Planning

Outline

- Introduction to the planning problem
- Formulating planning problems
- Languages for planning problems
- Planning algorithm - search in plan space

Planning

- The task of coming up with a sequence of actions that will achieve a goal
- What plans do you have for this evening/this weekend?
- Why should an intelligent agent be able to plan?
  - If you fail to plan, you plan to fail

    Benjamin Franklin
- Agents with planning capabilities differ from pure reactive agents
  - Which type is better?

Applications of Planning

- Mobile Robots
- Virtual agents
- Action choice + resource handling
  - for transportation of goods
  - at schools, hospitals
  - Hubble Space Telescope scheduler
  - Work-flow management
- Software test case generation

Plans and Actions in Jason Agent Programming Language

```
/* Initial beliefs and rules */
current_tactic(give_and_go(su_monday, ras_ruby)).

/* Initial goals */
!start_playing.

/* Plans */

+!start_playing  <- connect_to_SL("rasika", "Wonderland NW,178,164,1001");
                                  !check_connected.

+!check_connected: not connected <-.wait(2000);
                                  !!check_connected.

+!check_connected: connected <-  action("run","210,241,1001").

+successful_pass_by_upkick(OtherAgent, Me)[state(N)]:current_tactic(give_and_go(OtherAgent, Me))& .my_name(Me)
     <-  .term2string(OtherAgent,OtherAgentStr);
         .concat(OtherAgentStr, " passed ME the ball by an UP KICK: start monitoring for expectation", Msg);
          msg_sl(Msg);
         .start_monitoring("fulf", "reach_penaltyB", "expectation_monitor", NegShootRangeConjunction, FulfProposition, [N])


+fulf("fulf_reach_penaltyB", N) <-msg_sl("Expectation FULFILLED: pass the ball back");
                                 .stop_monitoring("fulf", "reach_penaltyB", "expectation_monitor");
                                .stop_monitoring("viol", "reach_penaltyB", "expectation_monitor");
                                 action("pass","up shot").

+viol("viol_reach_penaltyB", N)
                         <-.stop_monitoring("fulf", "reach_penaltyB", "expectation_monitor");
                         .stop_monitoring("viol", "reach_penaltyB", "expectation_monitor");
                         msg_sl("Expectation VIOLATED: run to penalty area");
                         !choose_and_enact_new_tactic.

+!choose_and_enact_new_tactic : .my_name(Me) <-
        -+current_tactic(solo_run(Me));
        action("run","54,101,790");
        -+current_tactic(give_and_go(su_monday, ras_ruby)).
```

Planning

- Given the current state and the state of the world you want to achieve (goals),
  - determine how (and when) to do it (plan).
  - I.e. Figuring out a sequence of actions to achieve the goal
- We will restrict this discussion to deterministic, fully observable, static and discrete situations => Classical Planning
  - Static => Changes occur only when the agent acts

A Planning Problem

- Goal: Have a birthday party
- Current state:
  - Agent is at home
  - There is enough flour in the larder
  - But no butter or sugar
- Things to do
  - Invite friends
  - Make a cake

Planning Problem

- Given:
  - A way to describe the world. I.e. state representation
  - An initial state of the world
  - A goal description
  - A set of possible actions to change the world
- Find:

  Planning problem

  - A sequence of actions to go from initial state to a state where the goal is achieved

Planning Requirements

- A way to express the planning problem
- An algorithm to solve the problem
- Planning algorithms based on logical approaches should take advantage of the logical structure of the problem
  - For this, the problem should be expressed in a suitable logical language

Language of Planning Problems

- The language should be
  - Expressive enough to describe a variety of problems
  - Restrictive enough to let efficient algorithms to operate on it
- It should represent **states**, **goals** and **actions**
- Basic language of classical planners - PDDL (Planning Domain Definition Language)
-

PDDL Syntax

- State representation
  - Conjunction of ground atomic fluents
  - Literals can be
    - Propositional (*Poor* ∧ *Unknown*)
    - First order ($At(Plane_1, Sydney) \wedge At(Plane_2, Perth)$)
  - First order literals must be
    - Ground ($At(x,y)$) ✖
    - Function-free($At(Father(Fred), Perth)$) ✖
  - **Closed-world assumption**: Whatever is not mentioned is assumed to be false
  - Unique name assumption: $Plane_1 \mathrel{!=} Plane_2$

PDDL Syntax

- Goal representation
  - Partially specified state
    - E.g.
      - *At (Plane*$_1$*, Melbourne)*
      - *Rich* $\wedge$ *Famous*
  - A propositional state *s* satisfies goal *g* if *s* contains all the atoms of *g*
  - E.g. State (*Rich* $\wedge$ *Famous* $\wedge$ *Happy*) satisfies the goal (*Rich* $\wedge$ *Famous*)

PDDL Syntax

- Action representation
  - Specified using an action schema containing
    - **Preconditions**: must hold to execute the action
    - **Effects:** changes caused by the action
  - E.g.

Action (*Fly(p, from, to*),

PRECOND: *At (p, from)* $\wedge$ *Plane(p)* $\wedge$ *Airport(from)* $\wedge$ *Airport(to)*

EFFECT: $\neg$*At (p, from)* $\wedge$ *At (p, to)*

PDDL Syntax

- Action name and parameter list
- Identifies the action

Action (*Fly(p, from, to)*,

PRECOND: *At (p, from)* $\wedge$ *Plane(p)* $\wedge$ *Airport(from)* $\wedge$ *Airport(to)*

EFFECT: ¬At (p, from) $\wedge$ *At (p, to)*

- States what should be true in a state for the action to execute
- Conjunction of function-free (+)ve literals
- Variables should appear in the parameter list

- Describes the state change when the action is executed
- Contains function-free literals
- Variables should appear in the parameter list
- Can be broken into an **add list** and a **delete list**

Action (*Fly(p, from, to*),

PRECOND: $At\ (p, from) \wedge\ Plane(p)\ \wedge\ Airport(fr) \wedge\ Airport(to)$

EFFECT: $\neg At\ (p, from)\ \wedge\ At\ (p, to)$

Action ($Fly(p, from, to)$,

PRECOND: $At(p, from) \land Plane(p) \land$
$Airport(from) \land \neg Airport(to)$

$EFFECT: \neg At(p, from) \land At(p, to)$)

PDDL Semantics

- An action is applicable in any state that satisfies the precondition
- How to establish applicability?

