Finish up Logic Planning

Jump to previous slides

AI: Representation and Problem Solving

Classical Planning
or Symbolic Planning

Instructor: Pat Virtue

Slide credits: CMU AI

Represent this Blocks World

A robot arm (yellow) can pick up and put down blocks to form stacks.

It cannot pick up a block that has another block on top of it.

It cannot pick up more than one block at a time.

Any number of blocks can sit on the table.

How would solve Blocks World problems with search, e.g., BFS?

state representation

costs

actions(state)

transistion(s, a) → s'

Goal test

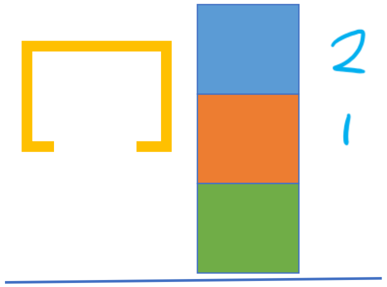
Represent this Blocks World

A robot arm (yellow) can pick up and put down blocks to form stacks.

It cannot pick up a block that has another block on top of it.

It cannot pick up more than one block at a time.

Any number of blocks can sit on the table.



How would solve Blocks World problems with logical planning?

$KB \leftarrow G_{0,0} \wedge O_{1,0} \wedge B_{2,0} \wedge H_{empty,0}$

φ

Search, Logic, and Classical Planning

Search Planning

- Assumes actions and transitions are provided for you, s’ = result(s, a)
- State changes as you take actions

Propositional Logic Planning

- Can reason about what actions are possible and their effects
- Represent world with only Boolean symbols
- Different symbols for different time points

Classical Planning

- Can reason about what actions are possible and their effects
- State changes as you take actions

Idea of Classical Planning

Represent objects/values separately from the state (instances)

A B C

Define predicates as true/false functions over the objects (propositions)

$onTable(B) \rightarrow T/F$

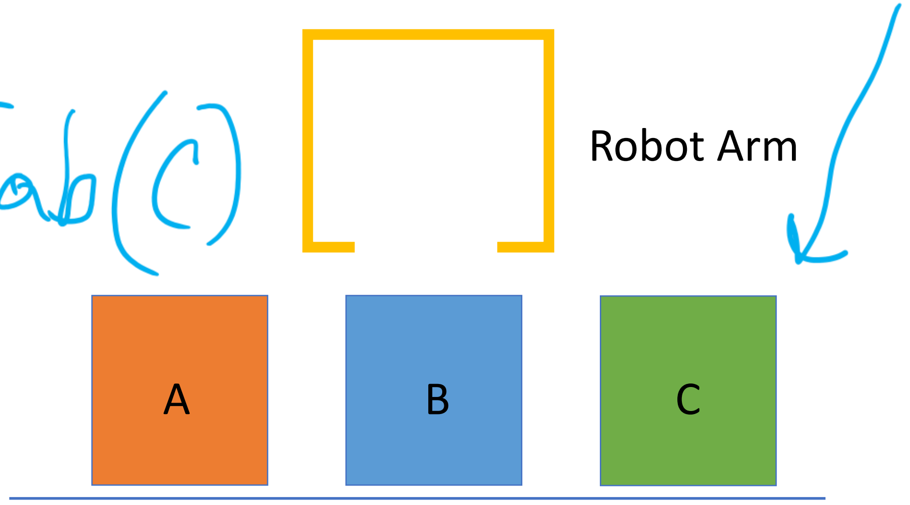
States are conjunctions of predicates

Goals are conjunctions of predicates

$onTab(A) \wedge onTab(B) \wedge onTab(C)$

Operators (actions)

$\wedge\ handEmpty()$



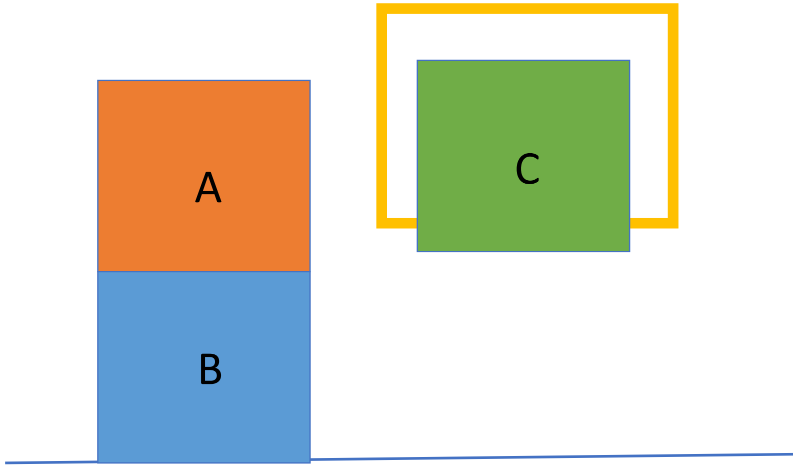
Poll 1

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

Instances: A, B, C

Predicates:

1) In-Hand(A)
2) In-Hand(B)
3) In-Hand(C)
4) On-Table(A)
5) On-Table(B)
6) On-Table(C)
7) On-Block(B,C)
8) On-Block(A,B)
9) HandEmpty()



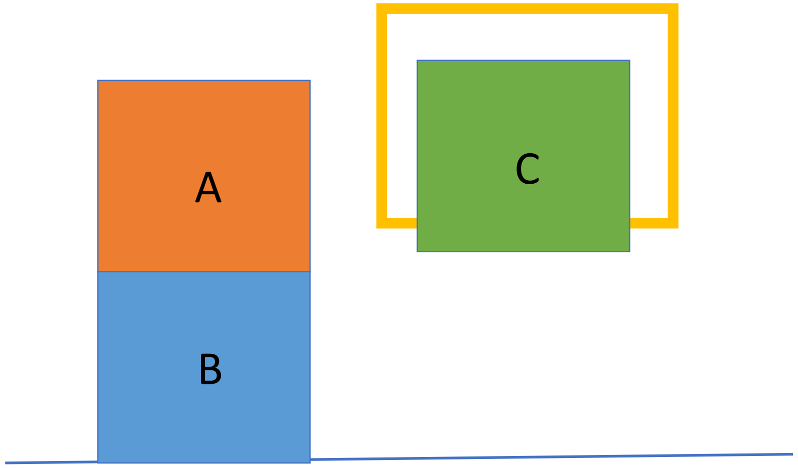
Poll 1

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

Instances: A, B, C

Predicates:

1) ~~In-Hand(A)~~
2) ~~In-Hand(B)~~
3) In-Hand(C)
4) ~~On-Table(A)~~
5) On-Table(B)
6) ~~On-Table(C)~~
7) On-Block(B,C)
8) On-Block(A,B)
9) HandEmpty()



Full State Description

Instances: A, B, C

Predicates:

In-Hand(C)

On-Table(B)

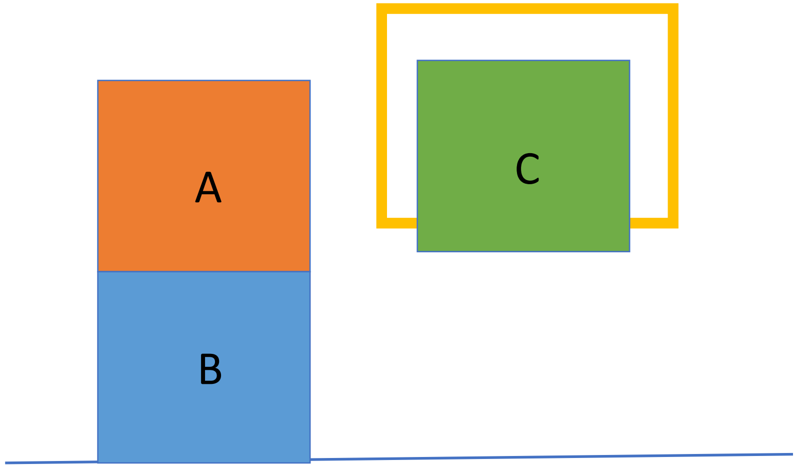
On-Block(A,B)

Clear(A)

Clear(C)

Optional: ~HandEmpty(), ~On-Table(C), ~On-Table(A), ~On-Block(B,A), ~On-Block(C,A), ~On-Block(B,C), ~On-Block(C,B), ~On-Block(A,C), ~Clear(B), ~In-Hand(A), ~In-Hand(B)

implicit



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

Rules of Blocks World

Blocks are picked up and put down by the hand

Blocks can be picked up only if they are clear

Hand can pick up a block only if the hand is empty

Hand can pick up and put down blocks on blocks or on the table

Pickup Block C from Table (State Transition)

Instances:

Blocks A, B, C

Possible Predicates:

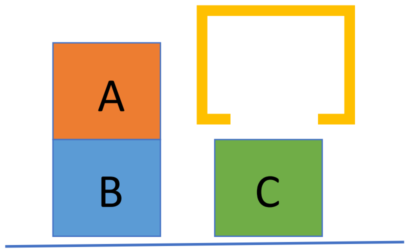
HandEmpty()

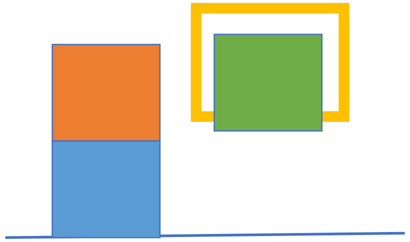
On-Table(block)

On-Block(b1,b2)

Clear(block)

In-Hand(block)



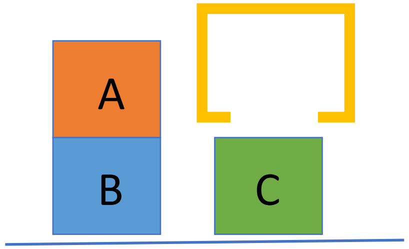


State:	State:
HandEmpty()	In-Hand(C)
On-Table(B)	On-Table(B)
On-Table(C)	On-Block(A,B)
On-Block(A,B)	Clear(A)
Clear(A)	Clear(C)
Clear(C)	

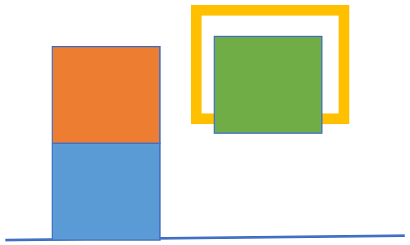
Pickup Block C from Table (Preconditions, Effects)

Instances:
Blocks A, B, C

Possible Predicates:
HandEmpty()
On-Table(block)
On-Block(b1,b2)
Clear(block)
In-Hand(block)

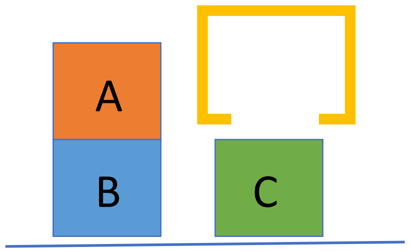


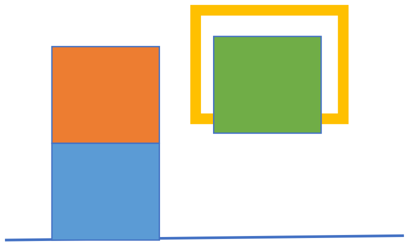
State:
HandEmpty()
On-Table(B)
On-Table(C)
On-Block(A,B)
Clear(A)
Clear(C)



State:
In-Hand(C)
On-Table(B)
On-Block(A,B)
Clear(A)
Clear(C)
Delete HandEmpty()
Delete On-Table(C)

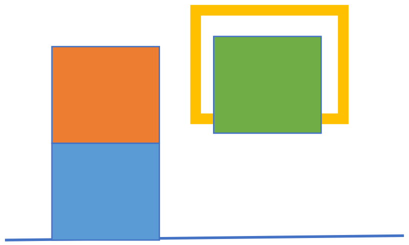
Operator: Pickup-Block-C from Table





Preconditions	Effects
HandEmpty()	Add In-Hand(C)
Clear(C)	Delete HandEmpty()
On-Table(C)	On-Table(C)

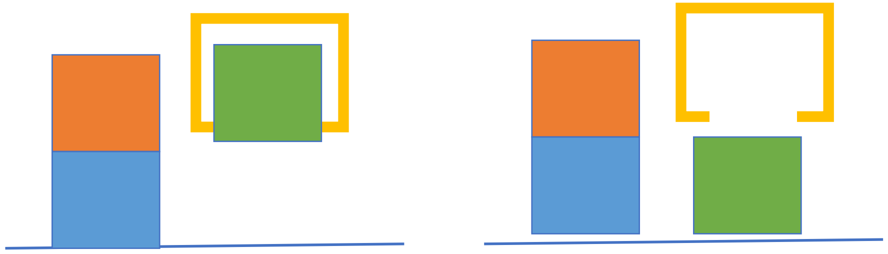
Operator: Pickup-Block from Table



<u>Preconditions</u>	<u>Effects</u>
HandEmpty()	Add In-Hand(block)
Clear(block)	Delete HandEmpty()
On-Table(block)	On-Table(block)

Create a variable that takes on the value of a particular instance for all times it appears in an operator.

