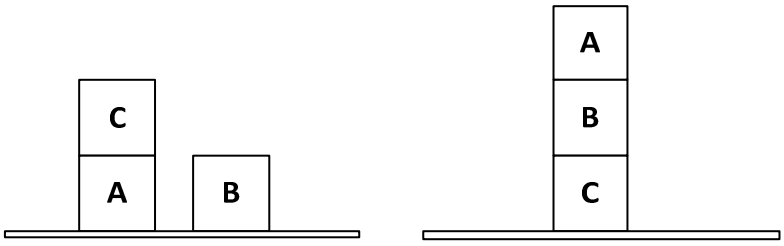
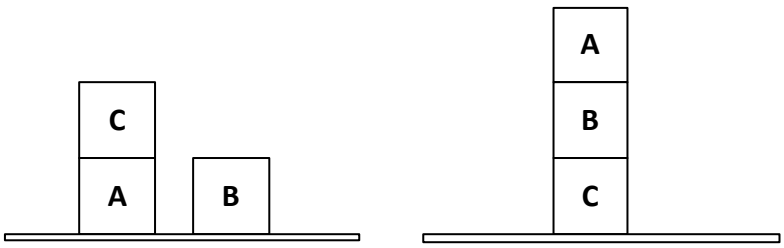
AI Course Fall HT 2/2015:
Exercise 4: Planning

Task 13 Classical State-Space Planning

Consider the following situation:



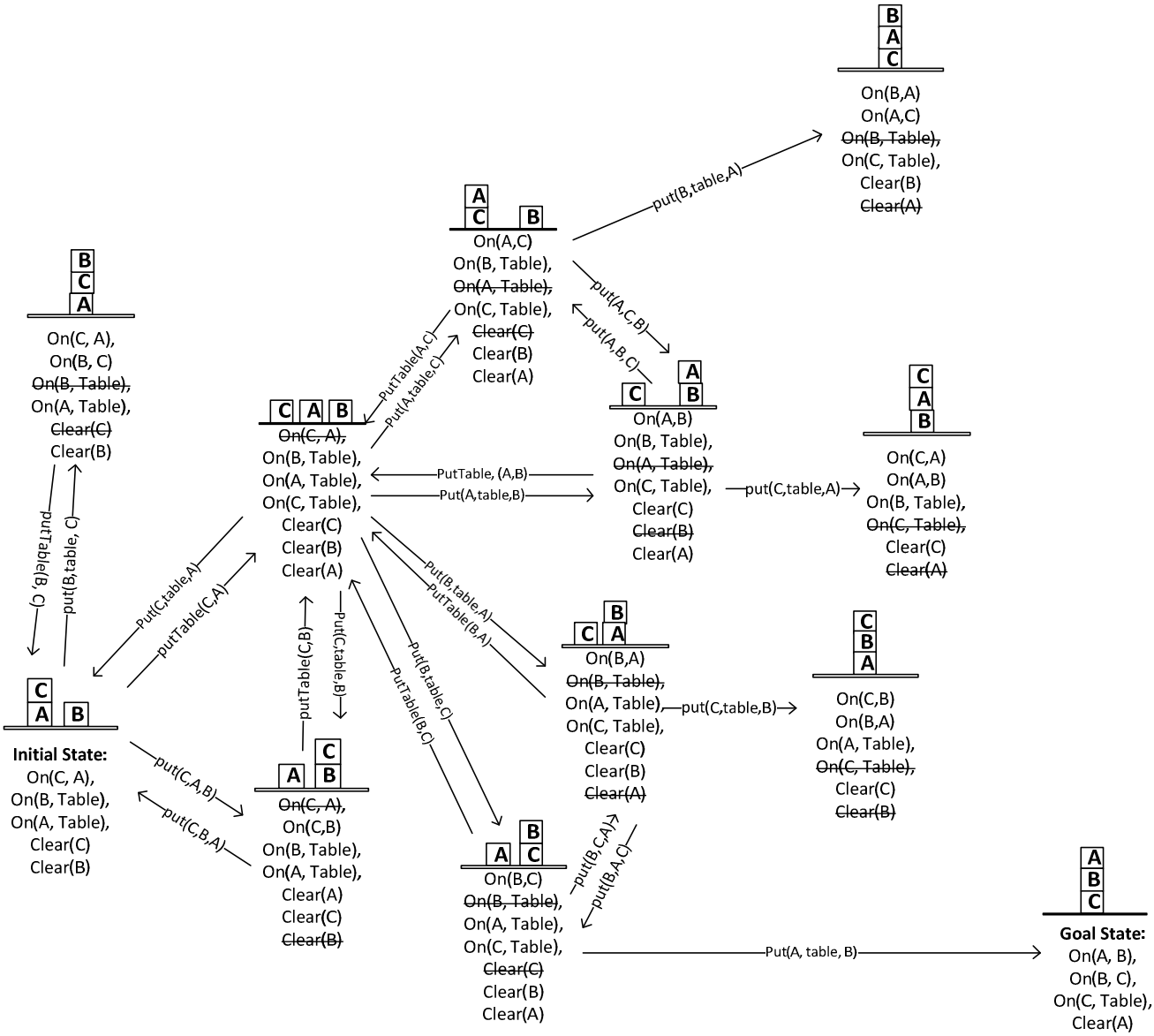
a) Describe it using the predicates used in the lecture
b) Solve it using progression search in a breadth-first way (when the branching becomes too much just indicate different paths). Discuss the application of different heuristics for improving search.
c) Solve it using regression search. The situation description is complete, so there are no additional blocks, etc.



