AI: Representation and Problem Solving

Classical Planning
or Symbolic Planning

Instructor: Tuomas Sandholm and Nihar Shah

Slide credits: CMU AI

Image: Shutterstock

Changing a Tire

You start with a flat tire on the axle of your car and a spare in the trunk

You can perform the following actions:

Remove(tire,axle)

Put(tire,axle)

Remove(tire,trunk)

Put(tire,trunk)

Your goal is to replace the flat tire on the axle with the spare in the trunk.

How would solve this problem with search, e.g., BFS?

Changing a Tire

You start with a flat tire on the axle of your car and a spare in the trunk

You can perform the following actions:

Remove(tire,axle)

Put(tire,axle)

Remove(tire,trunk)

Put(tire,trunk)

Your goal is to replace the flat tire with the spare.

How would solve this problem with logic?

Search, Logic, and Classical Planning

Search Planning

- State representation that changes as you act

Propositional Logic Planning

- Represent world with Boolean propositions and successor state axioms
- Different symbols for different time points

Classical Planning

- Represent the world with objects and Boolean predicates
- State changes as you act

Idea of Classical Planning

Represent objects/values separately from the state (instances)

Define predicates as true/false functions over the objects

States are conjunctions of predicates

Goals are conjunctions of predicates

Poll 1

Which predicates apply to this state? (Select all that apply)

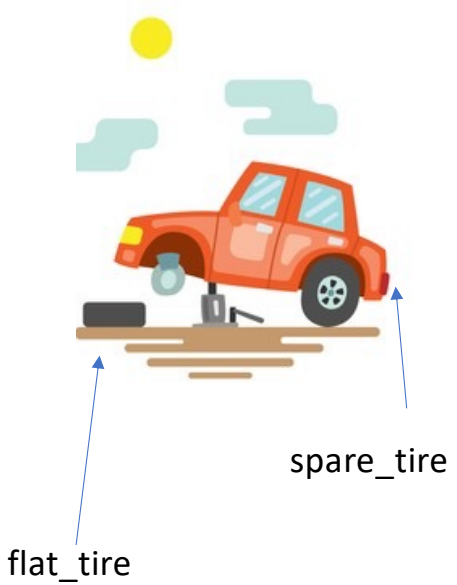
Instances:

TIRES: flat_tire, spare_tire

LOCS: axle, trunk, ground

Predicates:

1) On(flat_tire, axle)
2) On(spare_tire, axle)
3) On(flat_tire, trunk)
4) On(spare_tire, trunk)
5) On(flat_tire, ground)
6) On(spare_tire, ground)
7) Empty(trunk)
8) Empty(axle)



Full State Description

Instances:

TIRES: flat_tire, spare_tire

LOCS: axle, trunk, ground

Predicates:

On(spare_tire, trunk)

On(flat_tire, ground)

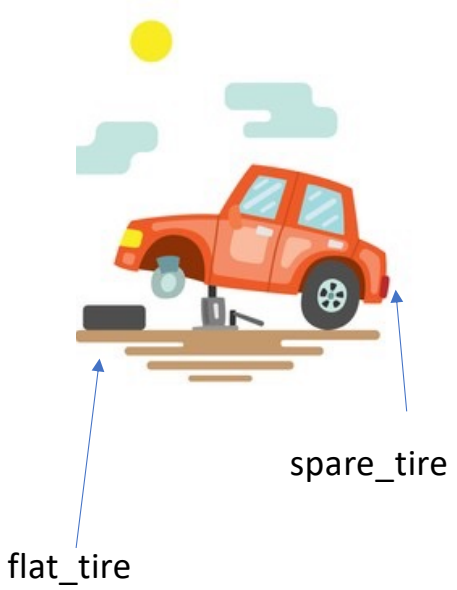
Empty(axle)

Optional: ~On(flat_tire, axle), ~On(spare_tire, axle),

~On(flat_tire, trunk), ~On(spare_tire, ground),

~Empty(trunk)

Why Optional?



Operators

Operators change the state by adding/deleting predicates

Preconditions:

Actions can be applied only if all precondition predicates are true in the current state

Effects:

New state is a copy of the current predicates with the addition or deletion of specified predicates

Unlike the successor-state axioms, we do not explicitly represent time and we can use our objects and predicates to more easily scale to new more complex problems (e.g., new objects, predicates, and operators).

Rules of Tire Fixing

A tire can be removed from an axle if it is on the axle (precondition)

Effect: the tire is on the ground

A tire can be put on an axle if it is on the ground (precondition)

Effect: the tire is on the axle

A tire can be removed from the trunk if it is in the trunk (precondition)

Effect: the tire is on the ground

A tire can be put in the trunk if it is on the ground (precondition)

Effect: the tire is in the trunk

Rules of Tire Fixing

A tire can be removed from an axle/trunk if it is on there (precondition)

Effect: the tire is on the ground

A tire can be put on an axle/trunk if it is on the ground (precondition)

Effect: the tire is on the axle/trunk

The trunk/axle can be empty if nothing is on it

At most one thing can be on the trunk/axle

NOTE: A successor state axiom in logic would have to be defined for EACH tire and EACH loc, but in classical planning, these rules are defined for an object type

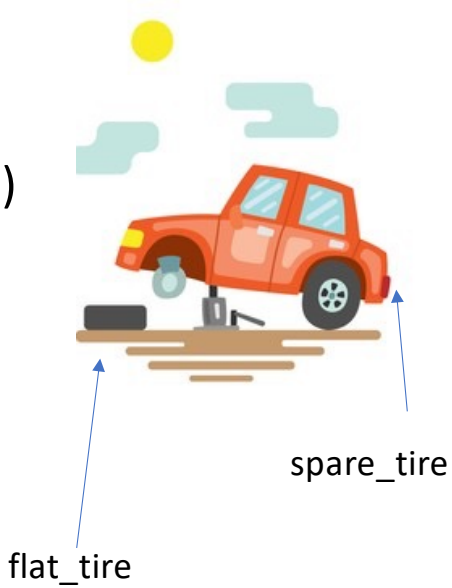
Remove Tire from Trunk (State Transition)

