Linköping University

# Automated Planning

## The State Space and Forward-Chaining State Space Search

Jonas Kvarnström

Automated Planning Group

Department of Computer and Information Science

Linköping University

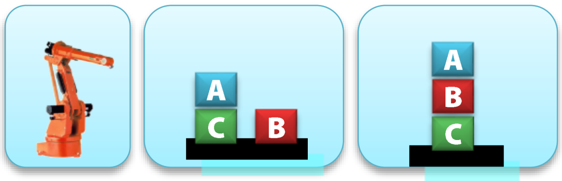
jonas.kvarnstrom@liu.se – 2018

# Exploring the State Space

# About Examples



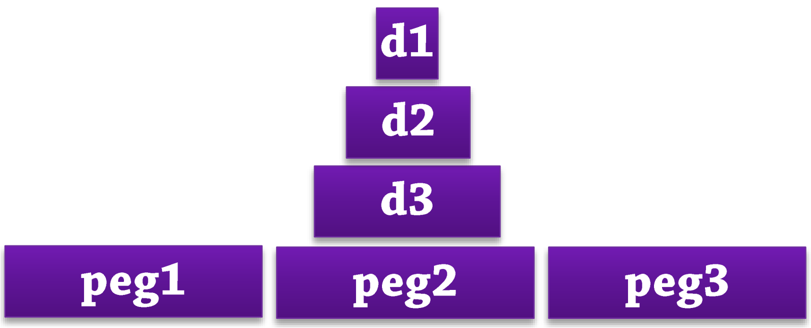
- Exploring the state space… of what?
  - As usual: **toy examples** in **very simple domains**
    - To learn *fundamental principles*
    - To focus on *algorithms and concepts*, not domain details
    - To create *readable, comprehensible examples*
  - Always remember:
    - Real-world problems are larger, more complex



# ToH 0: Towers of Hanoi



- Domain 1: Towers of Hanoi
  - A **modeling trick**:



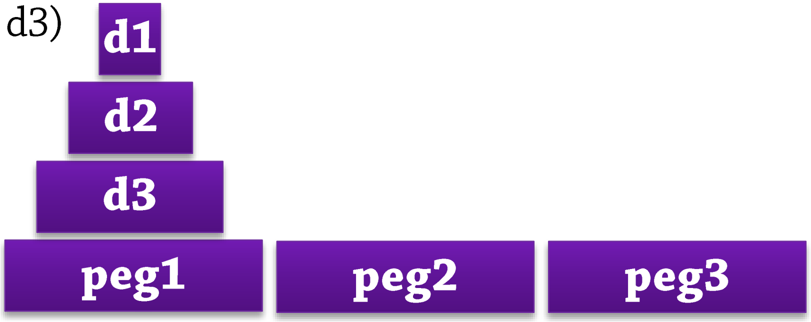

Disks and pegs are "equivalent"
Pegs are the *largest disks*,
so they cannot be moved

# ToH 1: Towers of Hanoi

- Domain 1: Towers of Hanoi
  - ```
    (define (domain hanoi)
      (:requirements :strips)
      (:predicates (clear ?x) (on ?x ?y) (smaller ?x ?y))

      (:action move
        :parameters (?disc ?from ?to)
        :precondition (and (smaller ?to ?disc) (on ?disc ?from) (clear ?disc) (clear ?to))
        :effect  (and (clear ?from) (on ?disc ?to) (not (on ?disc ?from)) (not (clear ?to))))
      )
    ```
  - ```
    (define (problem hanoi3) (:domain hanoi)
      (:objects peg1 peg2 peg3 d1 d2 d3)
      (:init
        (smaller peg1 d1) (smaller peg1 d2) (smaller peg1 d3)
        (smaller peg2 d1) (smaller peg2 d2) (smaller peg2 d3)
        (smaller peg3 d1) (smaller peg3 d2) (smaller peg3 d3)
        (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)
        (clear peg2) (clear peg3) (clear d1)
        (on d3 peg1) (on d2 d3) (on d1 d2))
      (:goal (and (on d3 peg3) (on d2 d3) (on d1 d2)))
      )
    ```

| | |
|---|---|
| clear: | "nothing on top of x" |
| on: | "x on top of y" |
| smaller: | "y is smaller than x" |



# ToH 2: Number of States




- How many **states** exist for this problem?
  - (**define** (**domain** hanoi)
    (:**requirements** :**strips**)
    (:**predicates** (clear ?x) (on ?x ?y) (smaller ?x ?y))

    (:**action** move
    :**parameters** (?disc ?from ?to)
    :**precondition** (**and** (smaller ?to ?disc) (on ?disc
    :**effect** (**and** (clear ?from) (on ?disc ?to) (**not** (or
    )
  - (**define** (**problem** hanoi3) (:**domain** hanoi)
    (:**objects** peg1 peg2 peg3 d1 d2 d3)
    (:**init**
    (smaller peg1 d1) (smaller peg1 d2) (smaller peg1
    (smaller peg2 d1) (smaller peg2 d2) (smaller peg2
    (smaller peg3 d1) (smaller peg3 d2) (smaller peg3
    (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)
    (clear peg2) (clear peg3) (clear d1)
    (on d3 peg1) (on d2 d3) (on d1 d2))
    (:**goal** (**and** (on d3 peg3) (on d2 d3) (on d1 d2)))
    )

**Answer**:

**Every** assignment of values
to the ground atoms
is one state

6 objects
$2^6$ combinations of "clear"
$2^{6*6}$ combinations of "on"
$2^{6*6}$ combinations of "smaller"

**$2^{78}$ combinations in total:**
302231′454903′657293′676544

# ToH 3: Without Rigid Predicates

- Suppose we don't include **fixed** predicates ("smaller") in the state?
  - (**define** (**domain** hanoi)
    (:**requirements** :**strips**)
    (:**predicates** (clear ?x) (on ?x ?y) (smaller ?x ?y))

    (:**action** move
    :**parameters** (?disc ?from ?to)
    :**precondition** (**and** (smaller ?to ?disc) (on ?disc
    :**effect** (**and** (clear ?from) (on ?disc ?to) (**not** (or
    )
  - (**define** (**problem** hanoi3) (:**domain** hanoi)
    (:**objects** peg1 peg2 peg3 d1 d2 d3)
    (:**init**
    (smaller peg1 d1) (smaller peg1 d2) (smaller peg1
    (smaller peg2 d1) (smaller peg2 d2) (smaller peg2
    (smaller peg3 d1) (smaller peg3 d2) (smaller peg3
    (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)
    (clear peg2) (clear peg3) (clear d1)
    (on d3 peg1) (on d2 d3) (on d1 d2))
    (:**goal** (**and** (on d3 peg3) (on d2 d3) (on d1 d2)))
    )