Operator: PutDown-Block on Table



<u>Preconditions</u>

In-Hand(block)

<u>Effects</u>

Add HandEmpty()
On-Table(block)

Delete In-Hand(block)

Why don't we need to check if ~HandEmpty() is true?

Full State Description

Instances: A, B, C

Predicates:

In-Hand(C)

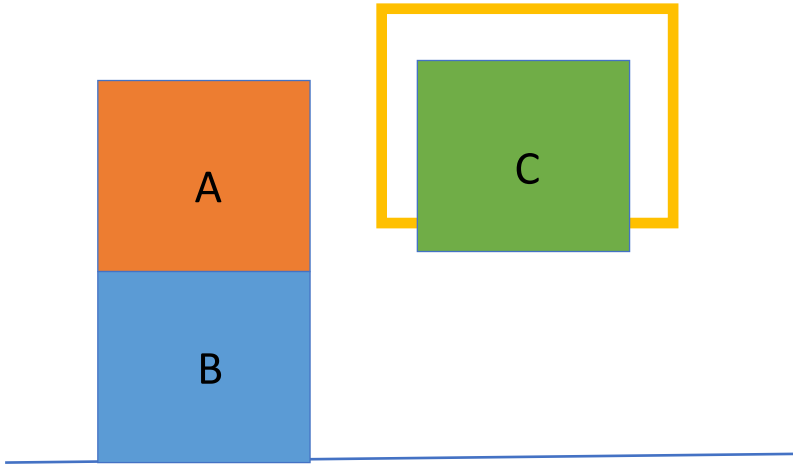
On-Table(B)

On-Block(A,B)

Clear(A)

Clear(C)

Optional: ~HandEmpty(), ~On-Table(C), ~On-Table(A), ~On-Block(B,A), ~On-Block(C,A), ~On-Block(B,C), ~On-Block(C,B), ~On-Block(A,C), ~Clear(B), ~In-Hand(A), ~In-Hand(B)



RULE OF THUMB: If you must match that Predicate is explicitly not true, you must include ~Predicate in the state description.

Operators for Block Stacking

Pickup_Table(b):

Pre: HandEmpty(), Clear(b), On-Table(b)

Add: In-Hand(b)

Delete: HandEmpty(), On-Table(b)

Pickup_Block(b,c):

Pre: HandEmpty(), On-Block(b,c), b!=c

Add: In-Hand(b), Clear(c)

Delete: HandEmpty(), On-Block(b,c)

Putdown_Table(b):

Pre: In-Hand(b)

Add: HandEmpty(), On-Table(b)

Delete: In-Hand(b)

Putdown_Block(b,c):

Pre: In-Hand(b), Clear(c)

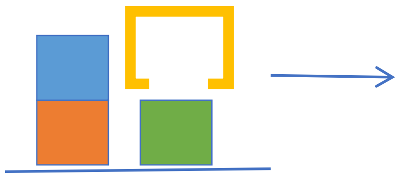
Add: HandEmpty(), On-Block(b,c)

Delete: Clear(c), In-Hand(b)

Why do we need separate operators for table vs on a block?

Example Matching Operators

HandEmpty() & On-Table(O) & On-Block(B,O) & Clear(B) & On-Table(G) & Clear(G)

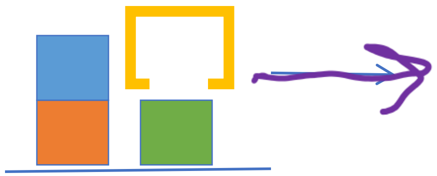


Example Matching Operators

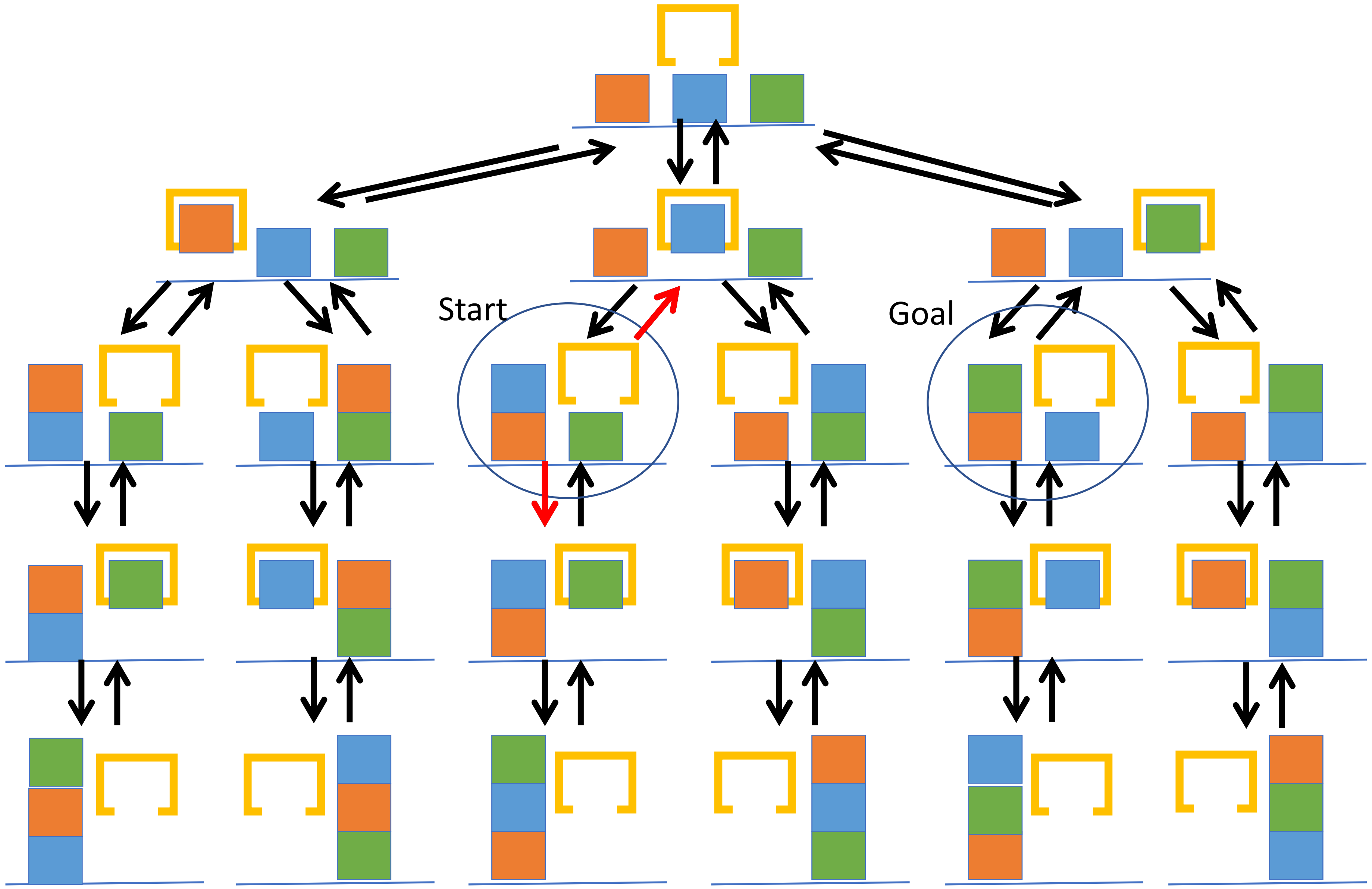
HandEmpty() & On-Table(O) & On-Block(B,O) & Clear(B) & On-Table(G) & Clear(G)

Pickup_Block(b,c):
Pre: HandEmpty(), On-Block(b,c), b!=c
Add: In-Hand(b), Clear(c)
Delete: HandEmpty(), On(b,c)

Pickup_Table(b):
Pre: HandEmpty, Clear(b), On-Table(b)
Add: In-Hand(b)
Delete: HandEmpty(), On-Table(b)



State Space Graph (also called Reachability Graph)



Example Matching Operators

HandEmpty() & On-Table(O) & On-Block(B,O) & Clear(B) & On-Table(G) & Clear(G)

Pickup_Block(B,O)

On-Table(O) & Clear(B) & On-Table(G) & Clear(G) & In-Hand(B) & Clear(O)

Putdown_Table(B)

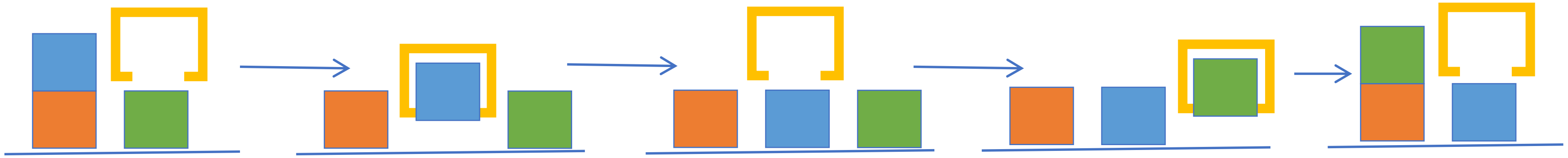
On-Table(O) & Clear(O) & On-Table(G) & Clear(G) & Clear(B) & On-Table(B) & HandEmpty()

Pickup_Table(G)

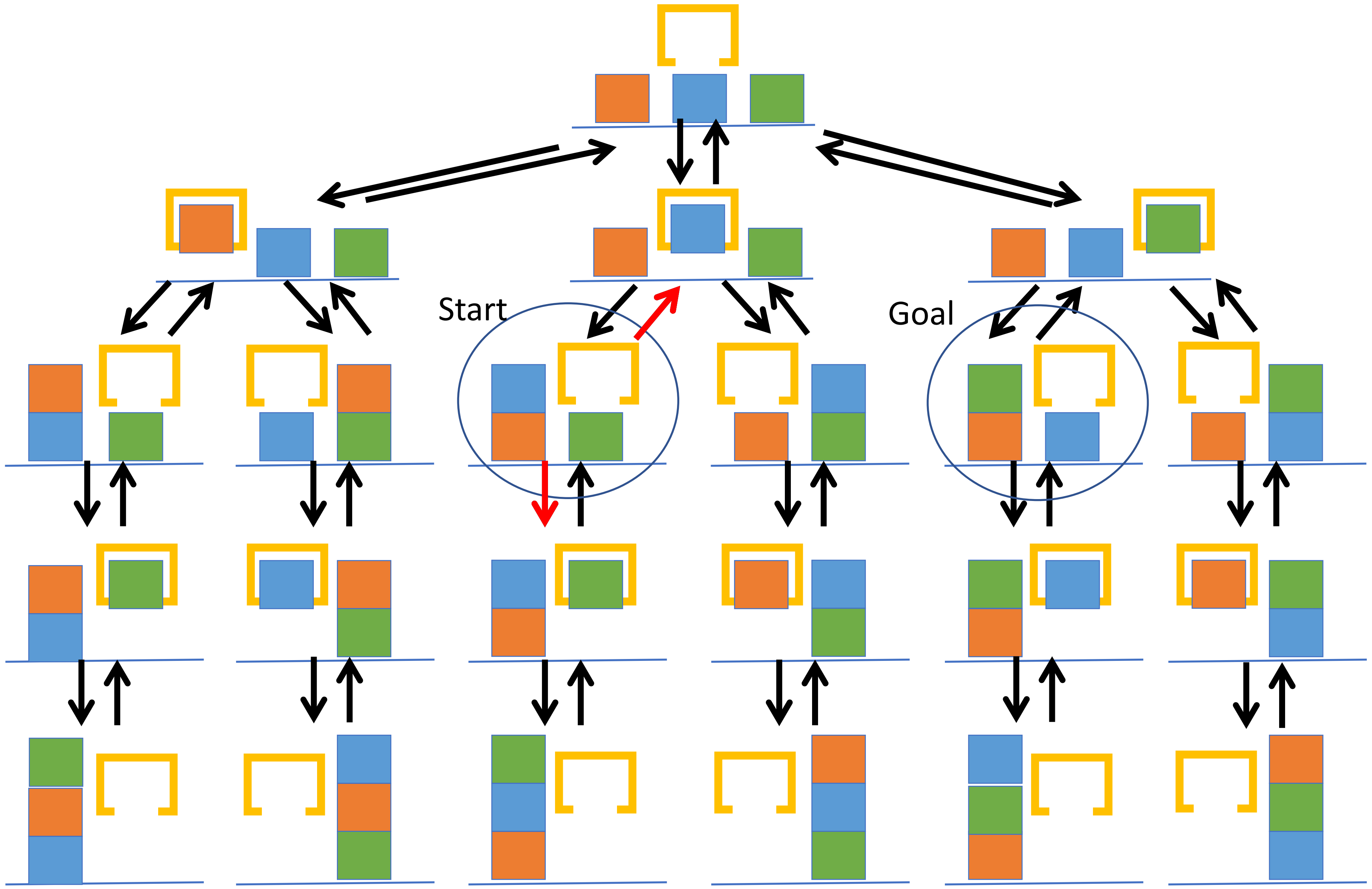
On-Table(O) & Clear(B) & Clear(G) & Clear(O) & On-Table(B) & In-Hand(G)