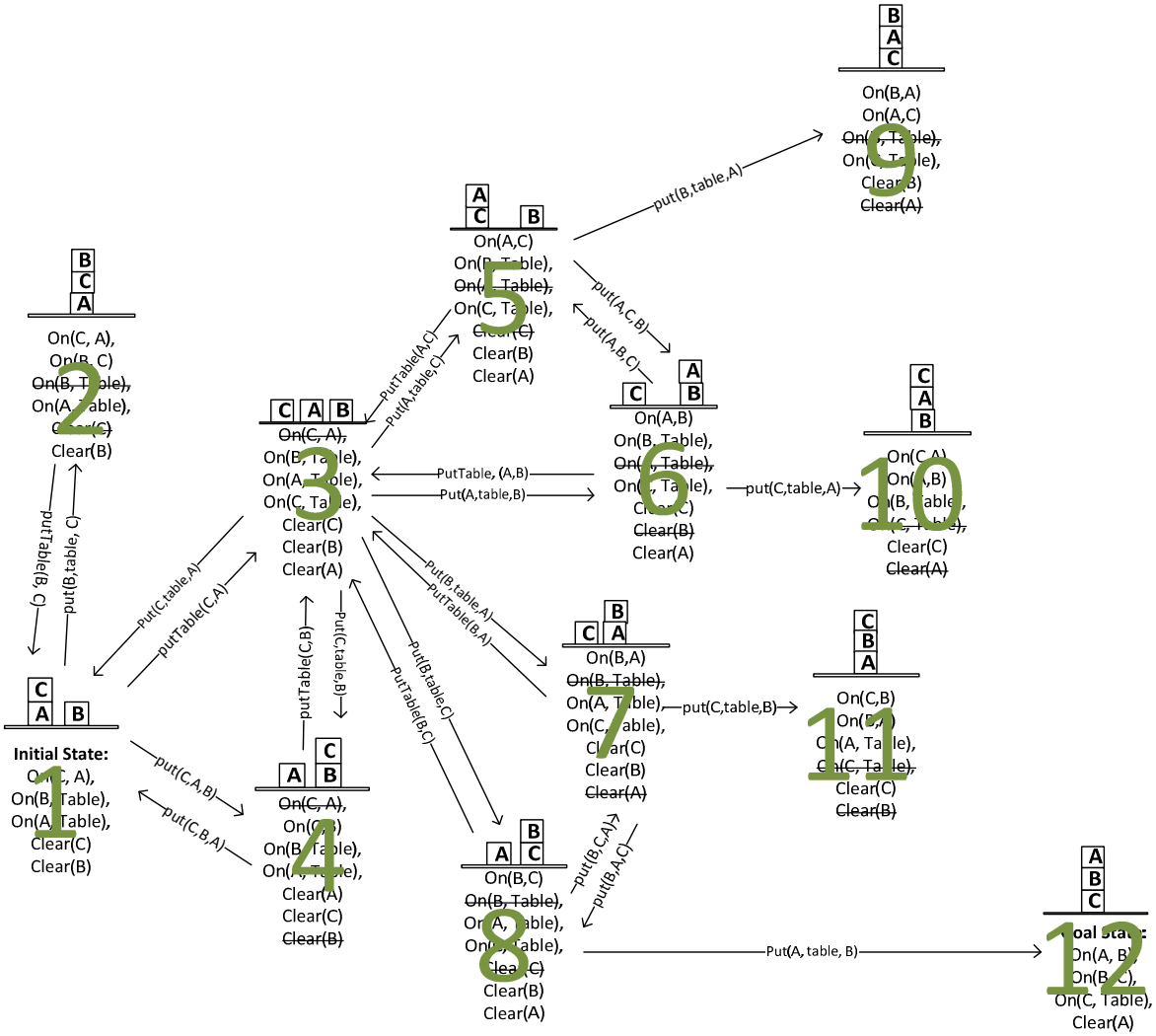
Initial State:
On(C, A),
On(B, Table),
On(A, Table),
Clear(C)
Clear(B)

