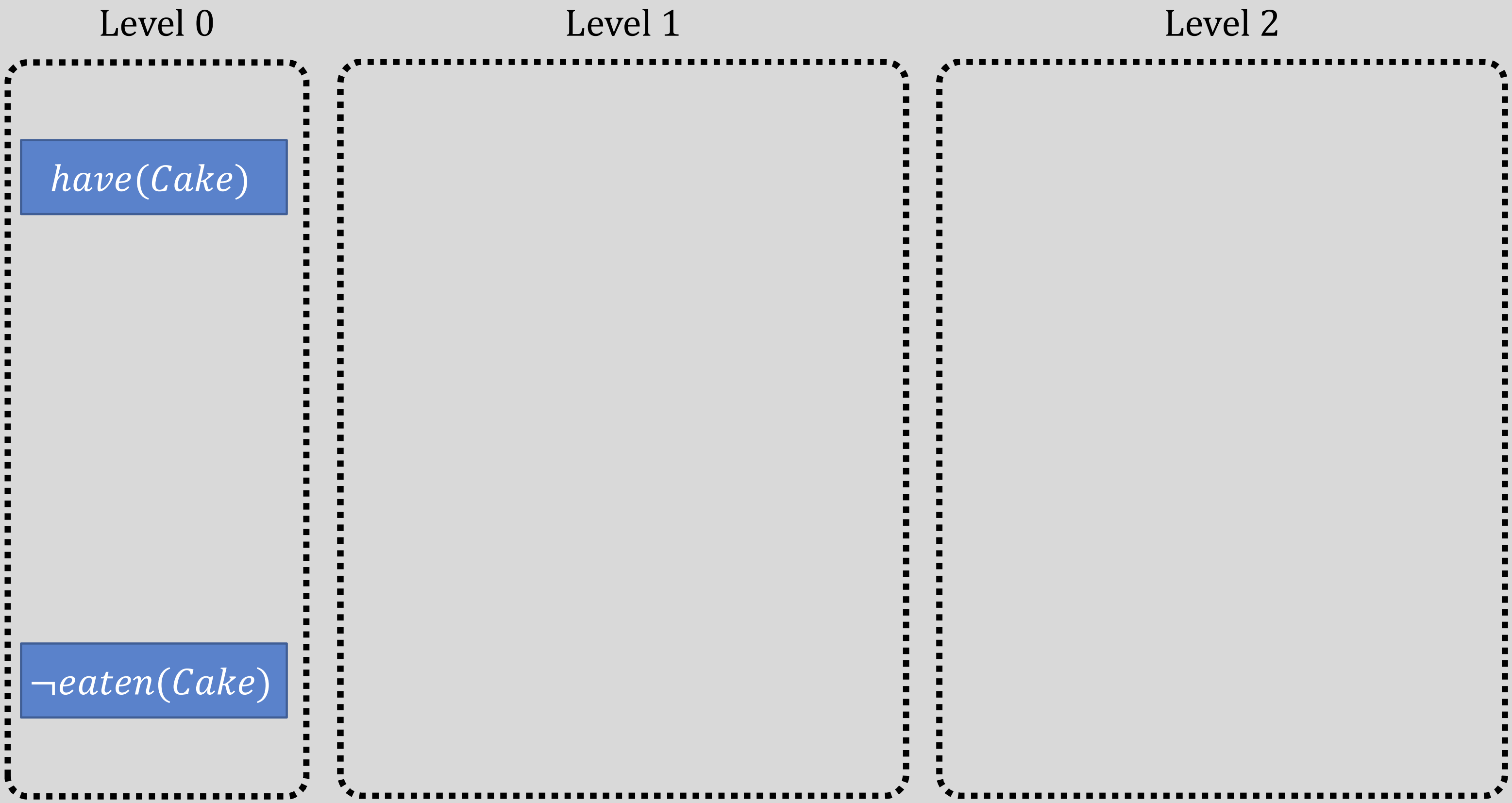
A Plan Graph

We'll begin with Level 0 – we recall it has NO step nodes, and its literal nodes represent anything true in our start state. Our start state is $have(Cake)$, so…



A Plan Graph

And again, in planning we use the “**closed world**” assumption. We didn’t specify *eaten(Cake)*, so we assume it must be false (i.e., $\neg eaten(Cake)$ is true).



Level 0

Level 1

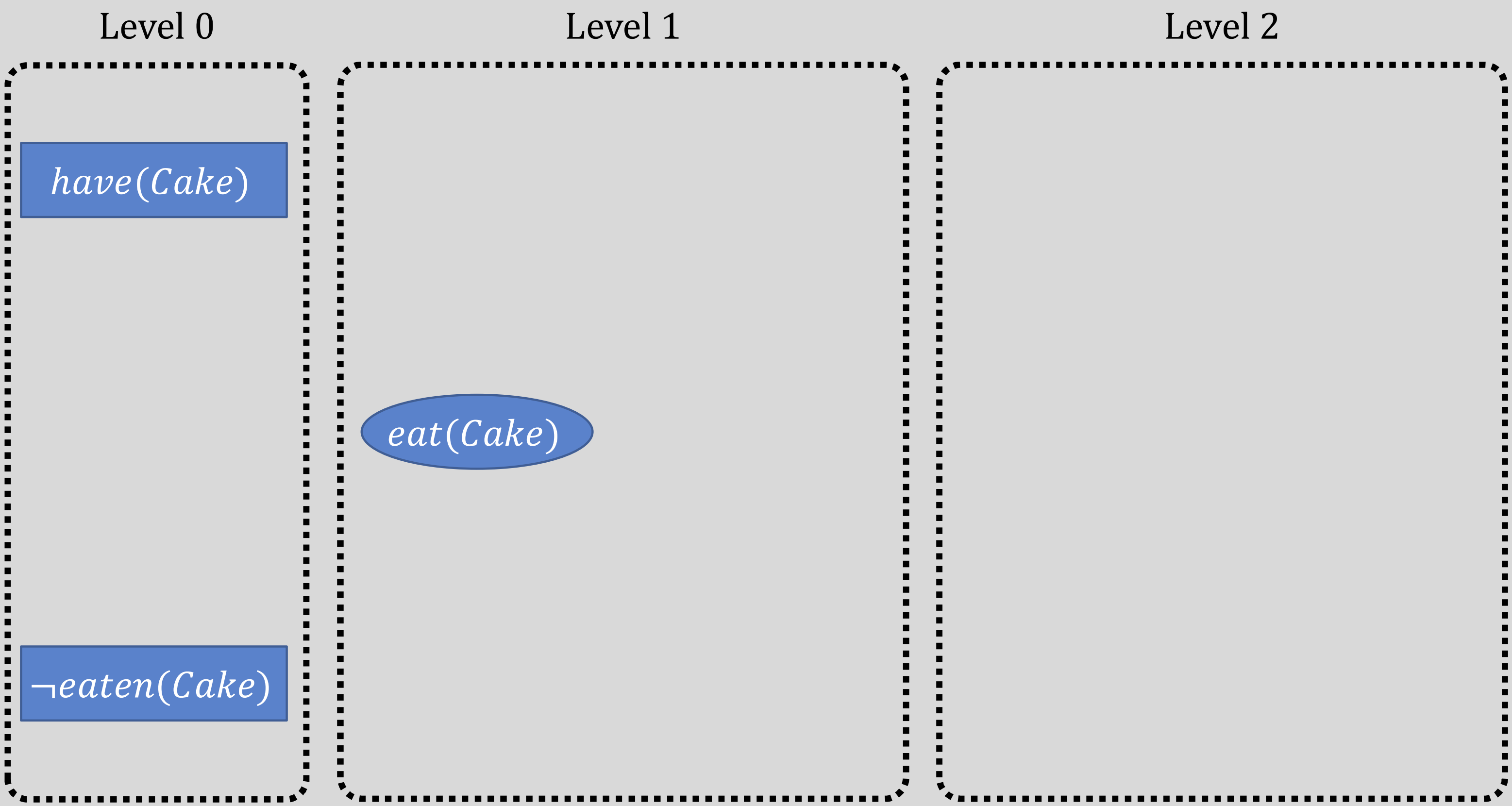
Level 2

$have(Cake)$

$\neg eaten(Cake)$

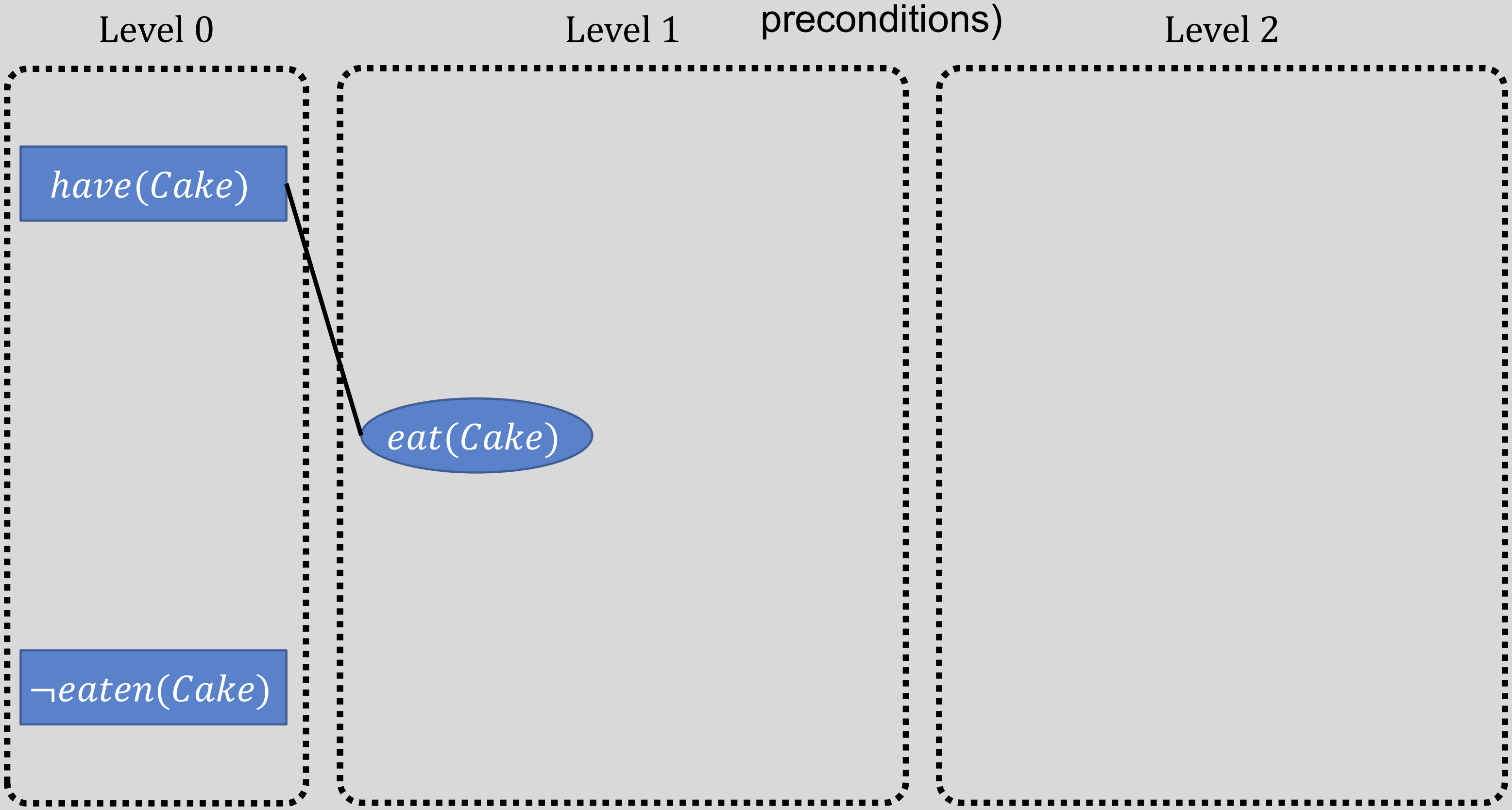
A Plan Graph

We move on to Level 1. Let’s add the step nodes first. We add any step node whose preconditions could be fulfilled via literals from the previous step.



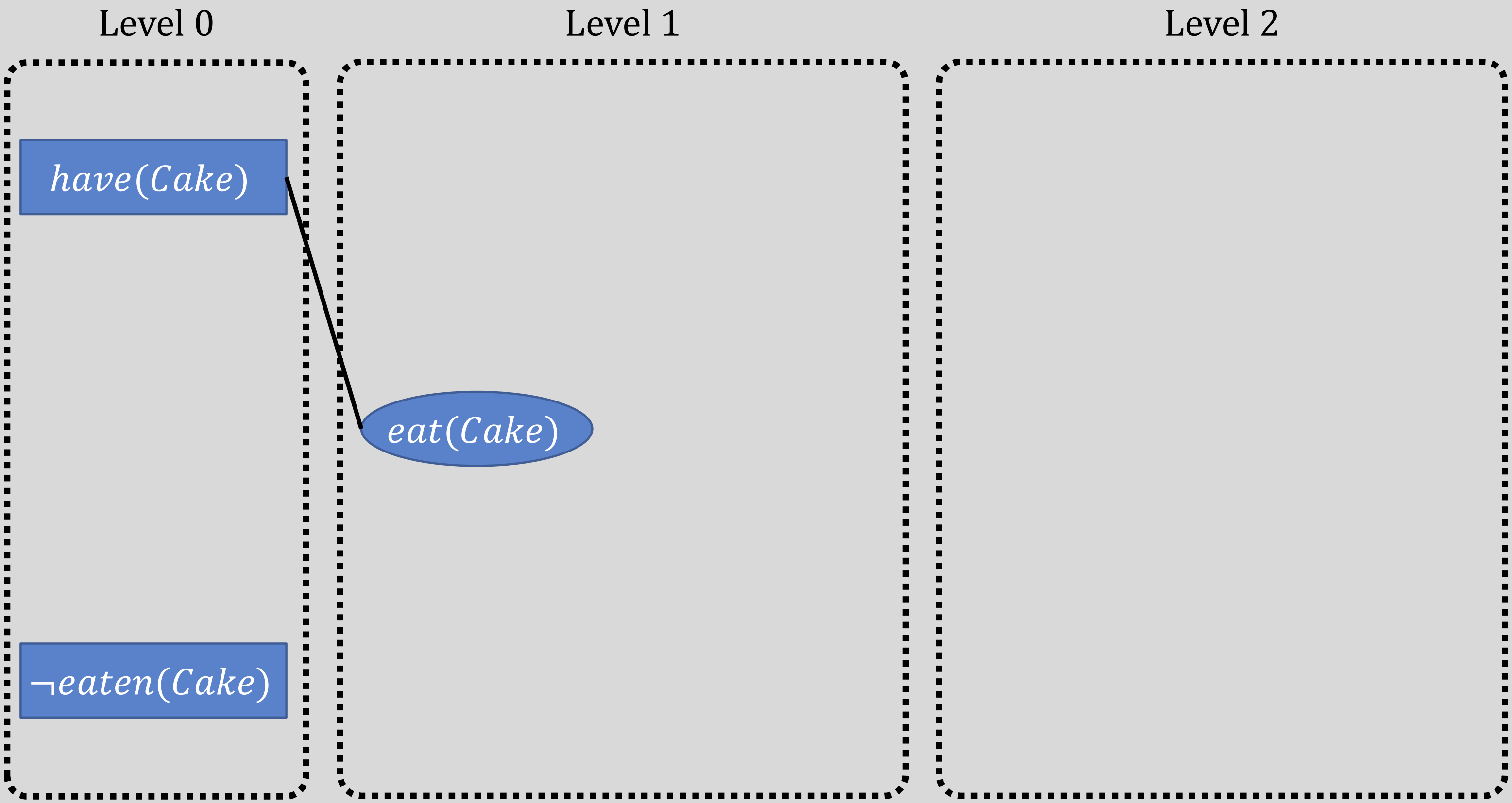
A Plan Graph

The precondition of *eat(Cake)* is that you *have(Cake). have(Cake)* does appear on the previous level, so we add *eat(Cake)* as a step node (and connect it to its preconditions)



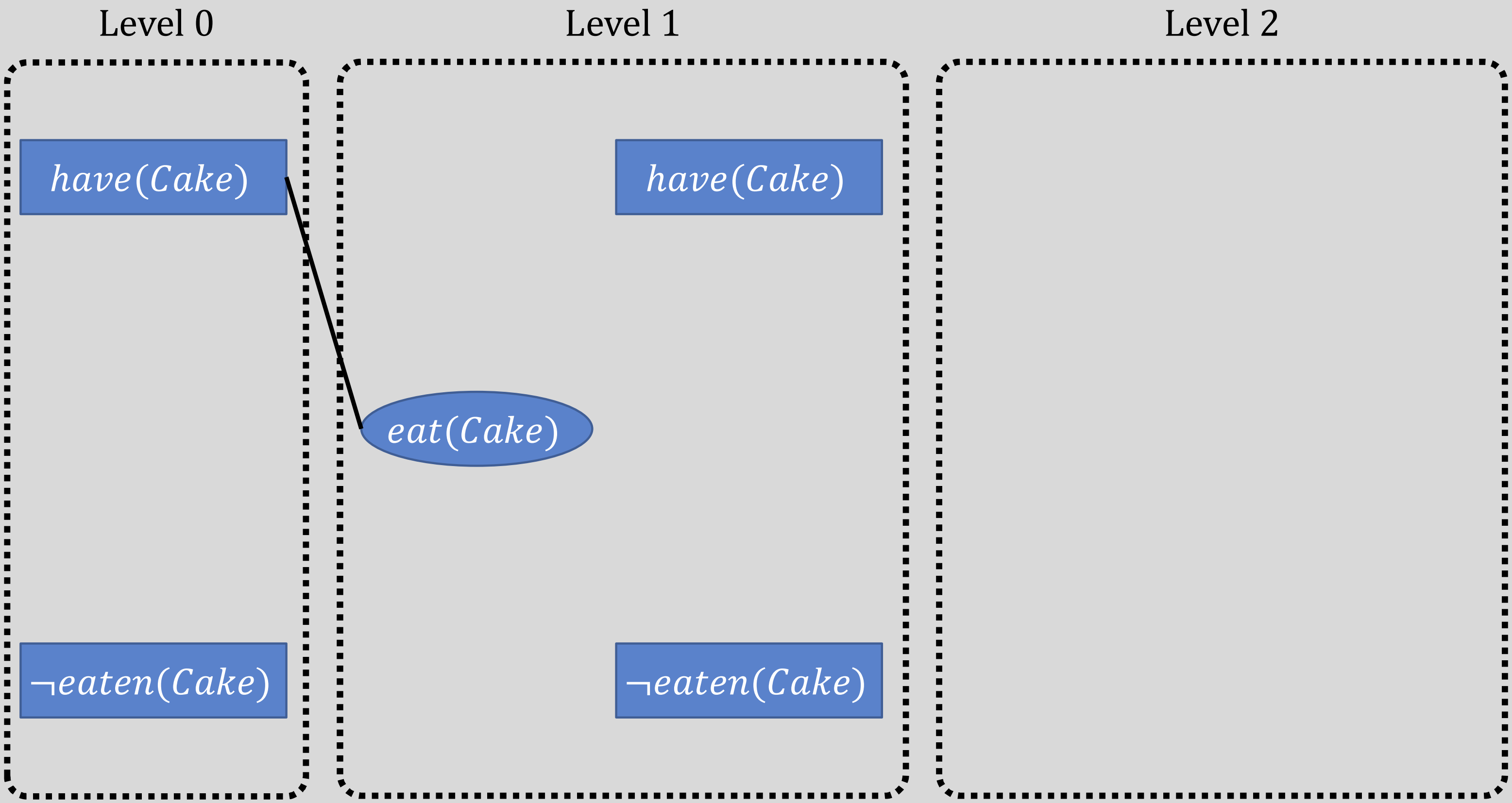
A Plan Graph

The precondition of our other action, *bake(Cake),* is that we explicitly $\neg have(Cake)$ But that doesn't appear in Level 0, so we can't add a step node for *bake(Cake)*