Instances:
Tires: flat, spare
Locs: axle, trunk, ground

Possible Predicates:
On(tire, loc)

Old State:
Empty(axle)
On(flat,ground)
On(spare,trunk)

New State:
Empty(axle)
On(flat,ground)
On(spare,ground)
Empty(trunk)



Remove Tire from Trunk (State Transition)

Instances:
Tires: flat, spare
Locs: axle, trunk, ground

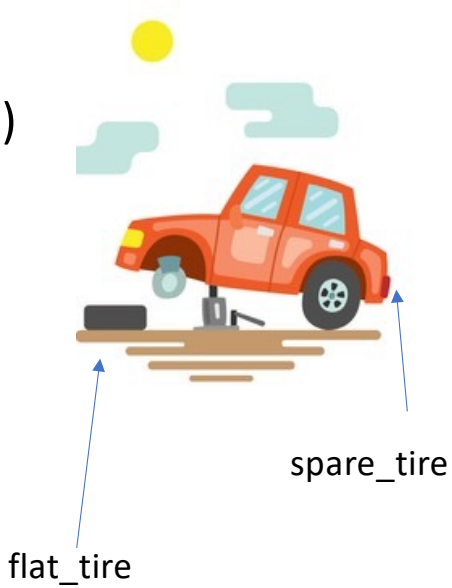
Possible Predicates:
On(tire, loc)

Old State:
On(flat,ground)
On(spare,trunk)

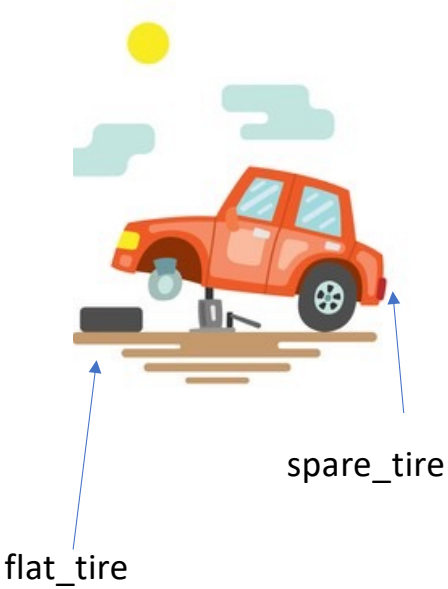
New State:
On(flat,ground)
On(spare,ground)
Empty(trunk)

Add Effect:
On(spare,ground)
Empty(trunk)

Delete Effect:
On(spare,trunk)



Operator: Remove(tire, trunk)



Preconditions

On(tire,trunk)

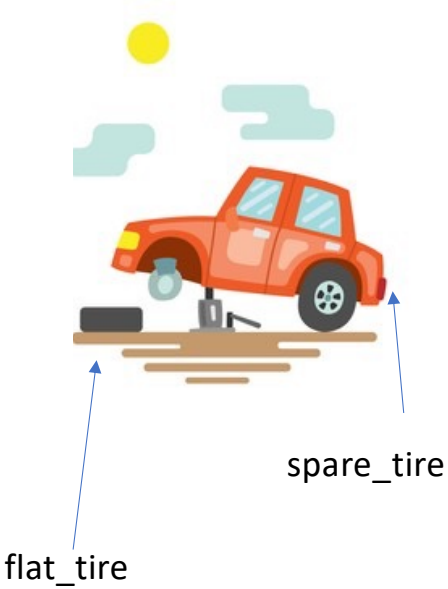
Effects

Add On(tire,ground)

Empty(trunk)

Delete On(tire,trunk)

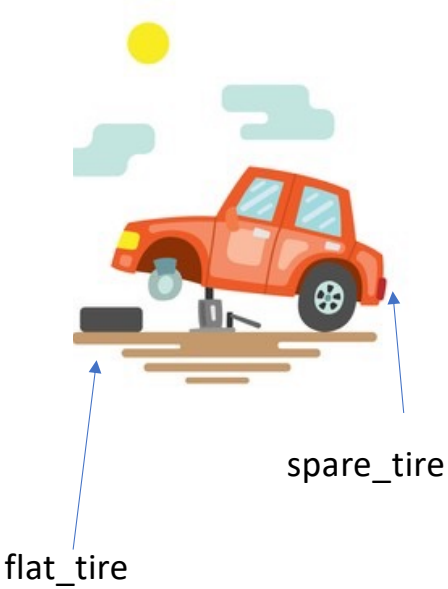
Operator: Remove(tire, loc)



Preconditions	Effects
On(tire,loc)	Add On(tire,ground)
	Empty(loc)
	Delete On(tire,loc)

WAIT: THIS DOESN'T WORK! WHAT IF LOC=GROUND???

Operator: Remove(tire, loc)



Preconditions

On(tire,loc)

loc != ground

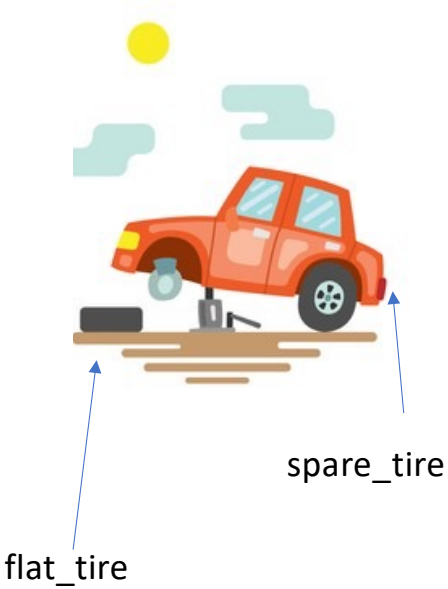
Effects

Add On(tire,ground)

Empty(loc)

Delete On(tire,loc)

Operator: Remove(tire, loc)



Preconditions

On(tire,loc)

~On(tire,ground)