6 objects
$2^6$ combinations of "clear"
$2^{6*6}$ combinations of "on"

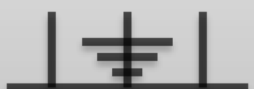
**$2^{42}$ combinations in total:**
4′398046′511104

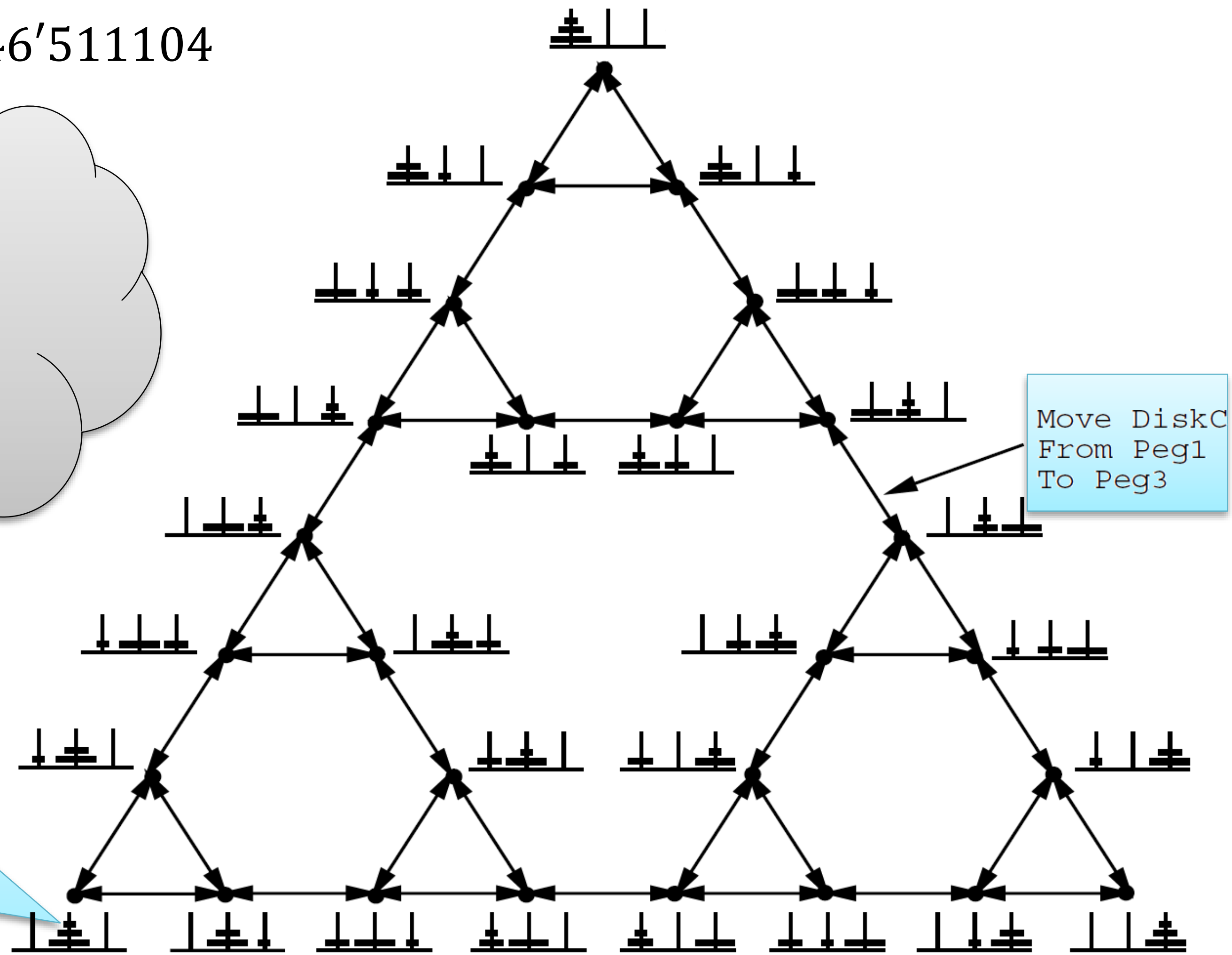
# ToH 4: Reachable From...



- How many **states** are **reachable from** the given **initial state**, using the given actions?
  - 27 out of 4′398046′511104

The other states still **exist** in $S$!






$s_{17}$ clear(peg1) is true
clear(peg2) is false
clear(peg3) is false
on(d1,peg1) is false
on(d3,peg22) is true
…

# ToH 5: Reachable States



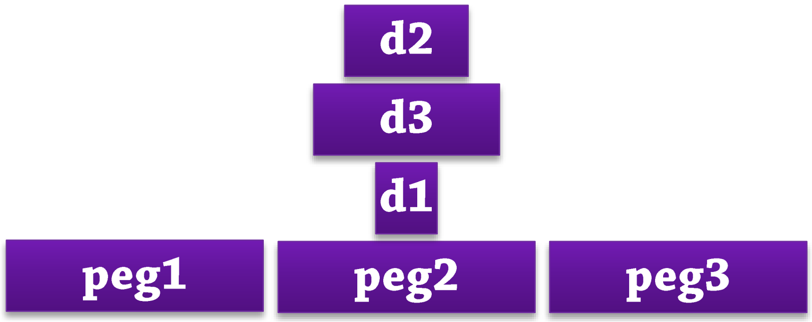
States are not **inherently** "reachable" or "unreachable"

They can be reachable **from a specific starting point**!

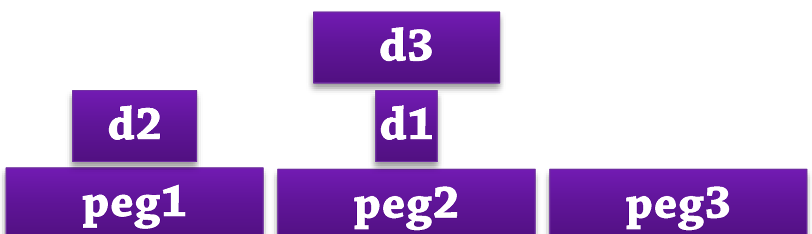
# ToH 6: Reachable from "Forbidden"



- Suppose **this** was your initial state
  - *Unreachable* from "all disks in the right order"!