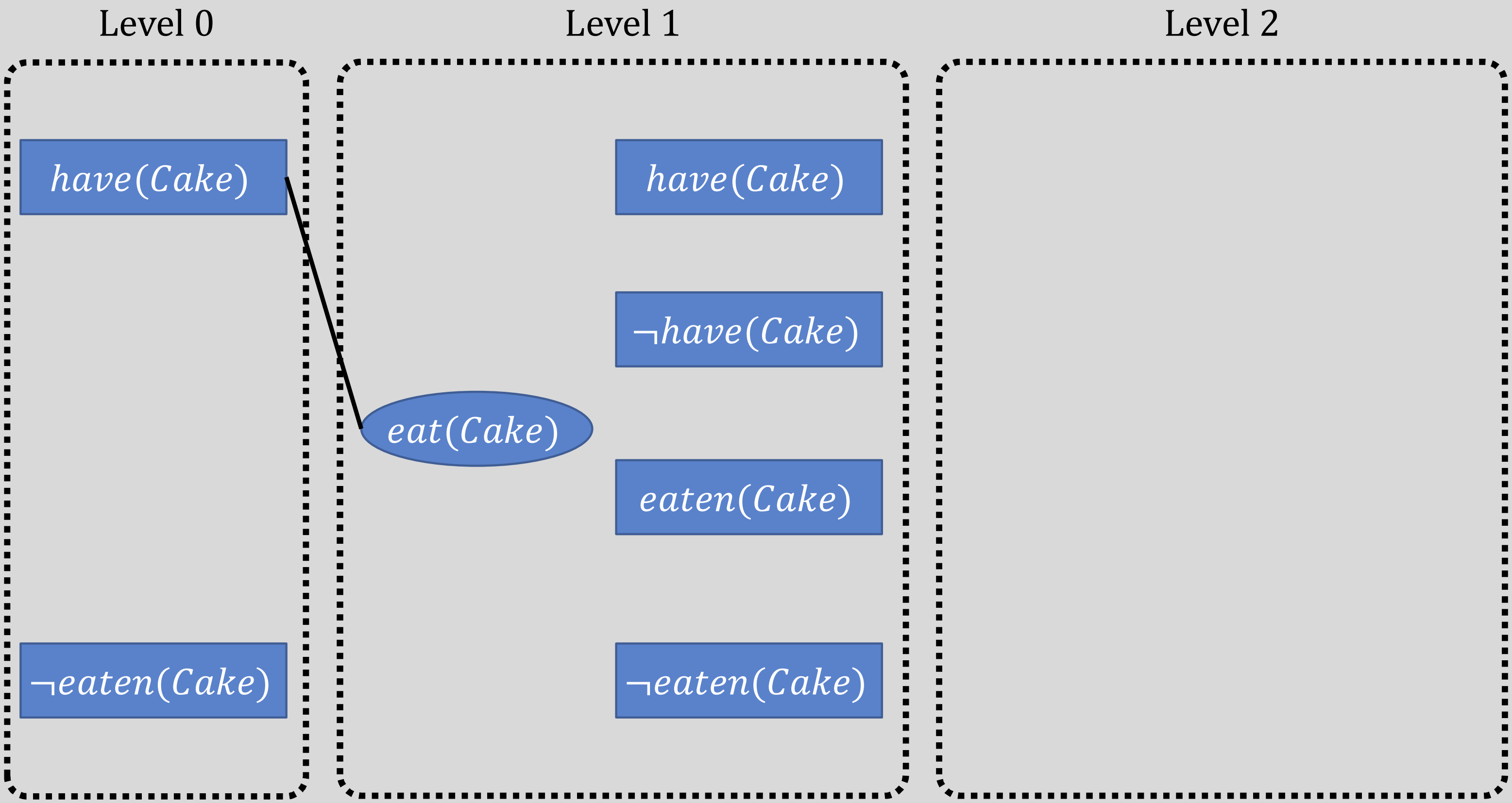
A Plan Graph

OK, let's add the literal nodes for Level 1. You automatically add nodes for any literals that appeared in the previous level, so that gets us two right away…



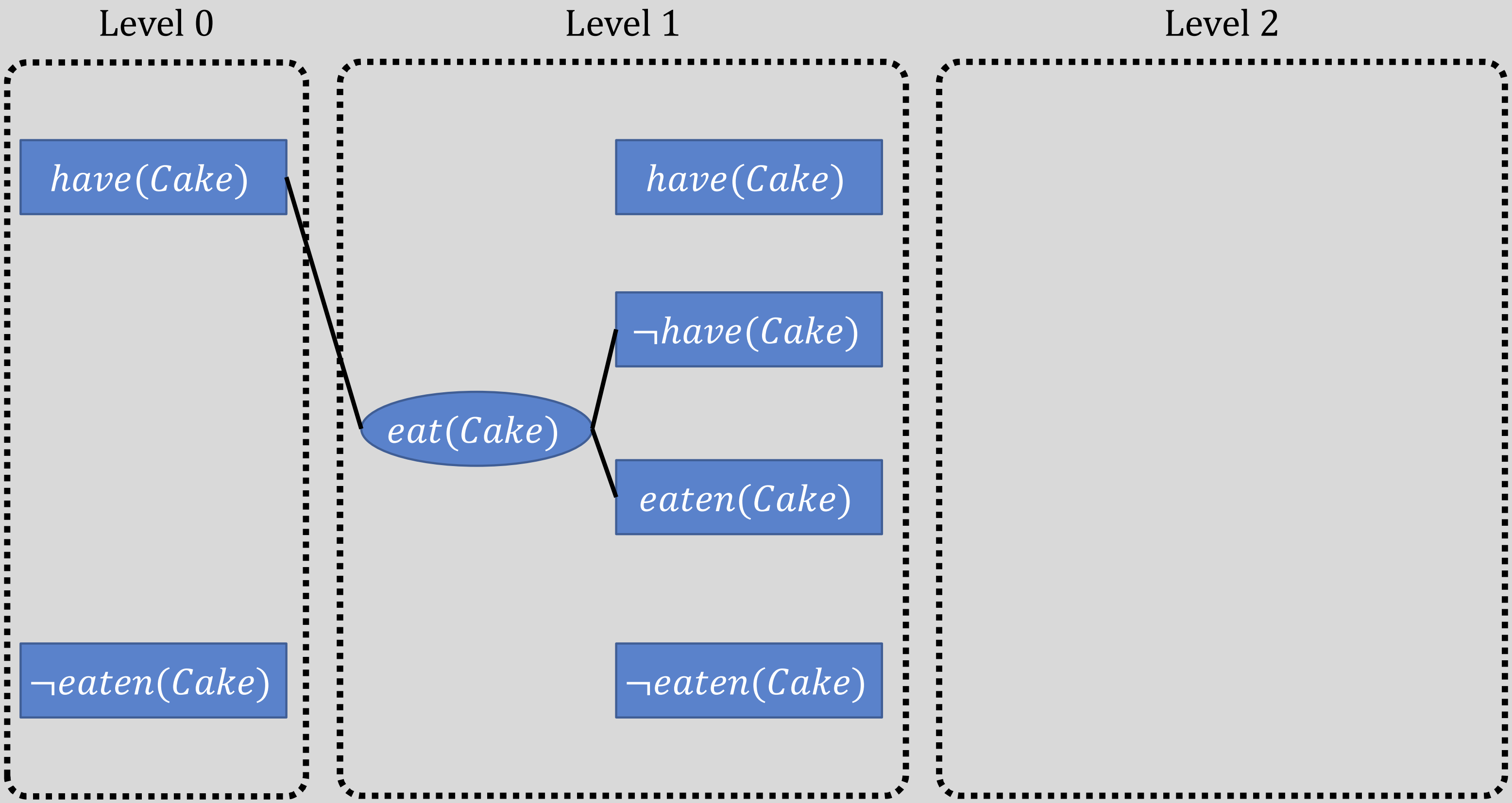
A Plan Graph

But we also add literal nodes representing the effects of any step node that exists at the same level. The effect of *eat(Cake)* is $eaten(Cake) \wedge \neg have(Cake)$, so we add them too!



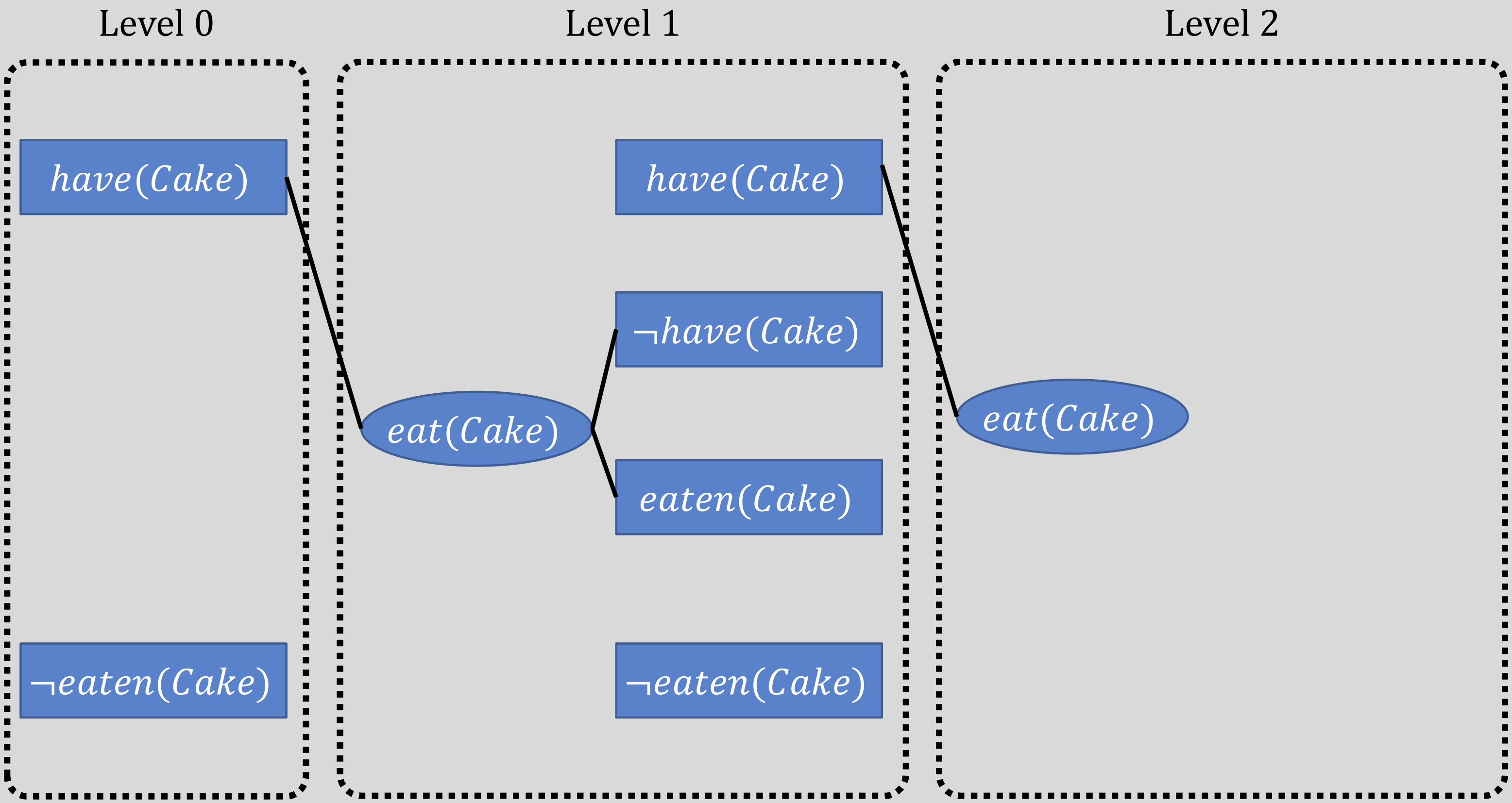
And we connect these literals to the steps that can make them possible. Here it’s just the one.

A Plan Graph



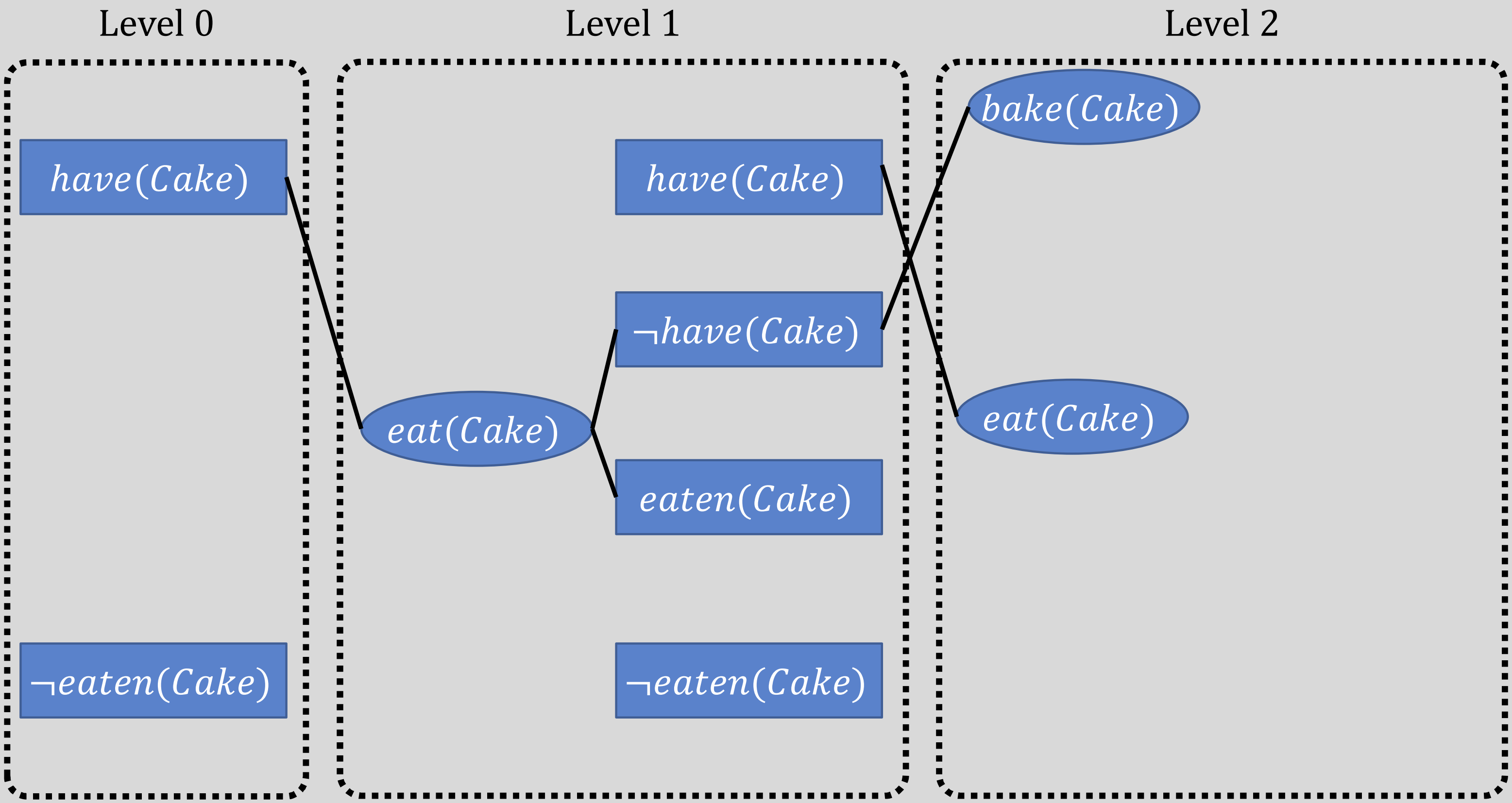
A Plan Graph

OK, exciting! Hopefully by now we see the pattern! We add the step nodes for Level 2. *have(Cake)* exists in the previous level, which is the precondition for *eat(Cake)*, so it's showing up again for sure.



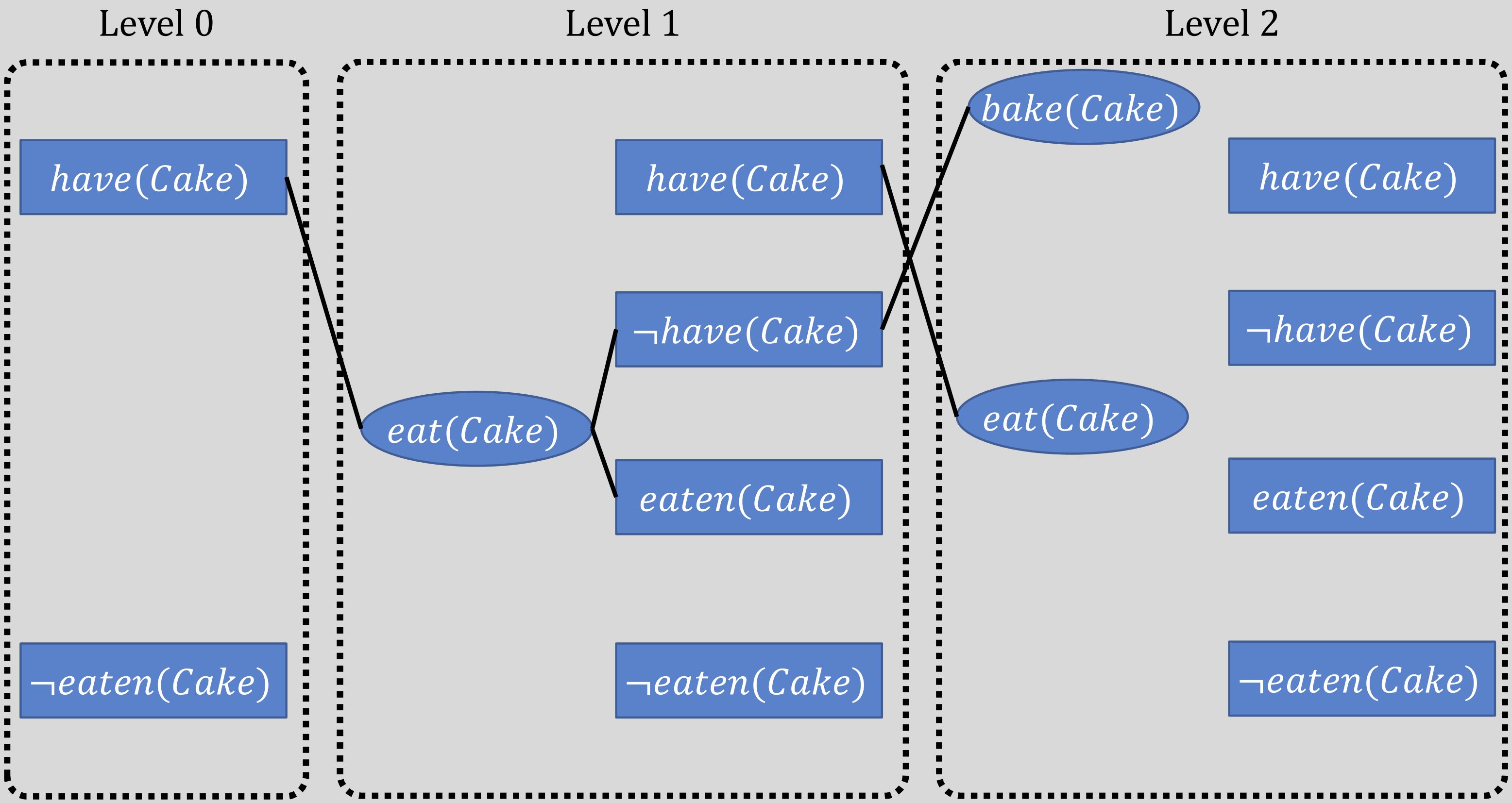
A Plan Graph

But now the preconditions for *bake(Cake)* *also* are in the previous level. $\neg have(Cake)$ might be true in Level 1, so level 2 has a second step node!



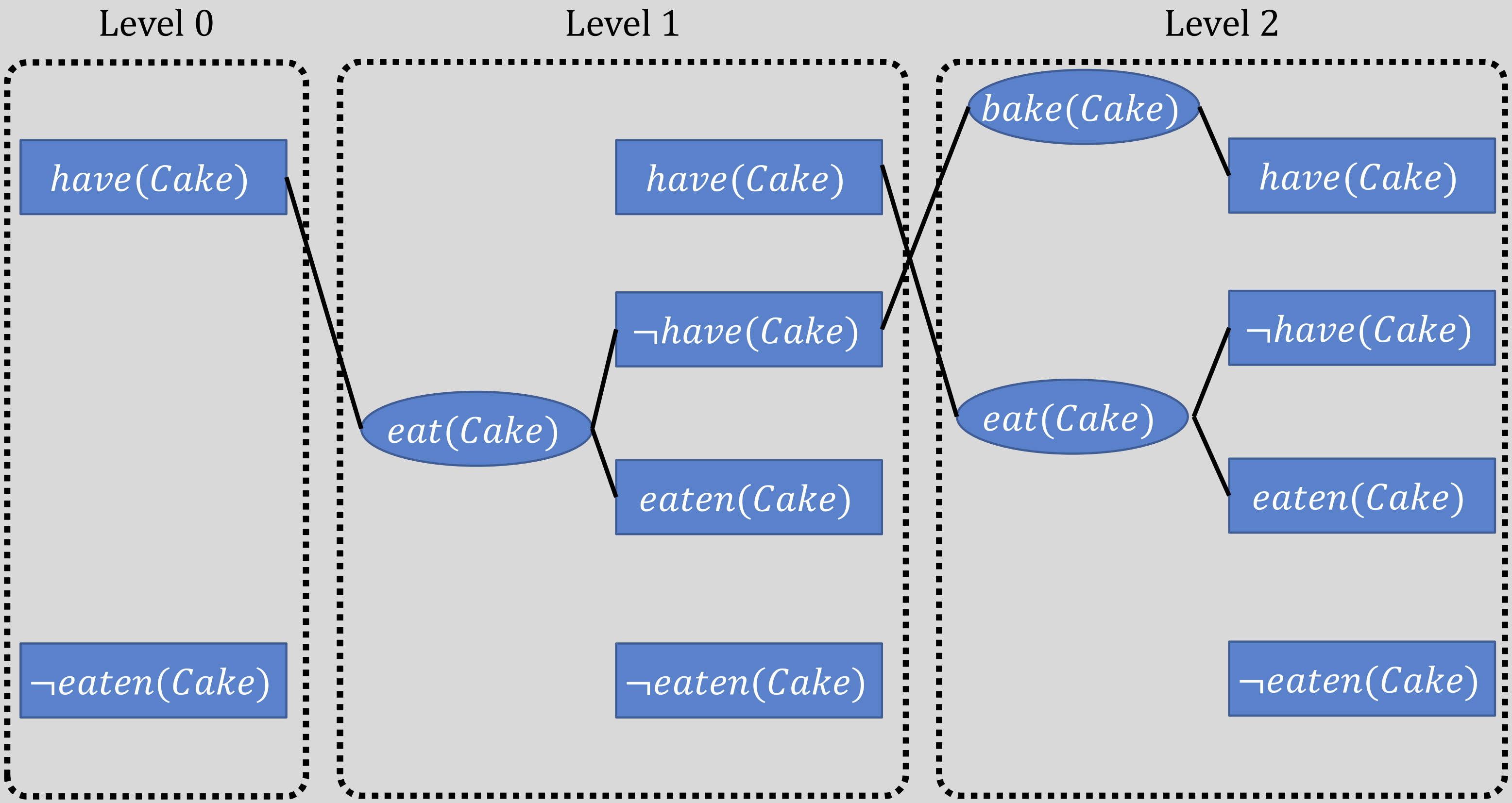
Finally, for the literal nodes of Level 2 – we know we copy all literals from the previous level, so, ah, we'll do that!