Effects

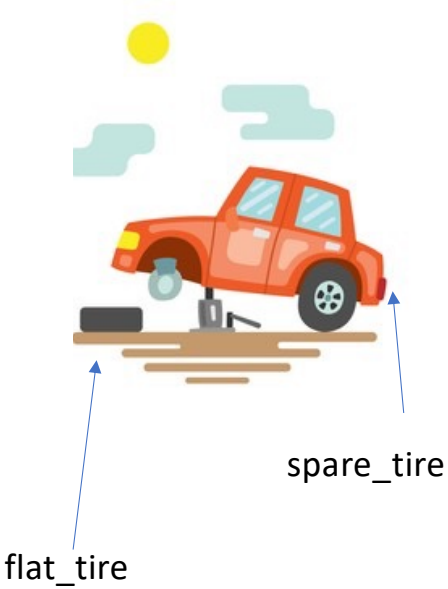
Add On(tire,ground)

Empty(loc)

Delete On(tire,loc)

~On(tire, ground)

Operator: Put(tire, loc)



Preconditions	Effects
On(tire,ground)	Add On(tire,loc)
loc != ground	Delete On(tire,ground)
Empty(loc)	Empty(loc)

Operators for Fixing a Tire

Put(tire,loc):

Pre: On(tire,ground), loc != ground
Empty(loc)

Add: On(tire,loc)

Delete: On(tire,ground), Empty(loc)

Remove(tire,loc):

Pre: On(tire,loc), loc != ground

Add: On(tire,ground), Empty(loc)

Delete: On(tire,loc)

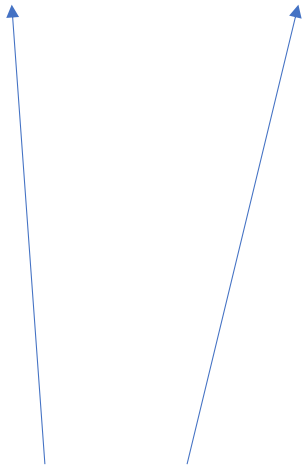
Why does ground get referenced directly instead of as a location like axle and trunk?
Why do we add Empty but not add ~Empty when it's full?

Example Matching Operators

On(flat, axle) AND On(spare, trunk)

Example Matching Operators

On(flat, axle) AND On(spare, trunk)



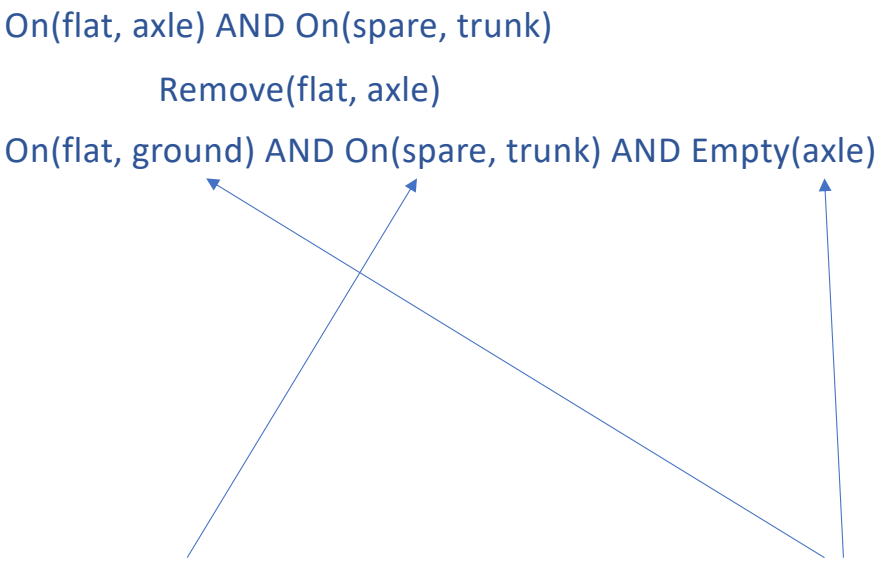
Remove(tire,loc):

Pre: On(tire,loc), loc != ground

Add: On(tire,ground), Empty(loc)

Delete: On(tire,loc)

Example Matching Operators



Remove(tire,loc):
Pre: On(tire,loc), loc != ground
Add: On(tire,ground), Empty(loc)
Delete: On(tire,loc)

Put(tire,loc):
Pre: On(tire,ground), loc != ground
Empty(loc)
Add: On(tire,loc)
Delete: On(tire,ground), Empty(loc)

Example Matching Operators

On(flat, axle) AND On(spare, trunk)

Remove(flat, axle)

On(flat, ground) AND On(spare, trunk) AND Empty(axle)

Remove(spare, trunk)

Remove(tire,loc):
Pre: On(tire,loc), loc != ground
Add: On(tire,ground), Empty(loc)
Delete: On(tire,loc)

Put(tire,loc):
Pre: On(tire,ground), loc != ground
Empty(loc)
Add: On(tire,loc)
Delete: On(tire,ground), Empty(loc)

Example Matching Operators

On(flat, axle) AND On(spare, trunk)

Remove(flat, axle)

On(flat, ground) AND On(spare, trunk) AND Empty(axle)

Remove(spare, trunk)

On(flat, ground) AND On(spare, ground) AND Empty(axle) AND Empty(trunk)

Remove(tire,loc):
Pre: On(tire,loc), loc != ground
Add: On(tire,ground), Empty(loc)
Delete: On(tire,loc)

Put(tire,loc):
Pre: On(tire,ground), loc != ground
Empty(loc)
Add: On(tire,loc)
Delete: On(tire,ground), Empty(loc)

Example Matching Operators

On(flat, axle) AND On(spare, trunk)

Remove(flat, axle)

On(flat, ground) AND On(spare, trunk) AND Empty(axle)

Remove(spare, trunk)

On(flat, ground) AND On(spare, ground) AND Empty(axle) AND Empty(trunk)

Put(spare, axle)

On(flat, ground) AND On(spare, axle) AND Empty(trunk)