- Then *other* states would be **reachable from this state**
  - If the preconditions hold, then *move* can be applied

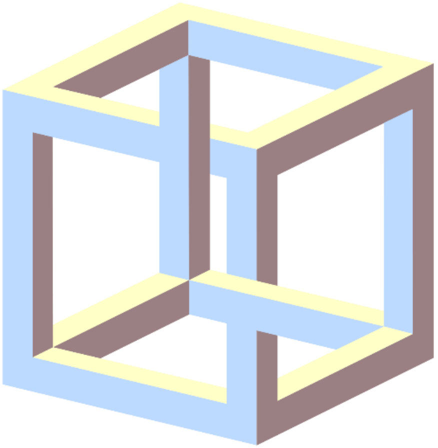



The **states** exist in $S$ – they obey no rules
**permitted transition** according to our operators

# ToH 7: Reachable from "Impossible"



- Suppose **<u>this</u>** was your initial state:
  - (**and**
    (on peg1 peg2)
    (on d1 d2)
    (on d2 d1)
    (on d3 d3)
    …
    )



- Then *other* states would be reachable
  - If the preconditions hold, then *move* can be applied

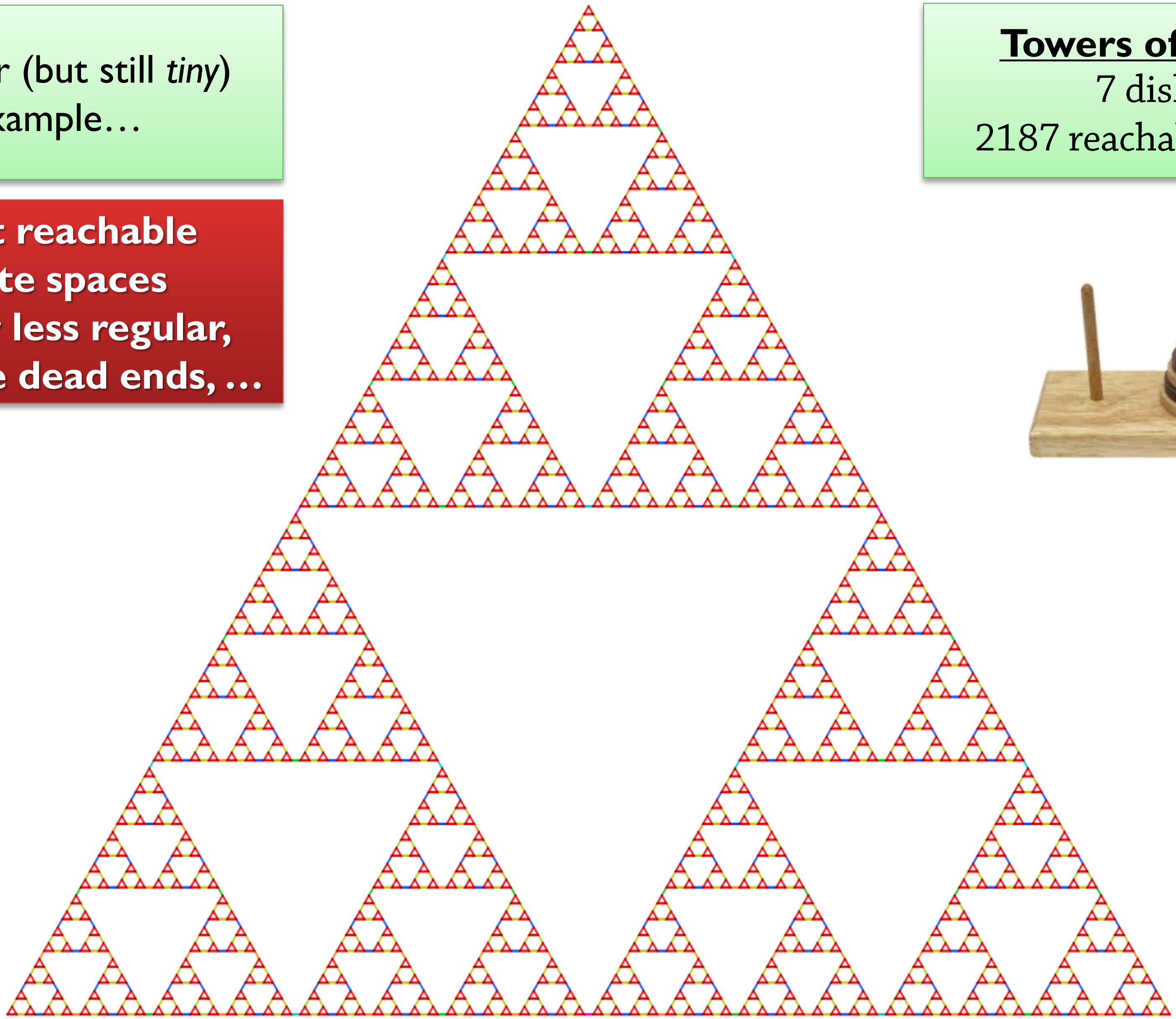
Can't even be visualized – physically impossible
But the **<u>states</u>** exist in $S$ – they are just combinations of true/false values

# ToH 8: Larger Reachable



A larger (but still *tiny*) example…

**Most reachable state spaces are far less regular, can have dead ends, …**



**Towers of Hanoi**
7 disks
2187 reachable states

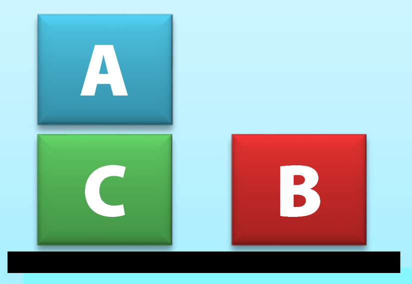
# State Space: Blocks World

# BW 1: Blocks World

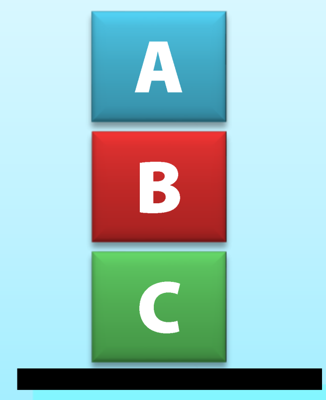
- Domain 2: The **Blocks World**

You

Initial State




Your greatest desire



# BW 2: Model



- We will generate classical **sequential plans**
  - One object type: **Blocks**
  - A common blocks world version, with **4 operators**
    - (**pickup** ?x) – takes ?x from the table
    - (**putdown** ?x) – puts ?x on the table
    - (**unstack** ?x ?y) – takes ?x from on top of ?y
    - (**stack** ?x ?y) – puts ?x on top of ?y
  - Predicates used:
    - (**on** ?x ?y) – block ?x is on block ?y
    - (**ontable** ?x) – ?x is on the table
    - (**clear** ?x) – we can place a block on top of ?x
    - (**holding** ?x) – the robot is holding block ?x
    - (**handempty**) – the robot is not holding any block