State - $At(P_1, JFK) \land At(P_2, SFO) \land Plane(P_1) \land Plane(P_2) \land Airport(JFK) \land Airport(SFO)$

satisfies

Precondition - $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$

If substituted with

$\{p/P_1, from/JFK, to/SFO\}$

PDDL Semantics

- Starting at state ***s***, result of executing action ***a*** is a state ***s'***
  - Contains (+)ve literals in the effect of *a*
  - (-)ve literals are removed
  - Otherwise same as *s*

- **Assumption** – every literal not mentioned in the effect remains unchanged
  - Avoids frame problem

- Solution to the planning problem – an action sequence that, when executed, results in a state that satisfies the goal

ADL (Action Description Language)

- In recent years, it has been noted that STRIPS is *not* sufficiently expressive

- This has resulted in many language variants. ADL is one of them.

- The same *Fly* action in ADL

Action(*Fly(p:Plane, from:Airport, to:Airport*),
PRECOND: $At\ (p, from) \land (from \neq to)$
EFFECT: $\neg At\ (p, from) \land At\ (p, to)$

STRIPS Vs ADL

Action (*Fly(p, from, to*),

PRECOND: $At\ (p, from) \land Plane(p) \land Airport(from) \land Airport(to)$

EFFECT: $\neg At\ (p, from) \land At\ (p, to)$

Action(*Fly(p:Plane, from:Airport, to:Airport*),

- PRECOND: $At\ (p, from) \land (from \neq to)$
- EFFECT: $\neg At\ (p, from) \land At\ (p, to)$

STRIPS Vs ADL

STRIPS Language	ADL Language
Only positive literals in states: $Poor \wedge Unknown$	Positive and negative literals in states: $\neg Rich \wedge \neg Famous$
Closed World Assumption: Unmentioned literals are false.	Open World Assumption: Unmentioned literals are unknown.
Effect $P \wedge \neg Q$ means add P and delete Q.	Effect $P \wedge \neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q.
Only ground literals in goals: $Rich \wedge Famous$	Quantified variables in goals: $\exists x\, At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place.
Goals are conjunctions: $Rich \wedge Famous$	Goals allow conjunction and disjunction: $\neg Poor \wedge (Famous \vee Smart)$
Effects are conjunctions.	Conditional effects allowed: **when** P: E means E is an effect only if P is satisfied.
No support for equality.	Equality predicate $(x = y)$ is built in.
No support for types.	Variables can have types, as in $(p : Plane)$.

Figure 11.1 Comparison of STRIPS and ADL languages for representing planning problems. In both cases, goals behave as the preconditions of an action with no parameters.

Example Planning Problem: Air Cargo

$$
\begin{aligned}
&Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK)\\
&\quad \wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)\\
&\quad \wedge Airport(JFK) \wedge Airport(SFO))\\
&Goal(At(C_1, JFK) \wedge At(C_2, SFO))\\
&Action(Load(c, p, a),\\
&\quad \text{PRECOND: } At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)\\
&\quad \text{EFFECT: } \neg At(c, a) \wedge In(c, p))\\
&Action(Unload(c, p, a),\\
&\quad \text{PRECOND: } In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)\\
&\quad \text{EFFECT: } At(c, a) \wedge \neg In(c, p))\\
&Action(Fly(p, from, to),\\
&\quad \text{PRECOND: } At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)\\
&\quad \text{EFFECT: } \neg At(p, from) \wedge At(p, to))
\end{aligned}
$$

$$
\begin{aligned}
&[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),\\
&\ Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P_2, SFO)]
\end{aligned}
$$

Example Planning Problem: The Spare Tire Problem

$Init(Tire(Flat) \wedge Tire(Spare) \wedge At(Flat, Axle) \wedge At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(obj, loc),$
  PRECOND: $At(obj, loc)$
  EFFECT: $\neg At(obj, loc) \wedge At(obj, Ground))$
$Action(PutOn(t, Axle),$
  PRECOND: $Tire(t) \wedge At(t, Ground) \wedge \neg At(Flat, Axle)$
  EFFECT: $\neg At(t, Ground) \wedge At(t, Axle))$
$Action(LeaveOvernight,$
  PRECOND:
  EFFECT: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$
    $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk))$

$$[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]$$

Planning Algorithms

- Use state-space search
  - Actions in a planning problem specify both preconditions and effects
  - Search can be performed in either direction
    - Forward state-space search
    - Backward state-space search

Planning Algorithms

- State-space search
  - Forward state-space search
  - Backward state-space search
- Plan space search
  - Partial order planning (POP)

State-space Search

- Possible because actions in a planning problem specify both preconditions and effects
- Nodes = states of the world
- Transitions between nodes = actions
- Path through the state space = plan

Forward state-space search

- Step cost is usually one per action. But it is easy to allow differential costs

- As usual, the main issue with Forward search is that it cannot avoid irrelevant actions
  - Thus it is inefficient
  - Need to use a good heuristic function to estimate distance from state to goal

Forward state-space search

- Also called progression planning

- Starting from the ***initial state***, consider sequences of actions that reach the goal

- For a given state, all ***actions*** whose preconditions are satisfied, are applicable
    - The successor state is determined by adding the positive effect literals and deleting the negative effect literals to the current state

- ***Goal test*** checks if the current state satisfies the goal

Forward State-space Search

Forward-search(O, s_0, g)

Forward State-space Search

- Forward search is sound
  - For any plan returned by any of its nondeterministic traces, this plan is guaranteed to be a solution
- Forward search is complete
  - If a solution exists then at least one of Forward search's nondeterministic traces will return a solution.