Putdown_Block(G,O)

On-Table(O) & Clear(B) & Clear(G) & On-Table(B) & On-Block(G,O) & HandEmpty()



Search with a State Space Graph



Finding Plans with Symbolic Representations

Breadth-First Search

Sound? Yes

Complete? Yes

Optimal? Yes

action(s)

result(s, a)

Soundness - all solutions found are legal plans

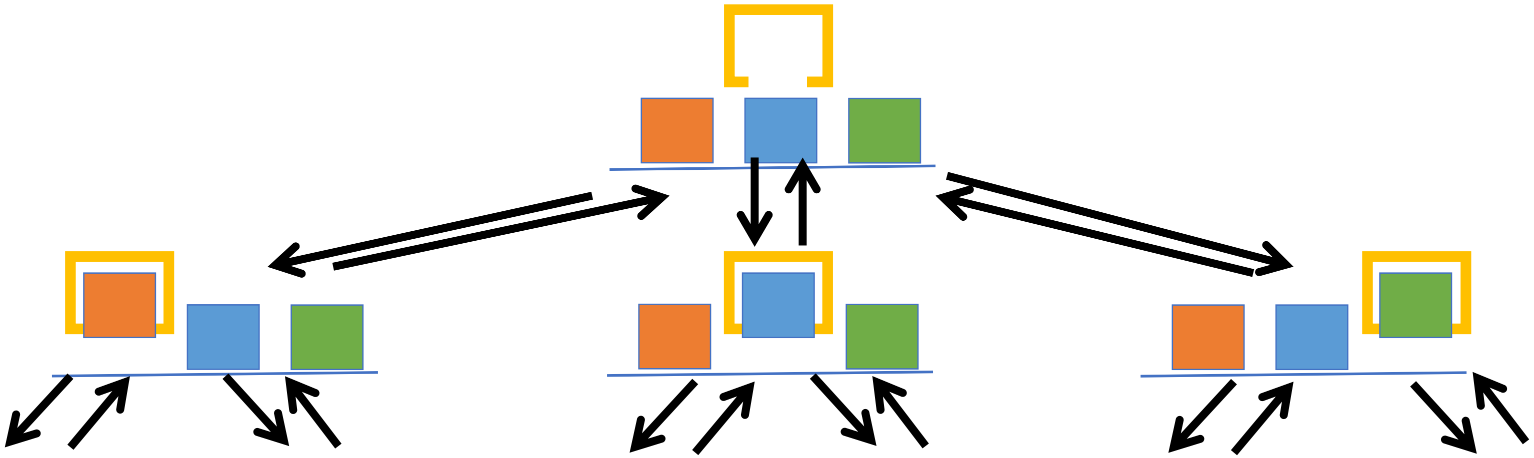
Completeness - a solution can be found whenever one actually exists

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

Size of the Search Tree

A planning tree's size is exponential in the number of predicates

Even if we use linear or non-linear planning, they use this graph



Can we reduce the size of the planning graph?

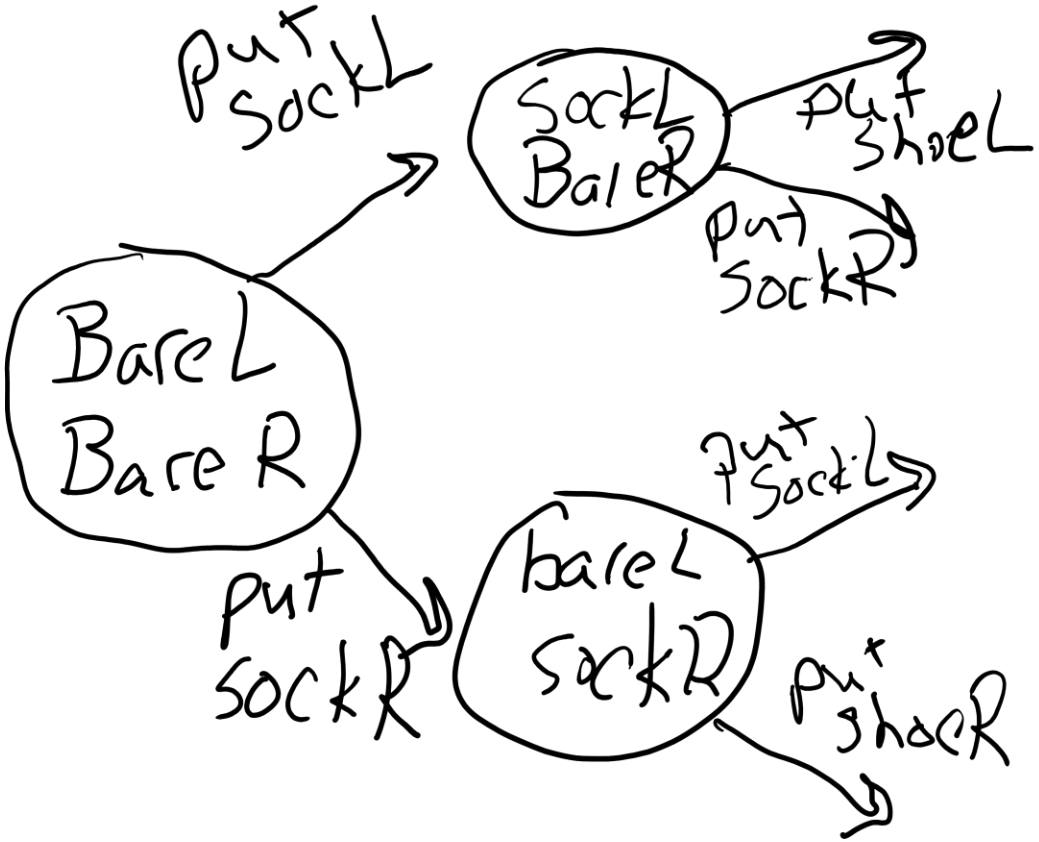
GraphPlan

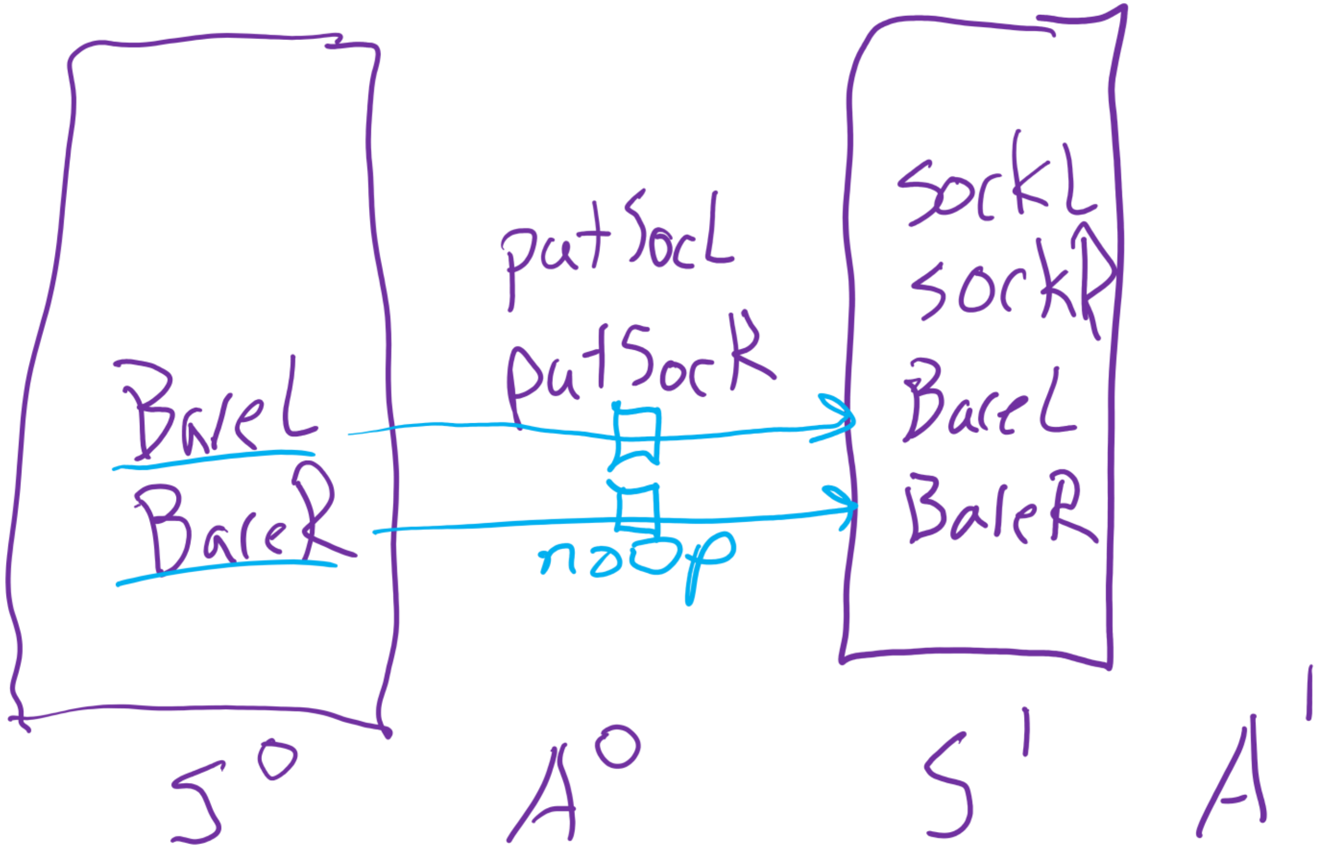
GraphPlan

1) actions simultaneously
2) always add predicates (don't delete)

GraphPlan is a relaxation of other classical planning search techniques

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



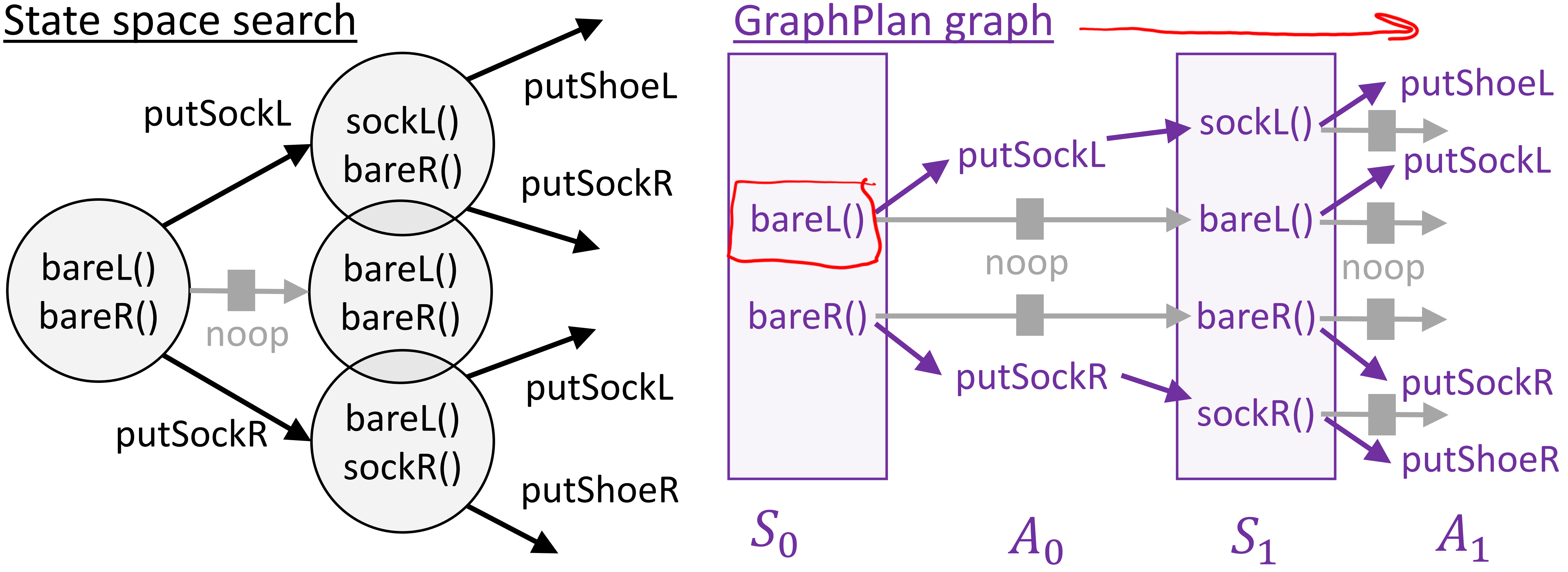


GraphPlan

1) Allow actions to be simultaneous
2) Always add predicates (don't delete)