Goal State:
On(A, B),
On(B, C),
On(C, Table),
Clear(A)

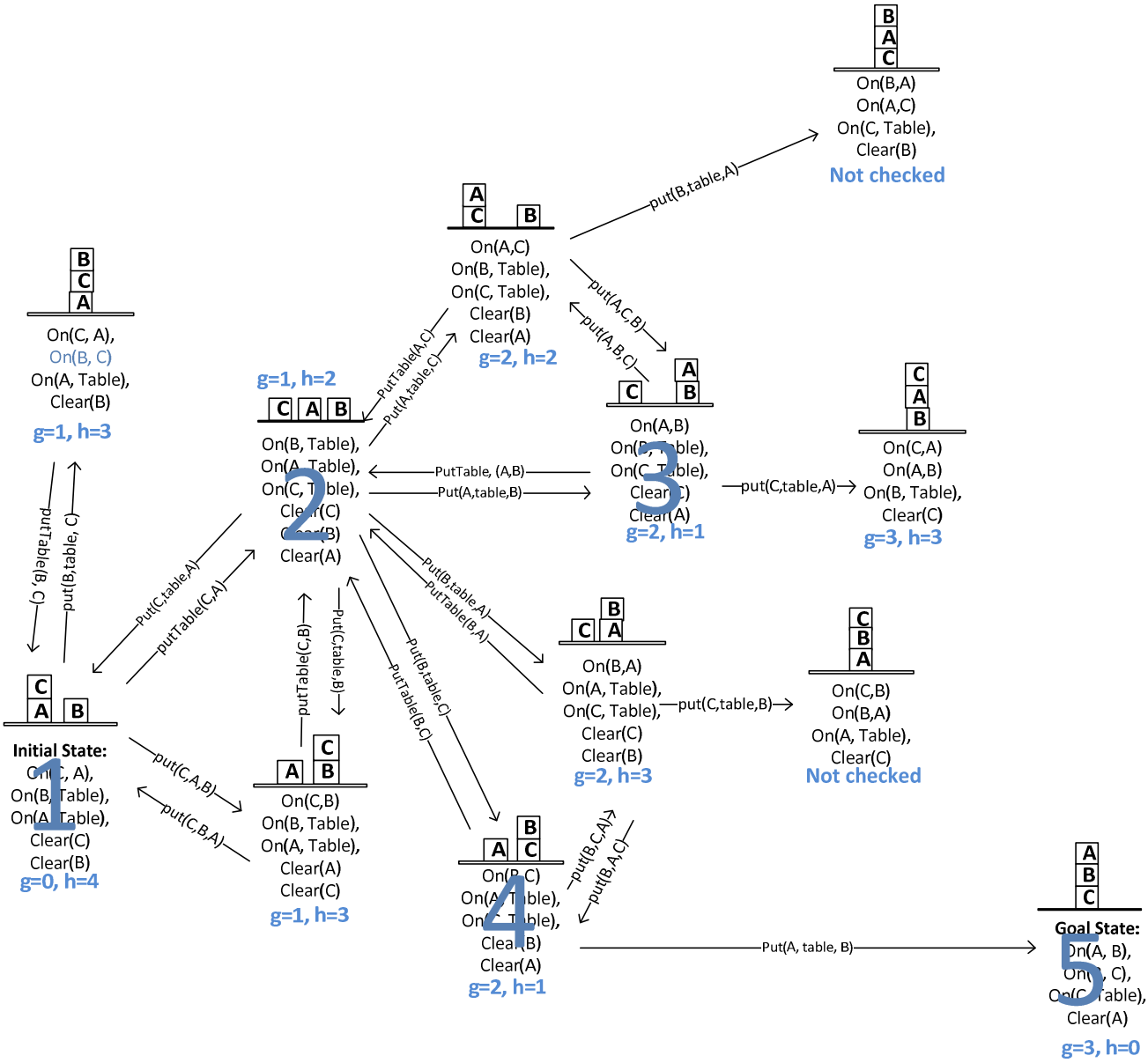
b) The full search tree for forward-chaining search / progression of the Sussmann Anomaly:



Breadth First Search Strategy for Progression with top-down strategy

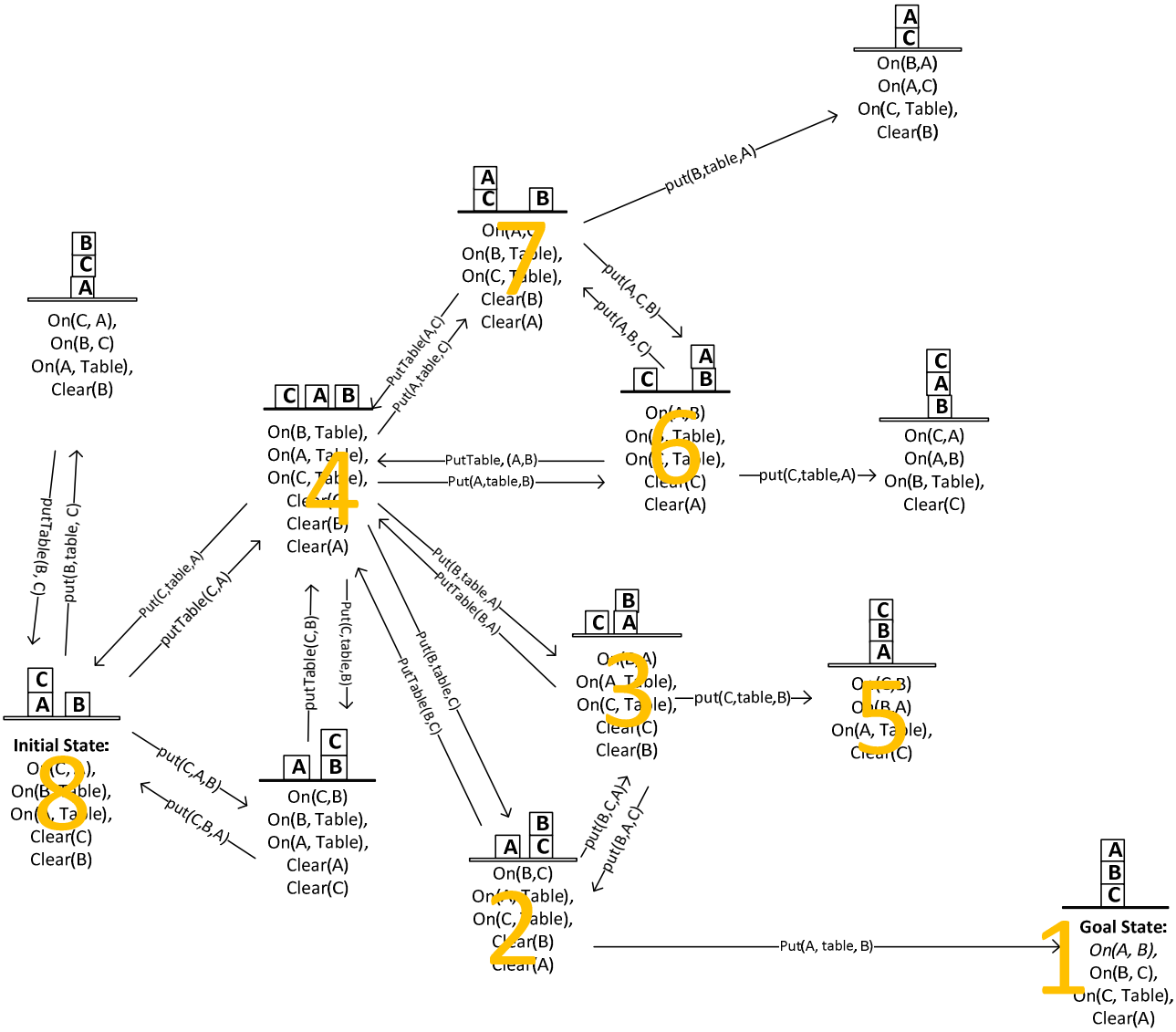


So, going through the search tree like that is not really smart.
What would be, if we would apply as a heuristic the number of sub-goal that are still to be fulfilled? → corresponds to ignoring both interaction between subgoals and that one action may destroy already achieved subgoals.



The heuristic works quite well – actually I just counted a node when we calculated its successors, for just calculating the heuristic I did not assume that this is a search step.

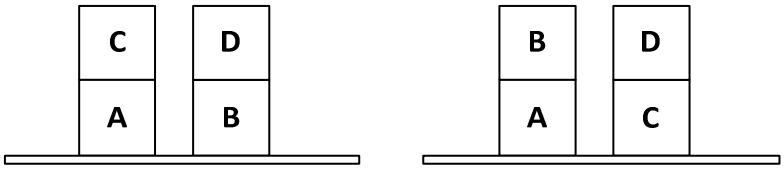
So what about Backward chaining ?



This is regression also with a bad ordering on how to test the childen. Nevertheless breadth-first search for backward chaining/regression seems