Deterministic Implementations

- Breadth-first search
- Depth-first search
- Best-first search (e.g., A*)
- Greedy search

Loop Checking

- Can be done by keeping a record of the state sequence (s_0, s_1, …., s_k) on the current path
- Modify the algorithm to return failure if there are nodes *i* and *k* where
  - $i<k$ s.t. $s_k = s_i$
  - $i<k$ s.t. $s_k \subseteq s_i$

Branching Factor

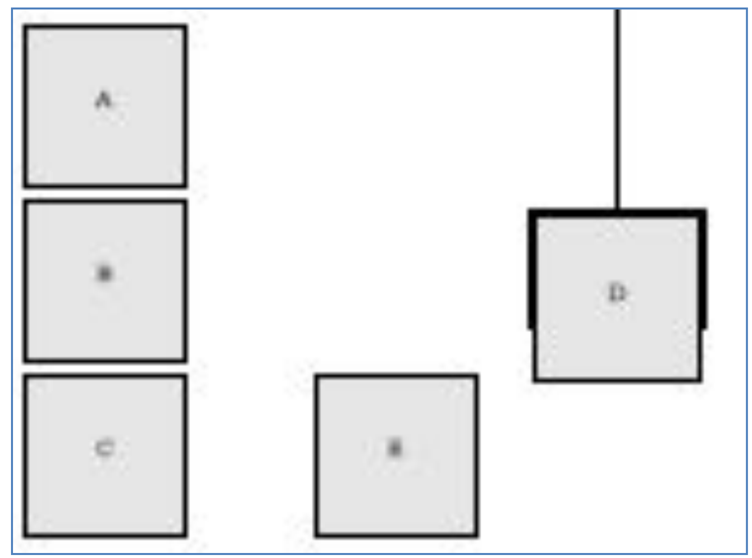
- Number of applicable plans that don’t progress towards goal
- Forward search has a large branching factor
- Example:
  - Blocks world

Blocks World

- Consists of a set of cube-shaped blocks sitting on a table.
- The blocks can be stacked
  - Only one block can fit directly on top of another
- A robot arm can pick up a block and move it to
  - The table OR
  - Top of another block
- The arm can pick up only one block at a time
  - Cannot pick up a block that has another one on it
- Goal => build one or more stacks of blocks according to the given order

Blocks World

- Possible state predicates
  - Holding(a)
  - handEmpty
  - onTable(a)
  - on(a, b)
  - clear(a)
- Possible actions
  - Pickup(a)
  - Putdown(a)
  - Stack(a, b)
  - unstack(a, b)