With $n$ blocks: $2^{n^2+3n+1}$ states

unstack(A,C) → putdown(A) → pickup(B) → stack(B,C)

# BW 3: Operator Reference



```
(:action pickup
  :parameters (?x)
  :precondition (and (clear ?x) (on-table ?x)
                     (handempty))
  :effect
  (and (not (on-table ?x))
       (not (clear ?x))
       (not (handempty))
       (holding ?x)))
```

```
(:action putdown
  :parameters (?x)
  :precondition (holding ?x)

  :effect
  (and (on-table ?x)
       (clear ?x)
       (handempty)
       (not (holding ?x))))
```

```
(:action unstack
  :parameters (?top ?below)
  :precondition (and (on ?top ?below)
         (clear ?top) (handempty))
  :effect
  (and (holding ?top)
       (clear ?below)
       (not (clear ?top))
       (not (handempty))
       (not (on ?top ?below))))
```

```
(:action stack
  :parameters (?top ?below)
  :precondition (and (holding ?top)
                     (clear ?below))
  :effect
  (and (not (holding ?top))
       (not (clear ?below))
       (clear ?top)
       (handempty)
       (on ?top ?below)))
```

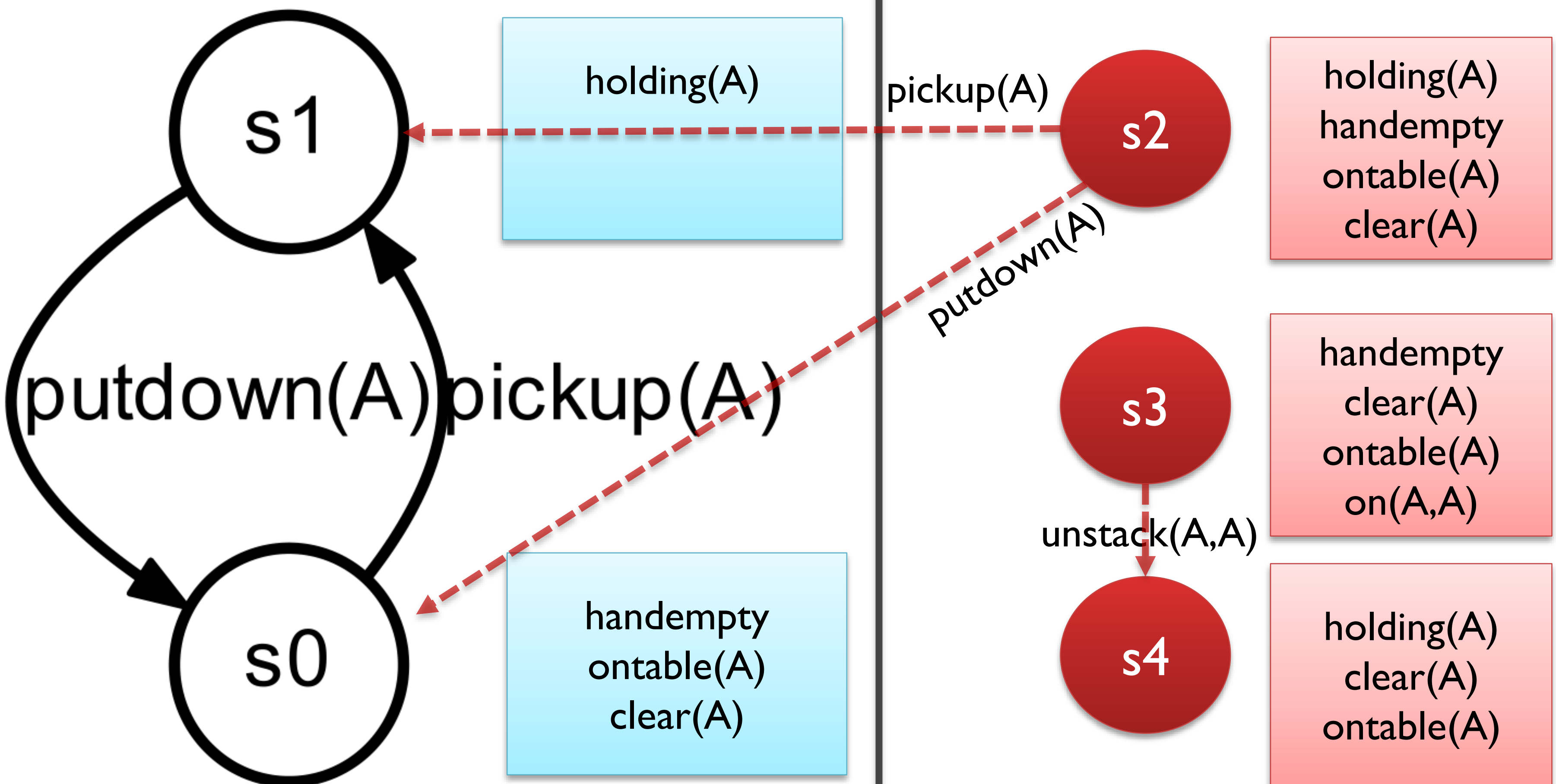
# BW 4: Reachable State Space, 1 block



We assume we know the initial state
Let’s see which states are *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