to be better than for forward-chaining. This is a question of the branching factor: how many actions are applicable in a state versus how many actions can produce a subgoal. Usually the second approach has to consider less branches.

Task 14 Plan-Space Search

Consider the following situation:



Initial State **Goal State:**

a) Describe it using the predicates used in the lecture

Initial State:

on(A, Table)
on(C, A)
clear(C)
on(B, Table)
on(D,B)
clear(D)

Goal State:

on(A, Table)
on(B, A)
clear(B)
on(C, Table)
on(D,C)
clear(D)

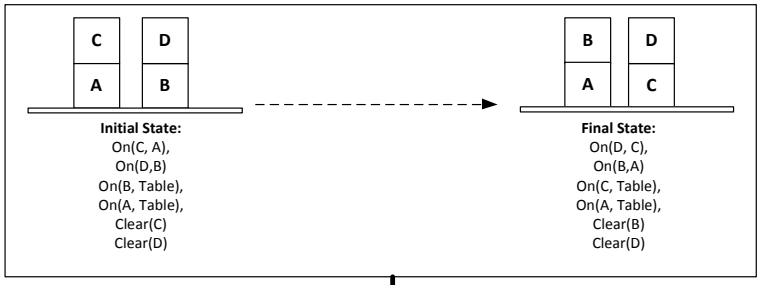
b) Write down a partial order plan that would solve the problem

A partial order plan is a set of actions with sequencing information. In this example we have so many interferences that this gives us a fully specified sequence.