A Plan Graph



A Plan Graph

And even though we would have copied them regardless, let's still connect them to actions that have them as effects.



Features of Plan Graphs

- There’s still a lot more to add to the graph, but let’s take a breather, and observe…
- The literals and steps at each level are monotonically increasing.
    - That is: the number always “stays the same or gets bigger” from level to level. It never goes down!
- It is possible that a literal and its negation (e.g., $\neg have(Cake)$ and $have(Cake)$) appear at the same level.
    - This is the abstraction at work!
    - Don’t think of the literals in a state as “the current state”, but rather “the components of all *possible* states after *n* actions.” Where ‘*n*’ is the current level.
- A plan graph can be built in polynomial time!

Plan Graph Heuristics.

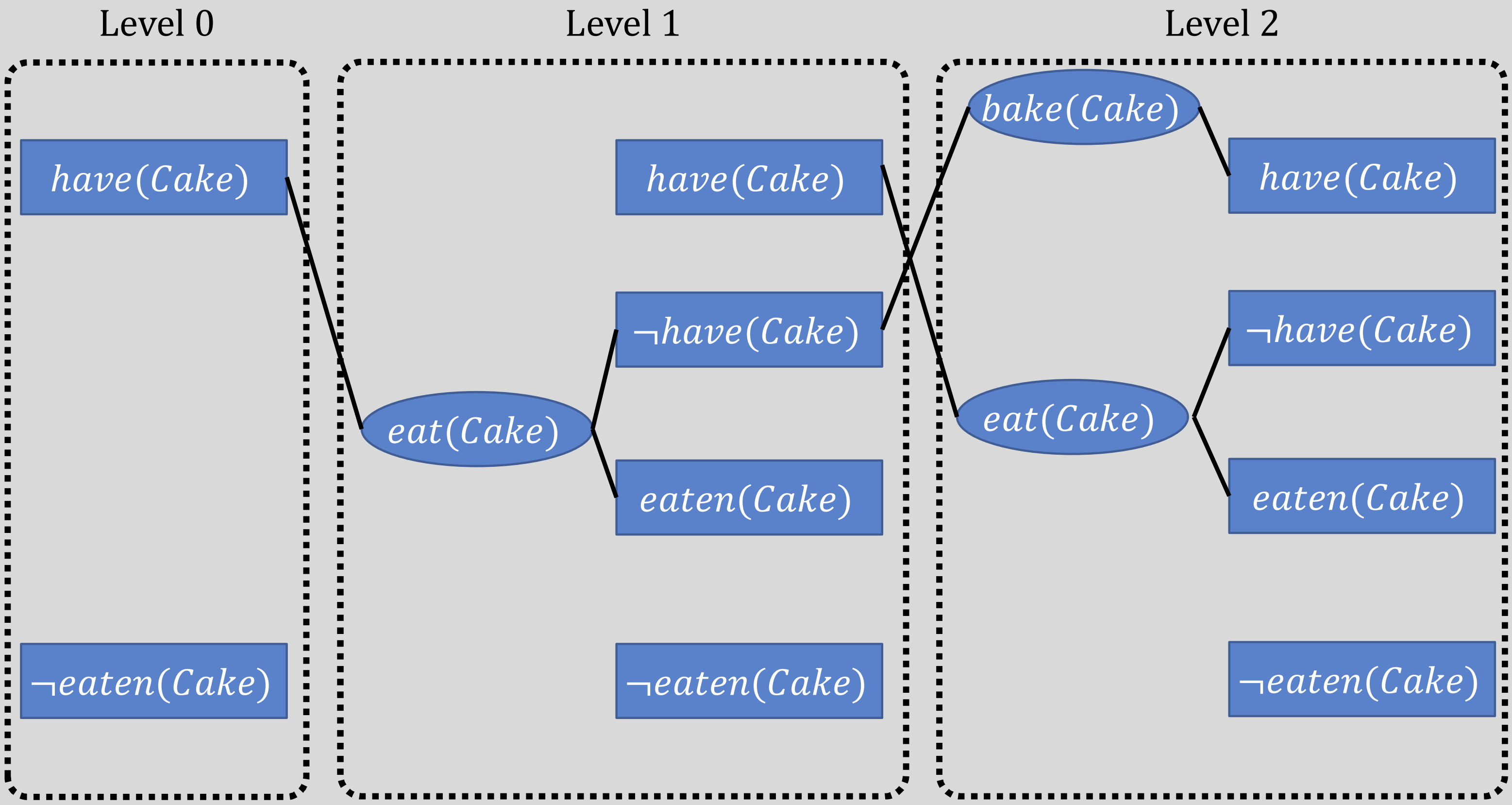
- Remember: Heuristics in AI are about estimating how close we are to our goal.
- With planning, it would be really helpful if we could estimate how far away we are from being able to perform a certain action, or producing a certain literal.
- Well… that's kind of what we just created!

Plan Graph Heuristics.

- As it turns out…
- The level at which a literal *first appears* is a good estimate of how many steps need to be taken before the literal can be achieved.
- And likewise, the level at which a step first appears is a good estimate of how many steps need to be taken before that step can be taken.
- Are these estimates admissible? Yes! Will certainly never overestimate!
- Will they ever underestimate? Yes. But that’s OK!

OK. Let’s dive back in. This is where we left off. And let’s ask ourselves some questions…

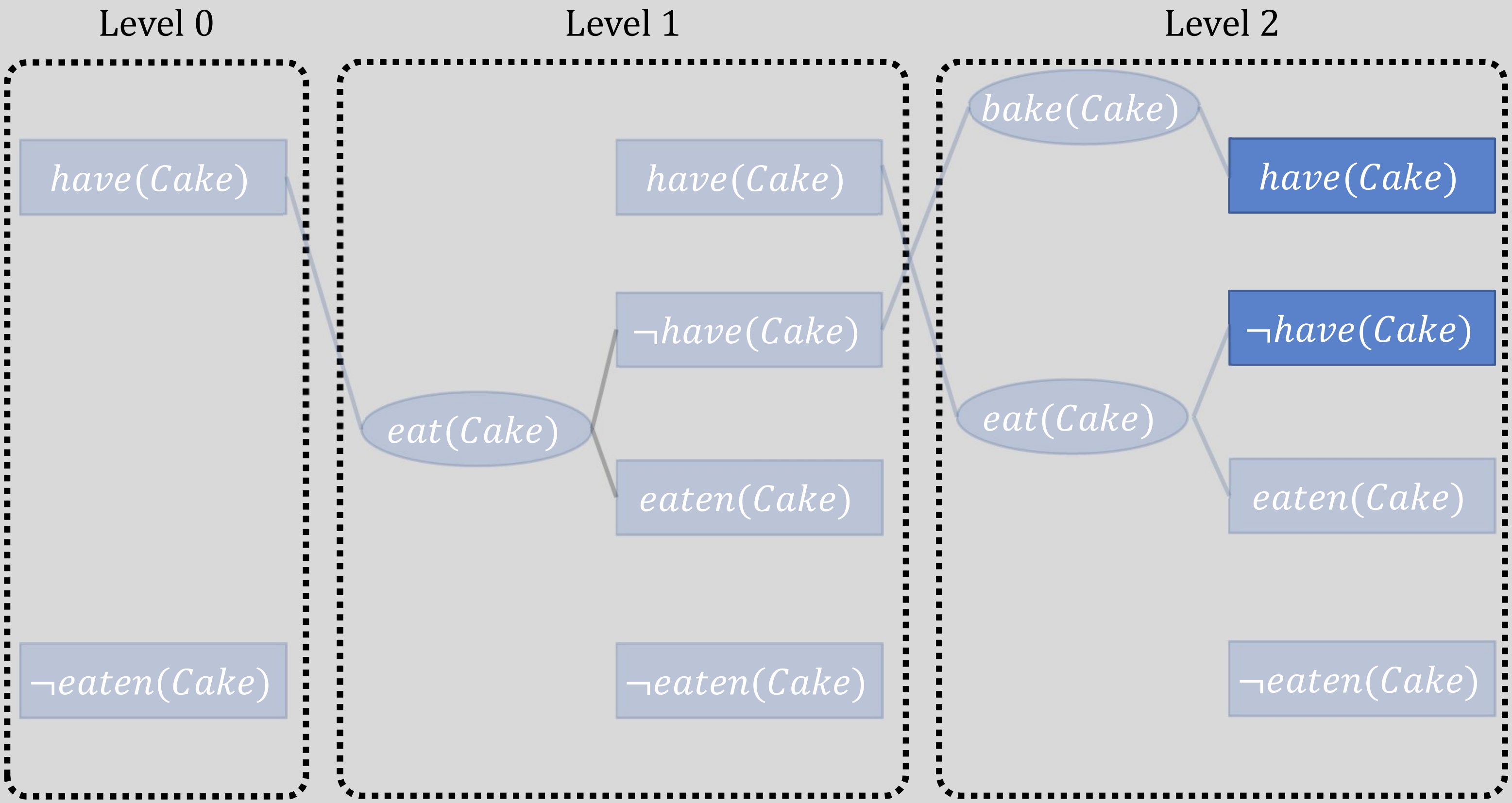
A Plan Graph



A Plan Graph

Can $have(Cake)$ and $\neg have(Cake)$ ever be true in the same state, i.e., at the same time?

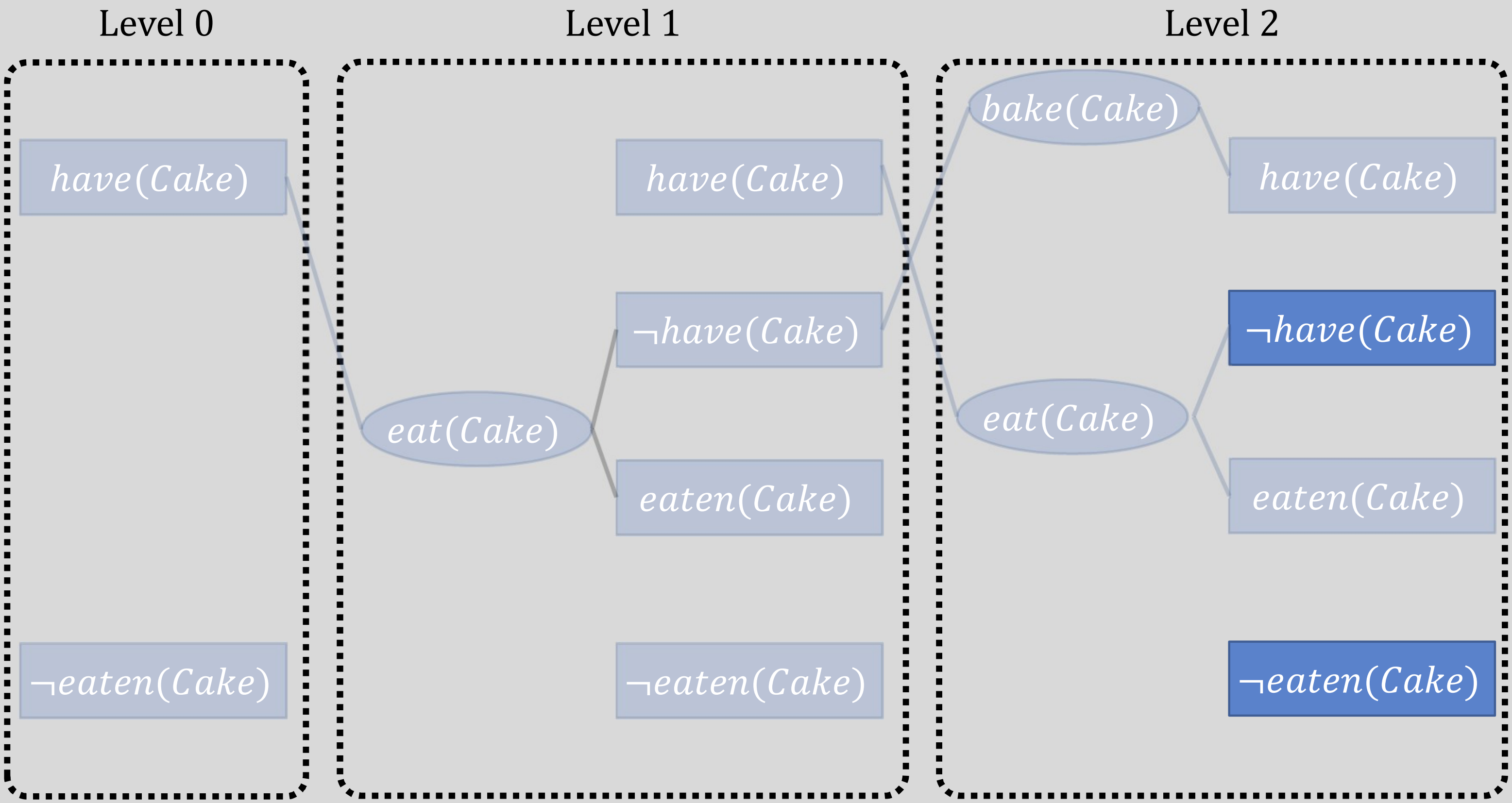
Nope!



A Plan Graph

Can $\neg have(Cake)$ and $\neg eaten\ (Cake)$ ever be true in the same state, i.e., at the same time?

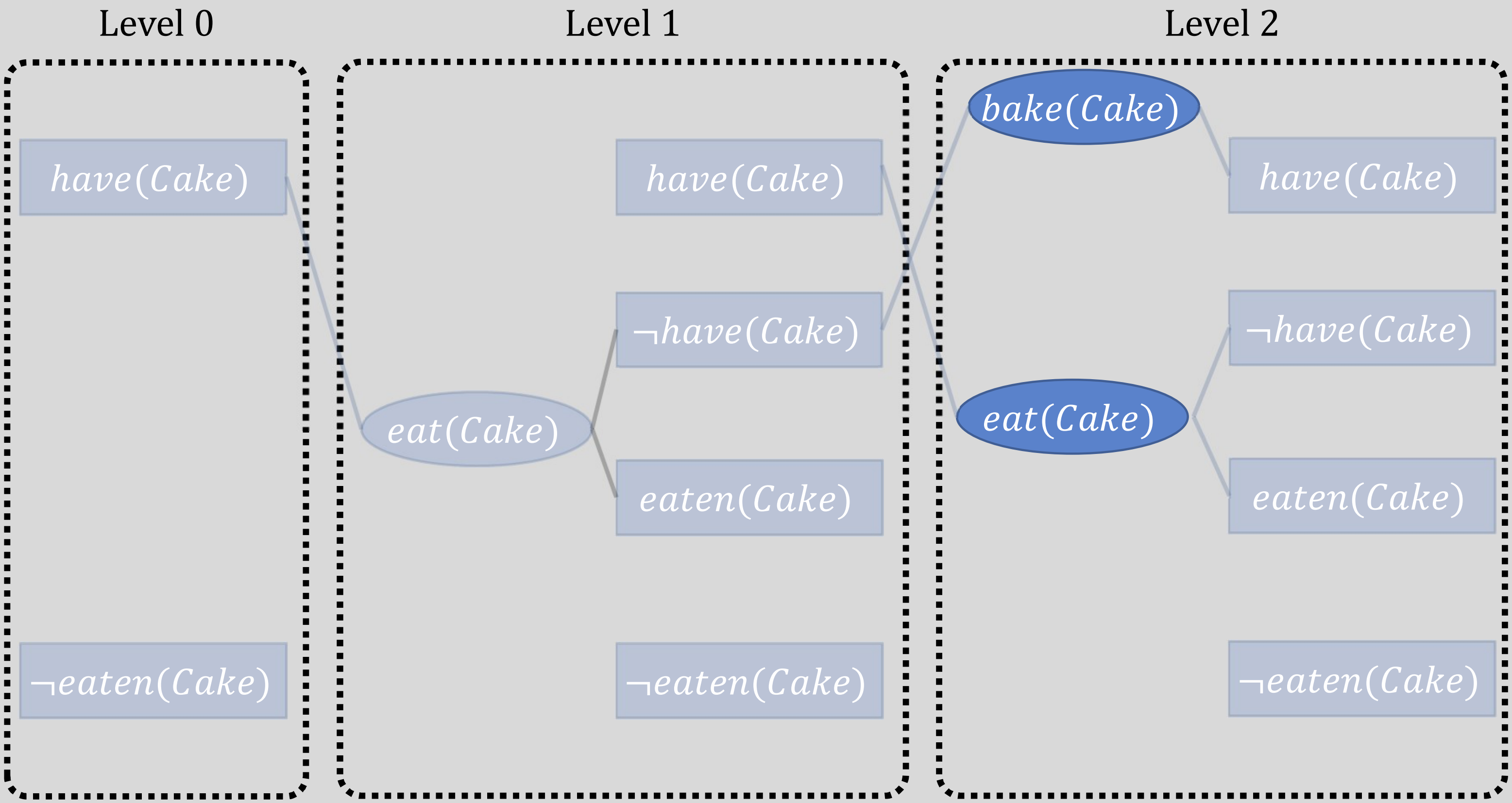
Also no! A little harder to see why, maybe? Any guesses?



(you start off having it, so the only way you could possible *not* have it is if you've eaten it)

A Plan Graph

Can the two actions $bake(Cake)$ and $eat(Cake)$ ever be taken in **parallel** (i.e., at the same time), without interfering with each other?



No again! You can only bake if you have no cake, you can only eat if you *do* have cake!

Increasing Accuracy

- The previous slides highlight truths that we likely implicitly already knew:

- There are some literals that cannot co-occur.
- There are some steps that cannot co-occur.

- So let’s improve our plan graph! Let’s mark which literals and states are mutually exclusive with each other! These are done with **mutex** links.

Persistence Steps

- Remember when we introduced the notion of time and fluents, and the Wumpus World arrow disappeared because we didn't capture what "stayed the same"?
- That was bad! We want to be able to capture "what stays the same"!
- We use **persistence steps** to handle that notion here.

- A persistence step is a "dummy" step that has a single literal, *L*, as it's precondition, and also has that same literal *L* as its effect.
- Every literal that appears at level *n* will have a persistence step connecting it to its equivalent at level *n + 1*.