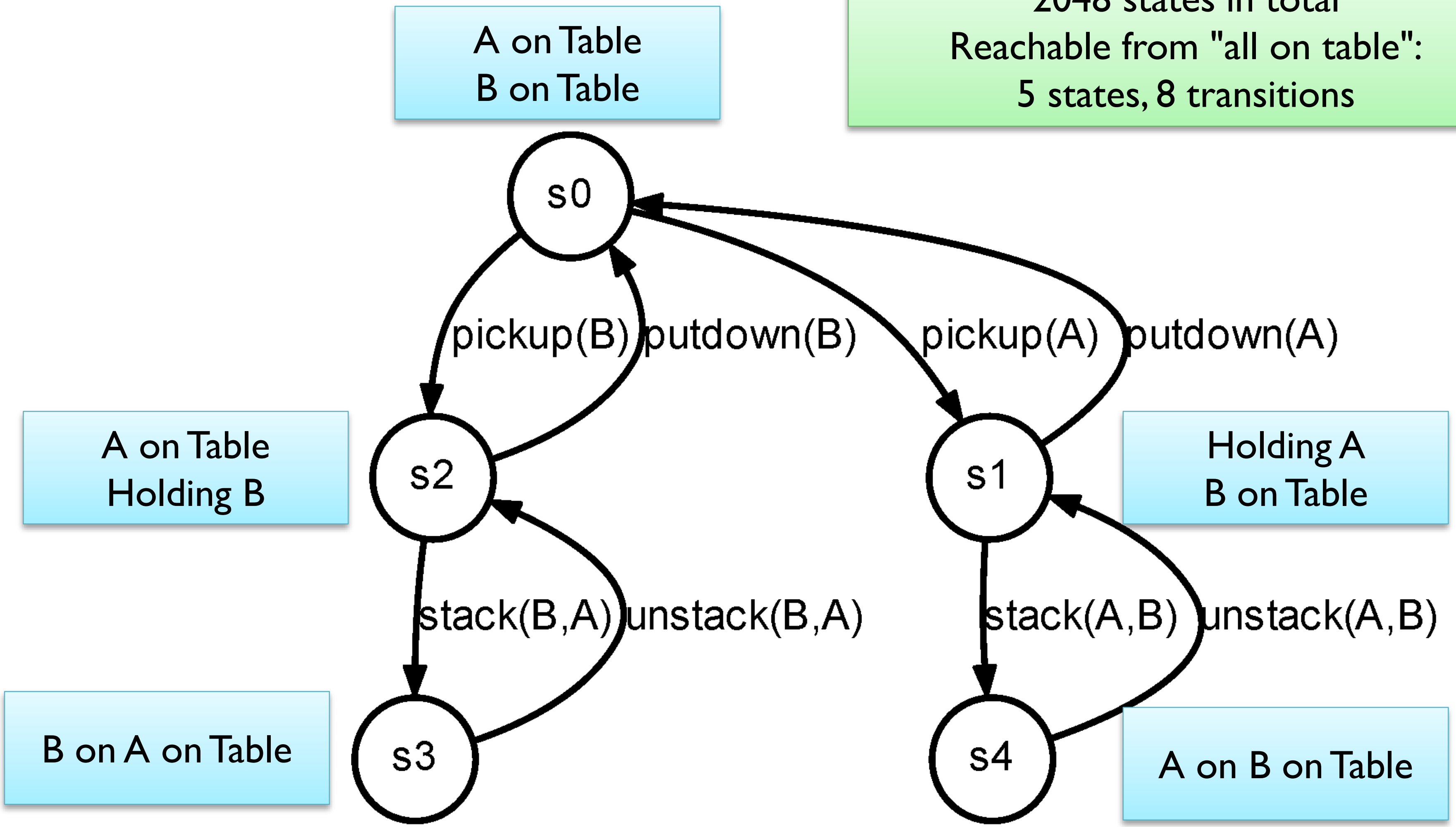


# BW 5: Reachable State Space, 2 blocks





2048 states in total
Reachable from "all on table":
5 states, 8 transitions

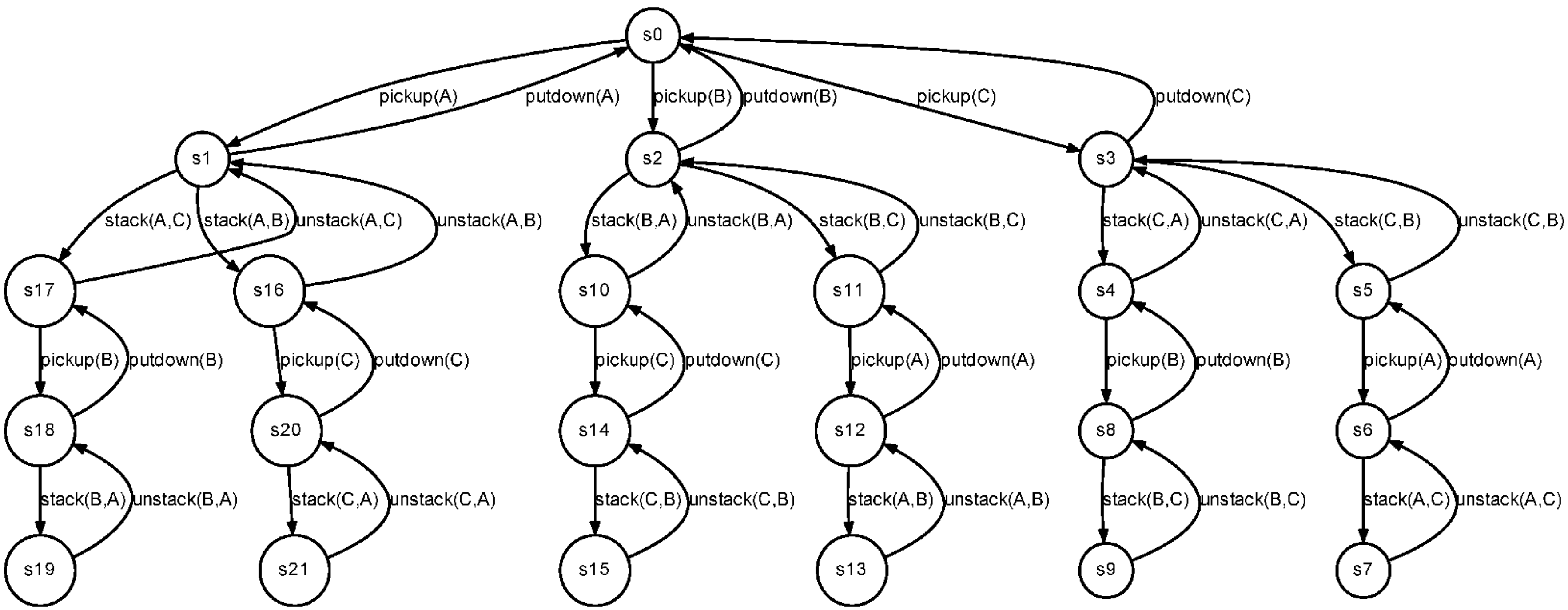


# BW 6: Reachable State Space, 3 blocks



524'288 states in total