PutOn(B,A), PutOn(D,C) and PutOn(C,Table) with the following constraints:
PutOn(C, Table) < PutOn(D, C) < PutOn(B,A)

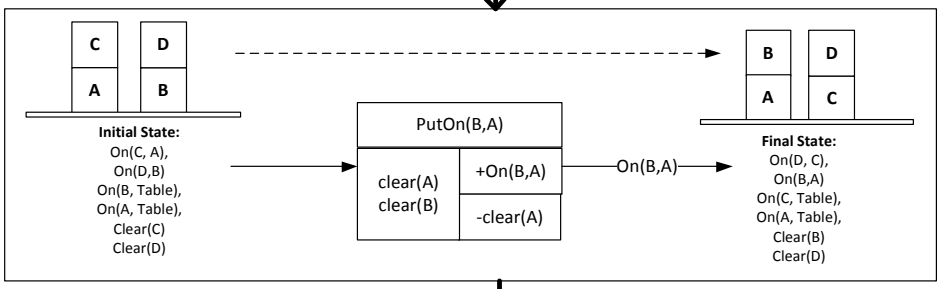
c) Solve it using **plan-space search** for generating a partial order plan

Pay attention: we just show one path through the graph... each step has several alternatives….



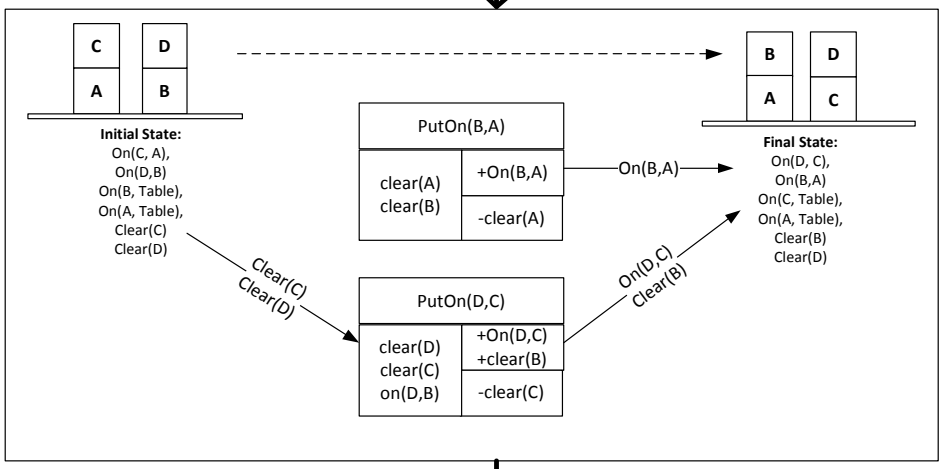
Start with adding a sequencing constraint – start comes before end

There are a number of requirements ... This is a search tree in plan space, we could follow different path adding different actions...



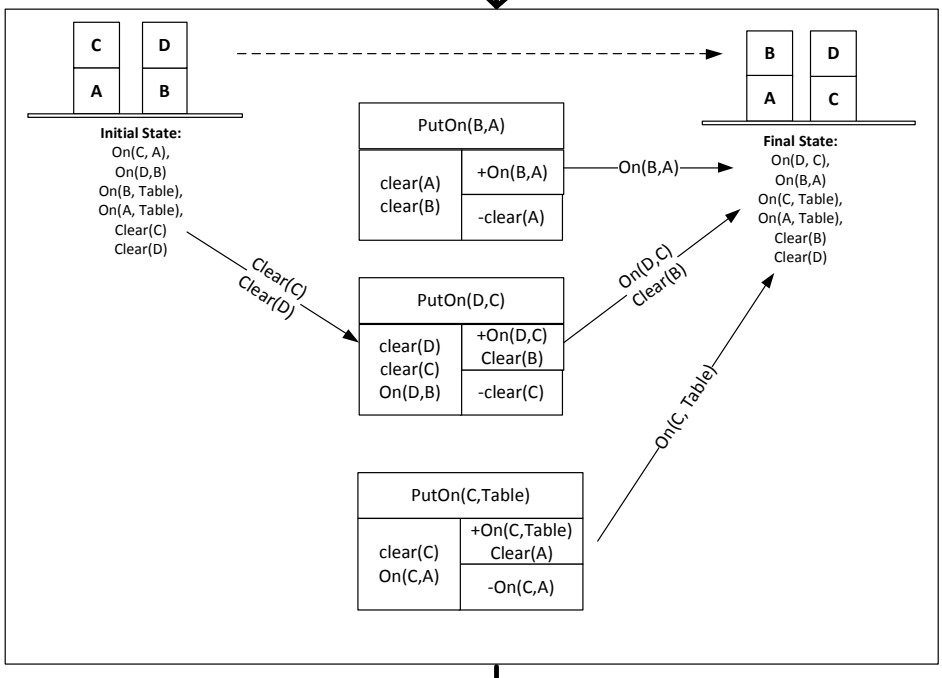
Add a plan step that eliminates the requirement on(B,A)