GraphPlan is a relaxation of other classical planning search techniques

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

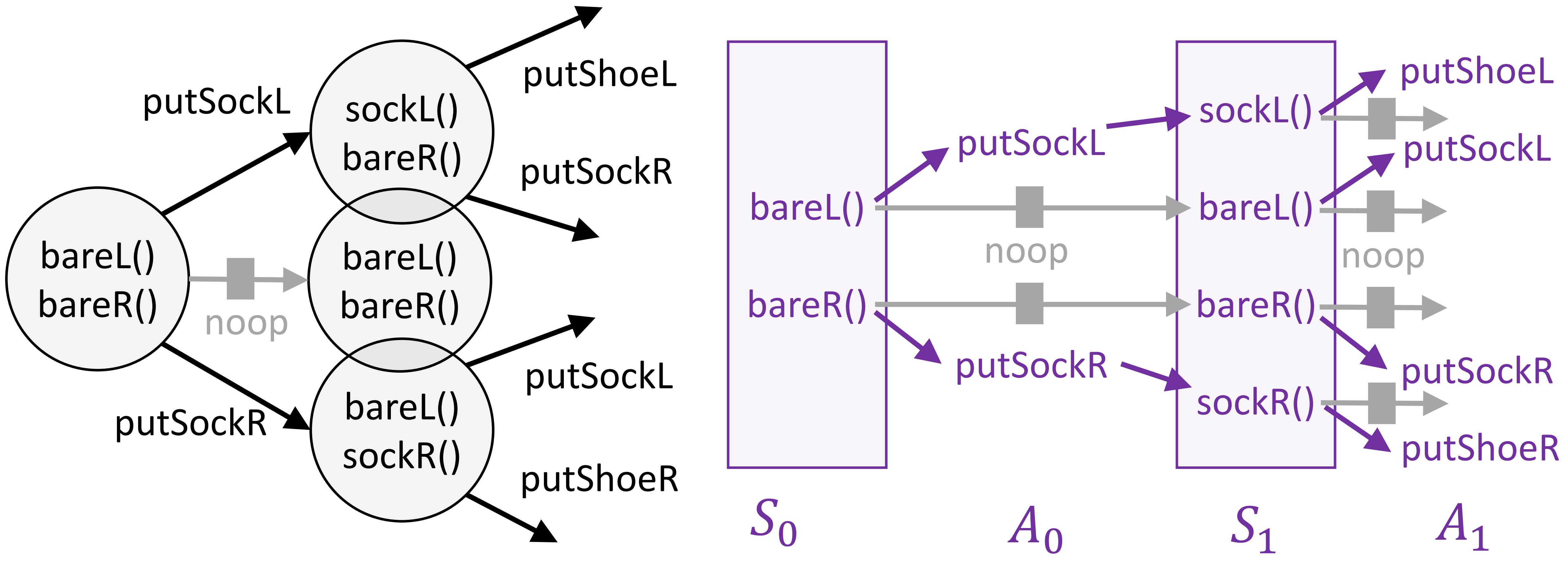


GraphPlan

1) Allow actions to be simultaneous
2) Always add predicates (don't delete)

GraphPlan is a relaxation of other classical planning search techniques

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



Building a GraphPlan Graph

Initialize S_0 with all predicates in the start state

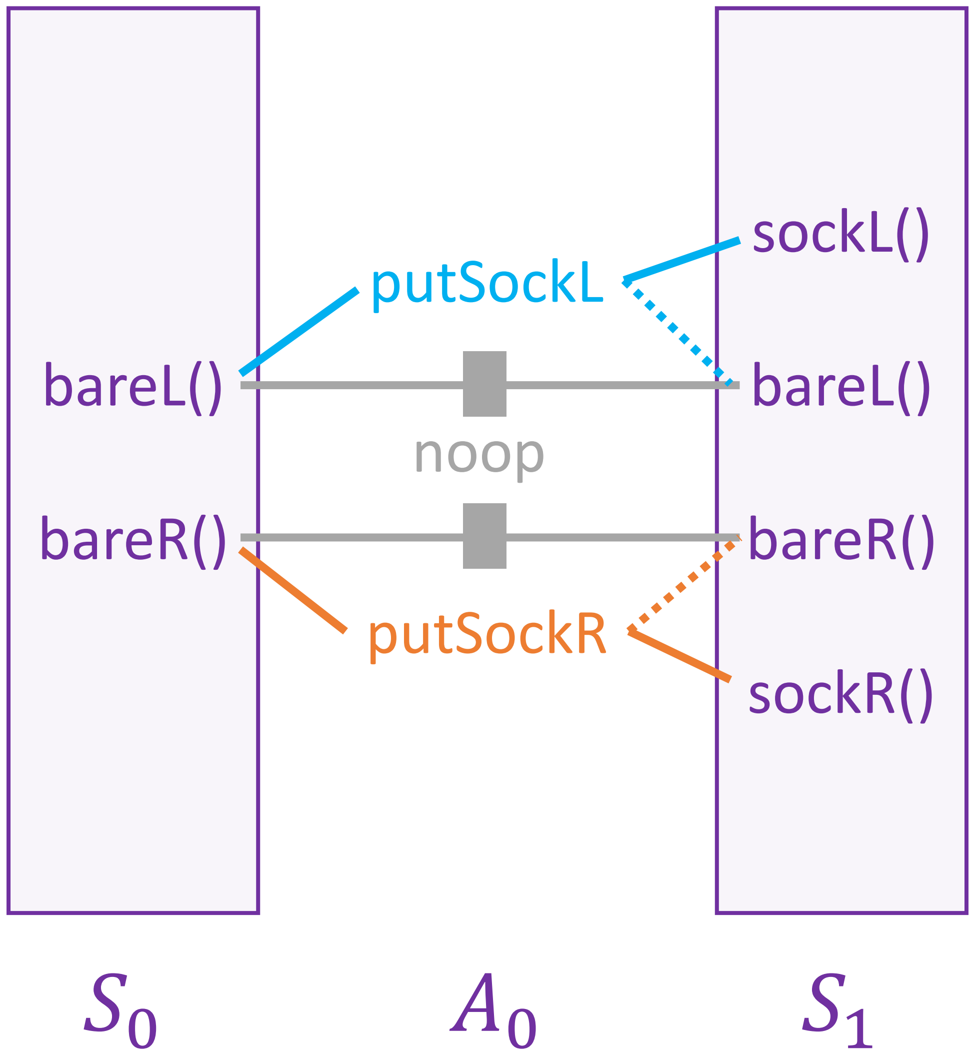
bareL()

bareR()

S_0 A_0 S_1

Building a GraphPlan Graph

Extend graph to S_1 with all actions in A_0 that can be taked from S_0



For now, we are just assuming that any actions (including no-ops) can be taken individually in any order to get us to the next state

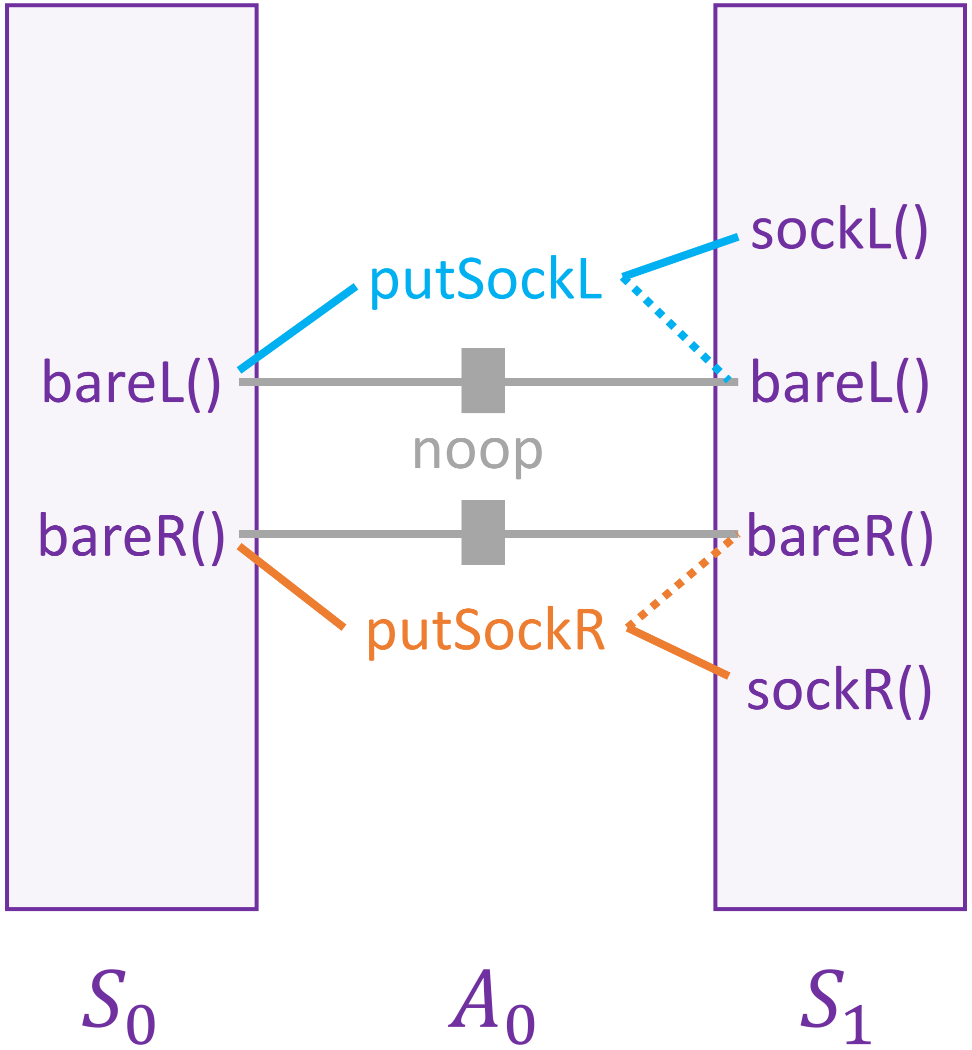
→ This certainly isn't always true

E.g., we can't put our socks on and then take the no-ops that require bare feet

(More on these exclusion checks later)