Reachable from "all on table":
22 states, 42 transitions

A on Table
B on Table
C on table

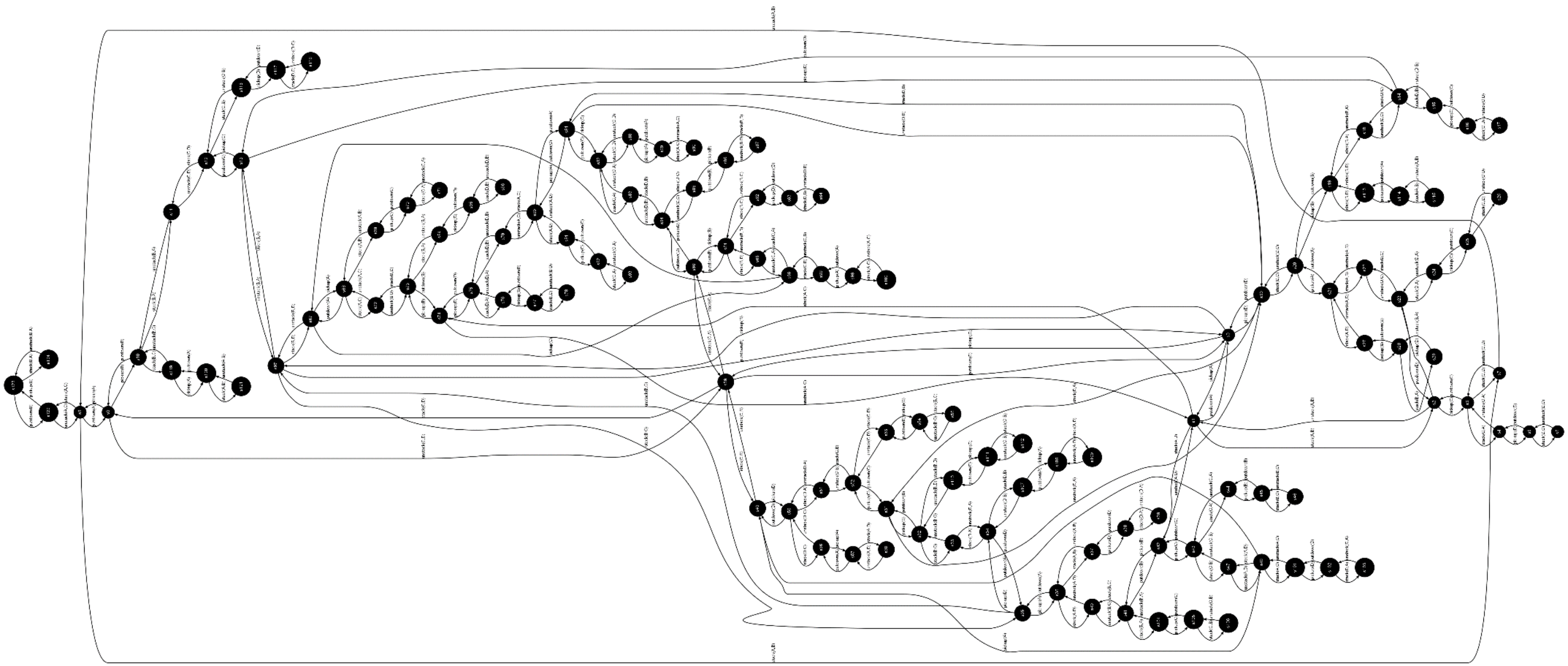



**Looking nice and symmetric…**

# BW 7: Reachable State Space, 4 blocks



536'870'912 states in total
Reachable from "all on table":
125 states, 272 transitions