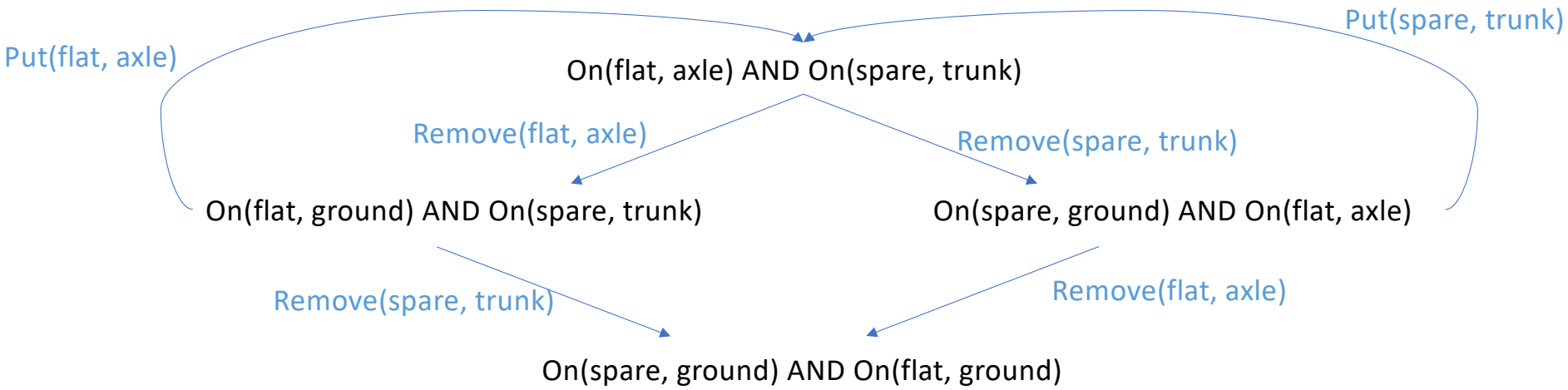
Put(flat, trunk)

On(flat, trunk) AND On(spare, axle)

Remove(tire,loc):
Pre: On(tire,loc), loc != ground
Add: On(tire,ground), Empty(loc)
Delete: On(tire,loc)

Put(tire,loc):
Pre: On(tire,ground), loc != ground
Empty(loc)
Add: On(tire,loc)
Delete: On(tire,ground), Empty(loc)

Matching Operators as Search



On(spare, axle) AND On(flat, trunk)

Finding Plans with Symbolic Representations

Breadth-First Search

Sound? Yes

Complete? Yes

Optimal? Yes, if equal costs of actions

Soundness - all solutions found are legal plans

Completeness - a solution can be found whenever one actually exists

Optimality - the order in which solutions are found is consistent with some measure of plan quality

Linear Planning

Since we have a conjunction of goal predicates, let's try to solve one at a time

- Maintain a stack of achievable goals
- Use BFS (or anything else) to find a plan to achieve that single goal
- Add a goal back on the stack if a later change makes it violated

Linear Planning Spare Tire Example

Start State
On(flat,axle)
On(spare,trunk)

Goal Stack
On(flat,trunk)
On(spare,axle)

Action Plan:
On(flat,trunk)
    Remove(flat,axle)
    Remove(spare,trunk)
    Put(flat,trunk)
On(spare,axle)
    Put(spare,axle)

Spare Tire Example with Tools

Suppose that you needed a tool like a wrench to remove and put the tire on the axle

Suppose that the wrench needed to be stored in a tool box as a goal

Start State
On(flat,axle)
On(spare,trunk)
In(wrench,box)

Goal Stack
On(flat,trunk)
In(wrench,box)
On(spare,axle)

Action Plan:
On(flat,trunk)
    Remove(wrench,box)
    Remove(flat,axle)
    Remove(spare,trunk)
    Put(flat,trunk)
In(wrench,box)
    Put(wrench,box)
On(spare,axle)
    Remove(wrench,box)
    Put(spare,axle)
In(wrench,box)
    Put(wrench,box)

What happened?

Is linear planning sound?
Is linear planning complete?
Is linear planning optimal?

Sussman's Anomaly

A weakness of linear planning is that sometimes you get long plans

- One goal can be achieved
- The second goal immediately undoes it

In fact, there are some problems for which solving goals one at a time will never result in a feasible plan.

Note: This isn't just a choice of goals. The anomaly can happen no matter which goal is first

Non-Linear Planning

Interleave goals to achieve plans

- Maintain a set of unachieved goals
- Search all interleavings of goals (in practice, this is very hard!)
- Add a goal back to the set if a later change makes it violated

Heuristics – Search Graph Representation

For planning, the state graph's size is potentially exponential in the number of predicates

It is possible that each action changes exactly one predicate

Can we reduce the size of the planning graph?

GraphPlan

GraphPlan is a relaxation of other classical planning search techniques

The GraphPlan search graph space is linear in the number of predicates

GraphPlan and GraphPlan Graph Representation

Graphplan graphs contain two types of layers

- Proposition layers – all reachable predicates
- Action layers – actions that could be taken
- Both layers represent one time step

GraphPlan algorithm includes two subtasks

- **Extend**: One time step (two layers) in the graphplan graph
- **Search**: Find a valid plan in the graphplan graph

GraphPlan finds a plan or proves that no plan has fewer time steps

- Each time step can contain multiple actions

Building a GraphPlan Graph

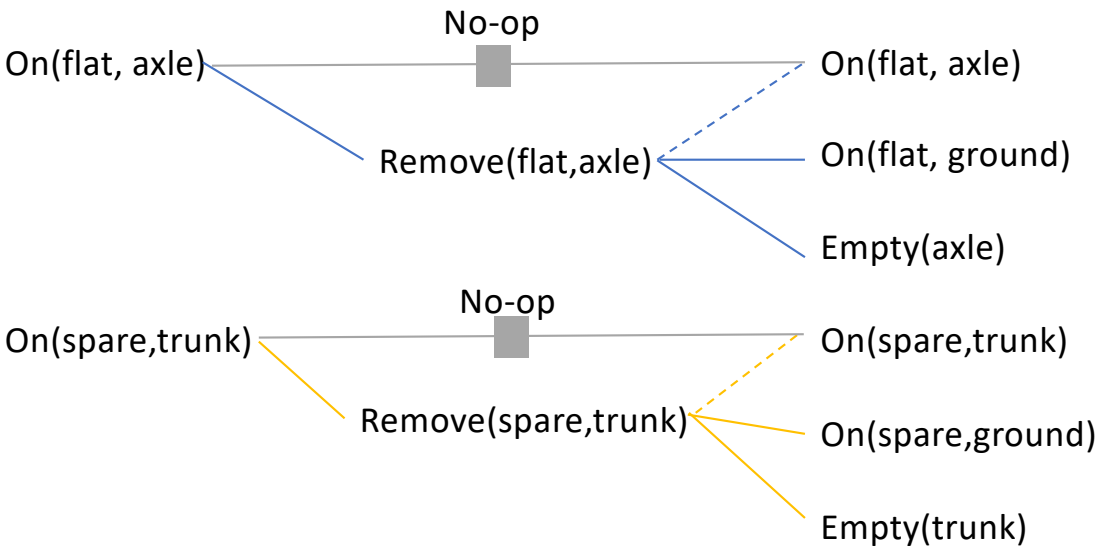
Start the planning graph with all starting predicates

