Questions from Last Week

Q: Why would I use forward chaining over backward chaining or vice versa? When does one outshine the other?

In class I gave an approximation for when you might prefer forward chaining over backward chaining but didn't formalize it with examples...

There are some great articles out there (a simple Google search away) that describe these scenarios, which I've outlined below.

á Use **forward chaining** when you need to reach a conclusion that you may not know at the onset or how to reach at the onset of your inference (deductive reasoning).

Forward chaining is also preferable when:

- You are precomputing and storing answers (remember that we generate all the facts we can)
- Facts are relatively inexpensive to store and compute
- Generated facts are largely static and will not change (i.e., knowledge generated is monotonic)

à So why not always perform forward chaining so that we have the most facts available to us when we start reasoning?

This brings us to backward chaining.

á Use **backward chaining** when you have a goal query to ask at the start and can look solely at dependencies to reduce your search space.

Backward chaining is therefore preferable when:

- There is a huge number of rules that might be used to reach a conclusion and you need to narrow down the possibilities

- Computed answers and facts have little chance of being reused
- Facts are expensive to store or compute

Some articles also offered a **hybrid** approach (similar to one someone mentioned in class) such that you only store / index facts that are used frequently in answering questions.

Planning

Planning is an important extension of first order logic that is really exactly what it sounds like:

- Represent states and actions as first order logic statements (actually a subset of FOL that doesn't have functions)
- Use a KB to represent our current states, which will determine whether or not we meet the conditions for taking an action that will change our KB.
- Attempt to reach a goal state specification through some sequence of actions for which we've met the requirements.

Hey wait a second... initial conditions? Actions? Goals?

> à What is this starting to sound like?

(I hope you guys remember Peewee's Playhouse from your childhood and their whole word of the day thing, otherwise that last joke will be weird for you... also, I can't make gifs)

But yes, because this is a class on search apparently, we're going to talk about planning as a search operation starting at some initial state and using actions (that we're capable of using) to hopefully get to a goal.

First, some definitions...

> á **Fluents** are our states of the world that are subject to change and consist solely of grounded FOL atoms (no functions)

Some examples of fluents include:

```
; Constants
LAX
Andrew
ProtocolObserved

; Functionless Predicates
ISA(Home, Building)
Thirsty(Liz)
```

á Our state of fluents uses **database semantics** under the **closed world** and **unique name** assumptions.

In other words, our conception of state will be contained in a knowledge base with certain properties:

- **Closed World:** our KB consists of a set of fluents assumed to be true, and any fluents *not* mentioned are assumed to be false
- **Unique Names:** constants with different names are assumed to refer to different objects (remember Skolemization?)

```
; For example:
Andrew1 ; me
Andrew2 ; my evil twin
Andrew1 ≠ Andrew2
```

So, our states might just look like KBs such as:

```
KB =
   1. At(Andrew, UCLA)
   2. Teaching(Andrew, CS161)
   3. AwakeAt(Andrew, 2:00) ; sorry 1A!
   ; ...
```

Now that we have our states, we should look at how to modify them.

In classical search this meant using actions, and planning is only slightly different...

Our actions in classical search were generated from legal movements to our atomic states that we knew were adjacent by one movement.

This same paradigm doesn't work so neatly for planning because we don't have atomic states! Our states are KBs with a bunch of different sentences.

á **Planning actions**, therefore, are a way of modifying our states that can be applied when their pre-conditions are met.

á **Preconditions** are criteria that our current state must meet in order to perform an action. They are in the form of (possible unground) FOL sentences.

Example

t Given the following state, are the following action preconditions satisfied?

```
; Example in honor of Dyer
KB =
   1. HasBat(MerryWidowMurderer)
   2. IsWith(MerryWidowMurderer, Charlotte)
   3. Alive(MerryWidowMurderer)
   4. Alive(Charlotte)

; Action 1: Bonk
; [!] Notice that our action has two "parameters"
; bonker (the one who's bonking) and bonkee (the one who's
; being bonked); these are FOL variables
Action(Bonk(bonker, bonkee)
  Precondition:
    Alive(bonker) ∧ HasBat(bonker) ∧ IsWith(bonker, bonkee)
  Effect:
    ; ...
)
```

á **Postconditions / Effects** are what happen to our state if we choose to take a certain action for which we have satisfied the preconditions.

Once we choose to take an action for which our state has satisfied the preconditions, the result is a new state KB where the set of fluents has changed.