Building a GraphPlan Graph

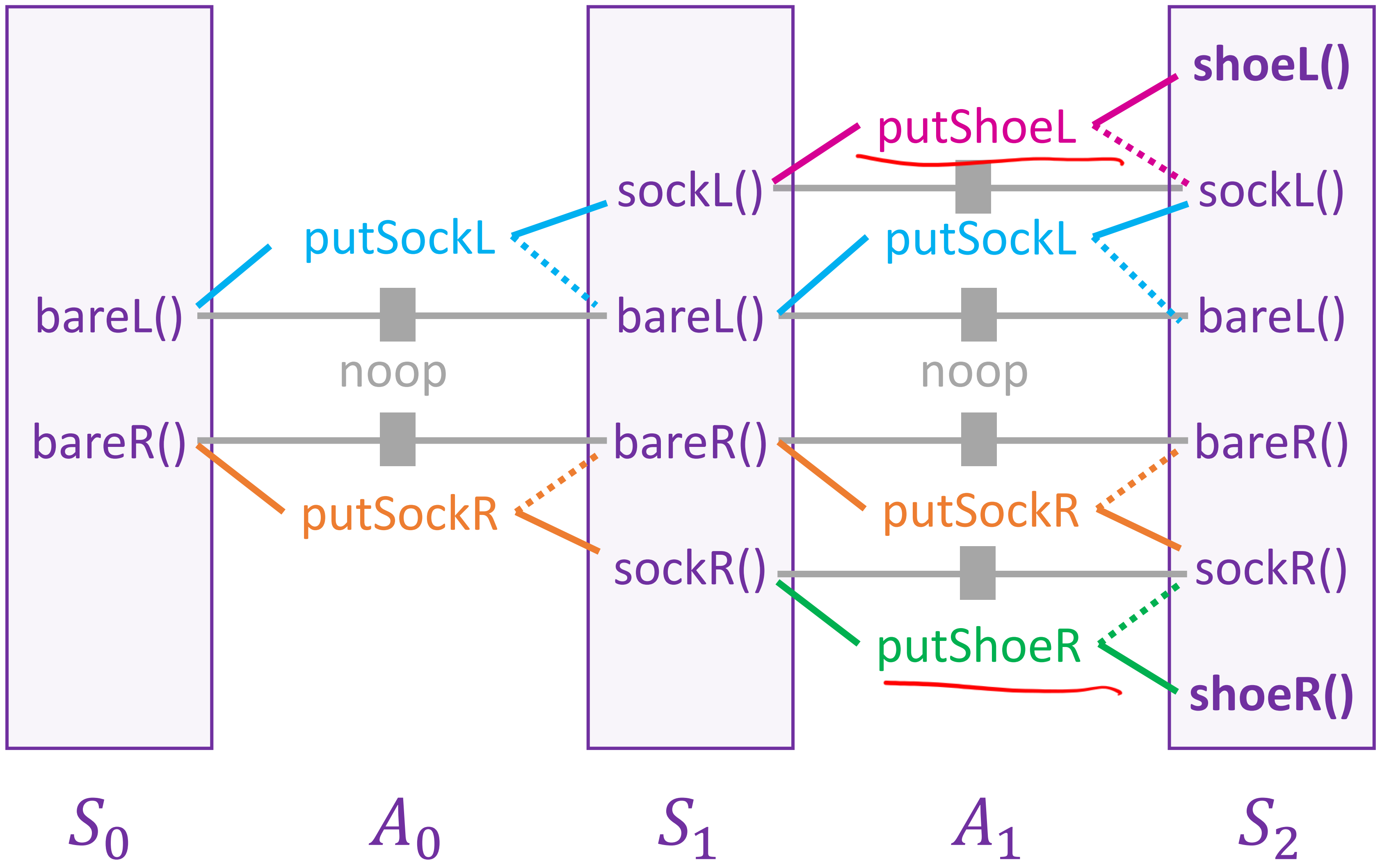
Search for solution. Does S_1 contain all goal propositions, shoeL(), shoeR()?



Nope, not yet

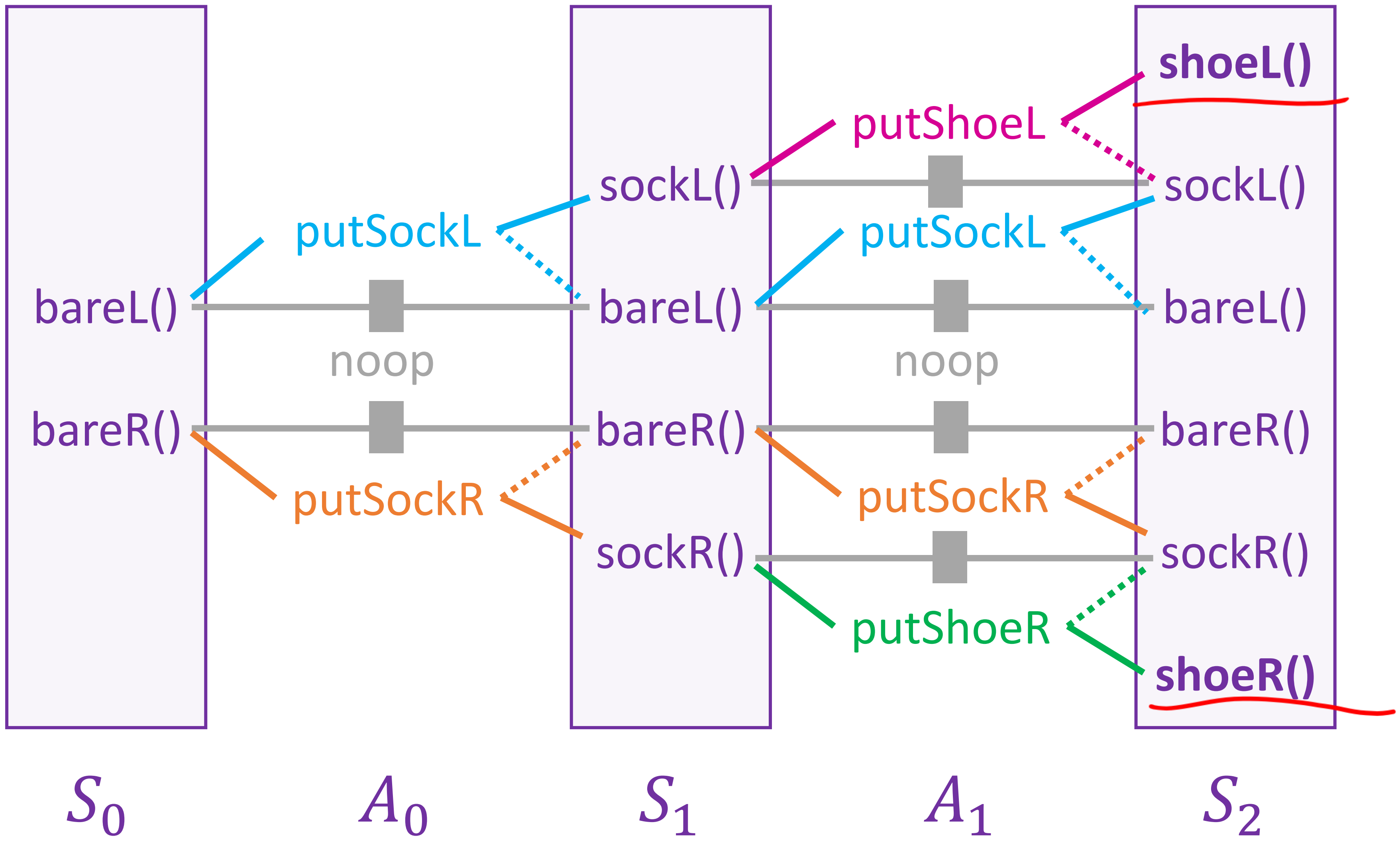
Building a GraphPlan Graph

Extend graph to S_2 with all actions in A_1 that can be taked from S_1



Building a GraphPlan Graph

Search for solution. Does S_2 contain all goal propositions, **shoeL()**, **shoeR()**?



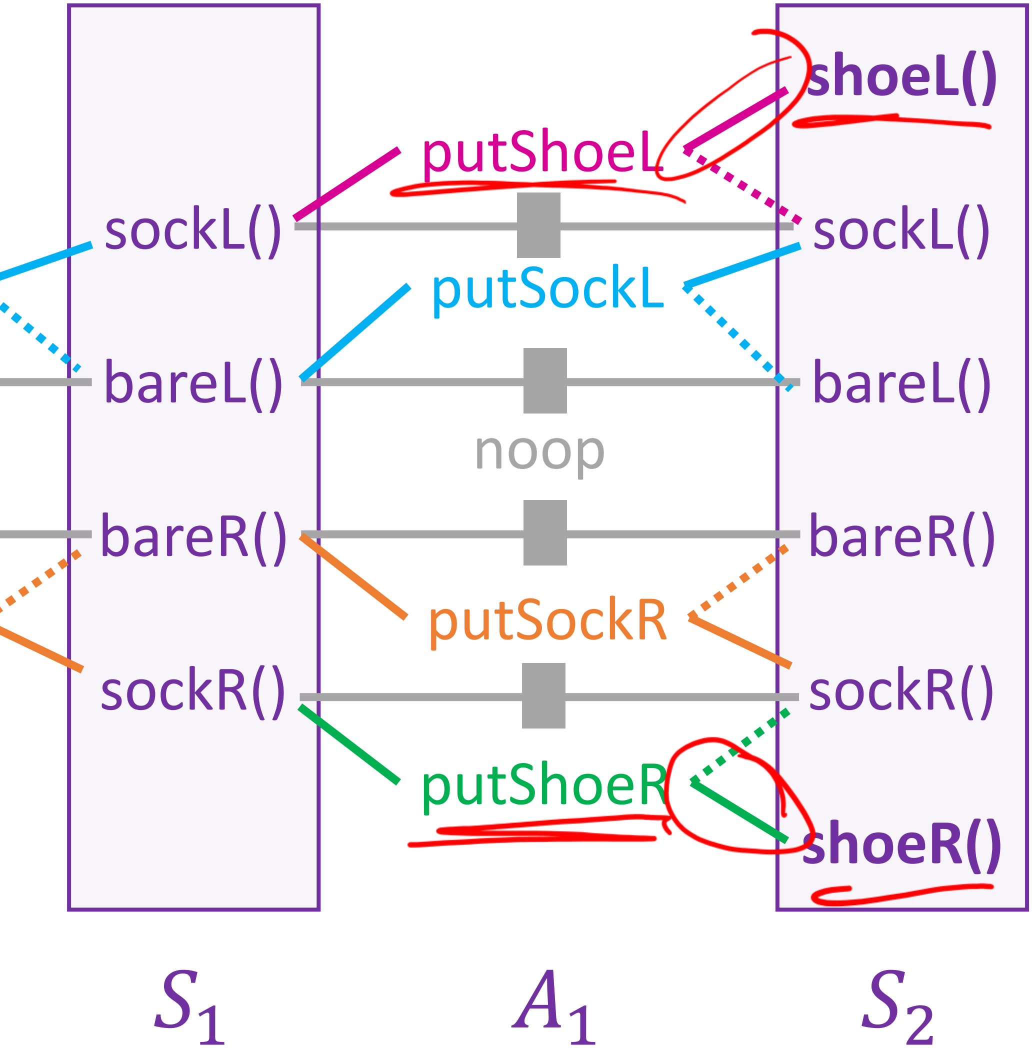
Yes, but
....is this ok??

Maybe ☺

We need to check that we can actually take the subset of actions that lead us the goal proposition

Building a GraphPlan Graph

Search for solution. Does S_2 contain all goal propositions, **shoeL()**, **shoeR()**?