On(flat, axle)

On(spare,trunk)

Building a GraphPlan Graph

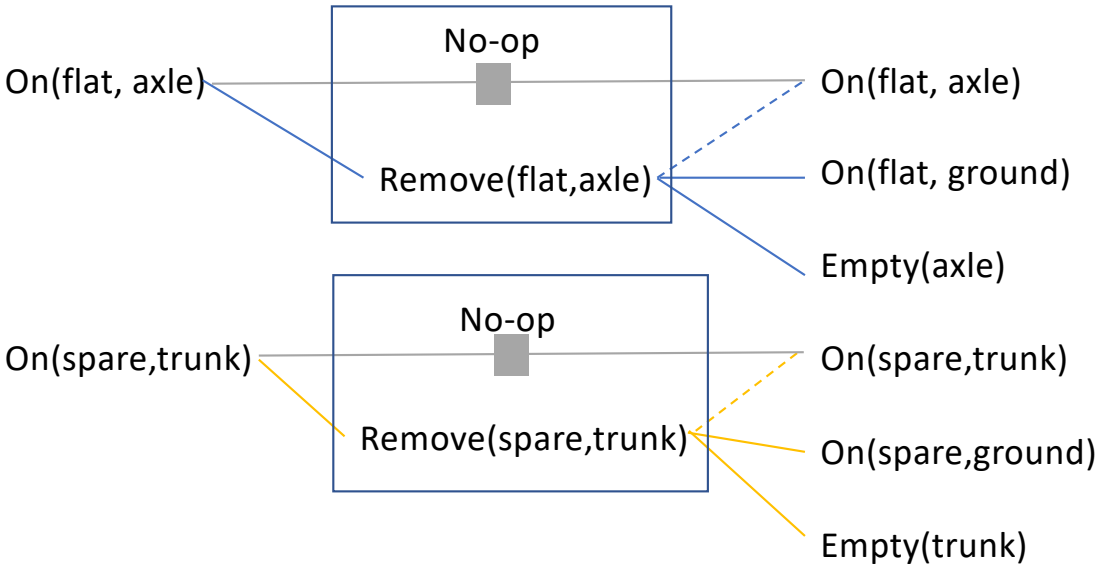
Extend the graph with all applicable actions. Designate all effects (add/delete)



Building a GraphPlan Graph

The search could be structured as a binary CSP where the variables are the actions at each action level. I will show a better search in a bit.

Search the graph to find feasible solutions. Determine mutually exclusive actions.



Actions A and B are ***exclusive (mutex)*** at action-level *i*, if:

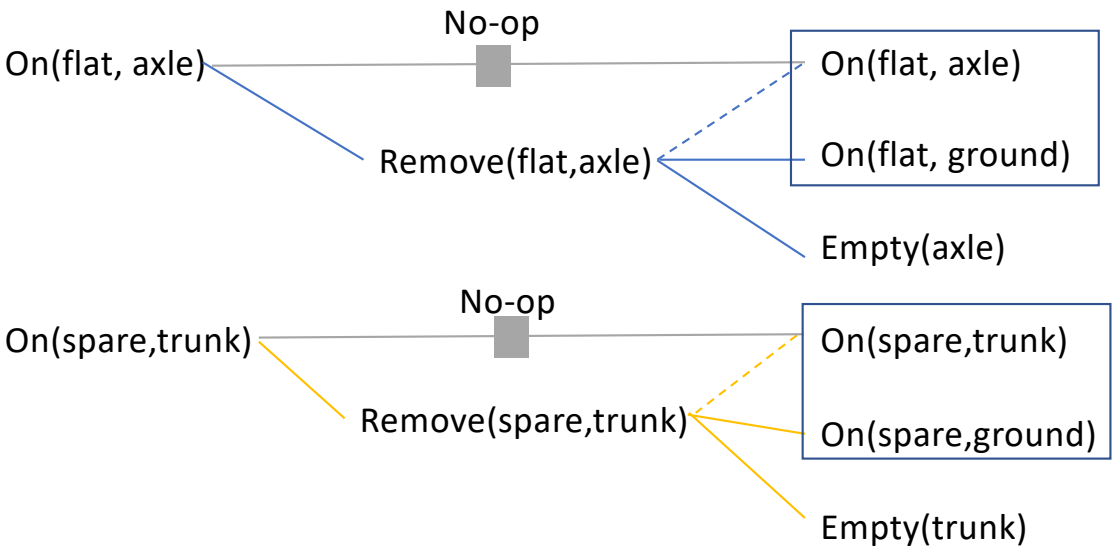
Interference: one action effect deletes or negates a **precondition** of the other

Inconsistency: one action effect deletes or negates the **effect** of the other

Competing Needs: the actions have preconditions that are mutex in prev. proposition-level

Building a GraphPlan Graph

Search the graph to find feasible solutions. Determine mutually exclusive predicates.