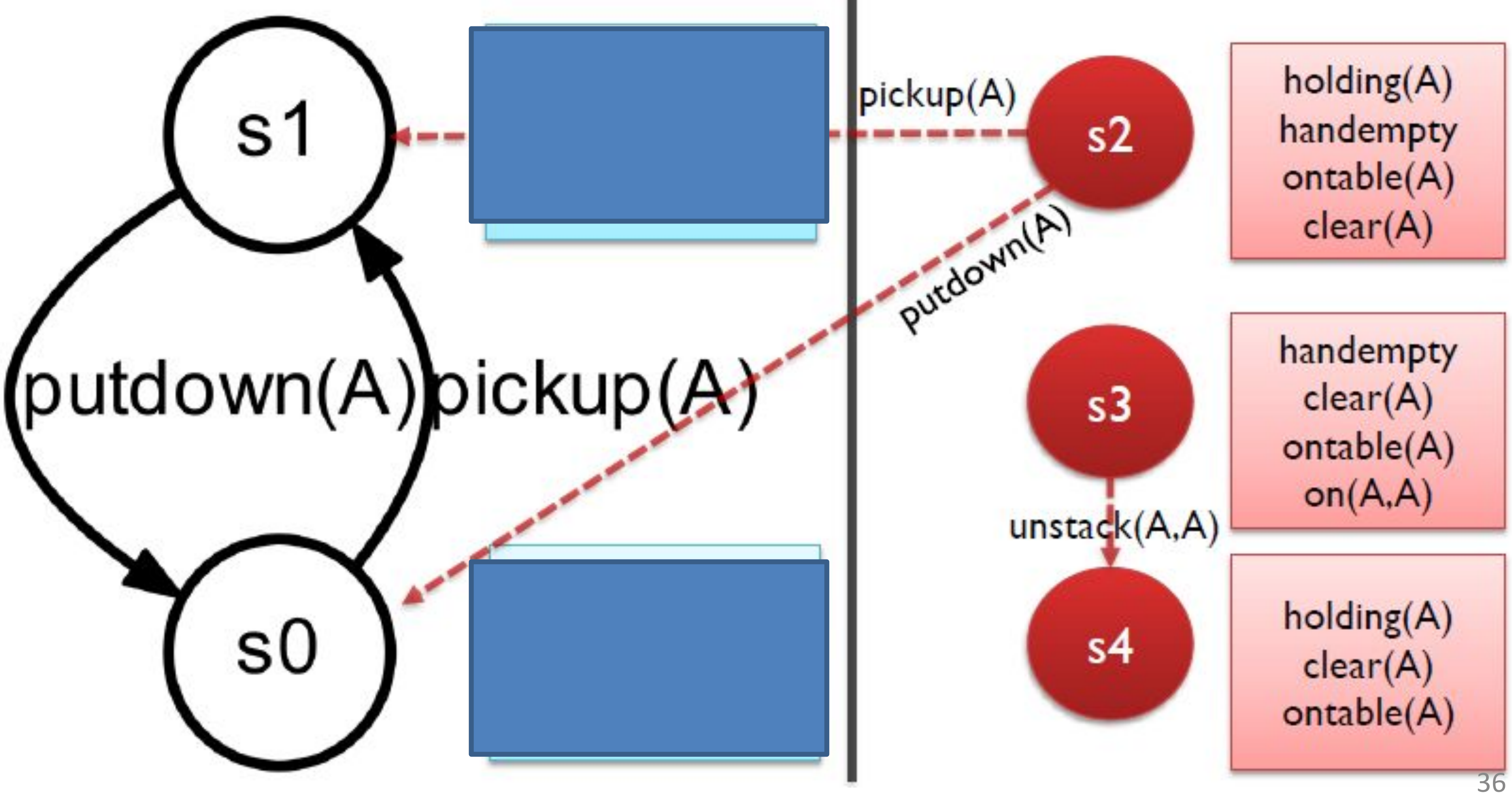


Blocks World - Size 1

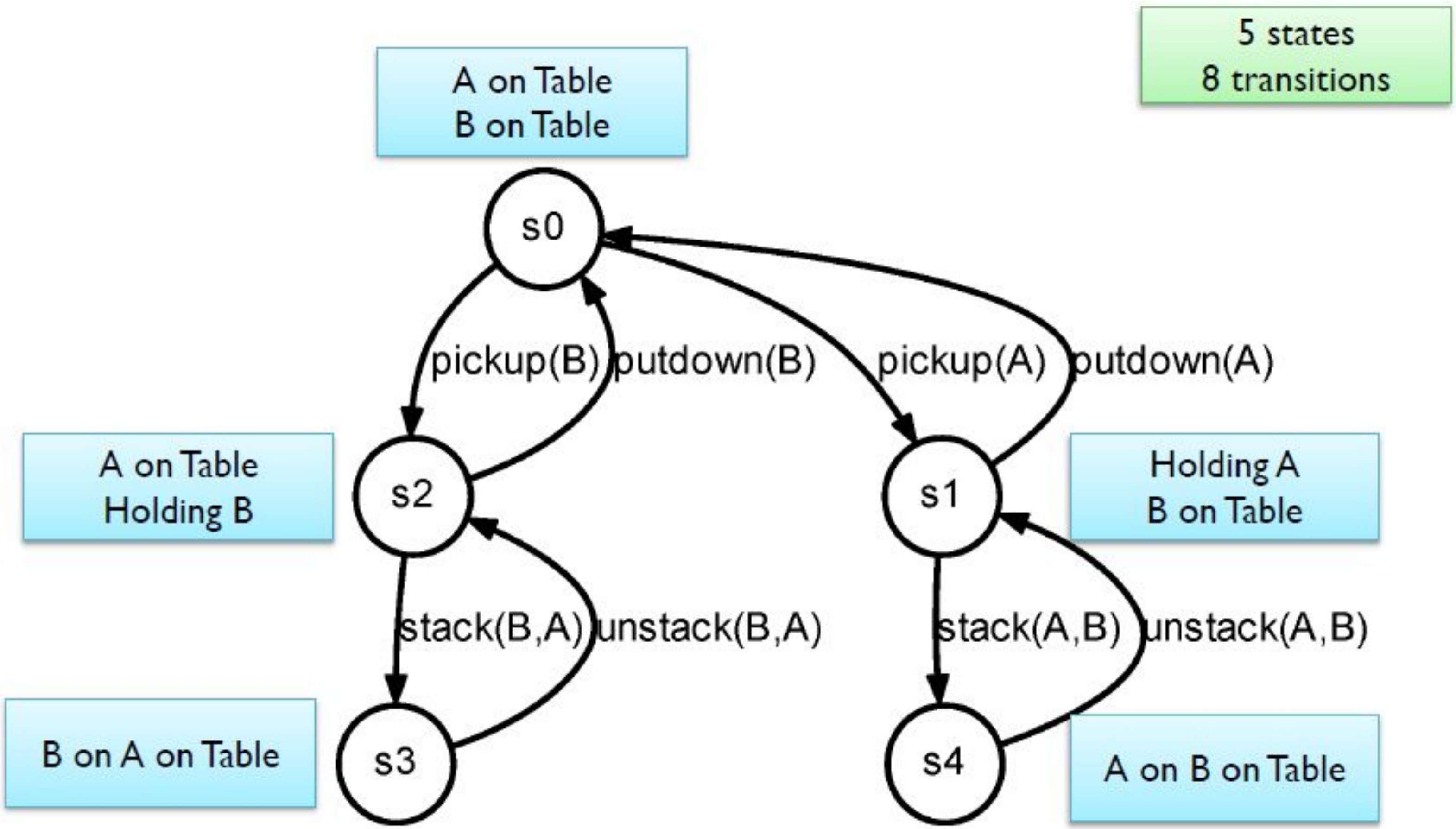
We assume we know the initial state
We only consider states *reachable* from there!

Here: Start with s0 = *all blocks on the table*

Many other states "**exist**",
but are not **reachable**
from the current starting state