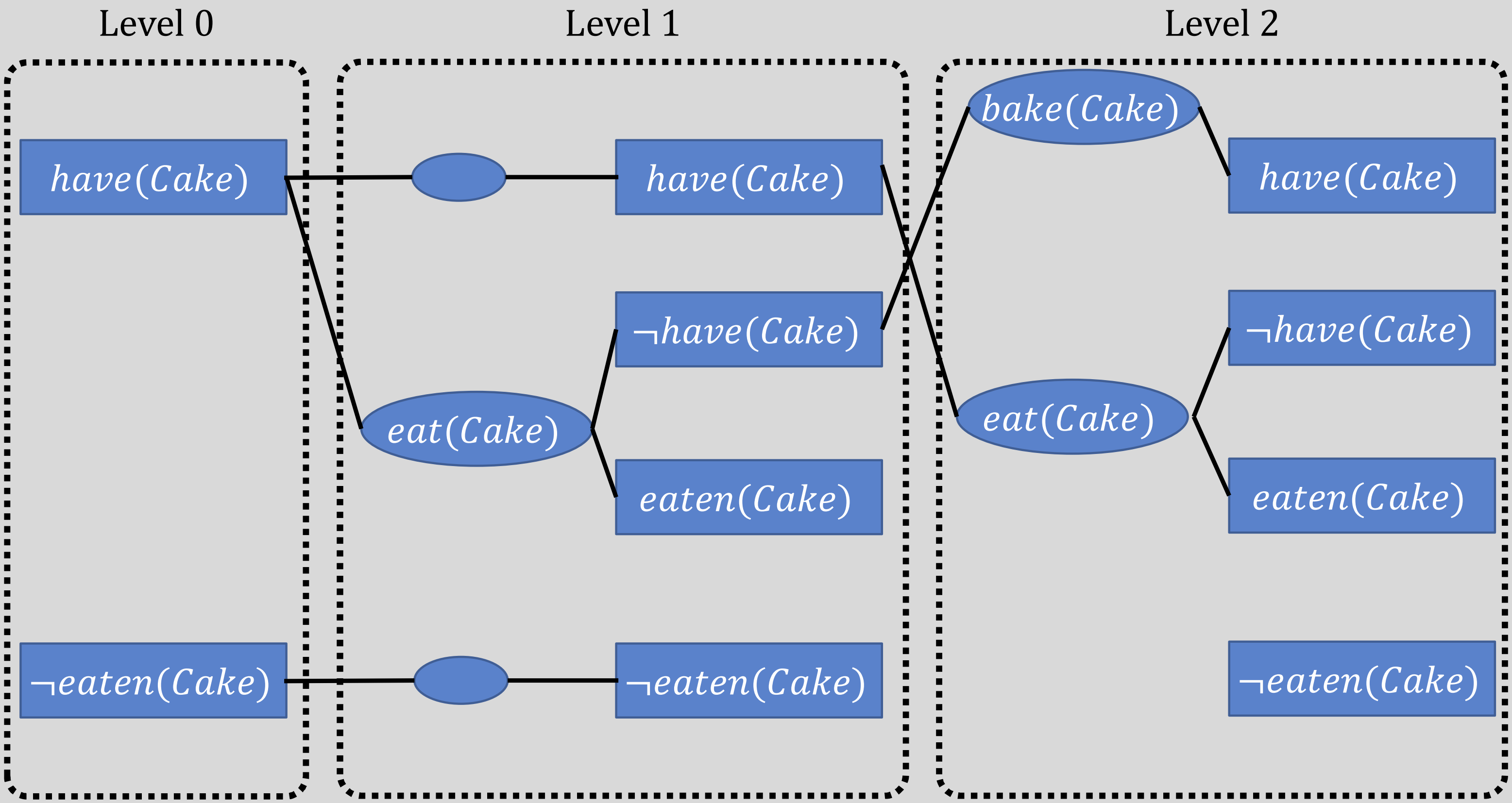
Persistence Steps

- Persistence Steps are how we represent frame axioms in plan graphs (i.e., capture the things that do not change).

- You can also think of them as **no-ops**
  - i.e., “if instead of taking an action you did nothing, then the stuff that was true in the previous time step will remain true in the future.”

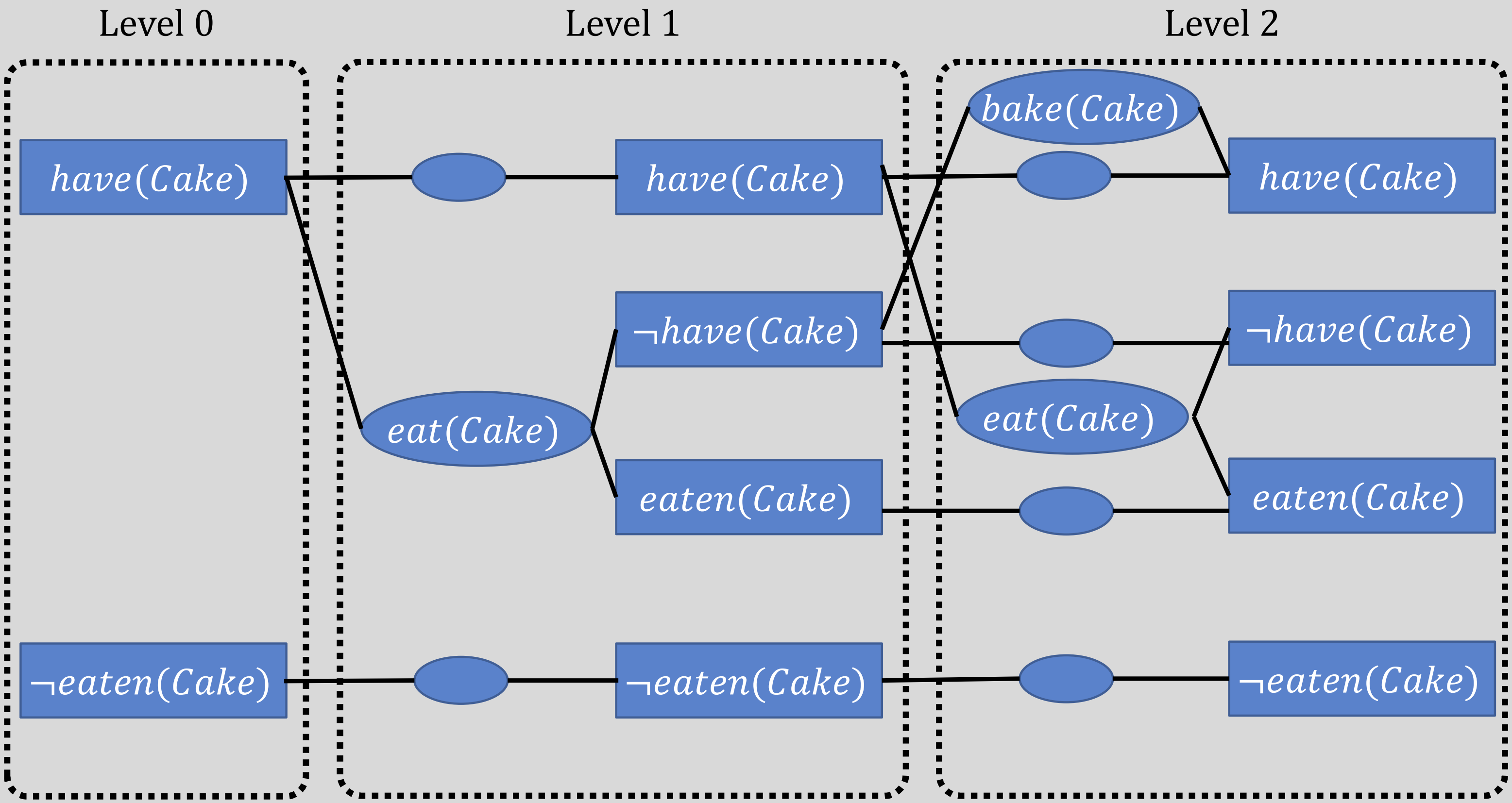
A Plan Graph

So, cool, let’s add those persistence steps!
Here we have the “level 1” persistence steps.
“if we have cake and don’t do anything, we still have cake.”
“If we haven’t eaten cake and don’t’ do anything, we still haven’t eaten cake”



And we'll add them to level 2, too!

A Plan Graph



Adding Mutex Links

- OK, we’re about to add a whole lot of additional lines to the graph, capturing those mutually exclusive relations.

- Before we add them, let’s be a little more formal about…
  - When, exactly, are two steps are mutually exclusive
  - And when, exactly, are two literals are mutually exclusive.

Adding Mutex Links: Step Mutex Relations

- Two steps cannot be taken in parallel (i.e., they are mutually exclusive) when one of the following is true:
  - They have **inconsistent effects**: The effect of one action negates the effect of another.
  - There is **interference**: One of the effects of one action negates a precondition of the other.
  - They have **competing needs**: The two actions have mutually exclusive preconditions.

inconsistent effects: The effect of one action negates the effect of another.
Interference: One of the effects of one action negates a precondition of the other.
Competing needs: The two actions have mutually exclusive preconditions.

Adding Mutex Links: Step Mutex Relations

- **Inconsistent effect** example: *eat(Cake)* and *have(Cake)*
  - **Why***: eat(Cake)* has the effect ¬*have(Cake)* but the persistence step of *have(Cake)* has the effect *have(Cake).* So eat(Cake) and that persistence step are mutex.
- **Interference** example *eat(Cake)* and *have(Cake)* :
  - **Why:** *eat(Cake*) and the persistence step of *have(Cake)* *also* interfere. The effect of *eat(Cake)* is ¬*have(Cake)* but the precondition of the dummy step *have(Cake)* is *have(Cake)*
- **Competing Needs** example *bake(Cake)* and *eat(Cake)* :
  - **Why***: bake(Cake)* and *eat(Cake)* are mutex as they compete on the value of the *have(Cake)* precondition. *bake(Cake)* needs ¬*have(Cake),* but *eat(Cake)* needs *have(Cake).*

Adding Mutex Links: Literal Mutex Relations

- On the other hand, two literals are mutex (i.e., they can’t be true in the same state), when one of the following is true:
  - They are **opposites**: one literal is the negation of the other.
  - They have **inconsistent support**: Every possible pair of actions that could achieve the two literals are mutually exclusive.

Opposites: one literal is the negation of the other.
Inconsistent support: Every possible pair of actions that could achieve the two literals are mutually exclusive.

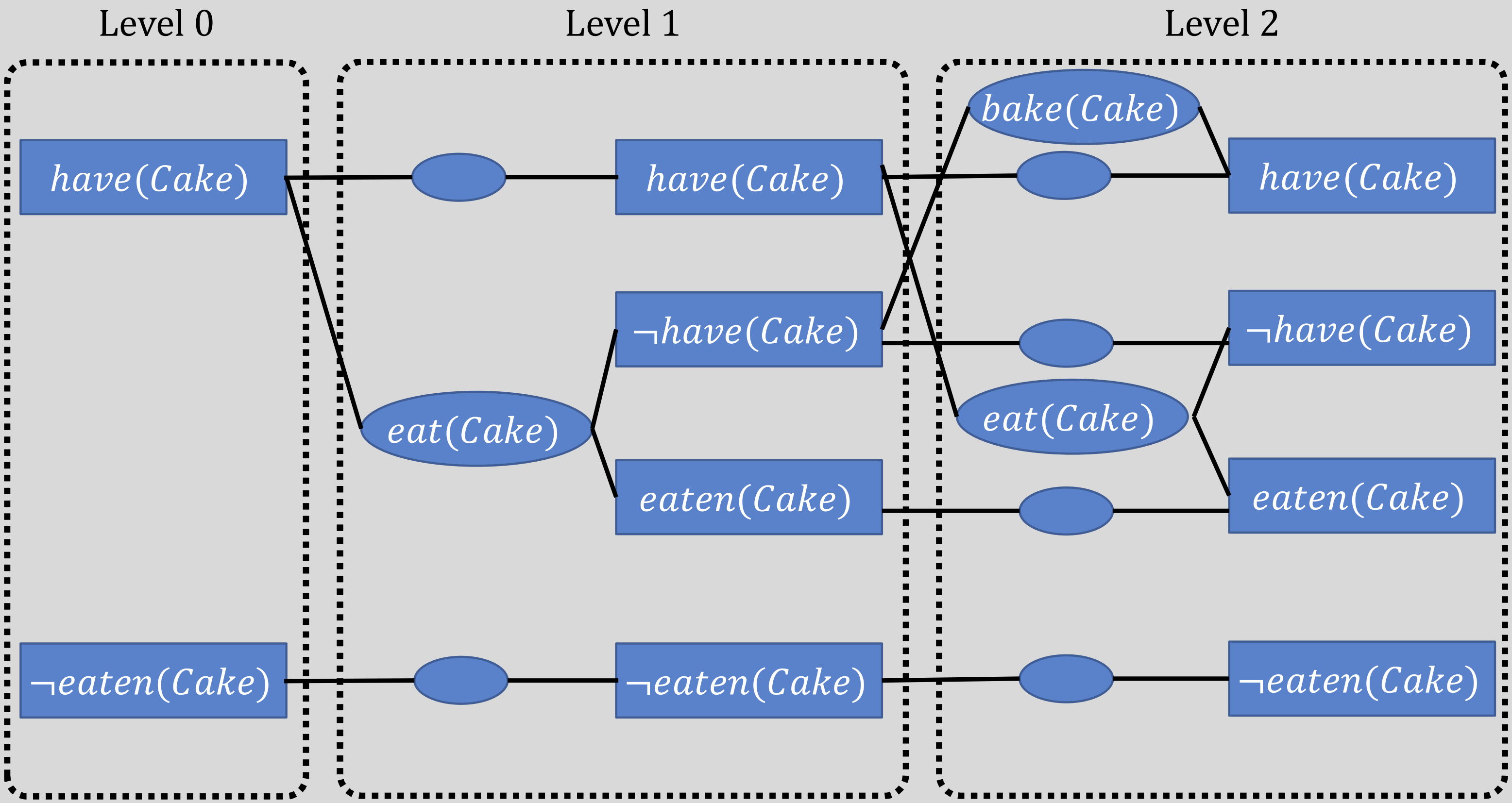
Adding Mutex Links: Literal Mutex Relations

- **Opposite** example:
  - $have(Cake)$ and $\neg have(Cake)$. easy!

- **Inconsistent Support** example:
  - *have(Cake)* and *eaten(Cake)* **specifically in Level 1** are mutex for this reason.
    - Because the only way to achieve *have(Cake)* in level 1 is the persistence action.
    - The only way to achieve *eaten(Cake)* in level 1 is the *eat(Cake)* action.
    - And those two actions are mutually exclusive (for two reasons: **inconsistent effects** and **interference**).
  - But as we'll see, this *isn't* the case for these literals at Level 2.

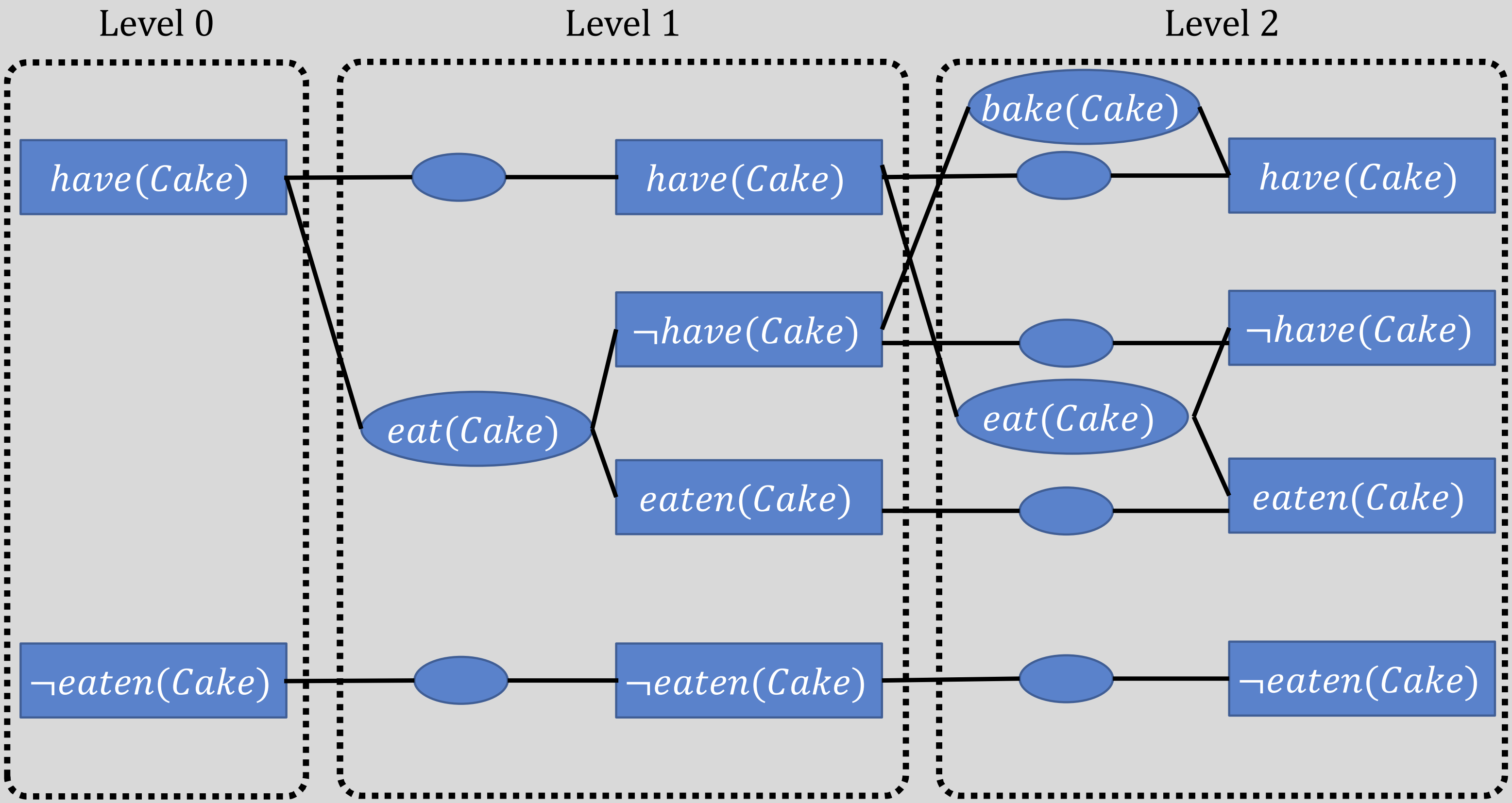
A Plan Graph

OK. Armed with our newfound mutex knowledge, let’s add some mutex links! Let’s start with steps.
NOTE: We are not adding *every* mutex link that would be on this graph, but hopefully enough to visualize how it works.



Inconsistent Effects: One action's effect negates an effect of the other.

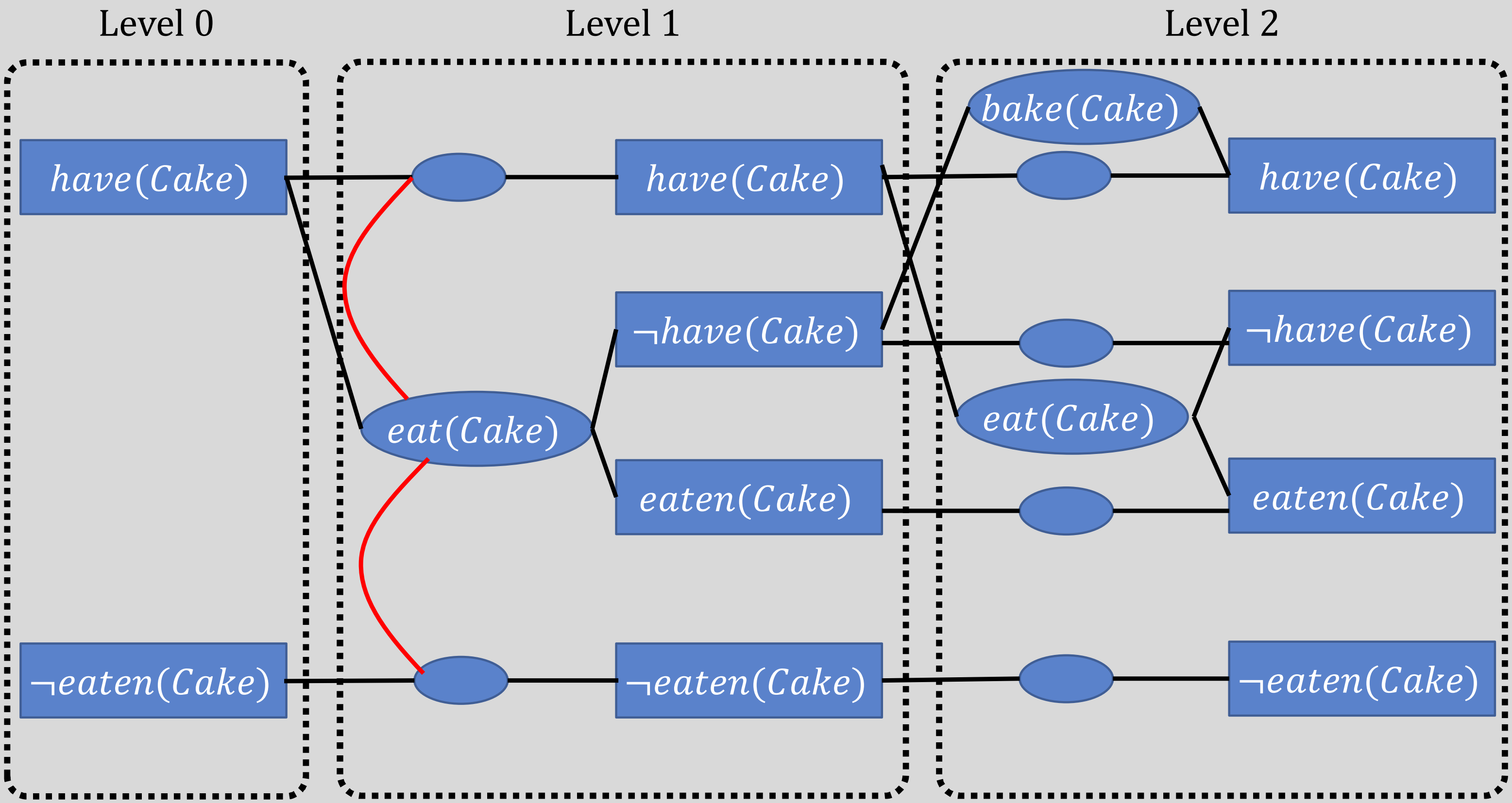
A Plan Graph



Inconsistent Effects: One action’s effect negates an effect of the other.