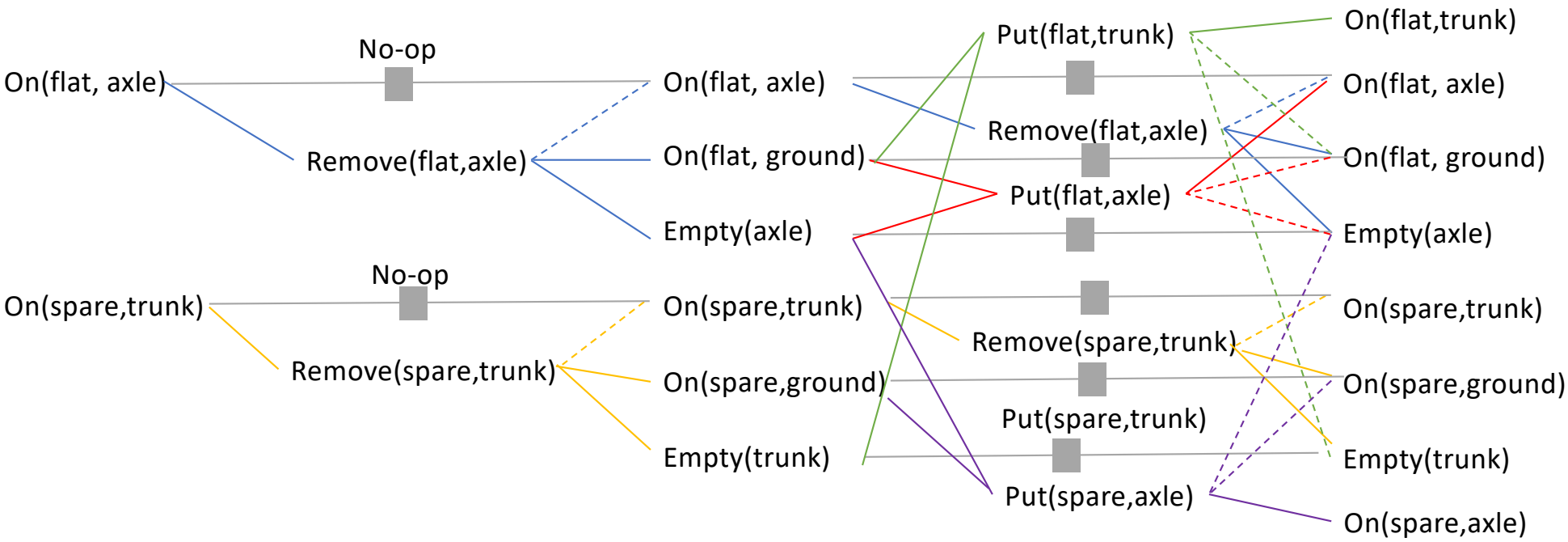
Propositions P and Q are ***exclusive (mutex)*** at proposition-level *i*, if:

Negation: They are the negation of each other or can't appear at the same time in a plan

Inconsistent Support: if there is no set of non-mutex actions in action layer i-1 that produce both P and Q

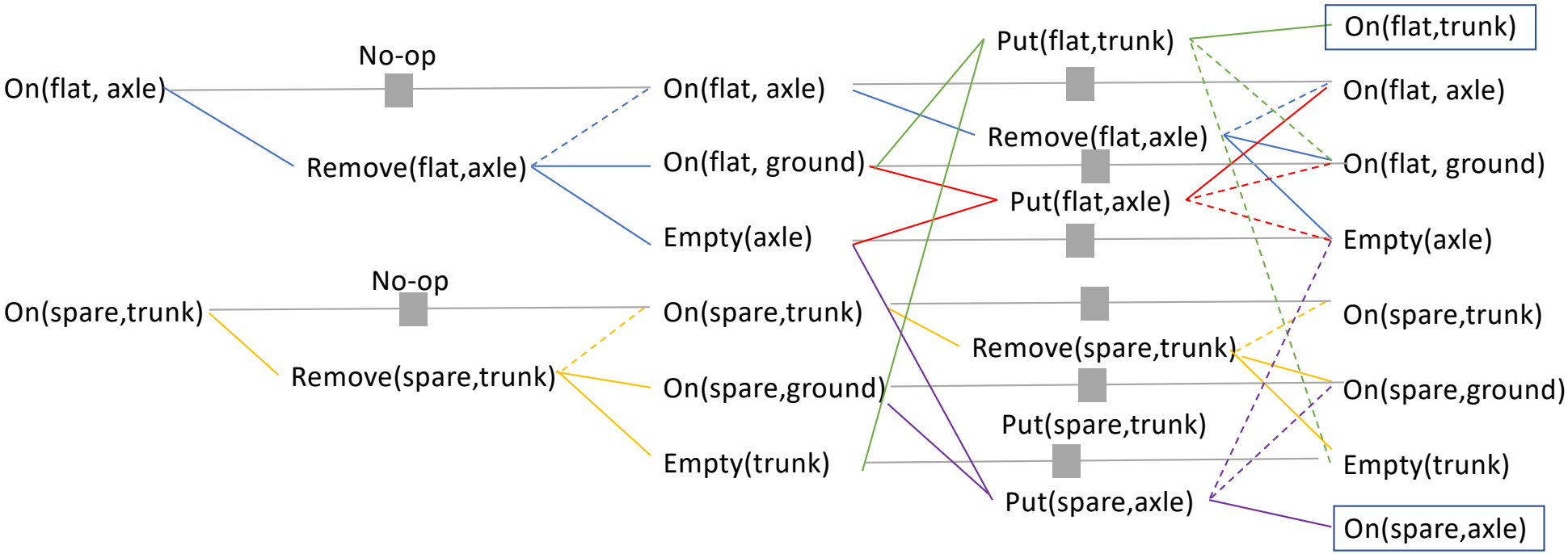
Building a GraphPlan Graph

Extend the graph with all applicable actions. Designate all effects (add/delete)



Building a GraphPlan Graph

Search the graph to find feasible solutions. Determine mutually exclusive actions/predicates.



The goal predicates are both present but are they mutex?

Can we perform this plan in 2 steps?

GraphPlan Big Picture

Construct a Graphplan graph as an approximation of the planning graph in polynomial space

The approximation: we do not delete any predicates that were EVER true since the start of the search. The GraphPlan graph computes the **possibly reachable** states although they aren't necessarily **feasible**

-> We can match multiple actions in one timestep if preconditions all match

Finds shorter than optimal plans if actions are sequential

How do we fix this?