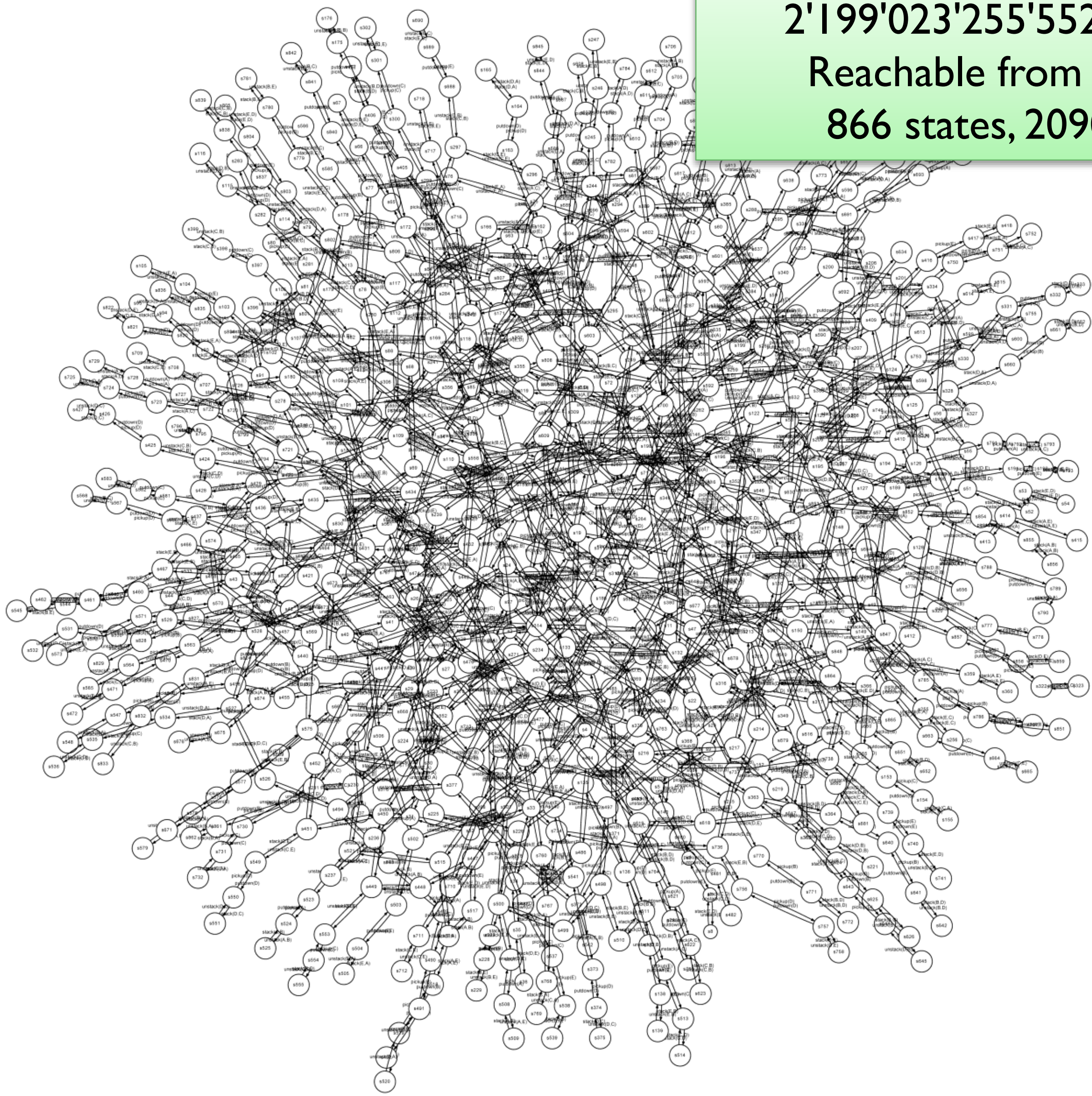


# BW 8: Reachable State Space, 5 blocks





2'199'023'255'552 states in total
Reachable from "all on table":
866 states, 2090 transitions



# BW 9: State Space Size





- Standard PDDL predicates:
  - (**on** ?x ?y)
  - (**ontable** ?x)
  - (**clear** ?x)
  - (**holding** ?x)
  - (**handempty**)
- Number of ground atoms, for $n$ blocks:
  - $n^2 + 3n + 1$
- Number of states:
  - $2^{n^2+3n+1}$

# BW 10: Reachable State Space, sizes 0–10



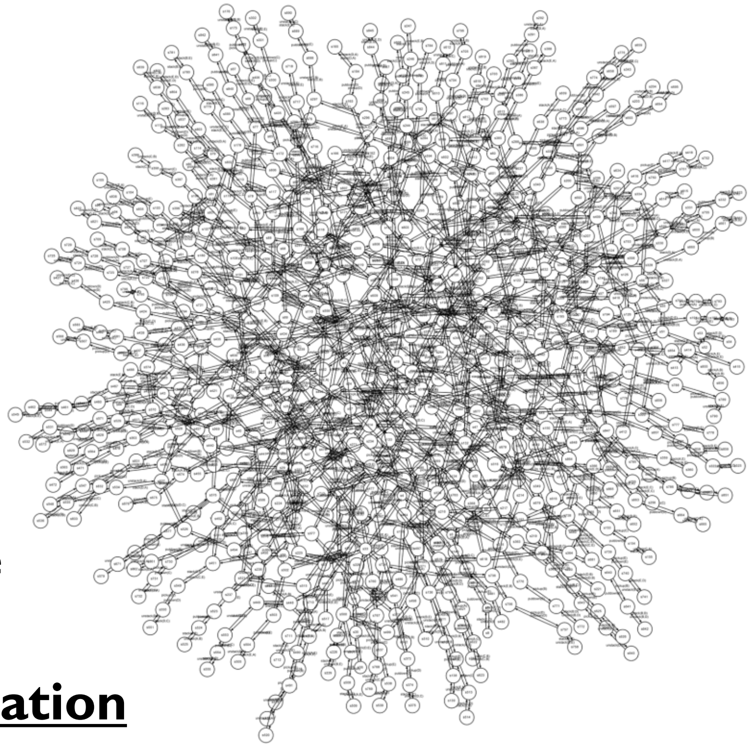
jonkv@ida

| Blocks | Ground atoms | States | States reachable from "all on table" | Transitions (edges) in reachable part |
|---|---|---|---|---|
| 0 | 1 | 2 | 1 | 0 |
| 1 | 5 | 32 | 2 | 2 |
| 2 | 11 | 2048 | 5 | 8 |
| 3 | 19 | 524288 | 22 | 42 |
| 4 | 29 | 536870912 | 125 | 272 |
| 5 | 41 | 2199023255552 | 866 | 2090 |
| 6 | 55 | 36028797018963968 | 7057 | 18552 |
| 7 | 71 | 2361183241434822606848 | 65990 | 186578 |
| 8 | 89 | 618970019642690137449562112 | 695417 | 2094752 |
| 9 | 109 | 649037107316853453566312041152512 | 8145730 | 25951122 |
| 10 | 131 | 2722258935367507707706996859454145691648 | … | … |

# BW 11: Reducing State Space Size, 5 blocks



jonkv@ida



- Reducing the State Space Size:
  - Standard PDDL model:
    - $2^{n^2+3n+1}$ = 2'199'023'255'552 states, 866 reachable
  - **Omit** (ontable ?x), (clear ?x)
    - In *physically achievable* states, these can be deduced from (on ?x ?y), (holding ?x)
    - $2^{n^2+n+1}$ = 2'147'483'648 states, 866 reachable
  - Also switch to a **state variable representation**
    - Add type *block-or-nothing*, size 6 (values A, B, C, D, E, nothing)
    - Use (= (block-below ?x) ?y), where ?y can be "nothing"
    - $(n+1)^n \cdot 2^{n+1}$ = 497'664 states, 866 reachable