Requirements left:
on(D,C),
on(C, Table)
clear(B)

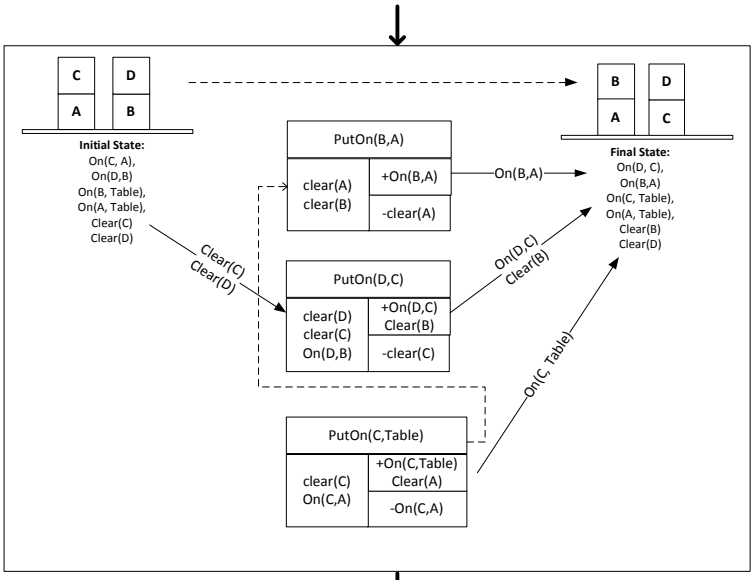


Add a plan step that eliminates a requirement, on(D,C)

Afterwards, we have left:
On(C, Table)

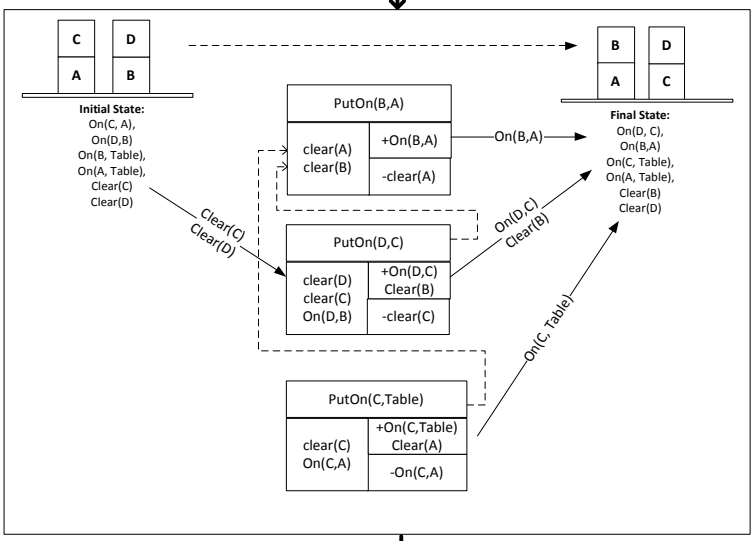


Add a plan step that eliminates a requirement On(C,Table)