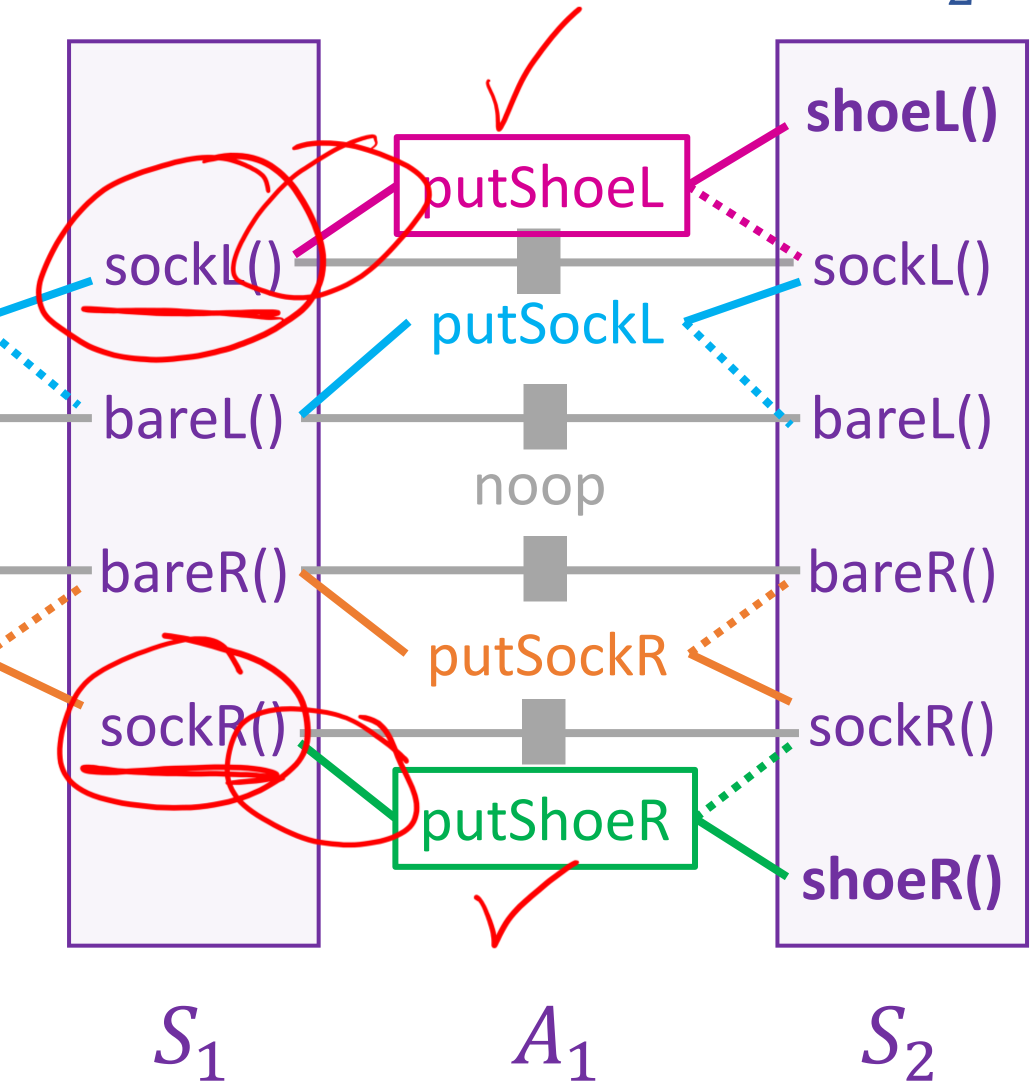


Are the goal propositions ok?

- Do they directly contradict each other (negation)?
- Are the actions that produced them ok (consistent support)?
  - We'll need to check putShoeL and putShoeR
  - Can we really do both of these actions in either order?

Building a GraphPlan Graph

Search for solution. Does S_2 contain all goal propositions, **shoeL()**, **shoeR()**?



Are the actions leading to the goals okay (not exclusive)?

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

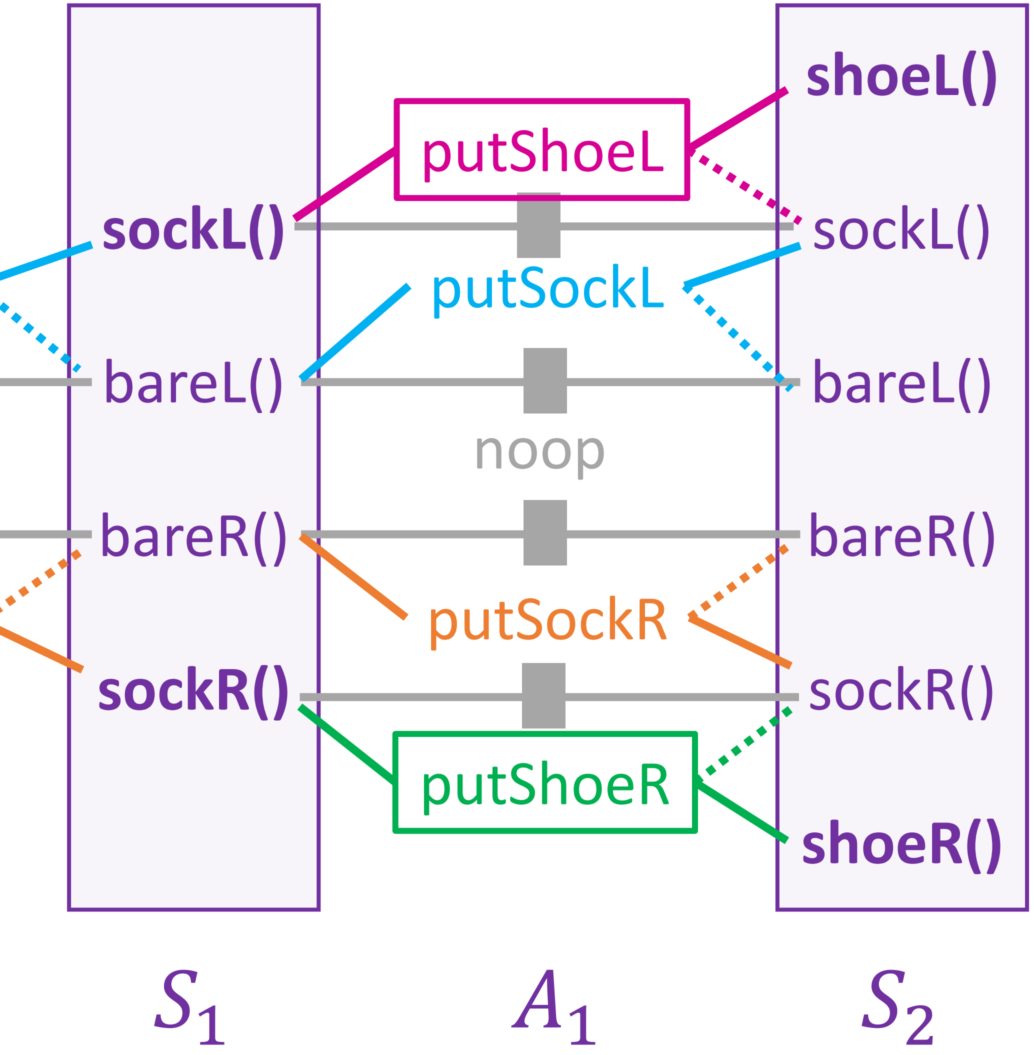
Interference: one action effect deletes or negates a **precondition** of the other

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 for solution. Does S_2 contain all goal propositions, **shoeL()**, **shoeR()**?



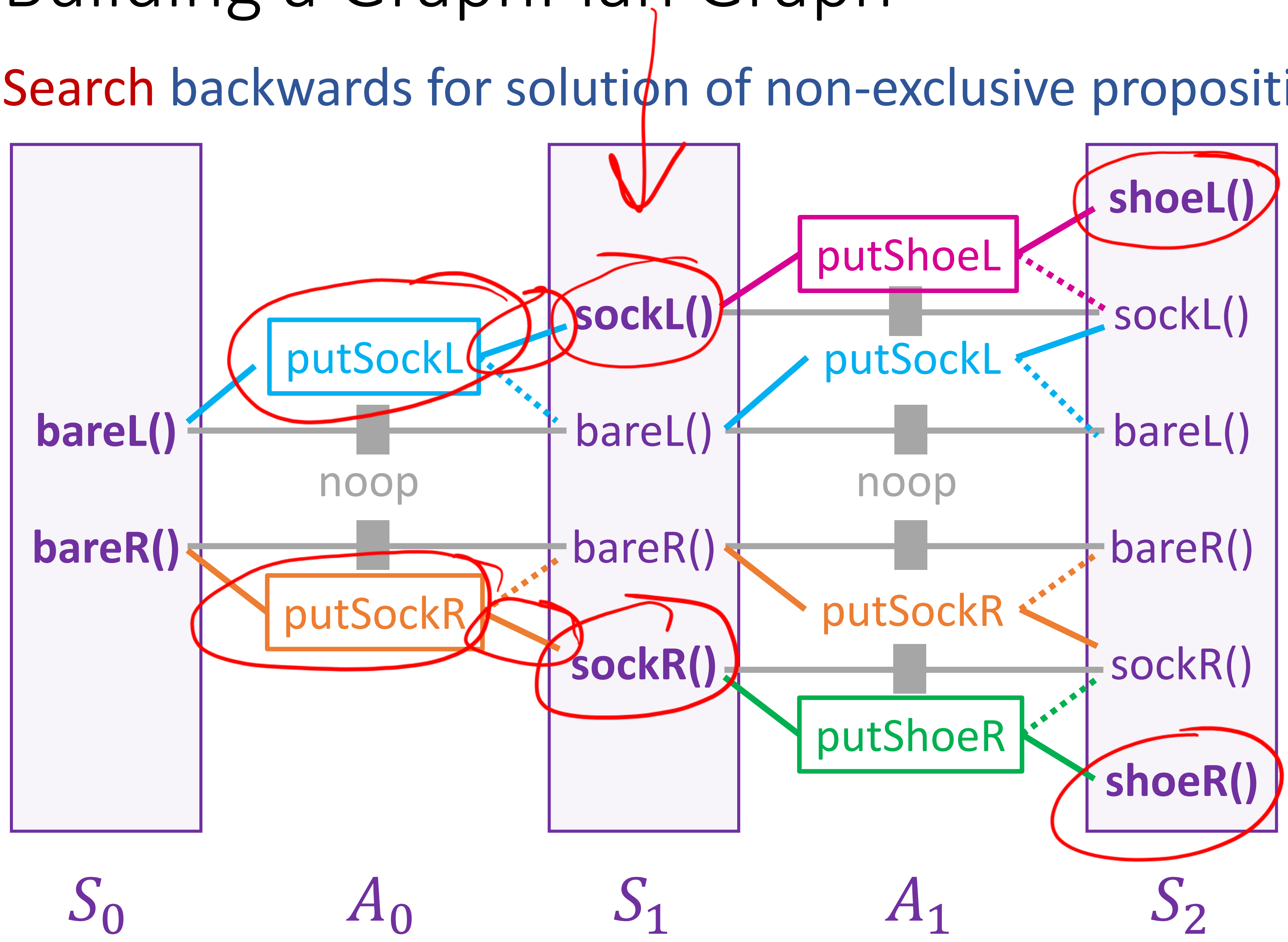
Are the actions leading to the goals okay (not exclusive)?

If yes, we need to check the S_1 pre-condition propositions of those actions: **sockL()** and **sockR()**

... and, then check the actions in A_0 that led us to that set of propositions...

Building a GraphPlan Graph

Search backwards for solution of non-exclusive propositions and actions



Solution found!

We can do putSockL and putSockR in any order, then

We can do putShoeL and putShoeR in any order

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 the goal are present in the added proposition level:

Search for a possible plan in the planning graph (see solution algorithm)

If plan found, return with that plan

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

Details: 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.

Poll

What kind of mutex are actions to each other? (select all that apply)

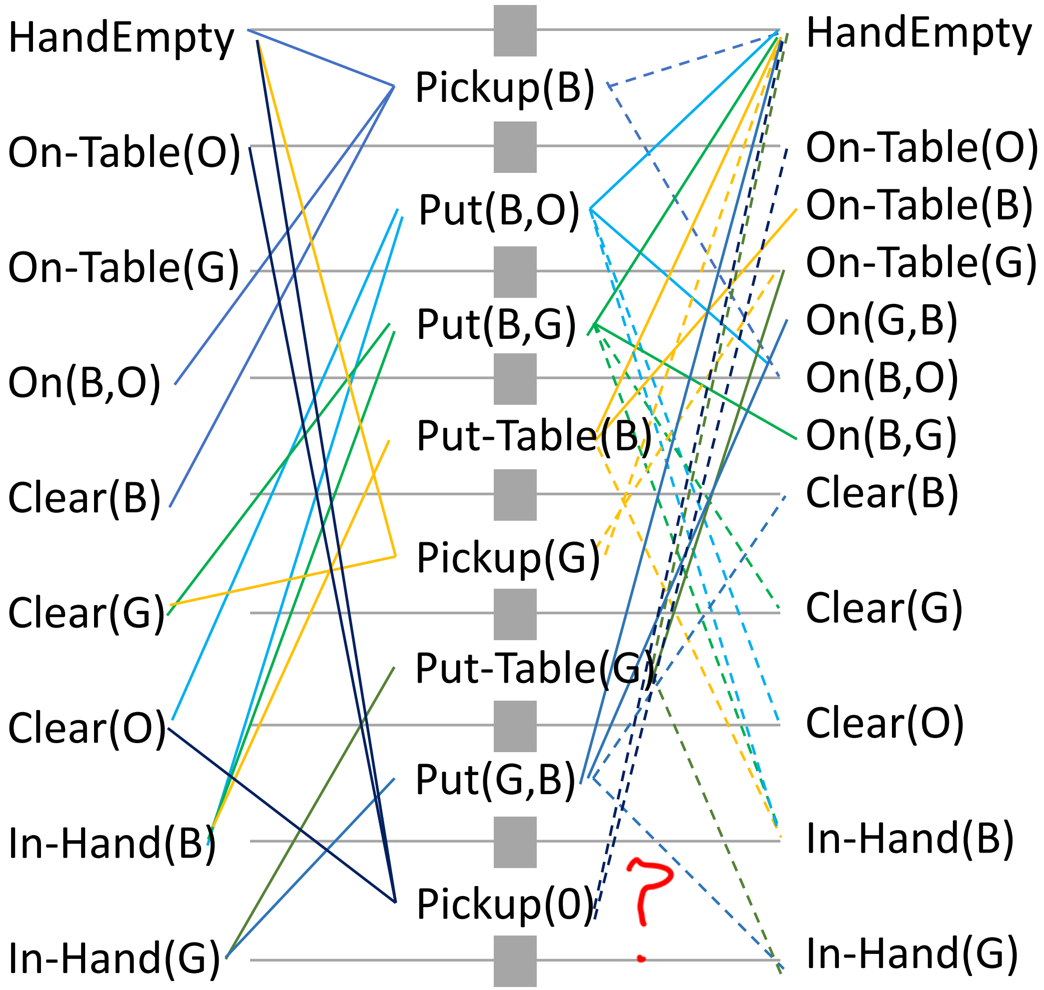
1) Pickup-pickup are interference
2) Pickup-pickup are inconsistent
3) Pickup-pickup are competing needs
4) Pickup-put are interference
5) Pickup-put are inconsistent
6) Pickup-put are competing needs