-> We have to handle the case that plans that couldn't be actually executed because one action negates another

GraphPlan High Level Algorithm

Initialize first proposition layer with proposition from initial state
Loop
    Extend the GraphPlan graph by adding an action level and
        then a proposition level
    If graph has leveled off (no new propositions added from previous level):
        Return NO SOLUTION
    If all propositions in goal are present w/o mutex in the added proposition level:
        Search for a possible plan in the planning graph
            (see solution algorithm)
    If plan found, return with that plan

Searching the GraphPlan Graph

- Search states: set of propositions in a proposition layer BUT it also includes an additional list of "goals" for that state. The "goals" for this initial state will be the set of planning goals propositions, but as you'll see below that will change as we search backwards.
- Initial search state: the set of propositions from the last level of the planning graph. We also keep track of the goals for this state, which are the goal propositions for the planning problem. Call this level S_i for now.
- Search actions: any subset of operators in the preceding action level, A_{i-1}, where none of these actions are conflicting at that level and their collective effects include the full set of goals we are considering in S_i
- Search transitions: lead to a next search state with the set of propositions in S_{i-1} and the "goals" for this state are the preconditions for all of the operators in the search action that was selected.
- Search goal: We keep searching to try to get to S_0, where the "goals" of that search state are all satisfied by S_0.

GraphPlan Takeaways

GraphPlan is a relaxation of other classical planning search techniques like BFS

It creates a different kind of graph that allows you to decide that **no plan is reachable at a given depth**.

If it finds a **reachable** solution, it may not be a feasible solution because it **allows you to perform multiple actions at the same time**.

- Can be made into a complete planning algorithm by continuing to add layers until either a feasible plan is found or a memoization called no-good set levels off too, in which case there is no feasible plan
- Each no-good represents a combination of goals that cannot be achieved by a given level of the graph
- No-goods are stored in a has table

The search graph is linear space in the number of predicates

Know the differences between the mutex conditions!!

We provide the GraphPlan implementation

In the programming assignment, you will create the representation, which will be passed into our GraphPlan implementation

In written assignments, you'll be asked to assess the graph plan graph for mutexes and goals.

Implementing Symbolic Representations

Literals: Each thing/object in our model

i = Instance("name",TYPE)

Variables: Can take on any TYPE thing

v = Variable("v_name",TYPE)

Spare Tire Example:

Instances: "flat", "spare" of type TIRE
Variable: "tire" of type TIRE
In an operator, tire can take on the value of any TIRE instance

Instances: "axle", "trunk", "ground" of type LOC
Variable: "loc" of type LOC
In an operator, loc can take on the value of any LOC instance

Implementing Symbolic Representations

Literals: Each thing/object in our model

i_spare = Instance(“spare”,TIRE), i_flat = Instance(“flat”,TIRE)

Variables: Can take on any TYPE thing

v_tire = Variable(“tire”,TIRE)

ALERT: no two literals nor variables can have the same string name!!

Implementing Symbolic Representations

Literals: Each thing/object in our model

i_spare = Instance("spare",TIRE), i_flat = Instance("flat",TIRE)

Variables: Can take on any TYPE thing

v_tire = Variable("tire",TIRE)

ALERT: no two literals nor variables can have the same string name!!

Propositions

Proposition("on", v_tire, v_loc) matches any tire and any loc

Proposition("on", v_tire, i_ground) matches any tire + the ground instance

Proposition("on", i_spare, i_axle) matches the spare tire and axle

Initial State and Goal State

Create lists of Propositions as the initial state and goal state (conjunctions)

initial = [Proposition(“on”, i_spare, i_trunk), Proposition(“on”, i_flat, i_axle)]

goal = [Proposition(“on”, i_spare, i_axle), Proposition(“on”, i_flat, i_trunk)]

Implementing Symbolic Representations

Operators: the actions we take change state

```
put = Operator("put", #name
        [Proposition("on", v_tire, i_ground), #preconditions
         v_loc != i_ground,
         Proposition("empty", v_loc)],
        [Proposition("on", v_tire, v_loc)], #add effects
        [Proposition("empty", v_loc), #delete effects
         Proposition("on", v_tire, i_ground)]
)
```

Lists are conjunctions!

All propositions with a variable must take on the same instance!

Variables that don't match name don't have to be the same but can be unless otherwise specified!

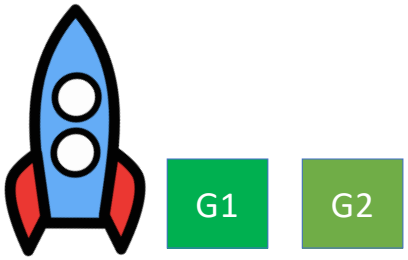
Another Example - Rocket Ship

Suppose we have a rocket ship that has enough fuel for 1 trip from A to B. It needs to transport 2 payloads G1 and G2. Both fit in the rocket together. The rocket can Move, and we can Load and Unload it.

Another Example - Rocket Ship

Suppose we have a rocket ship that has enough fuel for 1 trip from A to B. It needs to transport 2 payloads G1 and G2. Both fit in the rocket together.

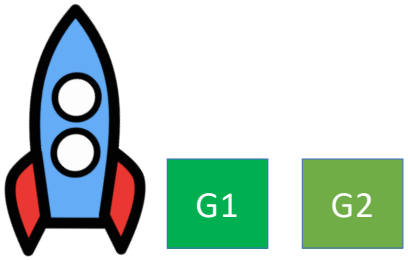
Literals?



Another Example - Rocket Ship

Suppose we have a rocket ship that has enough fuel for 1 trip from A to B. It needs to transport 2 payloads G1 and G2. Both fit in the rocket together.

Literals: Rocket, G1, G2, LocA, LocB

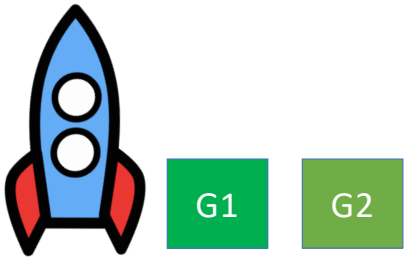


Another Example - Rocket Ship

Suppose we have a rocket ship that has enough fuel for 1 trip from A to B. It needs to transport 2 payloads G1 and G2.