A Plan Graph

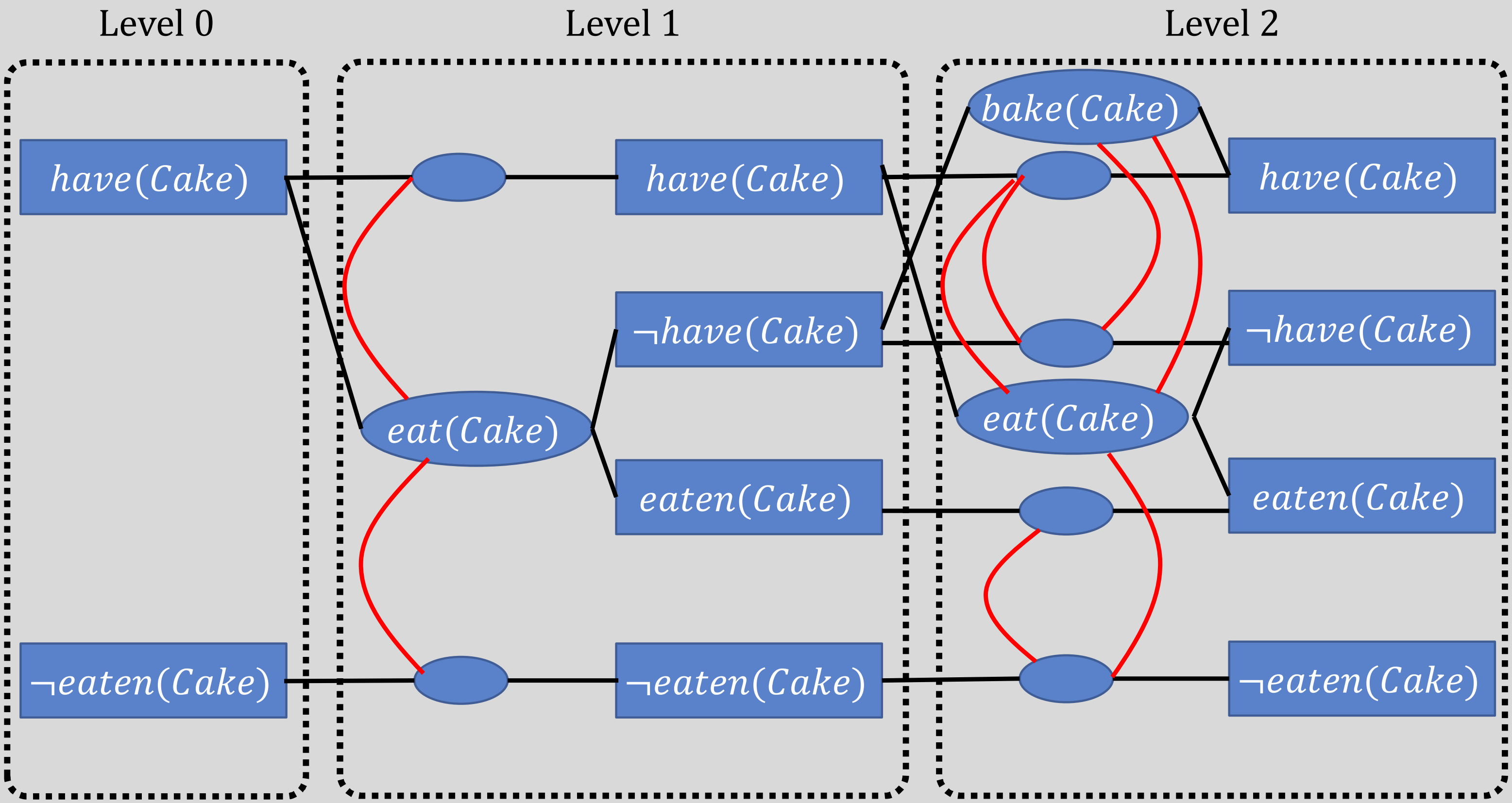
Starting with Level 1 stuff…



Inconsistent Effects: One action’s effect negates an effect of the other.

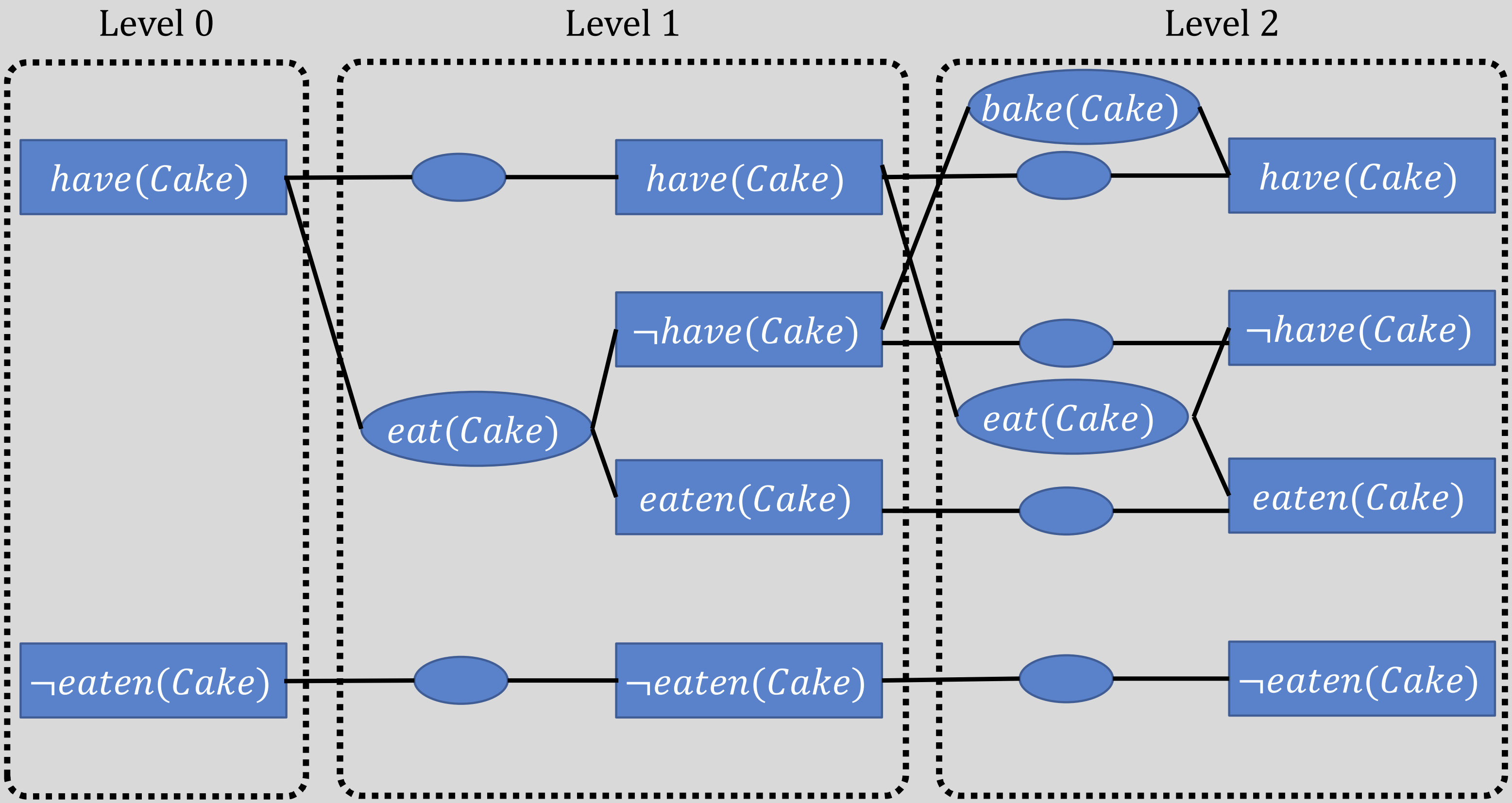
A Plan Graph

Adding in level 2 stuff



Interference: One action’s effect negates the precondition of another.

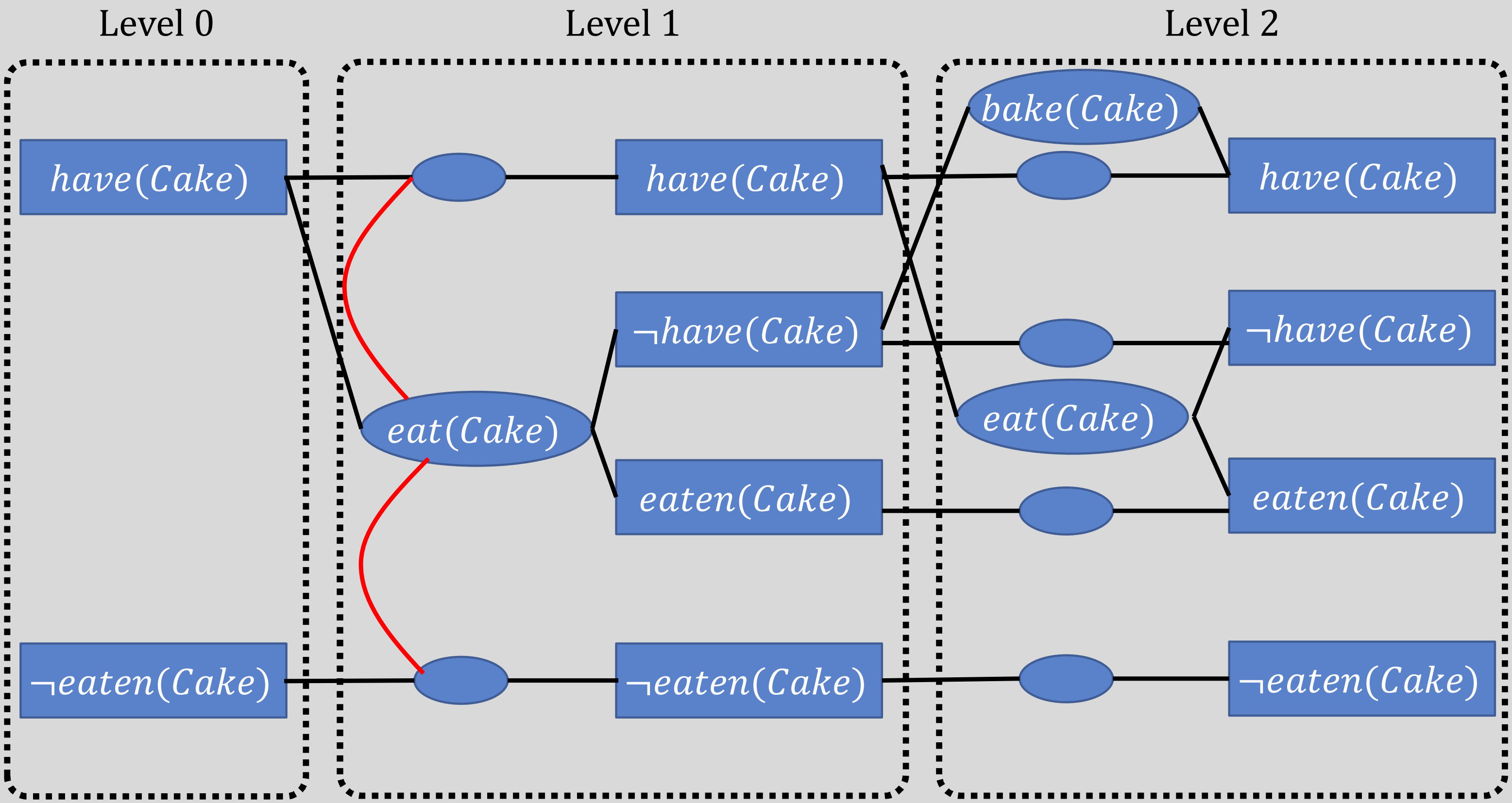
A Plan Graph



Interference: One action’s effect negates the precondition of another.

A Plan Graph

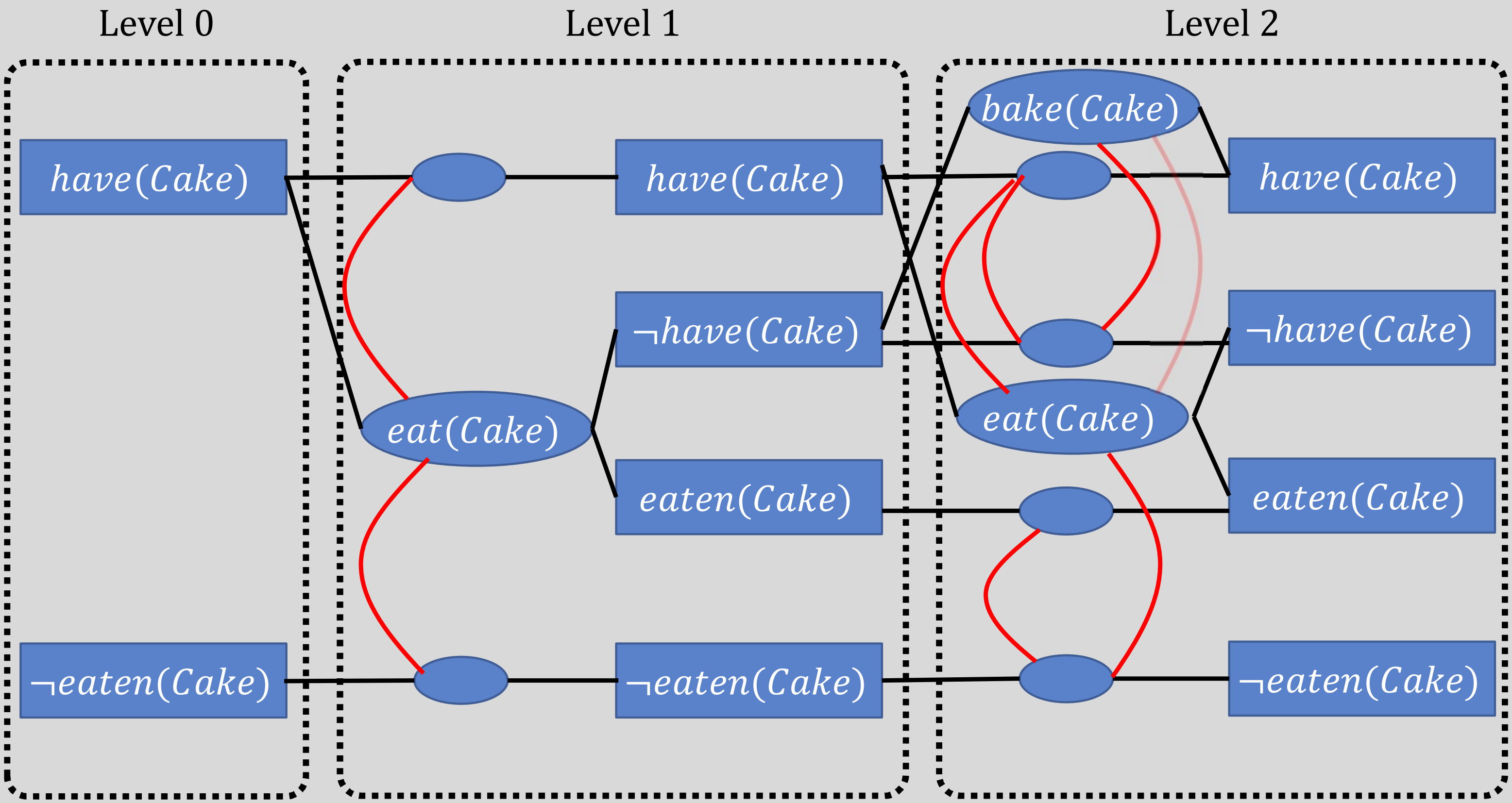
Starting with Level 1…



Interference: One action’s effect negates the precondition of another.