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 previous proposition-level



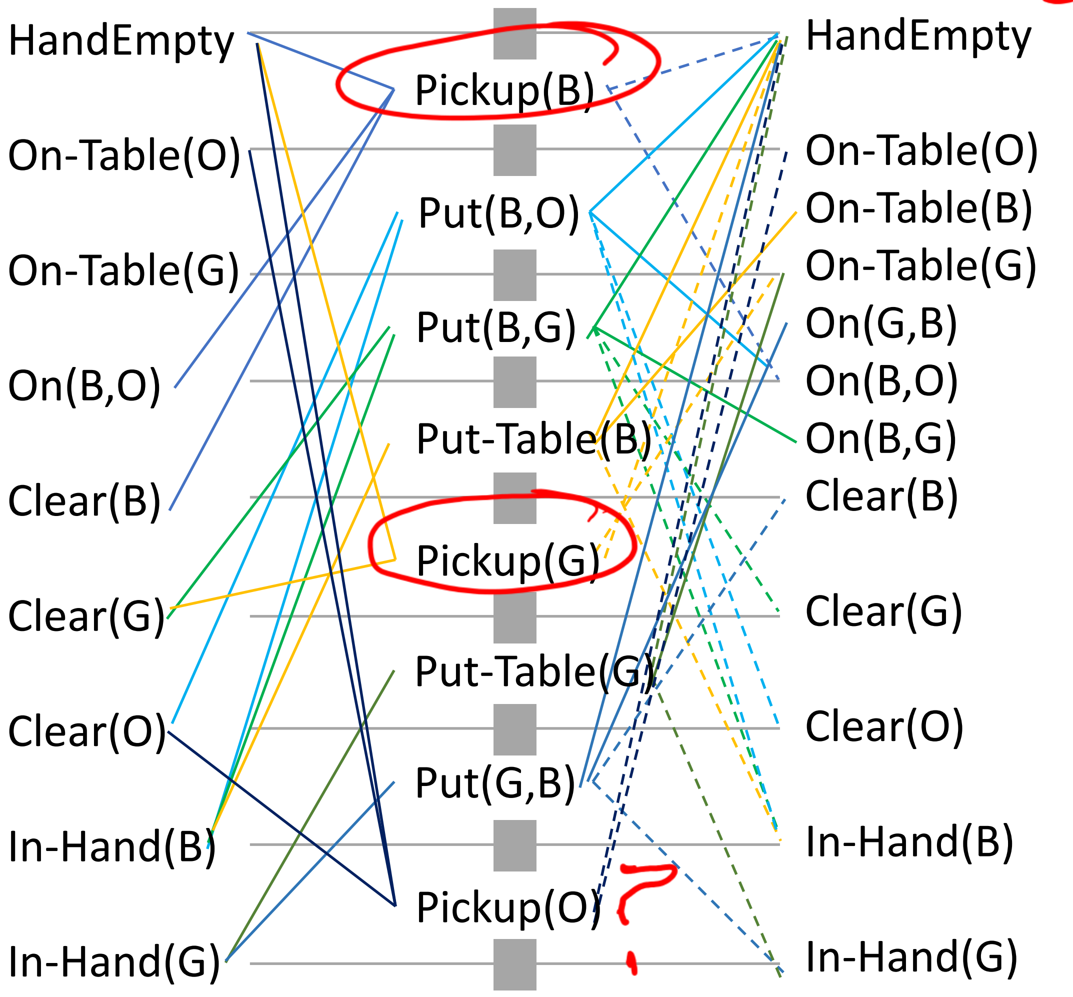
Poll

What kind of mutex are actions to each other? (select all that apply)

1) Pickup-pickup are interference
2) Pickup-pickup are inconsistent
3) Pickup-pickup are competing needs
4) Pickup-put are interference
5) Pickup-put are inconsistent
6) Pickup-put are competing needs

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 previous proposition-level



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

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 build graph plan graphs and assess the graph plan graph for mutexes, goals, leveling off, and solutions.

Implementation

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)

Block World Example:

Pickup_from_Table(b):

Pre: HandEmpty(), Clear(b), On-Table(b)

Add: In-Hand(b)

Delete: HandEmpty(), On-Table(b)

Instances: “A”, “B”, “C” of type BLOCK

Variable: “b” of type BLOCK

In this operator, b can take on the value of any block instance

Implementing Symbolic Representations

Literals: Each thing/object in our model

i_a = Instance("A",BLOCK), i_b = Instance("B",BLOCK)

Variables: Can take on any TYPE thing

v_block = Variable("b",BLOCK)

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

Block World Example:

Pickup_from_Table(b):

Pre: HandEmpty(), Clear(b), On-Table(b)

Add: In-Hand(b)

Delete: HandEmpty(), On-Table(b)

Implementing Symbolic Representations

Literals: Each thing/object in our model

i_a = Instance("A",BLOCK), i_b = Instance("B",BLOCK)

Variables: Can take on any TYPE thing

v_block = Variable("b",BLOCK)

Propositions: Predicate Relationships

p1 = proposition("relation", v_a, i, …)

NOTE: variables and instances do not have to start with i_ and v_

Block World Example:

HandEmpty(), Clear(b), On-Table(b), On-Block(b1,b2)

Proposition("handempty"), Proposition("clear",v_block),
Proposition("on-table",v_block), Proposition("on-block",v_block, i_a)

Initial State and Goal State

Create lists of Propositions as the initial state and goal state

initial = [Proposition(“handempty”), Proposition(“on-table”, i_c),
Proposition(“on-table”, i_b), Proposition(“on-block”, i_a, i_b),
Proposition(“clear”, i_a), Proposition(“clear”, i_c)]

Goal = [Proposition(“on-table”,i_b), Proposition(“on-table”,i_c),
Proposition(“on-block”,i_a, i_c), Proposition(“clear”,i_a),
Proposition(“clear”,”c”)]

Implementing Symbolic Representations

Operators: the actions we take change state

```
pickup_table = Operator("pick_table", #name
                    [ Proposition("handempty",), #preconditions
                      Proposition("clear", v_block),
                      Proposition("on-table", v_block) ],
                    [ Proposition("in-hand", v_block) ], #add effects
                    [ Proposition("handempty"), #delete effects
                      Proposition("on-table", v_block ]
               )
```

Lists are conjunctions!

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

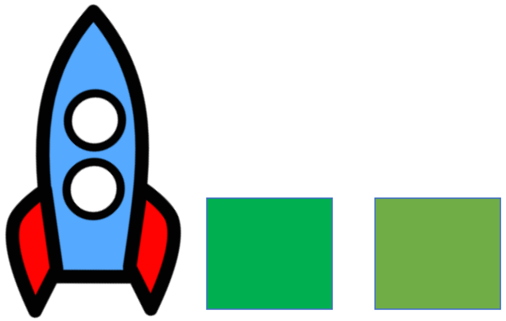
Variables that don't match name don't have to be the same but can be unless otherwise specified!

We provide the GraphPlan implementation

You will create the representation, which will be passed into our GraphPlan implementation

Another Example - Rocket Ship

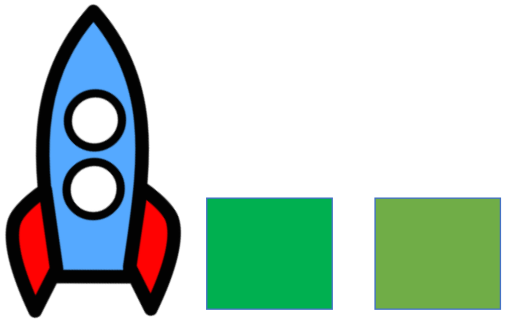
Suppose we have a rocket ship that can only be used once.
It has to carry two payloads.



Another Example - Rocket Ship

Suppose we have a rocket ship that can only be used once.
It has to carry two payloads.

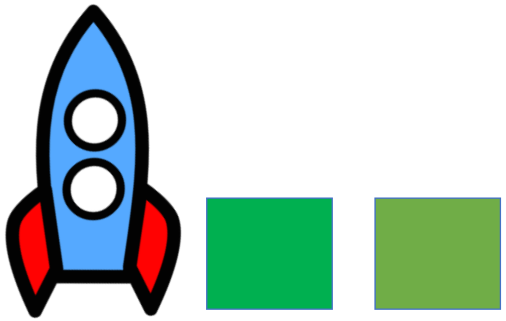
Literals?



Another Example - Rocket Ship

Suppose we have a rocket ship that can only be used once.
It has to carry two payloads.

Literals: Rocket, G, O, LocA, LocB

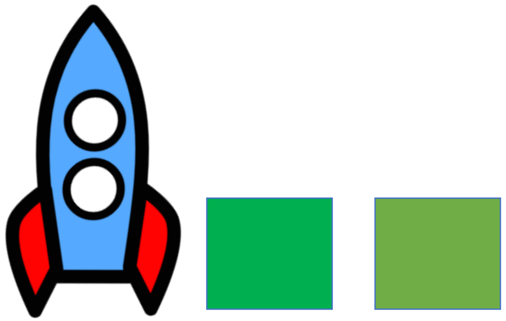


Another Example - Rocket Ship

Suppose we have a rocket ship that can only be used once.
It has to carry two payloads.

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

Start state:

At(Rocket, LocA), Has-Fuel(),
Unloaded(G,LocA), Unloaded(O,LocA)

Goal state:

At(Rocket, LocB), Unloaded(G,LocB), Unloaded(O,LocB)

As I create my operators, I will add variables.

Move:

Load:

Unload:

Another Example - Rocket Ship

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

Move:
P:
A:
D:

Another Example - Rocket Ship

Literals: Rocket, G, O, LocA, LocB

Start state:

At(Rocket, LocA), Has-Fuel(),
Unloaded(G,LocA), Unloaded(O,LocA)

Goal state:

At(Rocket, LocB), Unloaded(G,LocB), Unloaded(O,LocB)

Variables: L

The rocket starts at a location, and it could be either location. I need to add a location variable

Move:

P: At(Rocket,L)

A:

D:

Another Example - Rocket Ship

Literals: Rocket, G, O, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G,LocA), Unloaded(O,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G,LocB), Unloaded(O,LocB)
Variables: L

Move:
P: At(Rocket,L), Has-Fuel()
A:
D:

Another Example - Rocket Ship

Literals: Rocket, G, O, LocA, LocB
Start state:
 At(Rocket, LocA), Has-Fuel(),
 Unloaded(G,LocA), Unloaded(O,LocA)
Goal state:
 At(Rocket, LocB), Unloaded(G,LocB), Unloaded(O,LocB)
Variables: L, Dest

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

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

Another Example - Rocket Ship

Literals: Rocket, G, O, LocA, LocB
Start state:
  At(Rocket, LocA), Has-Fuel(),
  Unloaded(G,LocA), Unloaded(O,LocA)
Goal state:
  At(Rocket, LocB), Unloaded(G,LocB), Unloaded(O,LocB)
Variables: L, Dest

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

Another Example - Rocket Ship

Literals: Rocket, G, O, LocA, LocB

Start state:

    At(Rocket, LocA), Has-Fuel(),
    Unloaded(G,LocA), Unloaded(O,LocA)

Goal state:

    At(Rocket, LocB), Unloaded(G,LocB), Unloaded(O,LocB)

Variables: L, Dest

Load:

P:

A:

D:

Another Example - Rocket Ship

Literals: Rocket, G, O, LocA, LocB
Start state:
At(Rocket, LocA), Has-Fuel(),
Unloaded(G,LocA), Unloaded(O,LocA)
Goal state:
At(Rocket, LocB), Unloaded(G,LocB), Unloaded(O,LocB)
Variables: L, Dest, Pkg

The rocket needs to load a specific package G or O. The load action doesn't care which package it is. We need a variable pkg to use.