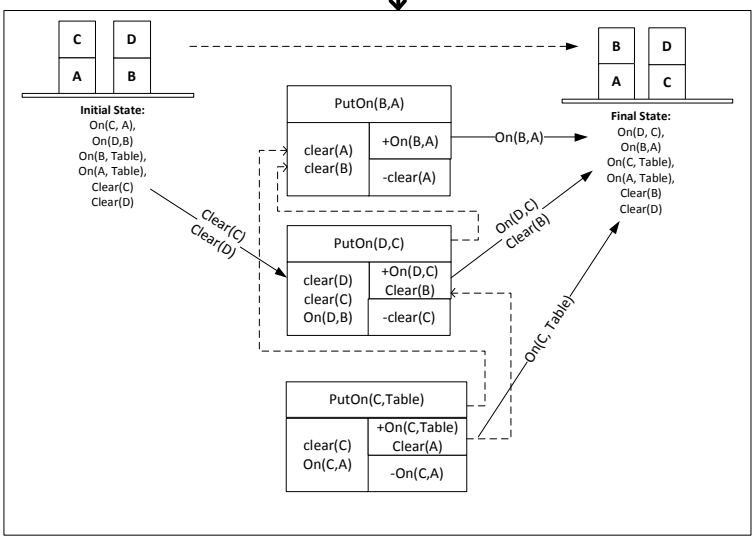
The goal test tells us that there are still problems to be solved:

PutOn(B,A) needs Clear(A) Which is produced by PutOn(C, Table) → so we add a relation to show that PutOn(C, Table) needs to be done before PutOn(B,A)



The goal test tells us that there are still problems to be solved:

PutOn(B,A) needs a clear(B) which is produced by PutOn(D,C) → so we add a relation that PutOn(D,C) must happen before PutOn(B,A)



The goal test tells us that there are still a problems to be solved:

PutOn(D,C) is a clobberer for clear(C) which is needed for PutOn(C, Table). It destroys that precondition... So we need to do PutOn(C, Table) before the clobberer

So, this is a search state in which our goal test says that all subgoals are fulfilled and all relations are settled.
→ goal state of the plan space search.
The result is a set of three actions: PutOn(B,A), PutOn(D,C) and PutOn(C,Table) with the following constraints:
PutOn(C, Table) < PutOn(D, C) < PutOn(B,A)

Task 15 Modeling Planning Problems

Automated driving is currently one of the hottest topics in AI research. It involves many levels of decision making. Formulate the action schemata that would be necessary for allowing the automated car to decide about overtaking a slower vehicle on a 2-lane highway.

Planning for overtaking? why this is a relevant task for automated planning?

1. discrete decisions to be done
2. execution just starts when sucsessful termination can be forseen (→ planning must be successful)
3. gains relevance with automated vehicles
4. good example to illustrate the relevance of abstraction

So, what do we need?

Actions:

* change lane

* pass

* eventually speedUp

Situation/State Description:

Object Types: car(C), lane(L)

rightTo (L1, L2) is true, if L1 is right of L2

infront (C1, C2), is true, if C1 drives infront of C2

distance (C1, C2) is a funkfion which returns the distance

onLane (C, L) is true, if car C goes on lane L

faster (C1,C2) C1 drives faster than C2

freeLane(L1) the lane is completely free

Initial state:

rightTo(lane1,lane2),
onLane(me,lane1),
onLane(slowCar,lane1),
infront(slowCar,me)

GoalState

onLane(me,lane1),
infront(me, slowCar)

ChangeLane(Car, FromLane, ToLane)

Precond: Lane(FromLane), OnLane (Car,FromLane), [rightTo (FromLane, ToLane) OR rightTo (ToLane,FromLane)], [freeLane(ToLane) OR (onLane(AnotherCar,ToLane) and distance(AnotherCar,Car)>100m)

Effect: NOT OnLane(Car,FromLane), OnLane(Car,ToLane)

Pass(Car, OtherCar)

Precond: onLane(Car,Lane1), onlane(OtherCar, Lane2), Lane1!=Lane2, faster(Car,Othercar), infront(OtherCar, Car)

Effect: infront(Car,OtherCar), NOT (infront(OtherCar, Car)

DoRelativeSpeedUp(Car, OtherCar)

Precond: [(onLane(Car,Lane) and NOT onlane(OtherCar, Lane] OR [onLane(Car,Lane) and onlane(OtherCar, Lane) and distance(Car,OtherCar)>100m] OR infront(Car,OtherCar)

Effect: faster(Car,OtherCar),

Now we would need to test whether this set of actions would be sufficient?

changeLane(car, mylane, leftlane)
doRelativeSpeedUp(car, othercar)
pass(car, othercar)
changeLan(car,leftlane, mylane)