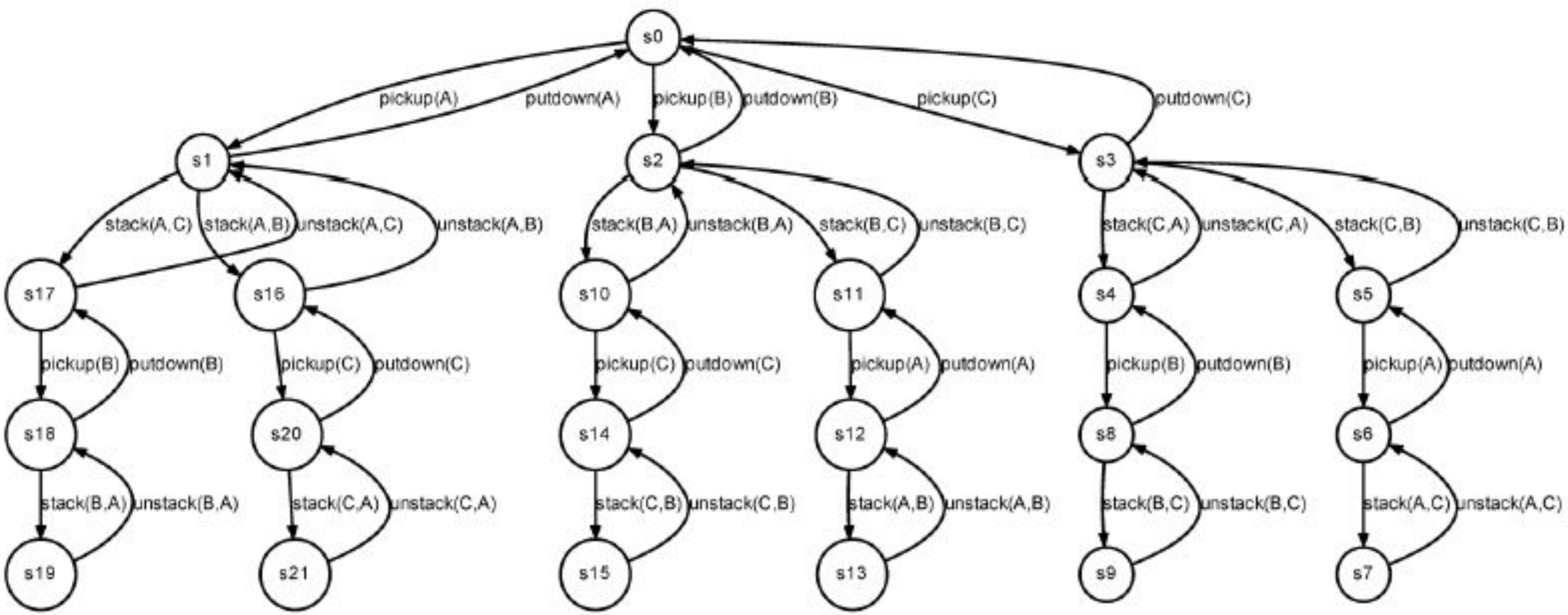
Blocks World - Size 2



Blocks World - Size 3

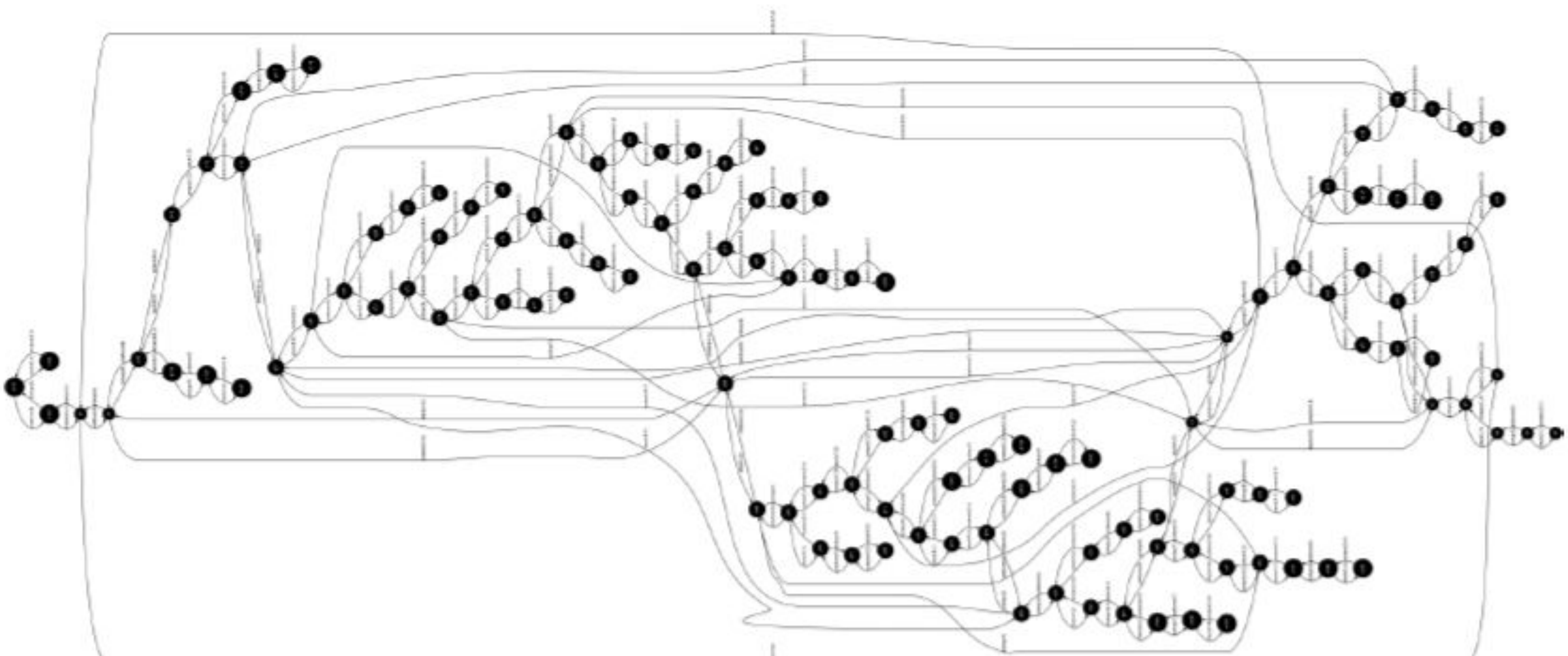
22 states
42 transitions

A on Table
B on Table
C on table

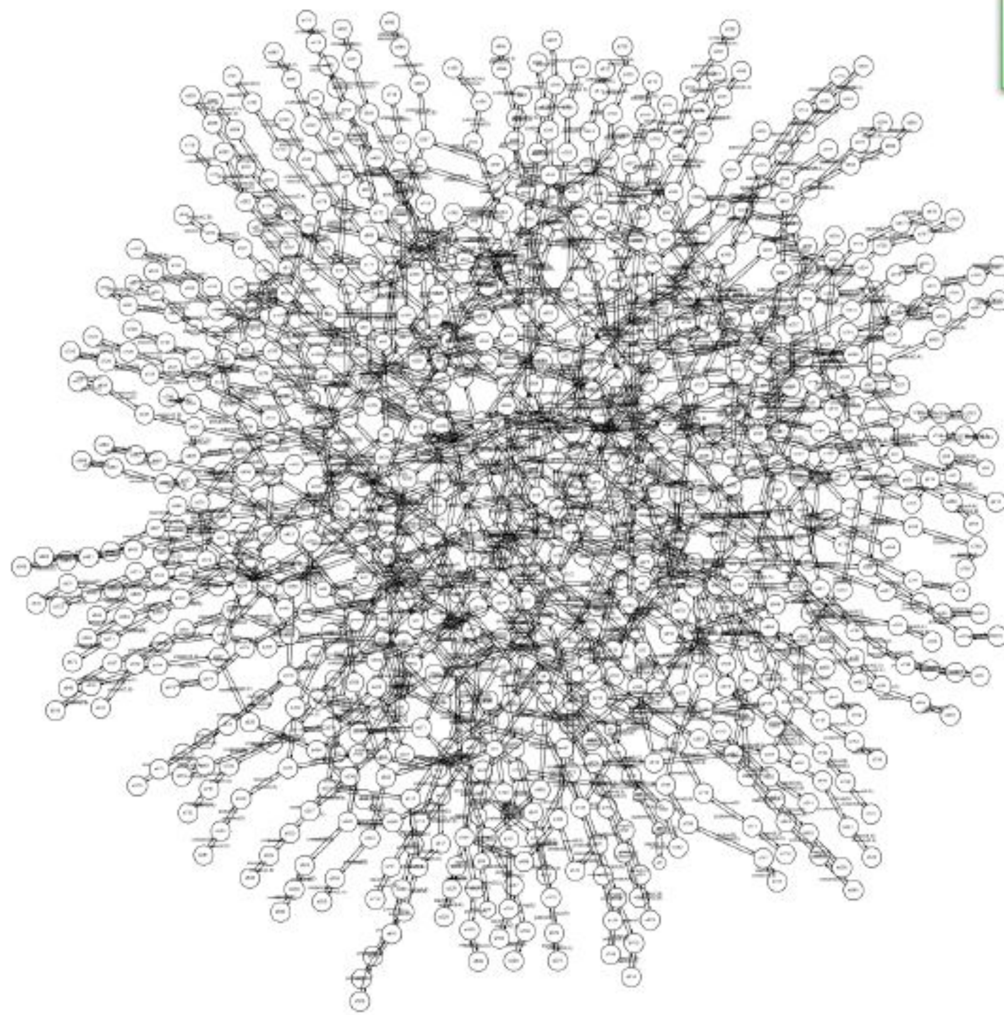


Blocks World - Size 4

125 states
272 transitions



Blocks World - Size 5



866 states
2090 transitions

Blocks World - Size 0-8

Blocks	States	States reachable from "all on table"	Transitions reachable
0	2	1	0
1	32	2	2
2	2048	5	8
3	524288	22	42
4	536870912	125	272
5	2199023255552	866	2090
6	36028797018963968	7057	18552
7	2361183241434822606848	65990	186578
8	618970019642690137449562112	695417	2094752
9	649037107316853453566312041152512	…	…
10	2722258935367507707706996859454145691648	…	…

Forward State Space Search

- Solution??
  - Domain-specific : search control rules, heuristics
  - Domain-independent : heuristics automatically generated from the problem description

Heuristic

A **heuristic** technique (/hjʉˈrɪstɨk/; Ancient Greek: εὑρίσκω, "find" or "discover"), often called simply a *heuristic*, is any approach to problem solving, learning, or discovery that employs a practical methodology not guaranteed to be optimal or perfect, but sufficient for the immediate goals. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution. Heuristics can be mental shortcuts that ease the cognitive load of making a decision. Examples of this method include using a rule of thumb, an educated guess, an intuitive judgment, stereotyping, profiling, or common sense.