á The changes a state undergoes from taking an action are as follows:

- If, in the postcondition, a fluent is positive, you add that to your KB. (the **add-list**)
- If, in the postcondition, a fluent is negative, you remove that fluent from your KB. (the **delete-list**)
- All other state fluents are left the same.
- Formally, the result of taking action a at state s is: **Result**(s, a) = (s - DEL(a)) ∪ **Add**(a)

Example

t Taking our example from before, what will our new state look like?

```
KB =
    1. HasBat(MerryWidowMurderer)
    2. IsWith(MerryWidowMurderer, Charlotte)
    3. Alive(MerryWidowMurderer)
    4. Alive(Charlotte)

; Action 1.1: Bonk
Action(Bonk(bonker, bonkee)
  Precondition:
    Alive(bonker) ∧ HasBat(bonker) ∧ IsWith(bonker, bonkee)
  Effect:
    ¬Alive(bonkee) ∧ ¬HasBat(bonker) ; the bat breaks, clearly
)

; Action 1.2: Bonk
Action(Bonk(bonker, bonkee)
  Precondition:
    Alive(bonker) ∧ HasBat(bonker) ∧ IsWith(bonker, bonkee)
  Effect:
    ¬Alive(bonkee) ∧ HasSplinter(bonker) ; the bat gave bonker splinters
)
```

à Why don't we add negative fluents to our KB?

So, we can formalize our planning search problem:

- The **initial state** is simply a list of positive fluents
- The **goal** is just a FOL description of conditions that our state must match for success--just like a precondition, except goals are our terminating condition.
- The **solution path** we want to find is a series of actions taken that are allowed from our initial state to our goal.

Example
t Determine a sequence of actions that reaches the goal state from the initial state in the following planning specification (from your book):

You must change a flat on your car using the following actions. You could try to get a ride from a friend, and leave it overnight, but your car is in a bad neighborhood so apparently that means that all of your tires, regardless of where they are, get stolen.

```
KB = Tire(Flat) ∧ Tire(Spare) ∧ At(Flat, Axle) ∧ At(Spare, Trunk)
Goal(At(Spare, Axle))

Action(Remove(obj, loc),
  Precondition:
    At(obj, loc)
  Effect:
    ¬At(obj, loc) ∧ At(obj, Ground))

Action(PutOn(t, Axle),
  Precondition:
    Tire(t) ∧ At(t, Ground) ∧ ¬At(Flat, Axle)
  Effect:
    ¬At(t, Ground) ∧ At(t, Axle))

Action(LeaveOvernight(),
  Precondition:
  Effect: ¬At(Spare, Ground) ∧ ¬At(Spare, Axle) ∧ ¬At(Spare, Trunk) ∧
          ¬At(Flat, Ground) ∧ ¬At(Flat, Axle) ∧ ¬At(Flat, Trunk))
```

Planning Algorithms

Just as we talked about backward and forward chaining as inference algorithms in first order logic, so do we have backward and forward search strategies for planning.

The paradigms between the two strategies are quite similar; we'll start by looking at backward relevant-states search:

Relevant State-space Search

á **Backward / Relevant States Search** starts with our goal and attempts to derive the initial state (or a subset of it) from relevant actions.

á A **relevant action** is one that unifies on at least one goal / subgoal fluent (either positive or negative) in its effects AND that does not contradict any goal fluents.

à Why is it sufficient to derive a subset of the initial state's positive fluents rather than the initial state in its entirety?

The gist is the same as for backward chaining: starting with the goal, look for actions that could be relevant for deriving the initial state and then generate sub-goals to satisfy relevant pre-conditions.

Note that our problem must allow us an easy means of working backwards to derive a new state from a hypothesized relevant action (that will hopefully work us from the goal to the initial state).

Luckily, using our PDDL representation of planning problems, this isn't difficult.

á Given a ground goal description g and ground action a, the regression (backward step) from g over action a gives us a new state g' described as:

```
g' = (g - ADD(a)) ∪ Precondition(a)
```

In other words, taking a step back is simply removing the add list of an action from our current state and the uniting over that actions preconditions.

The idea is that we "undo" the action's effects under the assumption that we met its preconditions in order to have taken it in the first place.

á **Partially uninstantiated** states are states we encounter along the way of taking regression actions in which not all variables are grounded.

Just like in backward chaining, we want to make sure we have a fully grounded KB.