A Plan Graph

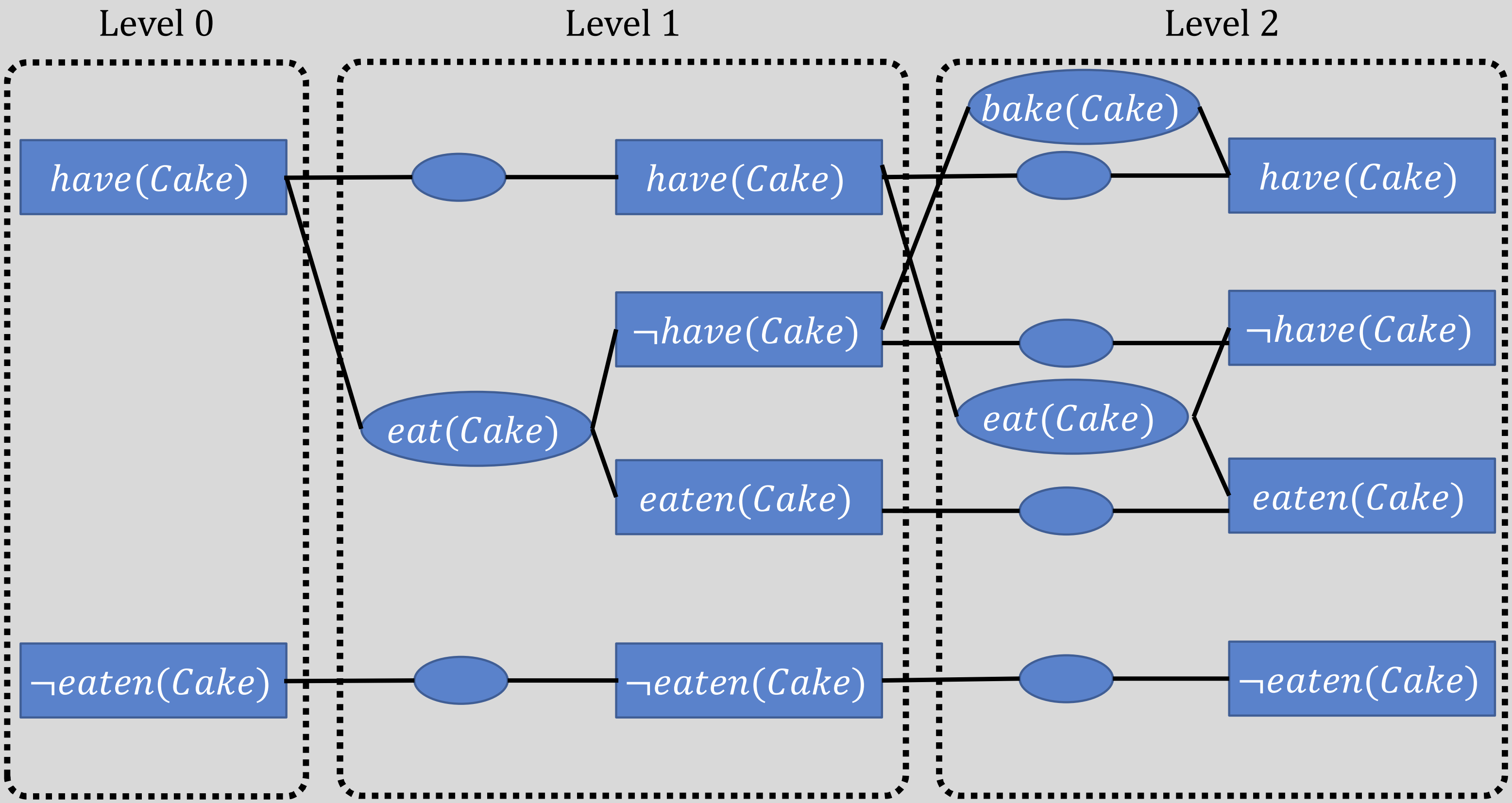
Adding in Level 2…



Competing Needs: Two actions have mutually exclusive preconditions

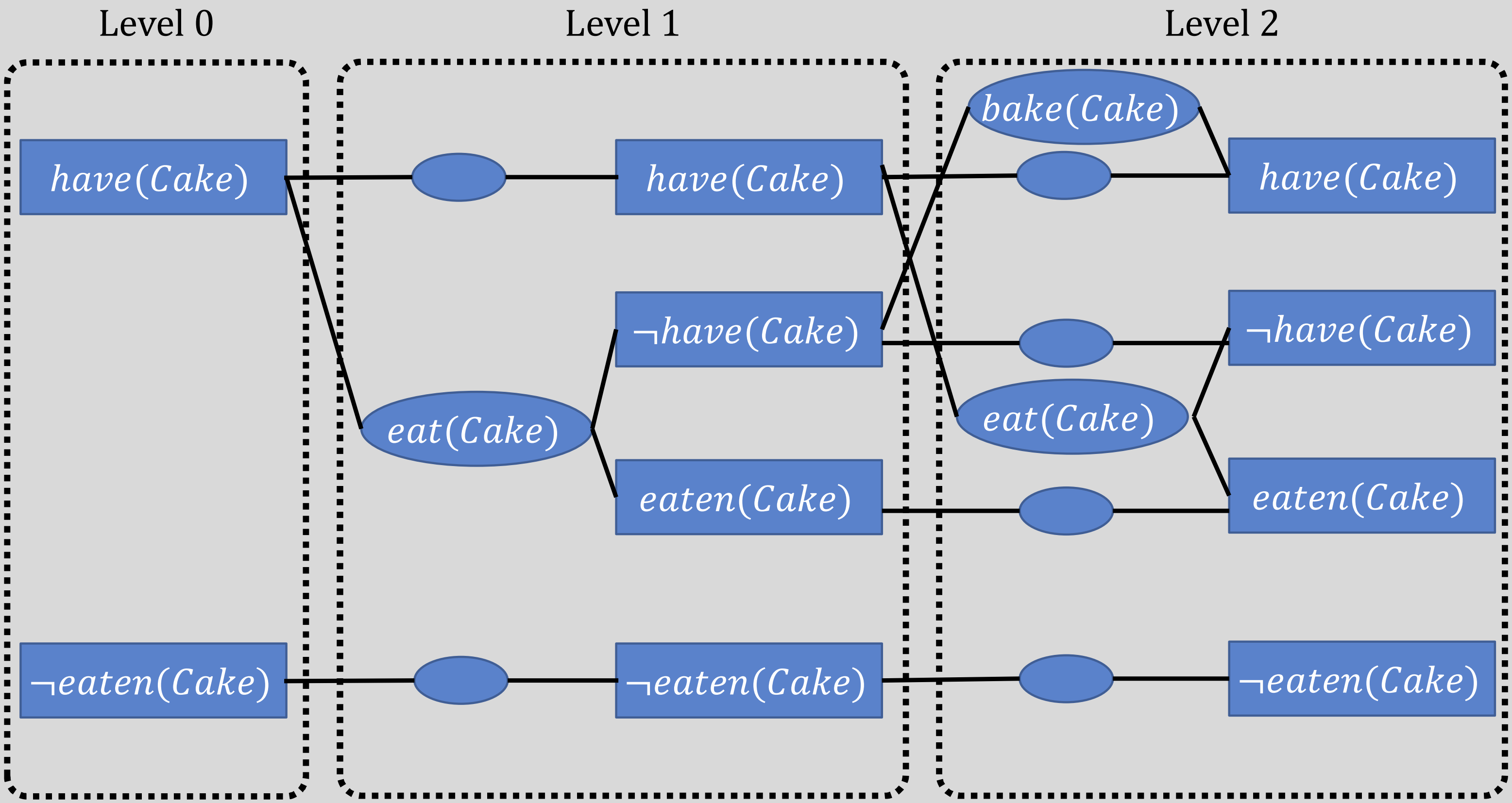
we will come back to this once we figure out which literals are mutex.
Speaking of which…

A Plan Graph



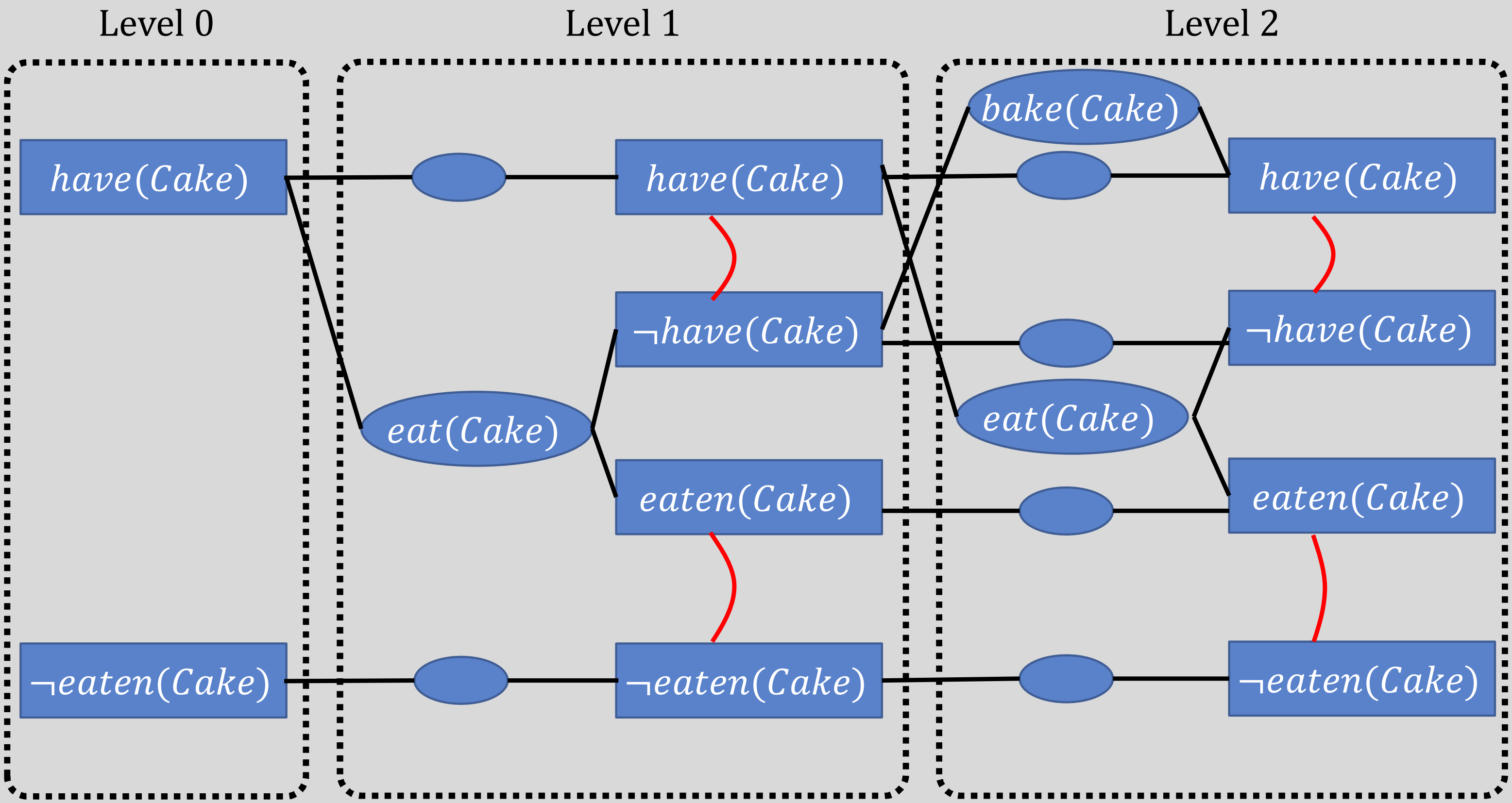
Opposites: One literal is the negation of the other.

A Plan Graph



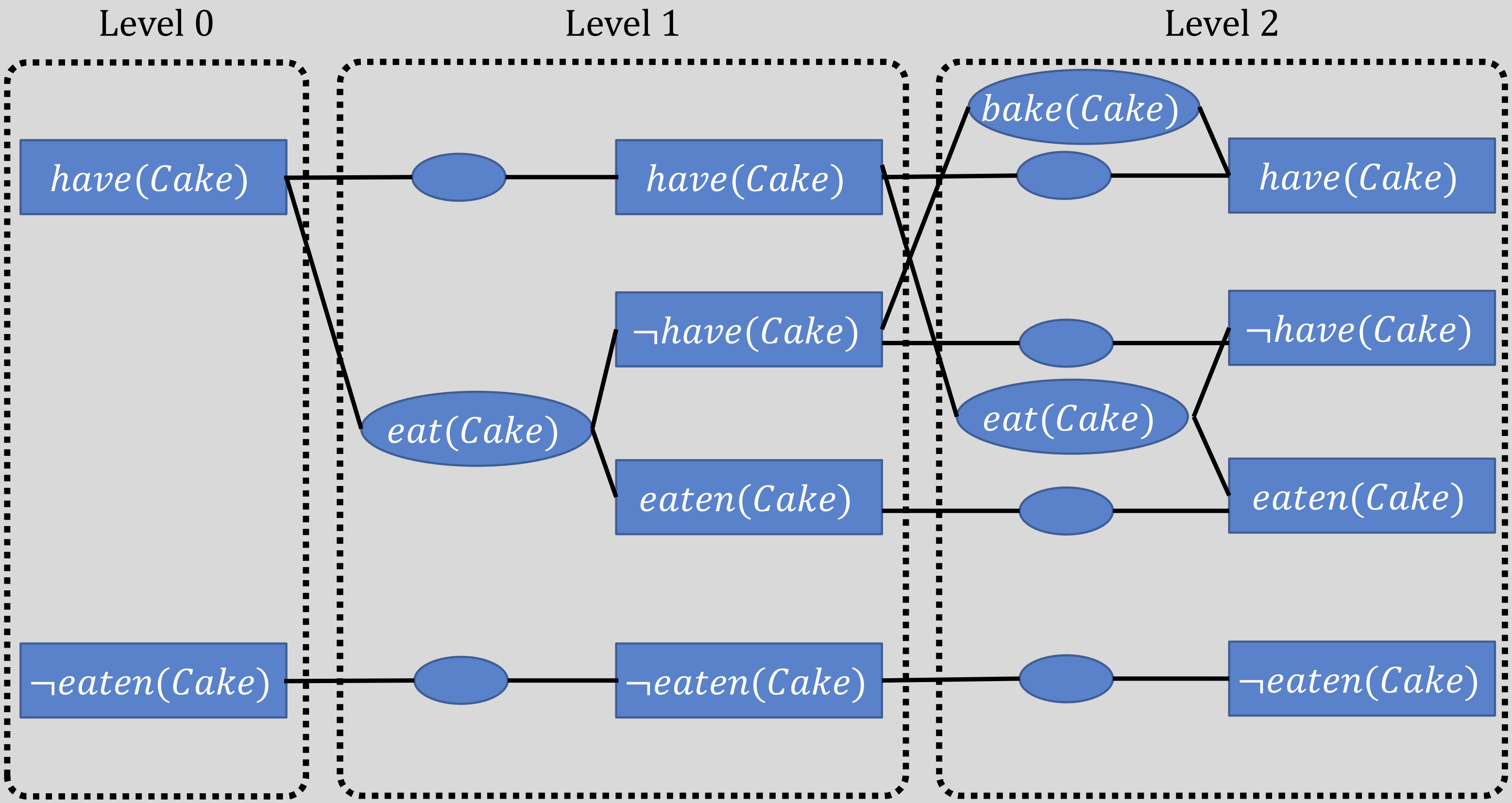
Opposites: One literal is the negation of the other.

A Plan Graph



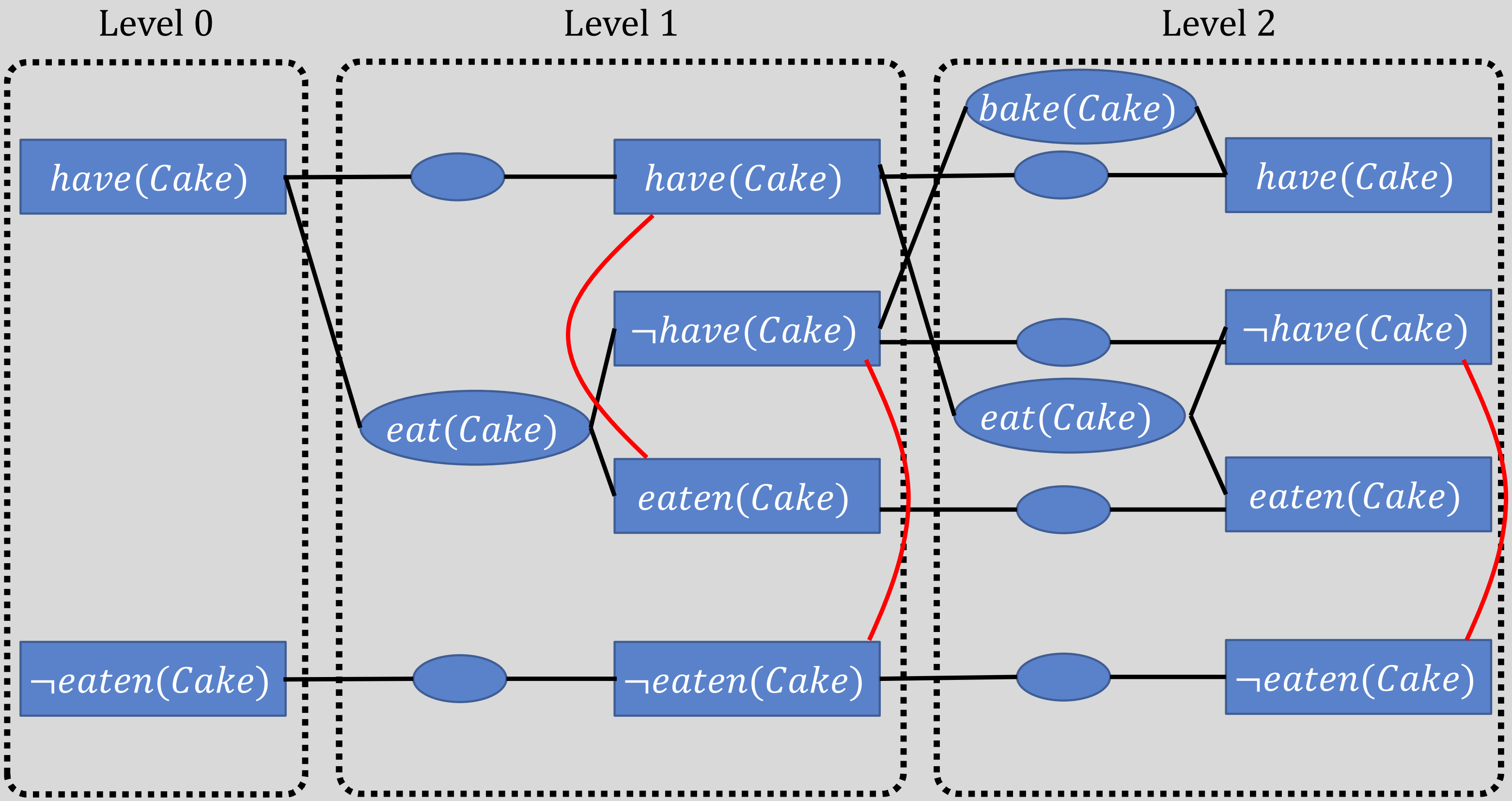
Inconsistent Support: Every pair of actions that achieves the literals are mutually exclusive.

A Plan Graph



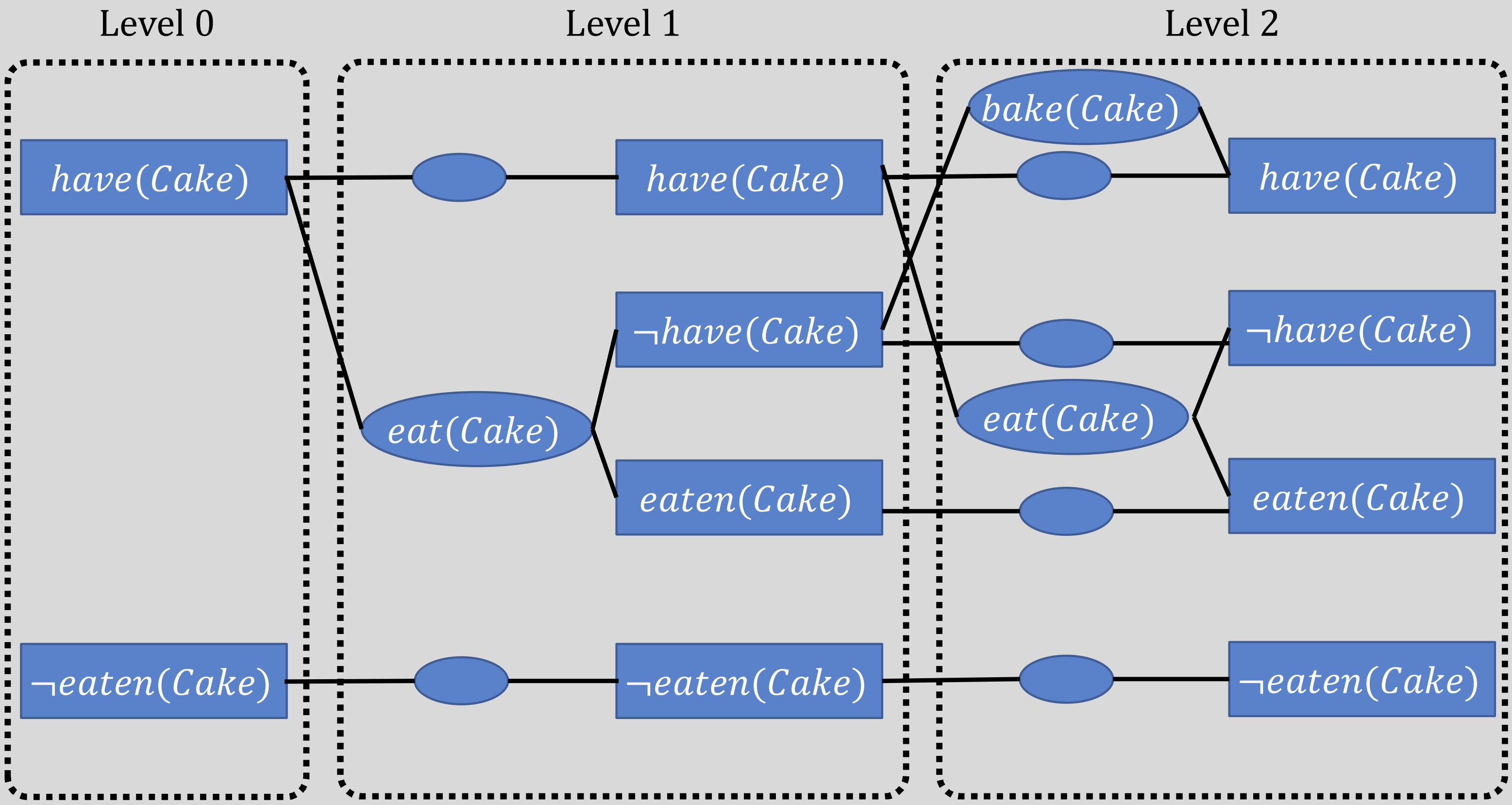
Inconsistent Support: Every pair of actions that achieves the literals are mutually exclusive.

A Plan Graph



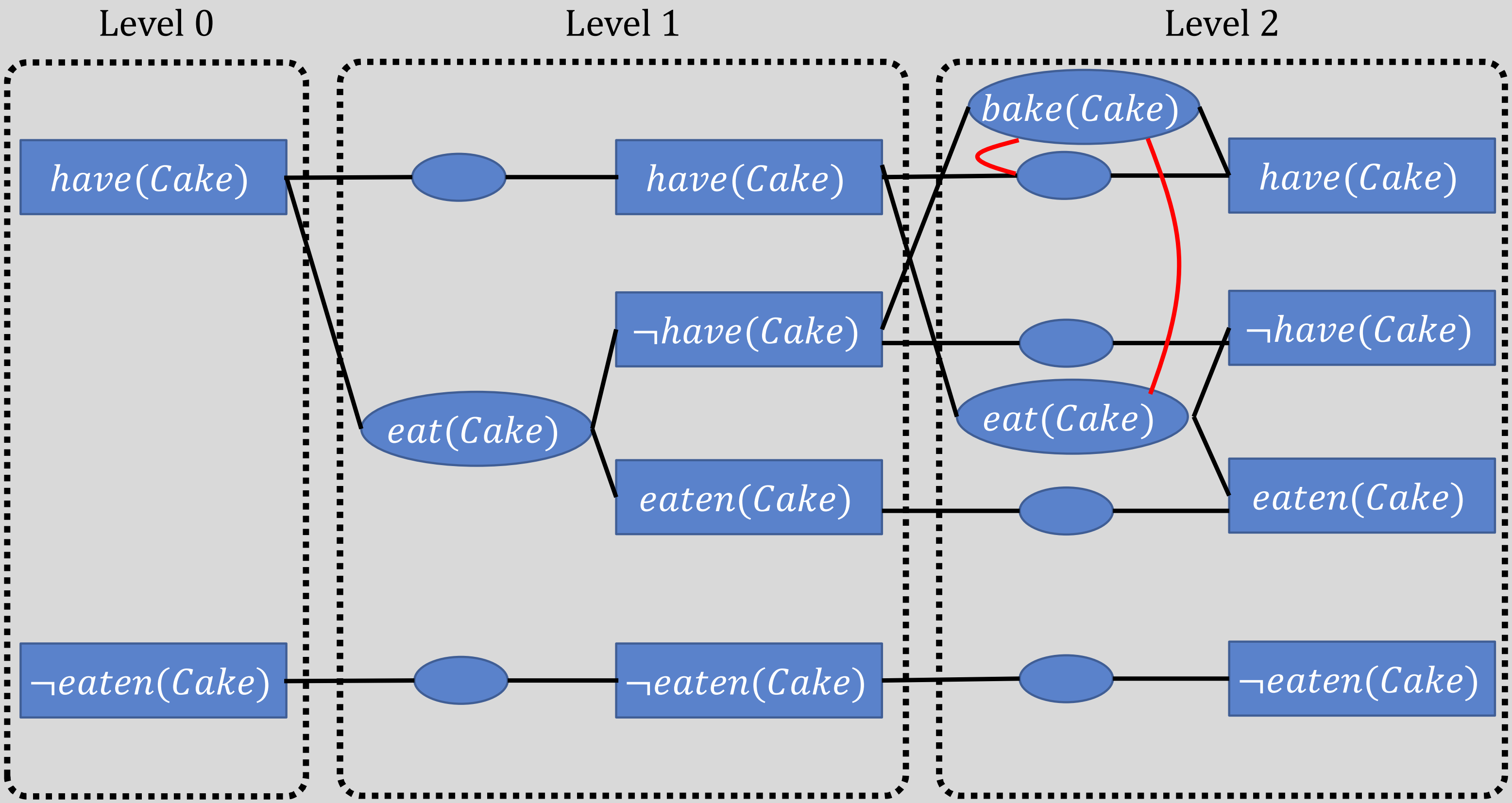
Competing Needs: Actions have mutually exclusive preconditions.