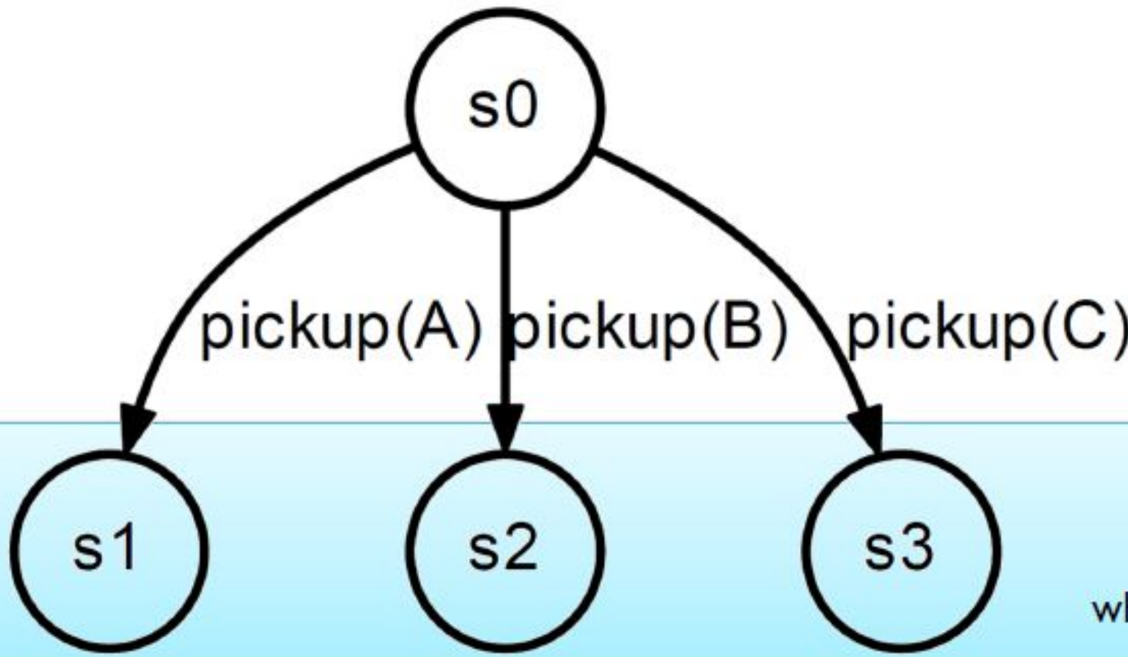
Use of Heuristics

Example: 3 blocks, all on the table in s0

We now have
1 *open node*,
which is *unexpanded*

Use of Heuristics

We visit s0 and expand it

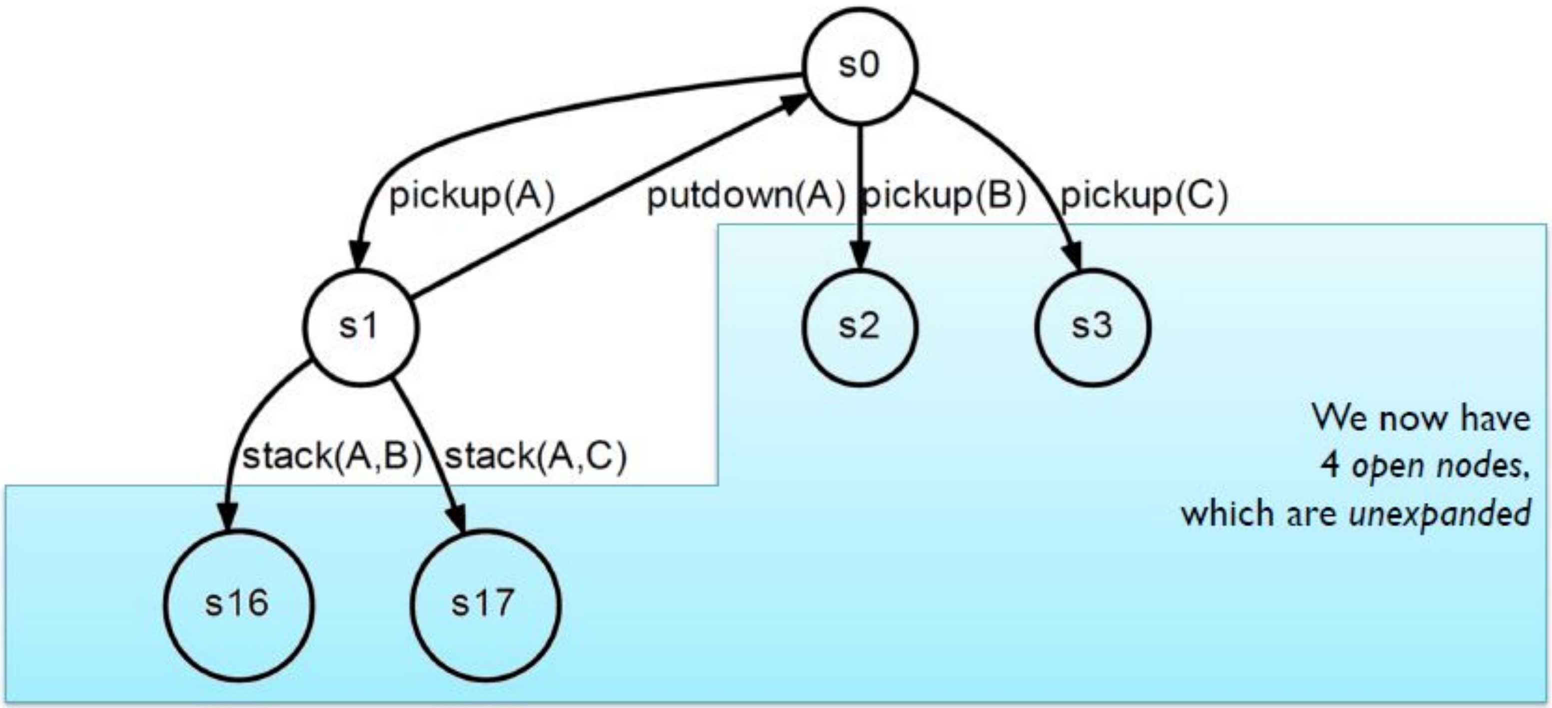


We now have
3 *open nodes*,
which are *unexpanded*

A ***heuristic function*** estimates the distance from each open node to the goal:
We calculate h(s1), h(s2), h(s3)
A ***search strategy*** uses this value (and other info) to *prioritize* between them

Use of Heuristics

If we choose to visit s1:



2 new heuristic values are calculated: h(s16), h(s17)
The ***search strategy*** now has 4 nodes to prioritize

Use of Heuristics

- Two aspects of heuristic guidance
  - Use a search strategy that can make use of heuristics
    - E.g. A*
  - Generating the actual heuristic guidance
    - Should be able to apply on the selected search strategy
- Heuristic function should be efficient
- Two **distinct** objectives for heuristic selection
  - Find a solution quickly
  - Find a good (cheap) solution