The process, then, is as follows:

1. Start with your query as the goal
2. Explore relevant actions starting from the goal specification hoping to derive the initial state (or a subset of it)
3. For each relevant action, search for groundings in our KB state that will derive the initial state (or a subset of it)
4. Return your plan if you derive some grounded subset of the initial set, or return failure if you have no relevant actions remaining

So, let's try using relevant state search on a fantastic example planning problem:

Forney Industries (the makers who brought you The Intelligent Sponge and PillowTurner2999) is developing a new general purpose buttler bot named ForneyBot3000.

It takes commands from its owner (currently only Andrew) to fetch and refill party libations.

During its stress test, Andrew commands it to refill another partier's beverage in order to quench that person's thirst.

Example

t Use relevant state search to derive a sequence of actions that will find our initial state starting with the goal of quenching our party goer's thirst.

```
Constants: Andrew, Partier2, Cup1, Cup2, ForneyBot3000
Goal: ¬Thirsty(Partier2)
Initial State:
  KB =
     1. Thirsty(Partier2)
     2. BotOwner(Andrew)
     3. ContainsLiquid(Cup2)
     4. Has(Partier2, Cup1)
Action(Drink(agent, cup)
  Precondition: Has(agent, cup) ∧ ContainsLiquid(cup)
  Effect: ¬Thirsty(agent) ∧ ¬ContainsLiquid(cup)
)
Action(Fetch(commander, object, recip)
  Precondition: ¬Has(recip, object) ∧ BotOwner(commander)
  Effect: Has(recip, object)
)
```

Solution:

```
; First step
Goal(¬Thirsty(Partier2))
g = ∅

; Relevant actions:
Action(Drink(agent, cup)
  Precondition: Has(agent, cup) ∧ ContainsLiquid(cup)
  Effect: ¬Thirsty(agent) ∧ ¬ContainsLiquid(cup)
)

; Groundings available
; Since we know our goal was to quench Partier2's thirst,
; then they must be our agent for the Drink action
; i.e. we choose a relevant action that can unify to our
; goal
Θ = { agent/Partier2 }

; Gives us KB g':
g' =
   1. Has(Partier2, cup')
   2. ContainsLiquid(cup')
```

Notice that we have multiple possible cups that we could unify our KB g' on, so we'll keep these partial instantiations and try to unify them later.

This is known as choosing the **most general unifier** since we don't need to unify on the cup yet.

We perform something like Skolemization to make sure that our cups are distinct, since we only need one cup to unify on ContainsLiquid(cup')

```
; Second step
g' =
   1. Has(Partier2, cup')
   2. ContainsLiquid(cup')

; Relevant actions:
Action(Fetch(commander, object, recip)
  Precondition: ¬Has(recip, object) ∧ BotOwner(commander)
  Effect: Has(recip, object)
)

; Groundings available
; Since we know that our Partier2 has to have some cup, we'll
; bind them as the recipient and object and then try to unify
Θ = { recip/Partier2, object/cup' }

; Gives us KB g'':
g'' =
    1. ContainsLiquid(cup')
    2. BotOwner(commander')

; Since we have constants Cup1 and Cup2 eligible for binding to the
; cup' placeholder, we can try each and find that Cup2 matches
; the fact in our initial state; similarly, we have only one choice
; for matching commander' with Andrew in our initial state

; [!] As it turns out, this binidng is sufficient for deriving a subset
; of our initial state:
Θ = {cup'/Cup2, commander'/Andrew}
KB =
   1. ContainsLiquid(Cup2)
   2. BotOwner(Andrew)

; So, we return our successful action sequence:
Fetch(Andrew, Cup2, Partier2) → Drink(Partier2, Cup2)
```

Forward State-space Search

Unimaginatively, forward state-space search is similar to forward chaining:

ά **Forward state-space search** starts with our initial state and attempts to use applicable actions to derive the goal state.

ň Warning! Needless to say, we'll need good heuristics to narrow down what actions we should take at a given state to guide us to a goal.

Without good heuristics, our expansions of possible actions can be totally intractable (we'll talk about heuristic formation later)

In the meantime, let's just run through our previous example with forward state-space search:

Example

t Use forward state-space search to derive the goal state from our initial state in the following specification:

```
Constants: Andrew, Partier2, Cup1, Cup2, ForneyBot3000
Goal: ¬Thirsty(Partier2)
Initial State:
  KB =
     1. Thirsty(Partier2)
     2. BotOwner(Andrew)
     3. ContainsLiquid(Cup2)
     4. Has(Partier2, Cup1)
Action(Drink(agent, cup)
  Precondition: Has(agent, cup) ∧ ContainsLiquid(cup)
  Effect: ¬Thirsty(agent) ∧ ¬ContainsLiquid(cup)
)
Action(Fetch(commander, object, recip)
  Precondition: ¬Has(recip, object) ∧ BotOwner(commander)
  Effect: Has(recip, object)
)
```

```
; Step 1
; Start at initial state
KB =
   1. Thirsty(Partier2)
   2. BotOwner(Andrew)
   3. ContainsLiquid(Cup2)
   4. Has(Partier2, Cup1)

; Choose an applicable action from the set of
; those that are applicable; in this case, only
; one is applicable:
{ Fetch(Andrew, Patier2, Cup2) }

; Applying that, we get the KB:
KB =
   1. Thirsty(Partier2)
   2. BotOwner(Andrew)
   3. ContainsLiquid(Cup2)
   4. Has(Partier2, Cup1)
   5. Has(Partier2, Cup2)
```

```
; Step 2
KB =
   1. Thirsty(Partier2)
   2. BotOwner(Andrew)
   3. ContainsLiquid(Cup2)
   4. Has(Partier2, Cup1)
   5. Has(Partier2, Cup2)

; Choose an applicable action from the set of
; those that are applicable; again, only one
; is possible:
{ Drink(Partier2, Cup2) }

; Applying that, we get the KB:
KB =
   1. BotOwner(Andrew)
   2. Has(Partier2, Cup1)
   3. Has(Partier2, Cup2)

; [!] This is consistent with our goal state, so we
; return the action sequence:
Fetch(Andrew, Cup2, Partier2) → Drink(Partier2, Cup2)
```

Example

t Note the additional exploration options we would have had to consider if we added the following to our initial state:
(5) Thirsty(Andrew)
(6) ContainsLiquid(Cup3)

á Note that both forward and backward state-space searches are **sound and complete** for the same reasons that forward and backward chaining were: they derive only those fluents that are entailed by allowable actions from the initial state.

So, the trick, it would seem, is in choosing a good heuristic to avoid unnecessary action explorations. Enter the planning graph...

Planning Graphs

Clearly the choice of a good heuristic is necessary to adequately navigate a planning search space, especially if the number of constants and fluents is very large.

As it turns out, however, the fact that backward search employs state sets to navigate its planning, finding good heuristics is generally quite difficult.

For this reason, forward state-space search is largely the preferred method, and planning graphs represent a useful heuristic for reducing the search space.

á A **planning graph** is a directed graph organized into levels that alternate between state and action levels.

So, we have the following decomposition of entities at each level where S_i represents the state levels and A_i the action levels:

- At level S_i, all fluents that **might possibly** be accessible (from some sequence of actions) in i steps appear (NOTE: we never list a fluent that might not possibly exist at level i)

- At level A_i, all actions that **might possibly** have their preconditions met given the previous state possibilities S_i appear (NOTE: we never list an action that might not possibly exist at level i)
- **Mutex** links illustrate mutually exclusive actions and fluents (i.e., if we were to choose one action / arrive at one fluent, then we necessarily could not have chosen any to which a mutex link exists)

The whole point of planning graphs is to prevent us from having to combinitorically choose one action after another and see if our effort reaches fruition.

Although imperfect (e.g., a fluent will never appear too late, but it might show up too early than actually possible), remember that planning graphs are a heuristic strategy...

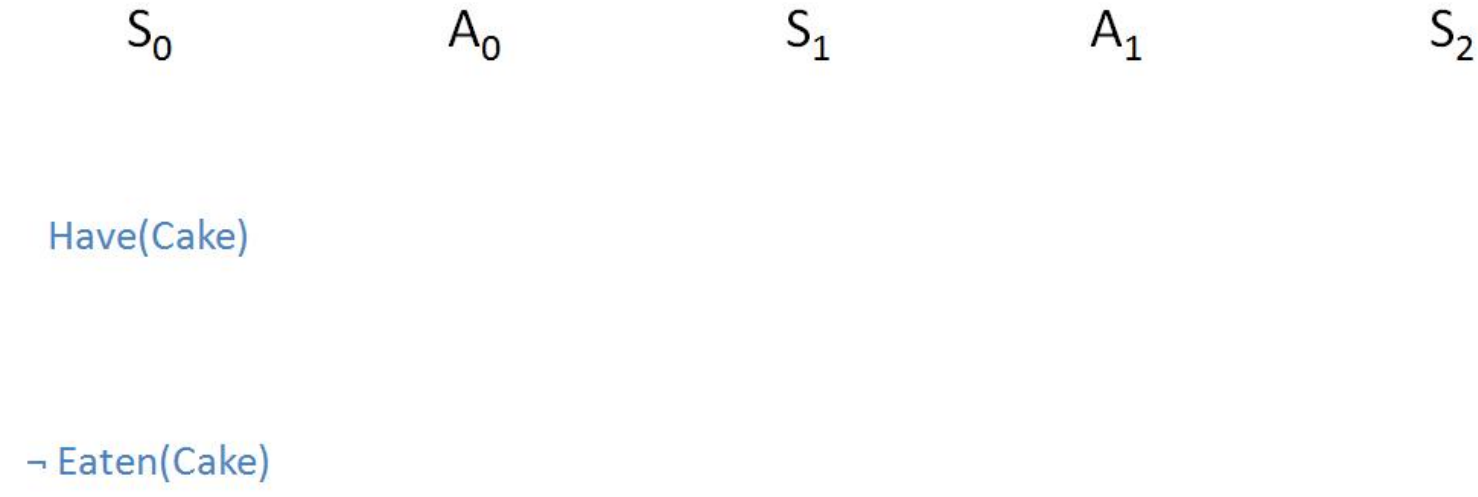
This means that they're meant to give us an estimate of how hard it is to attain a literal from the initial state.