A Plan Graph



Competing Needs: Actions have mutually exclusive preconditions.

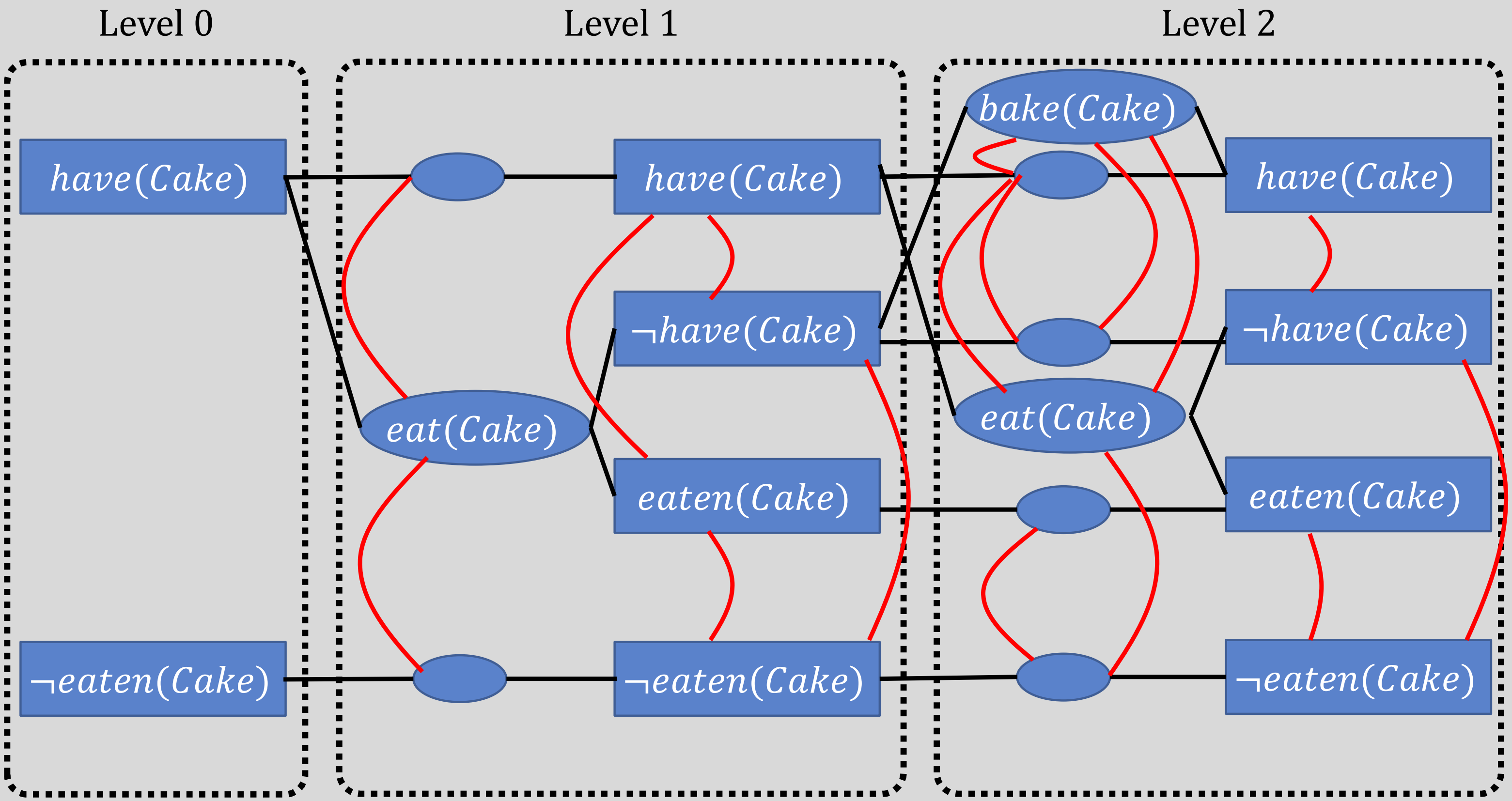
A Plan Graph



So, all together! These are (most of) the mutex links of the graph!

A Plan Graph

(some missing ones for you to ponder: if two literals are mutex at Level n, then their persistence actions will likely be mutex at Level n+1. Can you see why?)



Mutexes

- Another thing to consider: Is only marking pairs of nodes as mutually exclusive sufficient to capture everything that is *actually* mutually exclusive?

- No, consider this Blocks World problem:

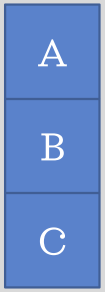
Goal: $on(A,B) \wedge on(B,C) \wedge on(C,A)$

Mutexes

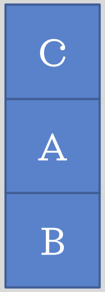
No, consider this Blocks World problem:

Goal: $on(A, B) \wedge on(B, C) \wedge on(C, A)$

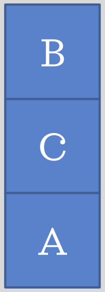
If you take any two of the goal's three conjuncts, you're in good shape… no two literals are mutually exclusive.



$on(A, B) \wedge on(B, C)$



$on(A, B) \wedge on(C, A)$



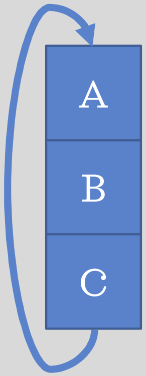
$on(B, C) \wedge on(C, A)$

Mutexes

No, consider this Blocks World problem:

Goal: $on(A, B) \wedge on(B, C) \wedge on(C, A)$

But things get weird if you try to have all three.



$on(A, B) \wedge on(B, C) \wedge on(C, A)$

C also has to be on A!

We’ve got on Ouroboros situation! An Ouroblockos situation!

Or in other words: the *three* goal literals taken together are mutually exclusive, but no *two* of them are mutually exclusive

Graphplan Algorithm

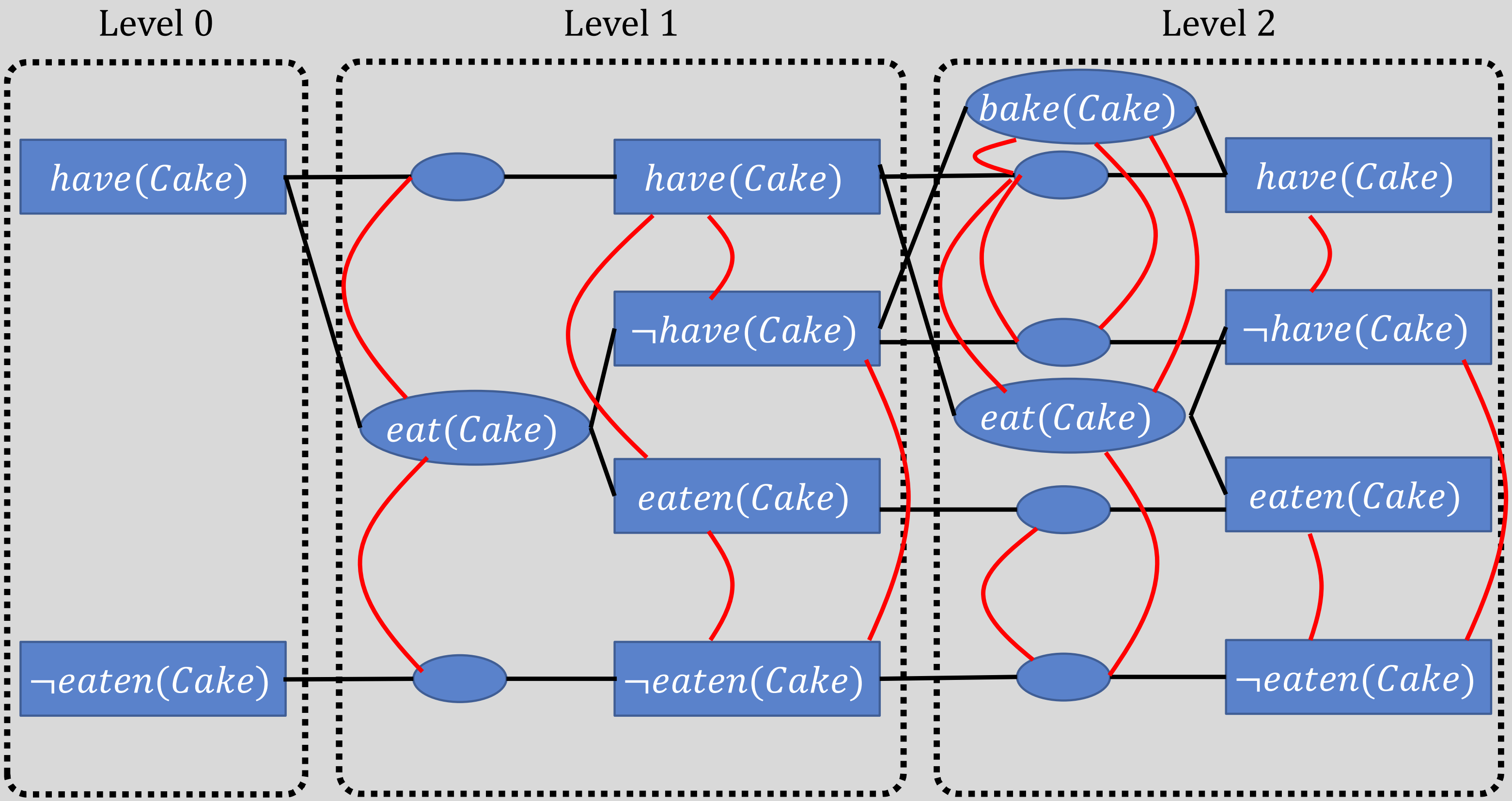
- As promised at the very beginning of this lecture…
- Plan Graphs aren’t *only* tools for deriving heuristic estimates of “how many steps until this literal becomes true / this action can be taken”…
- There are also algorithms that use plan graphs to search for actual plans themselves! Such as the **Graphplan** algorithm.

Graphplan Algorithm

```
Extend the plan graph until all goals are non-mutex.
Let G be the current goals, initially the problem’s goals.
Let n be the current level, initially the highest level.
To satisfy the goals in G at level n:
    If n = 0, return the plan as a solution.
    Choose a set of steps S which achieve all the goals in G.
    (Every pair of steps in S must be non-mutex).
    Add all steps in S to the plan.
    Let G = all the preconditions of the steps in S.
    Recursively satisfy all the goals in G at level n – 1.
If satisfying G fails, add a level to the graph and try again.
```

Graphplan Algorithm

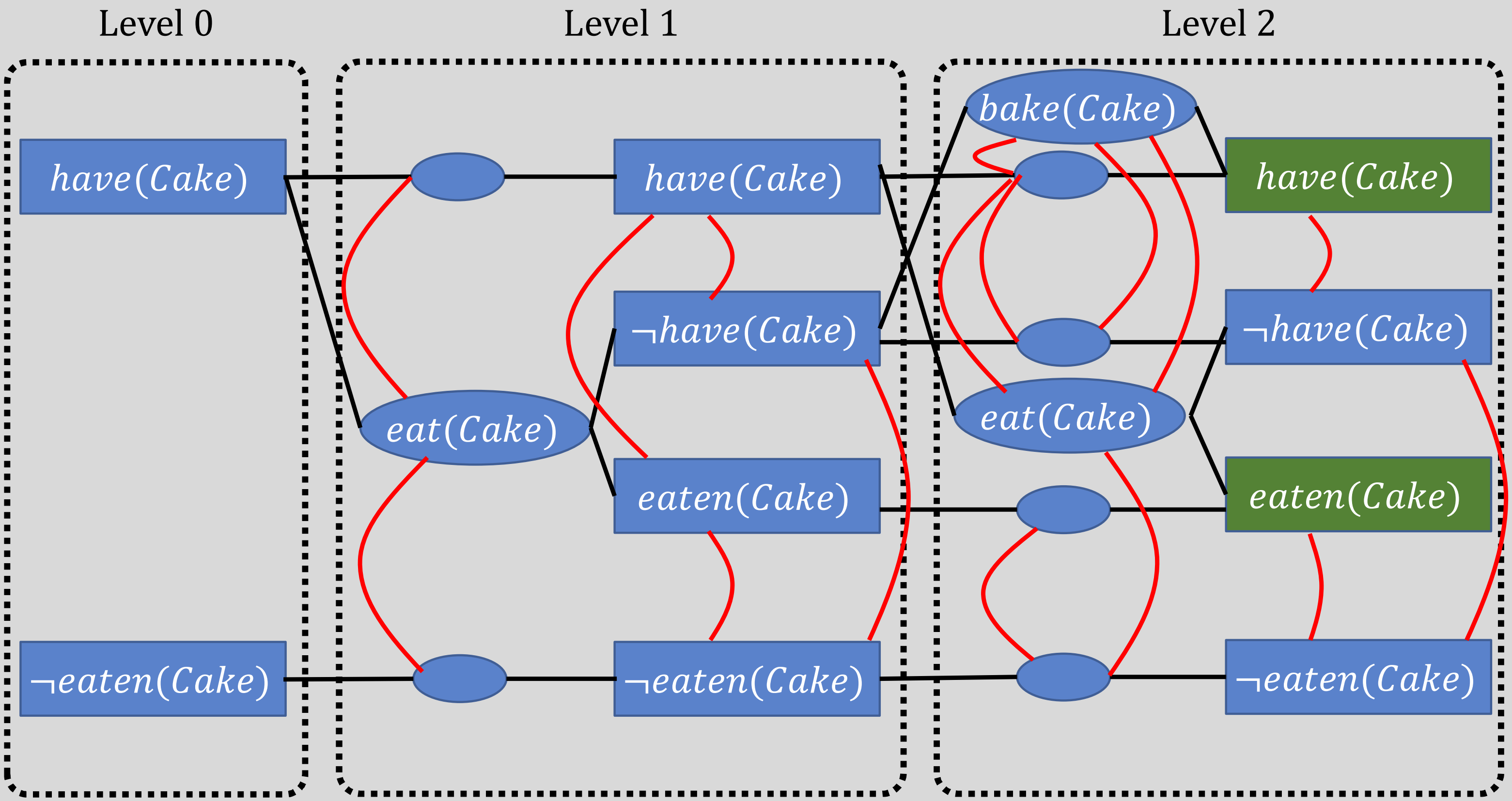
The good news is that building the graph is the hard part! Constructing a plan from it isn't actually all that bad!



Graphplan Algorithm

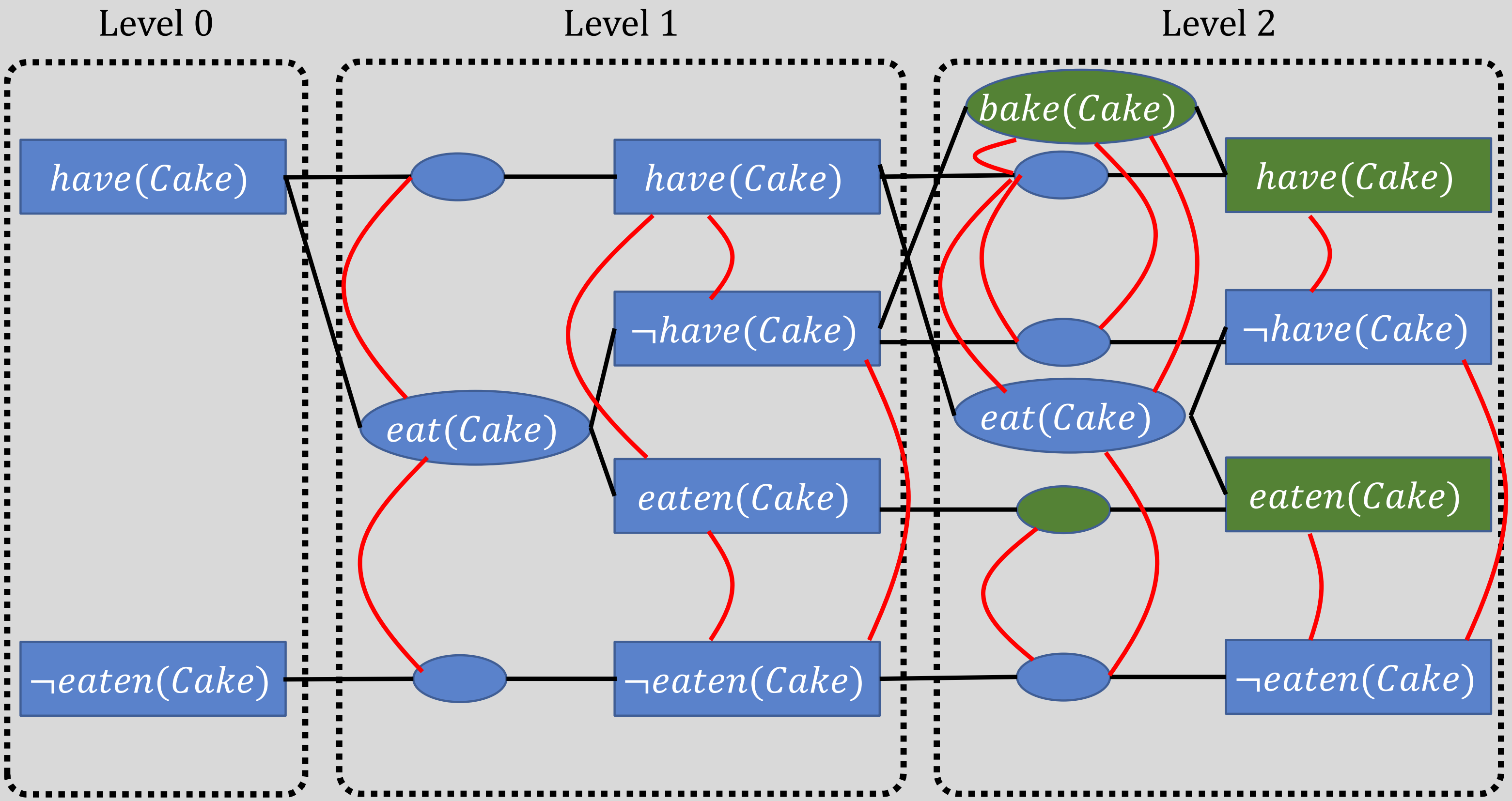
Current Goal: $have(Cake) \wedge eaten(Cake)$

And we see these two goals aren't mutually exclusive, so we've built the graph out enough to start.