Use of Heuristics

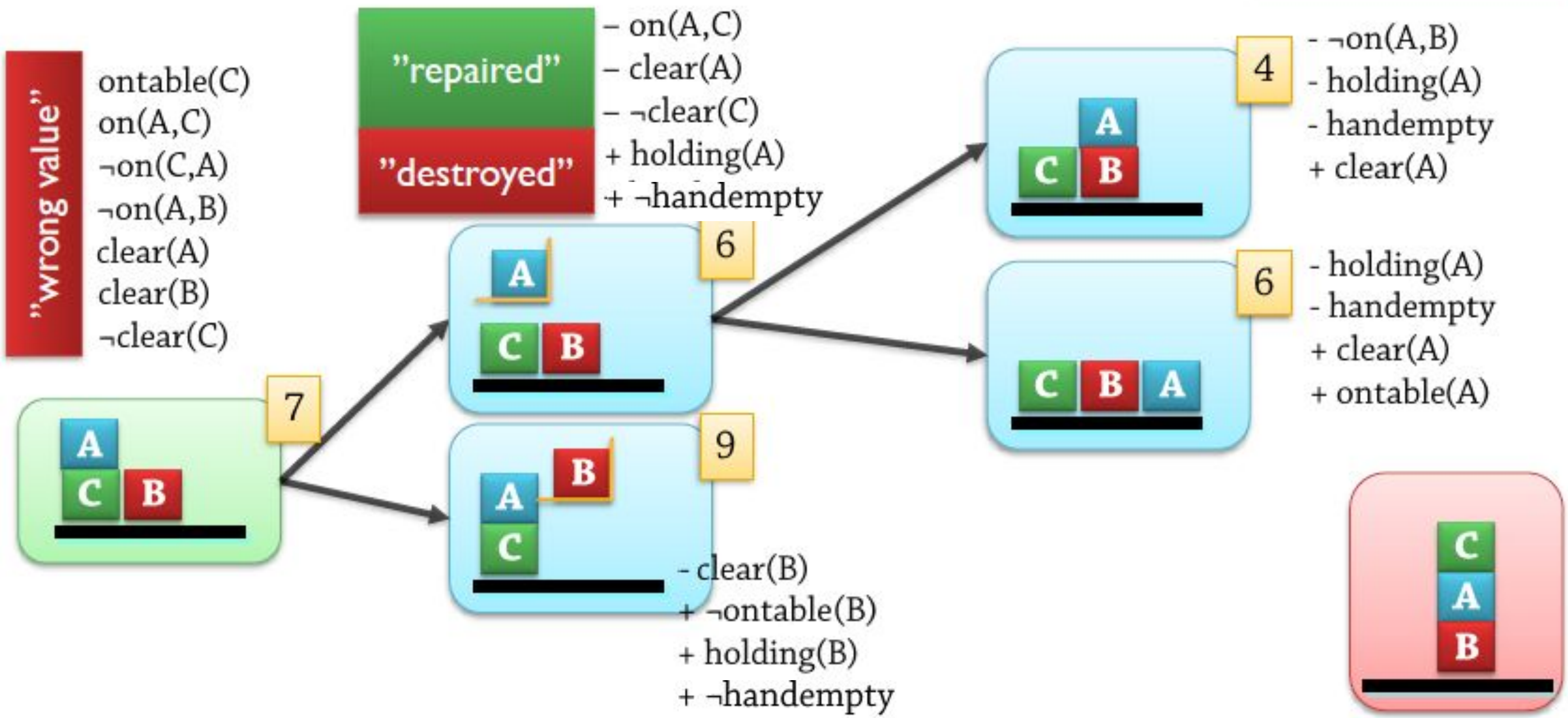
QUICK SOLUTION	GOOD (CHEAP) SOLUTION
Accumulated plan cost = 50	Accumulated plan cost = 5
Estimated distance to goal =10	Estimated distance to goal =30

Domain-Independent Heuristics

- Decide how close a state *s* is to the goal
  - Count how many facts are different
- No action costs, for simplicity
- Search strategy
  - Choose an open node with a minimal number of goal facts to achieve

- **Count** the number of facts that are "wrong"
  - Competely independent of the domain

Optimal:
unstack(A,C)
stack(A,B)
pickup(C)
stack(C,A)



Heuristic Functions

- Which is better?
  - Domain-specific??
  - Domain-independent??

Backward state-space search

- Also called regression planning
- Work backward from the goal(s)
- Generates only relevant actions
  - An action is relevant to a conjunctive goal, if it achieves one of the conjuncts of the goal
  - E.g. if the goal is: $At(C_1, B) \land At(C_2, B) \land At(C_3, B)$, then any action that satisfies $At(C_1, B)$, $At(C_2, B)$ *OR* $At(C_3, B)$ is relevant to the goal.
- Actions **must** be chosen such that they are **consistent**
  - An action must *not* undo any desired literals

Backward State Space Search

- Start at the goal and compute inverse state transitions
  - New set of subgoals = $\gamma^{-1}(g,a)$
- To define $\gamma^{-1}(g,a)$, must first define relevance:
- An action *a* is relevant for a goal *g* if
  - *a* makes at least one of *g's* literals true
    - $g \cap effects(a) \neq \varnothing$
  - *a* does not make any of *g's* literals false
    - $g^{+} \cap effects^{-}(a) = \varnothing$ and $g^{-} \cap effects^{+}(a) = \varnothing$

Backward State Space Search

- If *a* is relevant for *g,* then
  - $\gamma^{-1}(g,a) = (g - effects(a)) \cup precond(a)$
- Otherwise $\gamma^{-1}(g,a)$ is undefined

Backward State Space Search

Backward-search(O, s_0, g)

Efficiency of Backward Searching

- Backward search can also have a very large branching factor
- As before, deterministic implementations can waste lots of time trying all of them
- solution??
  - Lifting
  - STRIPS algorithm