How to Have a Planning Graph and Eat it Too

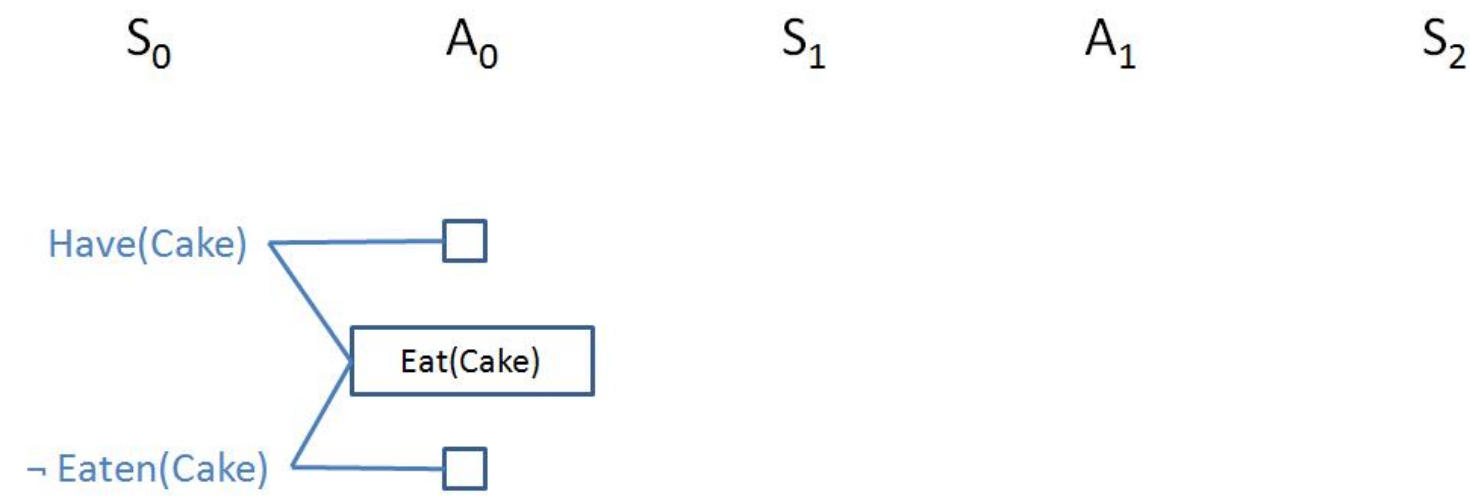
...oh wait, I think something got switched there... we're going to dissect the book's example of having your cake and eating it too, as follows:

```
Constants: Cake
Goals: (Have(Cake) ∧ Eaten(Cake))
KB =
  1. Have(Cake)
  (2. ¬Eaten(Cake))*

; * Notice that we still illustrate negative fluents in our
; planning graphs explicitly, even though they might be
; generated from the closed-world assumption

Action(Eat(Cake)
  Precondition:
    Have(Cake)
  Effect:
    ¬Have(Cake) ∧ Eaten(Cake))

Action(Bake(Cake)
  Precondition:
    ¬Have(Cake)
  Effect:
    Have(Cake))
```

H **Setup:** add your initial state to S_0, the first state level, making a node for each fluent.