Graphplan Algorithm

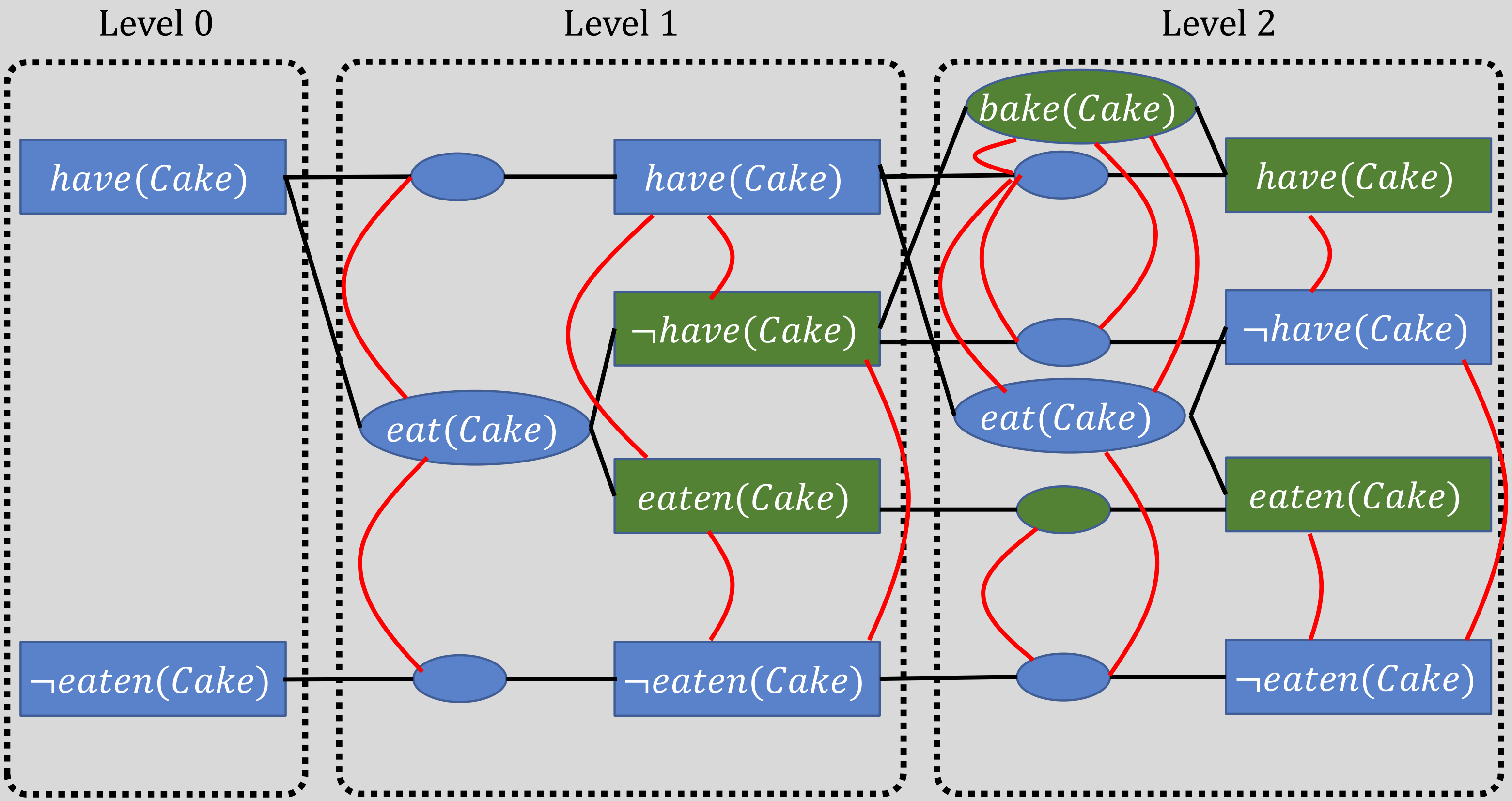
Current Goal: $have(Cake) \wedge eaten(Cake)$
We choose a set of steps (that aren't mutually exclusive) at this level that satisfies all these goal literals: here, *bake(cake)* and the *eaten(cake)* persistence step.



Graphplan Algorithm

Current Goal: $\neg have(Cake) \land eaten(Cake)$

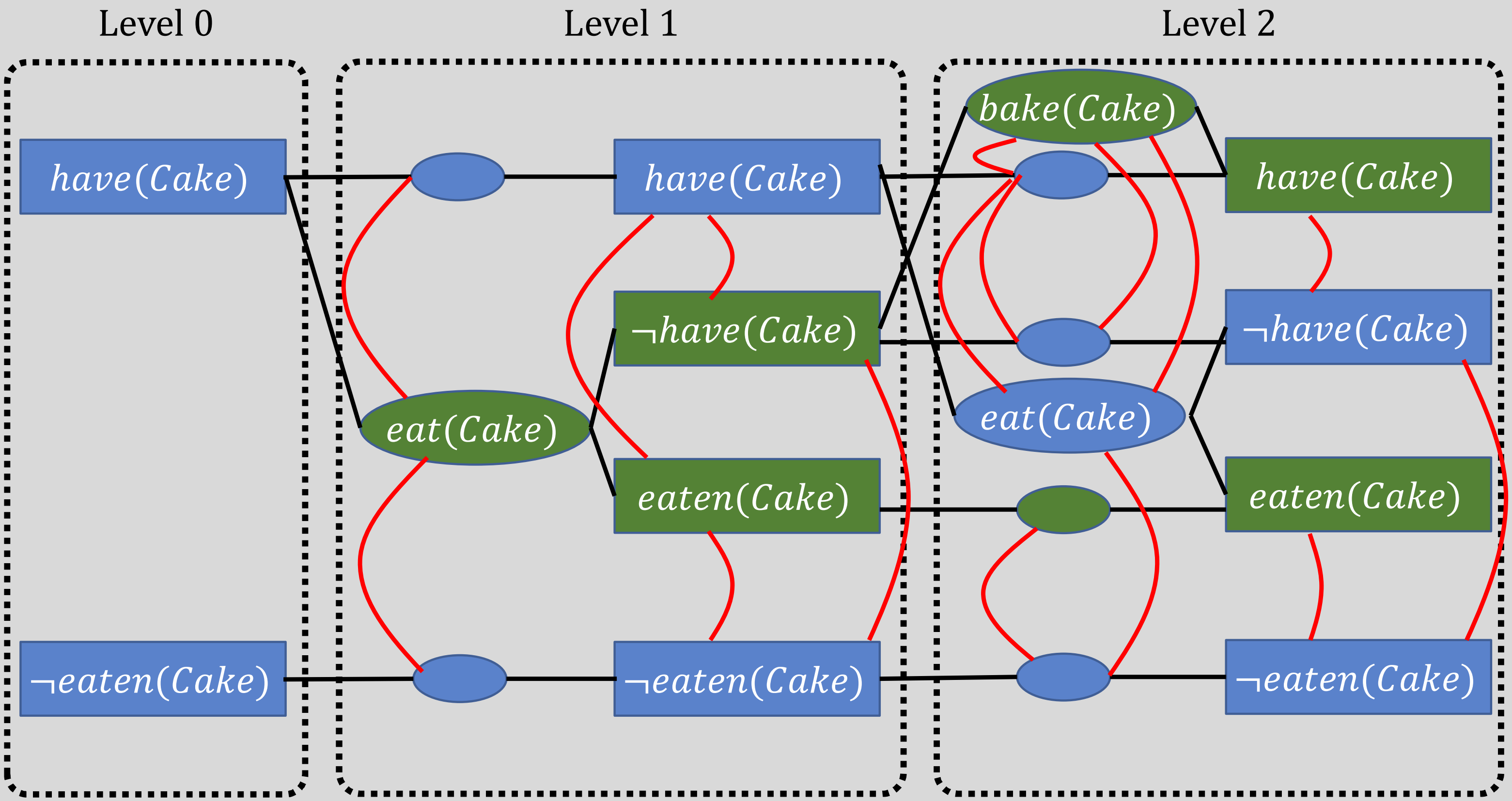
We recursively call graphplan, but now the "goal" we are solving are the preconditions for the actions we just chose.



Graphplan Algorithm

Current Goal: $\neg have(Cake) \wedge eaten(Cake)$

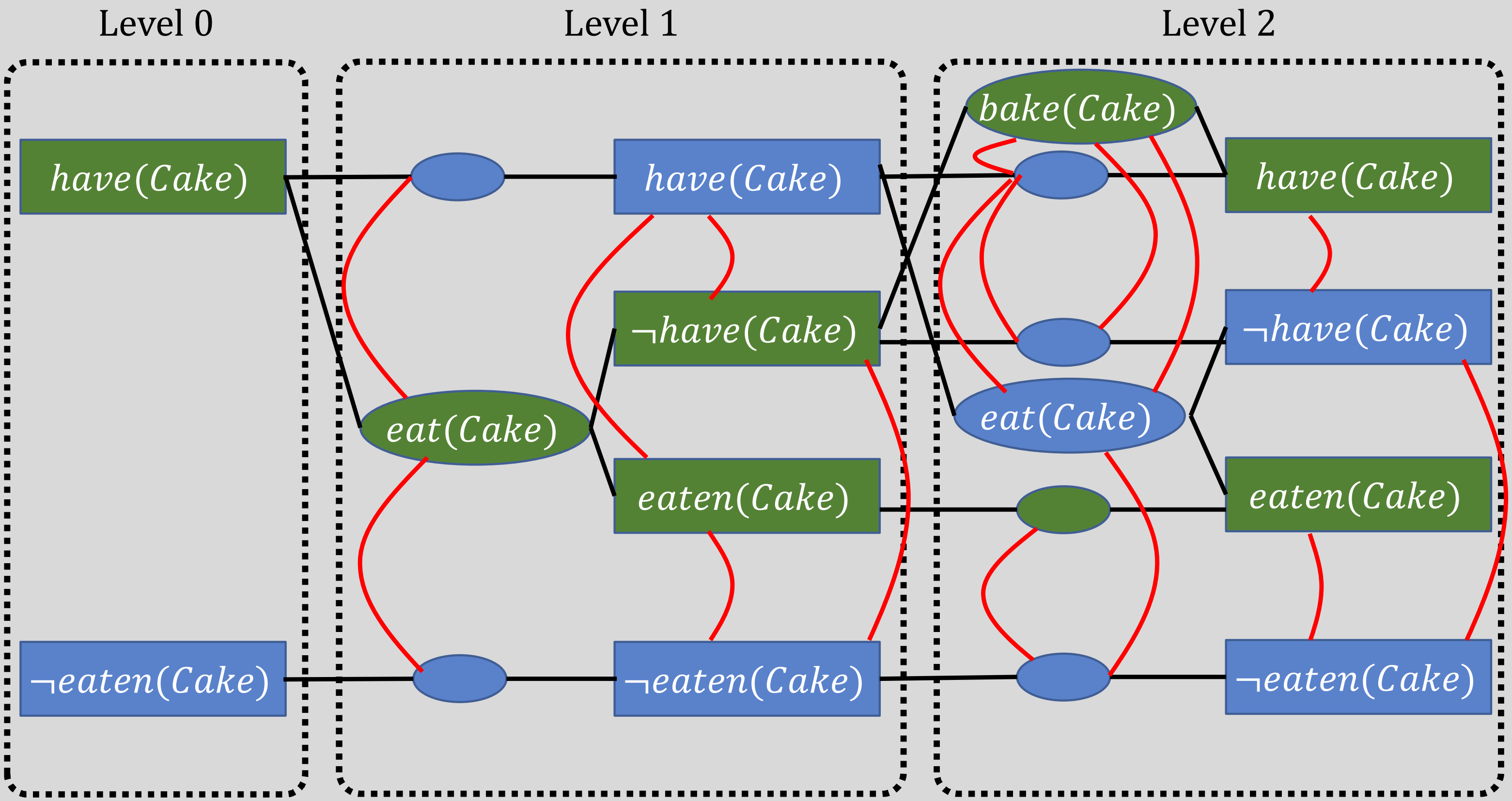
And again, we choose a set of actions that achieves all these new goals. Here, the single *eat(Cake)* action achieves everything!



Graphplan Algorithm

Current Goal:$have(Cake)$

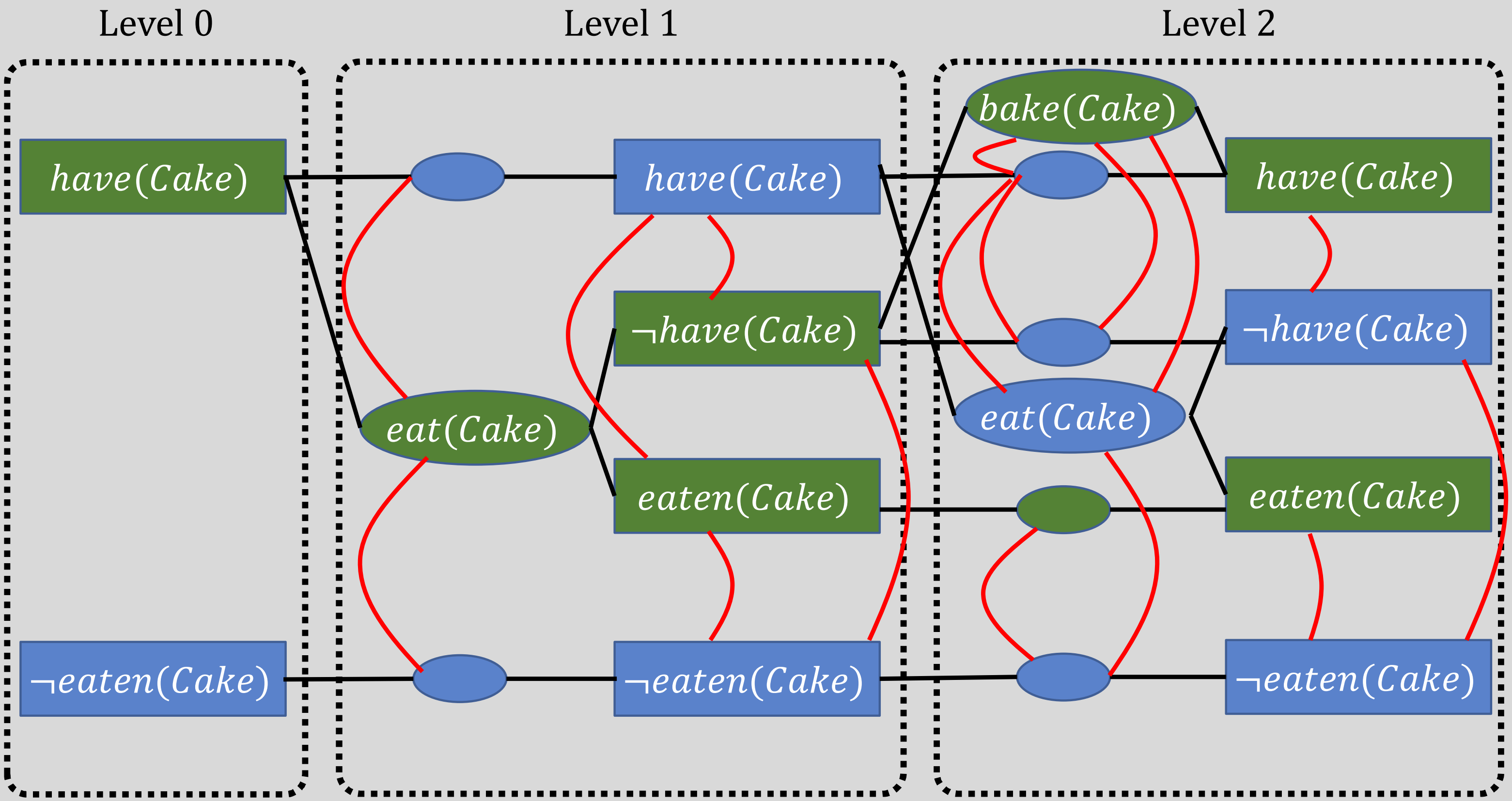
And once again, we recursively call graphplan, this time with the preconditions of *eat(Cake)* as the goal.



Graphplan Algorithm

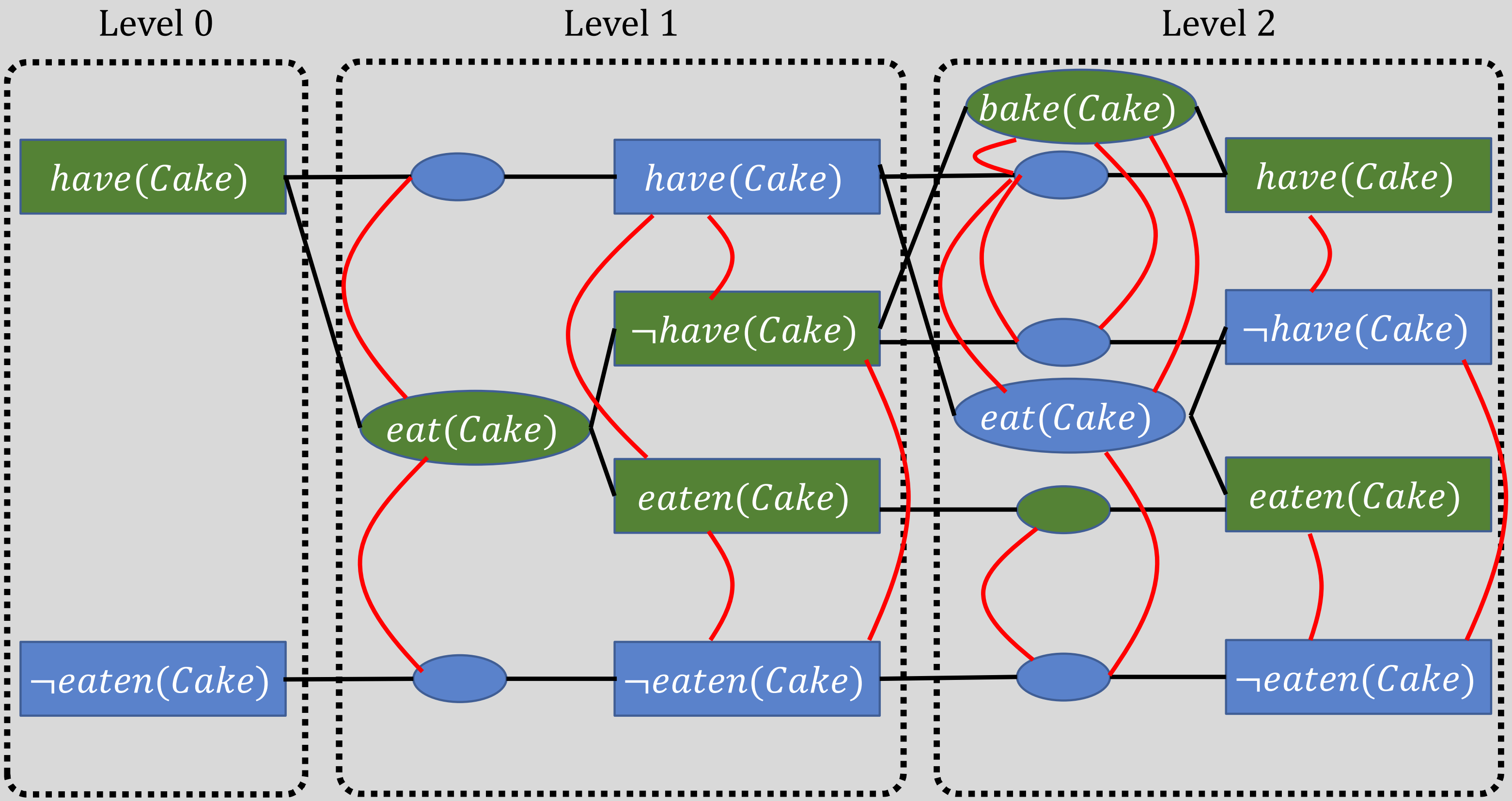
Current Goal: $have(Cake)$

And we hit Level 0, which is our end condition, so we're done!
Can you read the plan we constructed here?



Graphplan Algorithm

Basically, the plan is:
“First, do the eat(Cake) action”.
“Then, do *bake(Cake)* and the eaten(Cake) persistence action concurrently (which is safe to do, since they aren’t mutex).



Backtracking Search

- As we’ve seen many times, you might have to backtrack again with this algorithm if you hit a dead-end.

- In other words: if you reach a level where you can’t do something (e.g., a set of actions that aren’t mutually exclusive that achieves the current goal literals doesn’t exist), you’ll have to go back to a previously visited level and try something else.

No-Goods

- If a complete backtracking search fails to satisfy a set of goals at a particular level, it means that set cannot be achieved at that level.

- We can record this set as a **no-good**, adding it to a list. Later in the algorithm, if we are ever asked to solve that set of goals and that same level again, we can just refer to our no-good list, and fail right away without re-doing the search.

- You can think of no-good's as just another form of mutex if you like (one that potentially involves more than two nodes).

Extending the plan-graph

- As we discussed, once a literal appears at a level, it will forever appear at all higher levels.
- Once a step appears at a level, it will appear at all higher levels.
- Once a mutex appears, it might disappear at higher levels (things which were mutex early on might become non-mutex later)
  - This was pivotal for us! At level 1 *have(Cake)* and *eaten(Cake)* were mutex, but not so at Level 2.
- Once a no-good appears, it might disappear at a higher level.

Levelling off and Termination

- Eventually, the literals and goals at each level will stop increasing.
- Eventually, the mutexes and no-goods will stop decreasing.
- When both of these conditions have occurred (i.e., when you have two levels that are the same), then the graph has **levelled off**.
  - Extending it more at this point will do nothing. It will forever be the same.
- If Graphplan fails when the graph has levelled off, then we know that no solution to the problem exists (i.e., there is no plan that takes us from the initial state to the goal state).