Literals: Rocket, G1, G2, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)

I create literals and variables as I go through the problem. In order to create the start state and the goal state, I need the literals defined.



Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB

Start state:

At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)

Goal state:

At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)

As I create my operators Move, Load, and Unload, I will add variables.

Move: Load: Unload:

Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)

Move:
P:
A:
D:

Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)
Variables: Start

Move:
P: At(Rocket,Start)
A:
D:

The rocket starts at a location, and it could be either location. I need to add a location variable. There is only 1 rocket, so I don't need a variable for it.

Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB

Start state:

At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)

Goal state:

At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)

Variables: Start

Move:

P: At(Rocket,Start), Has-Fuel()

A:

D:

The rocket needs fuel.

Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)
Variables: Start, Dest

Move:
P: At(Rocket,Start), Has-Fuel(), Start !=Dest
A: At(Rocket,Dest)
D:

The rocket needs to go to a destination, which needs to be different from the start location. We need to define a dest variable, which we will add after moving the rocket.

Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)
Variables: Start, Dest

Move:
P: At(Rocket,Start), Has-Fuel(), Start !=Dest
A: At(Rocket,Dest)
D: Has-Fuel(),At(Rocket,Start)

Once the rocket has moved, it has no more fuel and it is no longer at Start.

Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)
Variables: Start, Dest

Load:
P:
A:
D:

Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)
Variables: Start, Dest, Pkg

Load:
P: At(Rocket,Start), Unloaded(Pkg,Start)
A:
D:

The rocket needs to load a specific package G1 or G2. The load action doesn't care which package it is because both are loaded the same way. We need a variable pkg to use, and we need to say it starts at the starting location unloaded from the rocket.

Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)
Variables: Start, Dest, Pkg

Load:
P: At(Rocket,Start), Unloaded(Pkg,Start)
A: Loaded(Pkg,Rocket)
D: Unloaded(Pkg,Start)

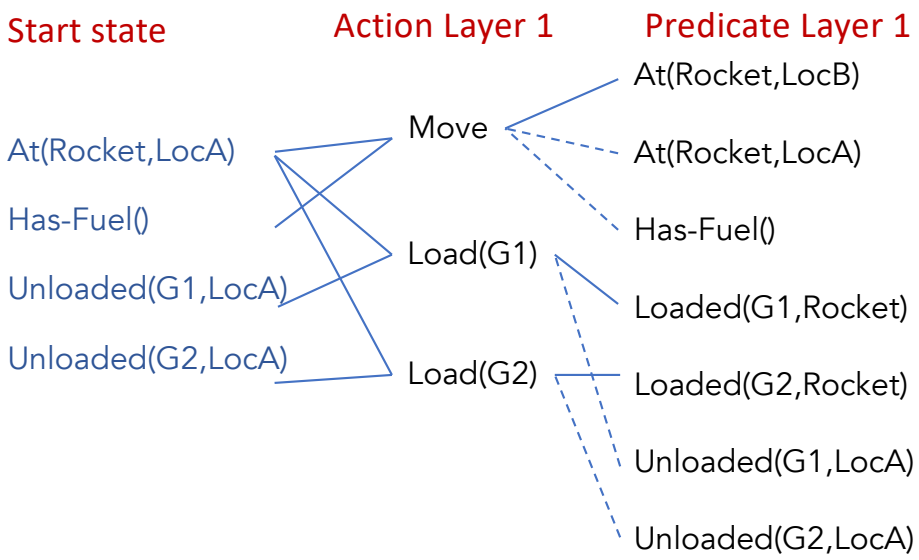
Another Example - Rocket Ship

Literals: Rocket, G1, G2, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G1,LocA), Unloaded(G2,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G1,LocB), Unloaded(G2,LocB)
Variables: Start, Dest, Pkg

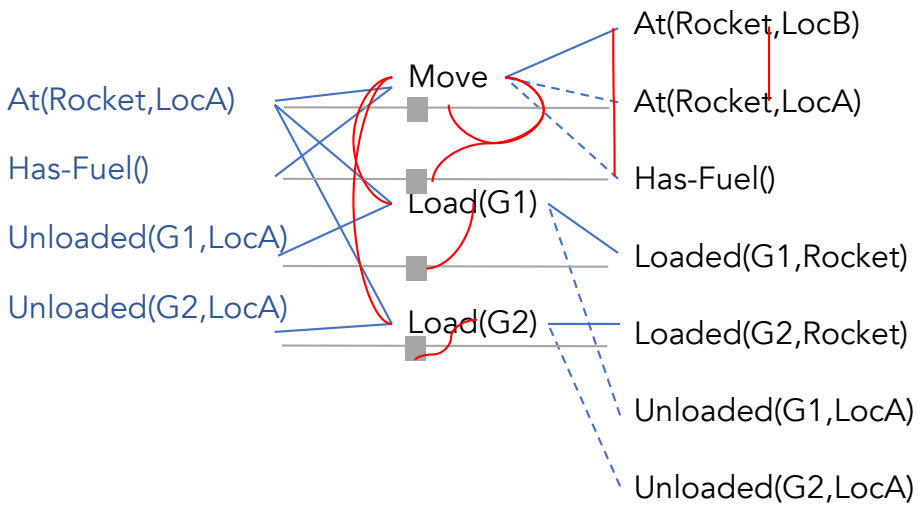
Unload:
P: At(Rocket,Dest), Loaded(Pkg,Rocket)
A: Unloaded(Pkg,Dest)
D: Loaded(Pkg,Rocket)

No new variables needed for unload, which works like Load but in reverse.

Rocket Ship GraphPlan Graph



Rocket Ship GraphPlan Graph



• • •

Mutex Actions
Interference:
Move deletes At which is a precondition of Load
Inconsistent:
Move deletes At but noop adds it
Move deletes Has-Fuel but noop adds it

Mutex Propositions:
- At(Rocket,LocB) and At(Rocket,LocA) because Move and noop are mutex actions
- What else?