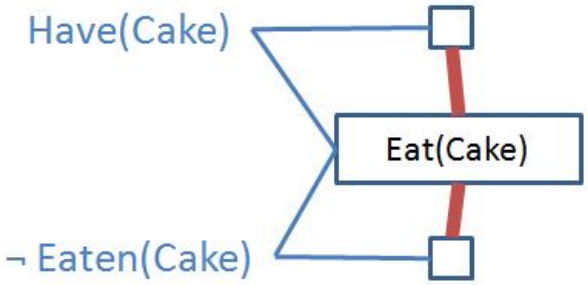
H **Creating Action Levels:** for action level i, (1) add a persistence action (small box) for each fluent in S_i, and (2) add any actions that can be performed using any combo of fluents from S_i.



H **Adding Action Mutex Links:** there are three criteria for adding a mutex (mutual exclusivity) between actions:

- **Inconsistent effects:** action1 sets X and action2 sets ¬X are mutually exclusive.
- **Interference:** an effect of action1 is the negation of a precondition for action2
- **Competing needs:** a precondition of action1 is mutex with a precondition of action2

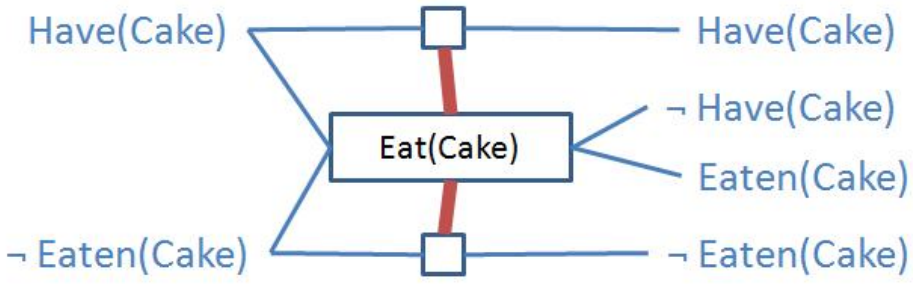
S_0 A_0 S_1 A_1 S_2



à What mutex rule is represented in our addition of the two mutex additions above?

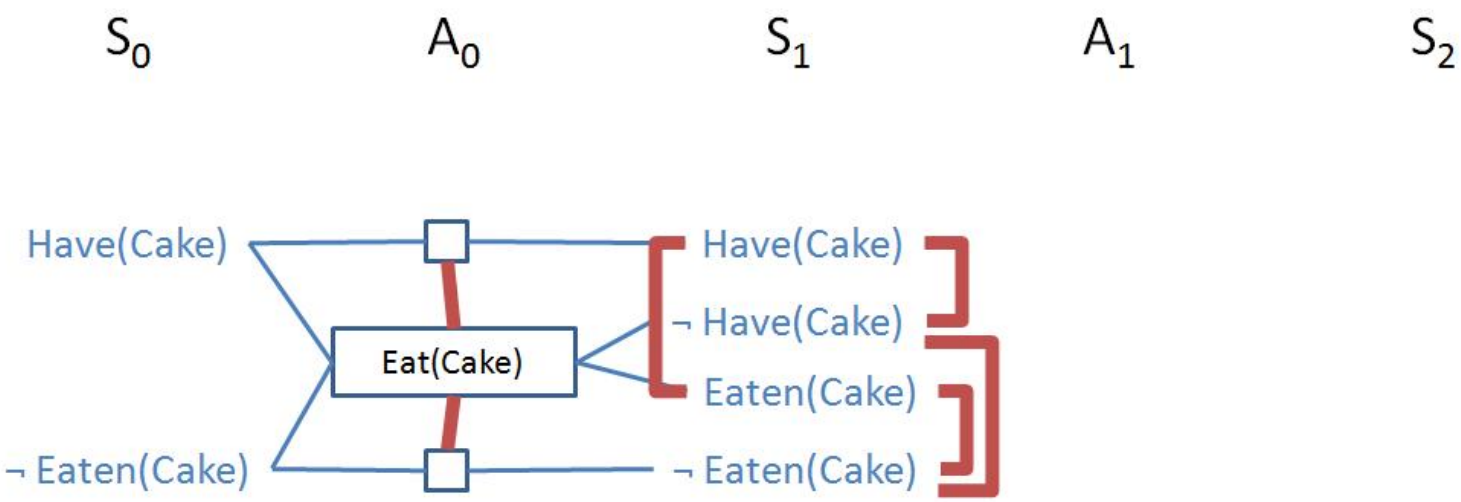
H **Creating State Levels:** add fluents resulting from each action (including persistence actions) at the previous action level.