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

Another Example - Rocket Ship

Literals: Rocket, G, O, LocA, LocB
Start state:
    At(Rocket, LocA), Has-Fuel(),
    Unloaded(G,LocA), Unloaded(O,LocA)
Goal state:
    At(Rocket, LocB), Unloaded(G,LocB), Unloaded(O,LocB)
Variables: L, Dest, Pkg

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

Another Example - Rocket Ship

Literals: Rocket, G, O, LocA, LocB

Start state:

At(Rocket, LocA), Has-Fuel(),
Unloaded(G,LocA), Unloaded(O,LocA)

Goal state:

At(Rocket, LocB), Unloaded(G,LocB), Unloaded(O,LocB)

Variables: L, Dest, Pkg

No new variables needed for unload.

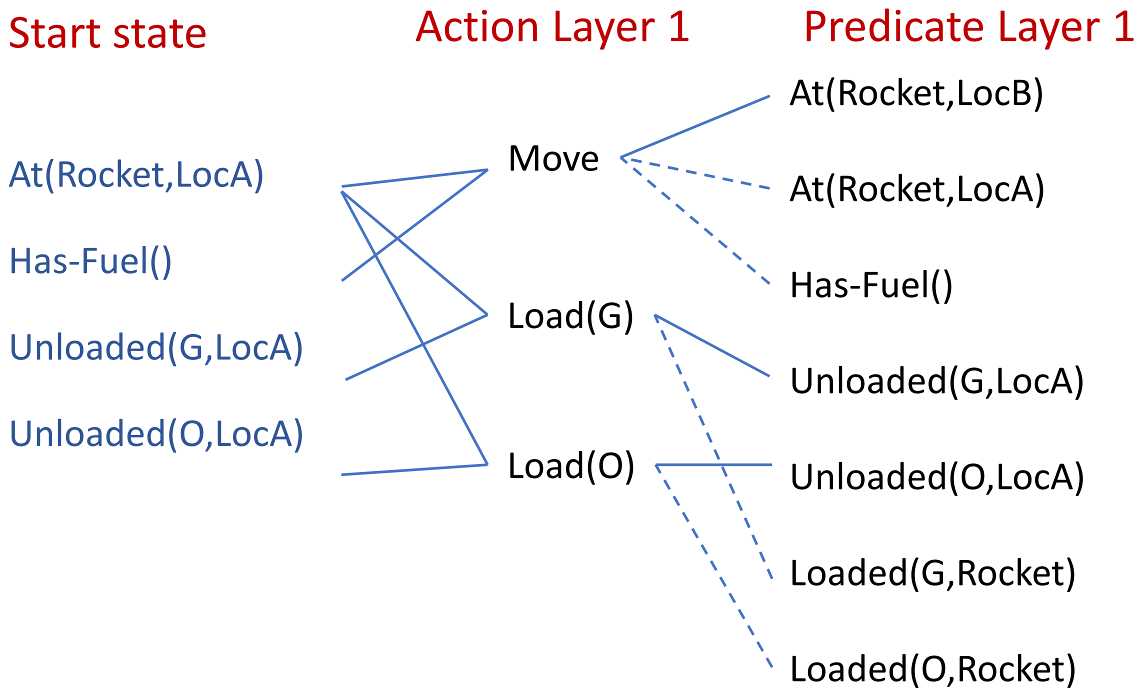
Unload:

P: At(Rocket,Dest), Loaded(Pkg,Rocket)

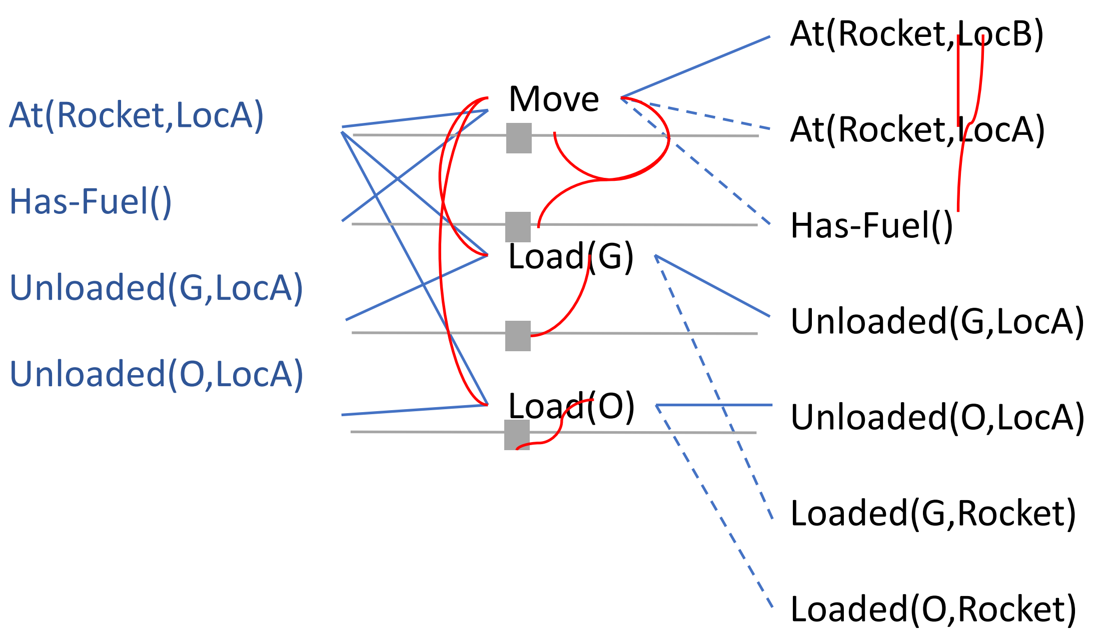
A: Unloaded(Pkg,Dest)

D: Loaded(Pkg,Rocket)

Rocket Ship GraphPlan Graph



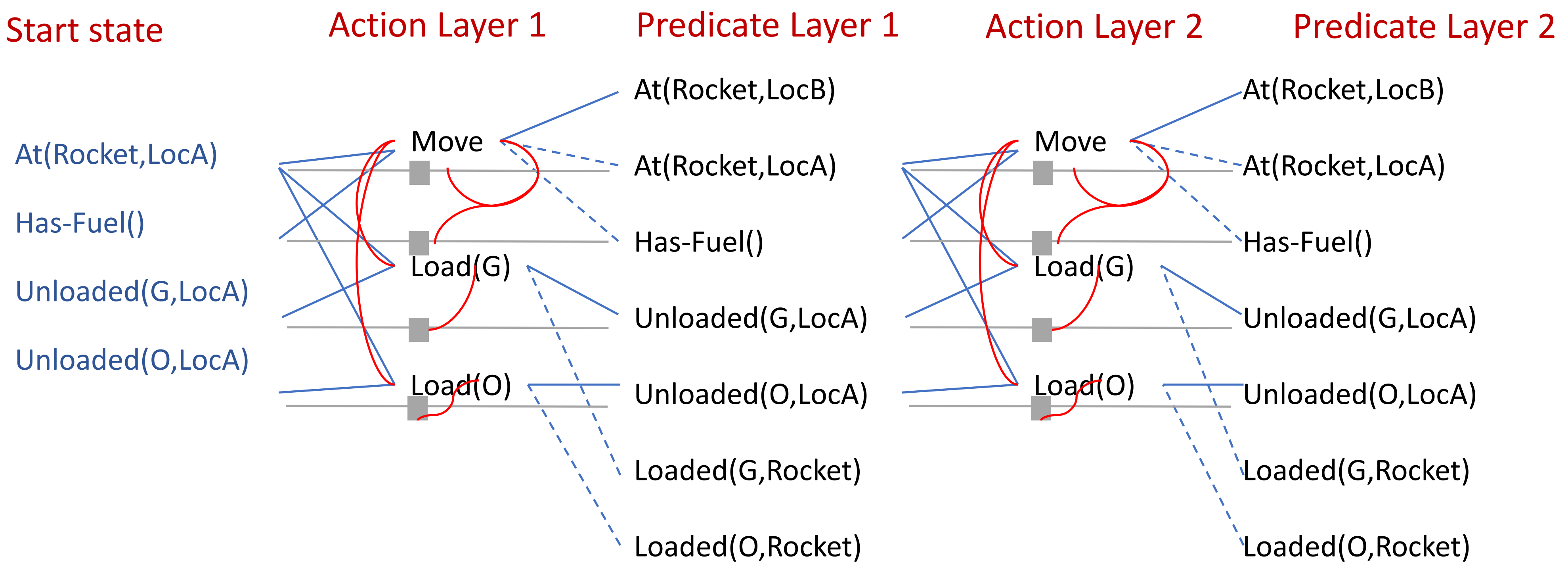
Rocket Ship GraphPlan Graph



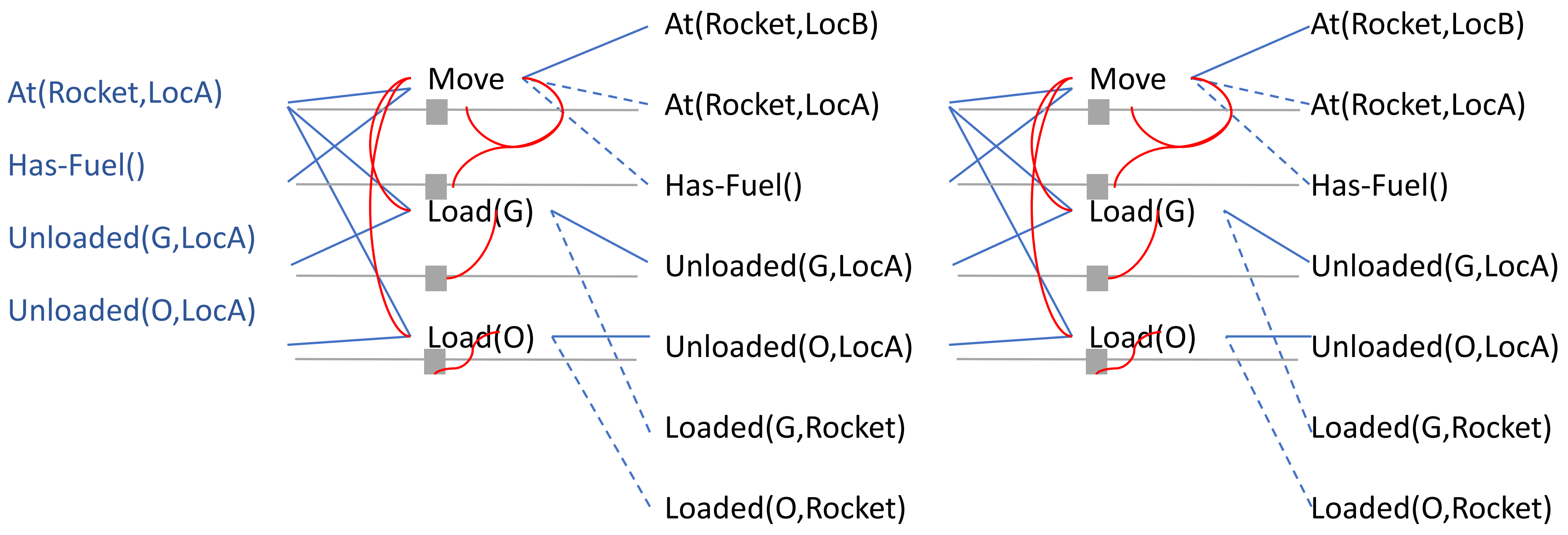
Mutex Actions
Interference:
Move deletes At which is a precondition of Load
Inconsistent:
Move deletes At but noop adds it
Move deletes Has-Fuel but noop adds it

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

Rocket Ship GraphPlan Graph



Rocket Ship GraphPlan Graph



At time 1: Move can be performed OR both Load actions

At time 2: Possible plans include:

Load(G), Load(O), Move(LocB) ← reachable goal in two steps but feasible in three

Load(G), Move(LocB)

Load(O), Move(LocB)