S_0 A_0 S_1 A_1 S_2



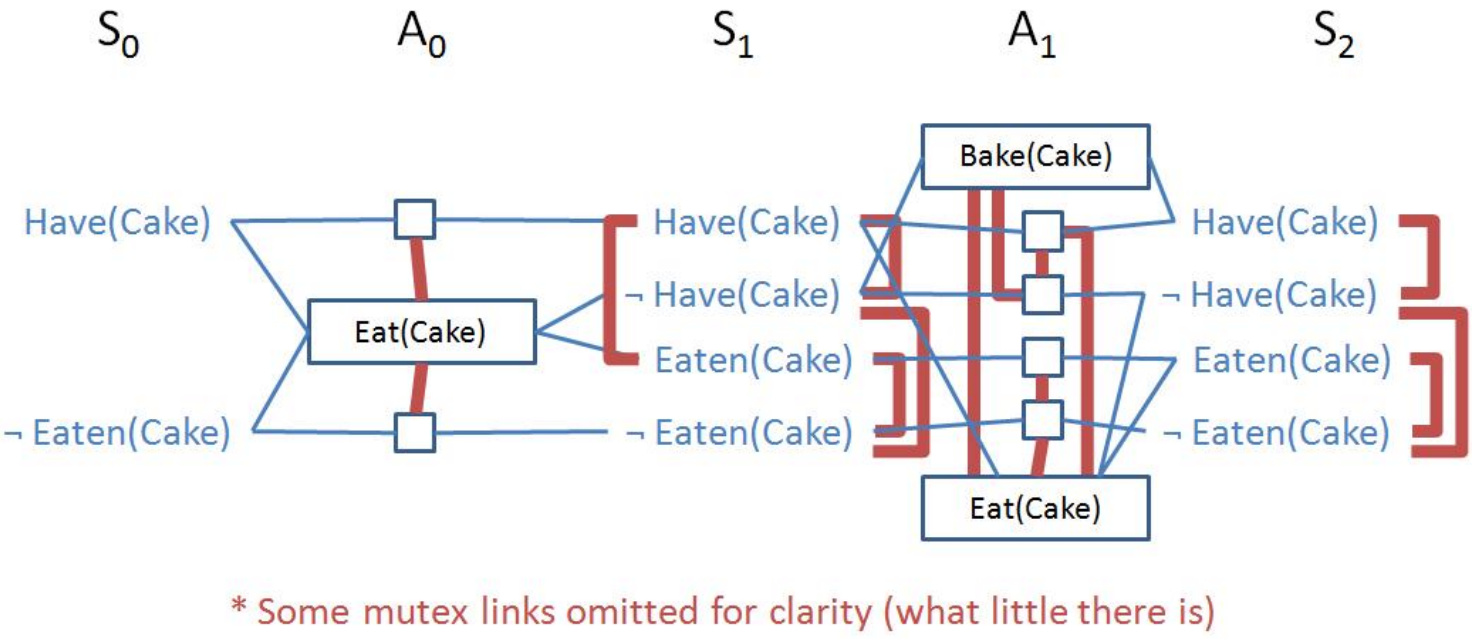
H **Adding Fluent Mutex Links:** there are two criteria for adding a mutex (mutual exclusivity) between fluents:

- If the two fluents are contradictory (e.g., X and ¬X)
- If **every** pair of actions that could produce them is also mutex



à Why were each of the above 4 fluent mutexes added?

H We stop generating new levels when we have **leveled off**, meaning that two consecutive state levels (S_i and S_(i+1)) are identical.



We observe that S_1 and S_2 are identical, so we've leveled off and do not continue generating levels.

Using the Planning Graph

Alright, so we've generated our planning graph, have all of our mutexes in place... now what? ...what were we trying to do again?

Oh, right... reduce the number of action explorations our forward state space search will have to check to derive a plan if it exists.

á The **set-level** heuristic finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutex.

So, once we have our planning graph, we can use set-level to give us the following:

- We are guaranteed that if we **never** derive the goal state's literals before we level off that there exists no solution plan.
- We are NOT guaranteed that if we derive the goal state's literals that a plan certainly exists; only that it might!*
- Lemma: We only compare mutual exclusions between two actions / fluents at a time, leaving more complex exclusions between 3 or more actions / fluents an unaccounted-for possibility.

à So why don't we check for 3-consistency, 4-consistency, etc. in our planning graph actions / fluents?

The same reason explains our persistence actions in planning graphs--they might not be completely accurate, but they simplify the heuristic by making sure it doesn't become too expensive.

Hierarchical Planning

Our previous discussion of planning gave us the basic elements to enact a plan, but with very low-level action effects that were atomic and did little abstraction.

It would be nice, then, if we were capable of programming our intelligent systems to have some level of action abstraction so that we don't have to spell every little thing out.

á A **hierarchical decomposition** of actions attempts to group small actions into larger, more abstract ones to make large planning problems tractable.

á **High-level actions (HLAs)** are defined as 1 or more refinements that consist of some number of preconditions and then some number of steps to satisfy that refinement.

á A refinement **step** may be either another (recursively defined) HLA or a primitive action.

á A **primitive action** is an atomic action instruction like we've dealt with previously (Chapter 10).

```
; Example of HLA:
Refinement(Go(Home, SFO)
  Preconditions:
    Fueled(Car)
  Steps:
    [Drive(Car, Home, SFOLongTermParking),
     Shuttle(SFOLongTermParking, SFO)]
)
```

Notice that our steps are primitive actions in the above case that are performed in sequence.

á A list of steps containing only primitive actions is called an **implementation**.

So, in order to use HLA to find plans from an initial state to a goal, we search for an implementation that obeys all preconditions and leads to the goal state.

Example

t Find an implementation using the following HLAs that allow us to make tea!

```
Constants = { Water, Tea }
KB =
   1. IsLiquid(Water)
   2. Temperature(Water, 68) ; Farenheit, room temp

g = Steep(Tea, Water)

; HLAs:
Refinement(Brew(tea, water)
  Steps:
    [Boil(water), Steep(tea, water)])

Refinement(Boil(liquid)
  Precondition:
    IsLiquid(liquid) ∧ Temperature(liquid, 100)
  Steps: [])

Refinement(Boil(liquid)
  Precondition:
    IsLiquid(liquid)
  Steps:
    [ApplyHeat(liquid), Boil(liquid)])

Refinement(ApplyHeat(object)
  Precondition:
    Temperature(object, x)
  Steps:
    [Temperature(object, x+1)])
```

(example credit to Evan Lloyd)

Nondeterministic Planning

Previously, our knowledge bases have operated under the closed world assumption because we assumed that we had full percept of our intelligent agent's environment...

Of course, in practice, this might not always be the case, and we should treat **uncertainty** with special deference in our reasoning system.

We can divide this process into three primary diversions from our traditional Action representations:

H Fluents can be true, false, or unknown (we remove the closed world assumption in favor of the open world one in which uncertainty is possible)

Now, our state semantics have been modified such that we no longer know that a missing fluent is automatically false.

So, we need to add mechanisms to reduce uncertainty:

H Add actions that do nothing except learn the values of unknown fluents. We'll call these actions **Percepts**.

```
; Example percepts
Percept(Color(x, c)
  Precondition:
    Object(x) ∧ InView(x))
Percept(ContainsLiquid(x, liquid)
  Precondition:
    Cup(x) ∧ InView(x))

; Might also need a mechanism for causing
; an object to come into view (one object
; at a time)
Action(LookAt(x),
  Precondition:
    InView(y) ∧ (x ≠ y)
  Effect:
    InView(x) ∧ ¬InView(y))
```

By adding percepts, we assume that our agent has sensors to set the proper parameters from the perception findings, and thus give us more information about our environment's objects.

Finally, by using these percepts, we can approach our planning problem to react to different scenarios with appropriate actions.

We can react to unknown values and randomness using contingent plans, which are really just if-then-else actions based on the knowledge from our percepts.

```
; Contingent Planning
Constants: Andrew, Partier2, Cup1, Cup2, ForneyBot3000
; [!] No longer closed world assumption!
KB =
   1. Thirsty(Partier2)
   2. BotOwner(Andrew)
   3. ContainsLiquid(Cup2)
   4. Has(Partier2, Cup1)

Action(LookAt(x),
  Precondition:
    InView(y) ∧ (x ≠ y)
  Effect:
    InView(x) ∧ ¬InView(y))

Percept(ContainsLiquid(x, liquid)
  Precondition:
    Cup(x) ∧ InView(x))

; ...other actions from example omitted

; Contingent plan:
[Observe(Cup1)
  if ContainsLiquid(Cup1)
  then Drink(Partier2, Cup1)
  else [Observe(Cup2)
         if ContainsLiquid(Cup2)
         then Fetch... ; too lazy to fill out
         else [Explode(ForneyBot3000)] ; still some kinks to work out...
       ]
]
```