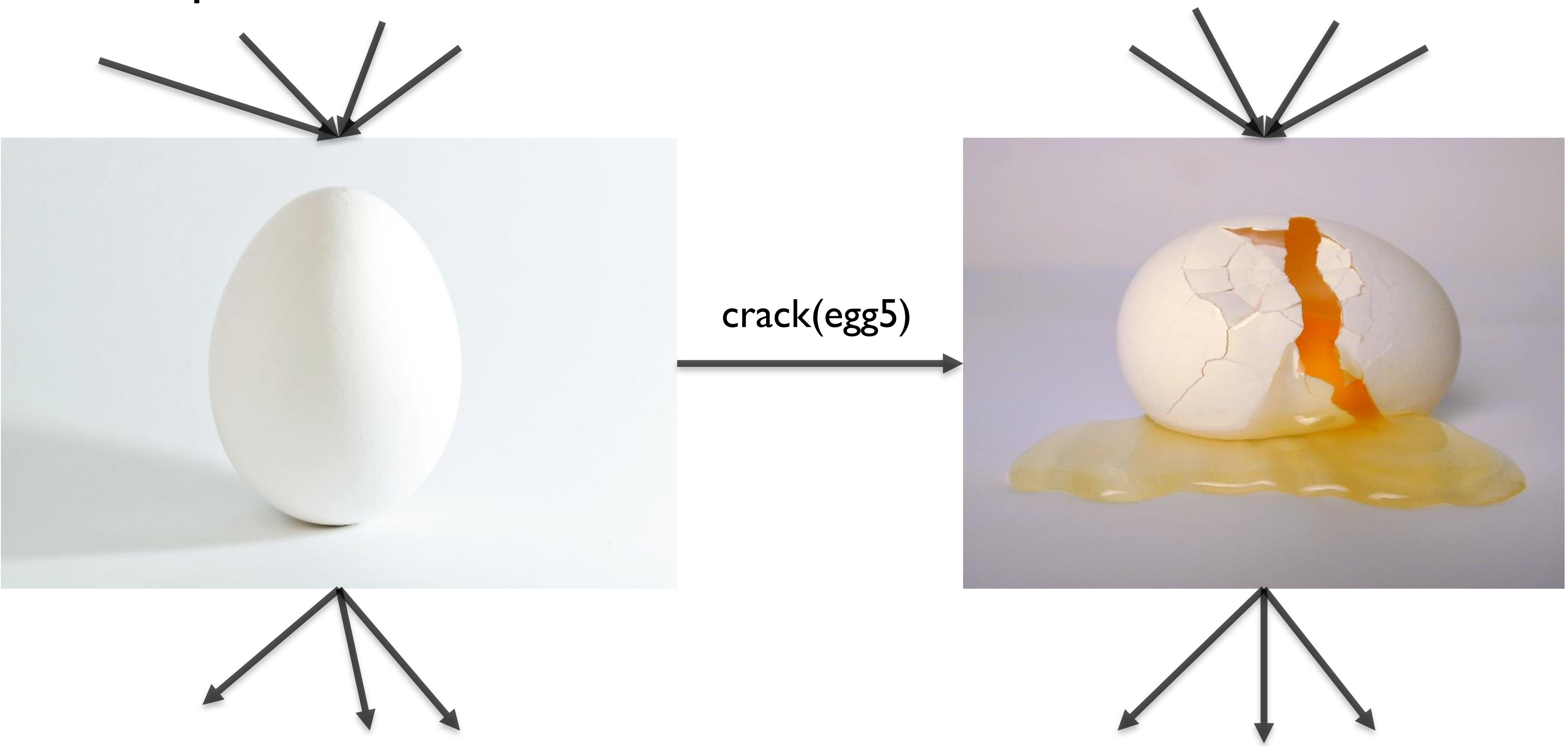
**Is planning time reduced with fewer unreachable states?**

Depends on the planning algorithm!

# State Space: Not Symmetric



- Example: Unable to return

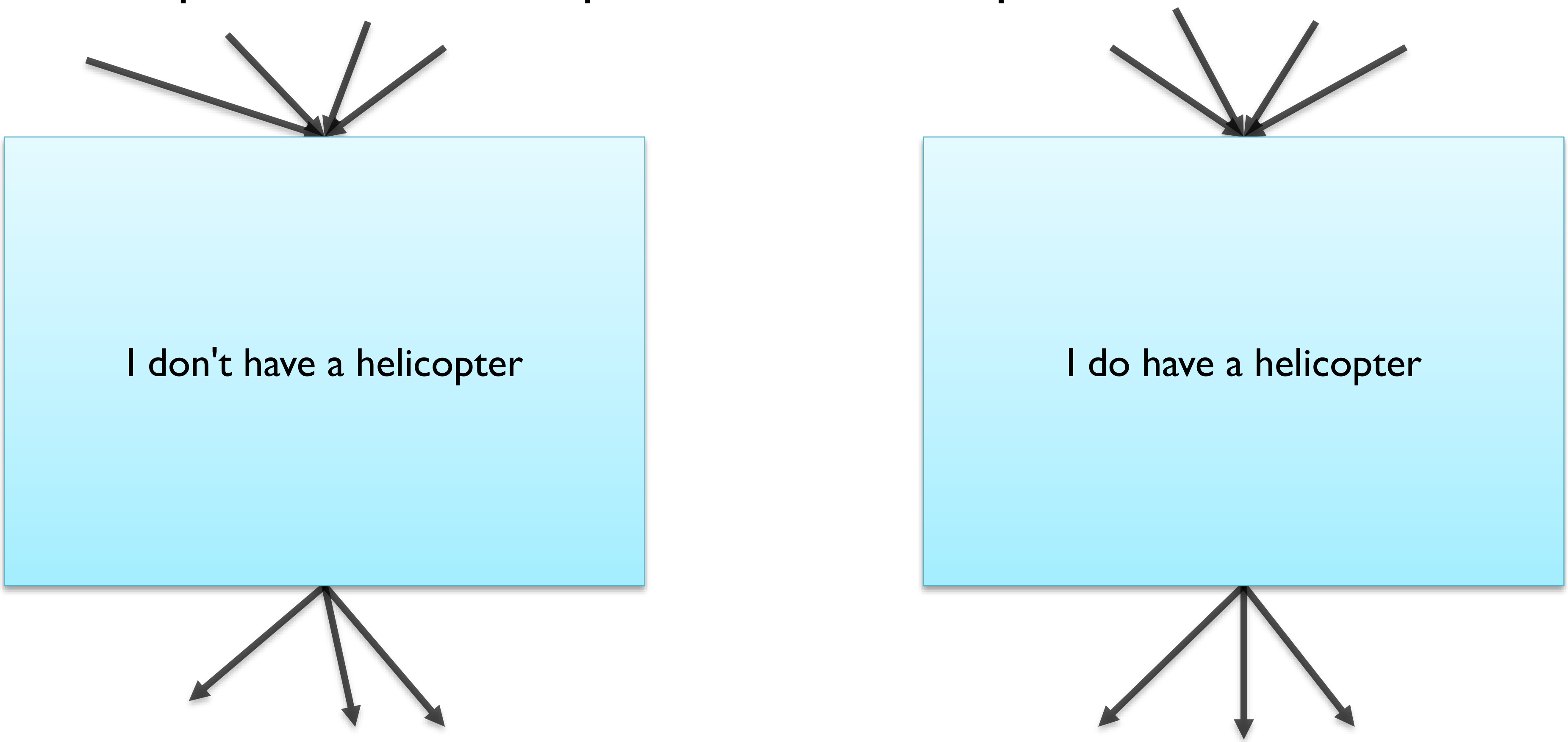



**Can never return to the leftmost part of the state space**

# State Space: Disconnected



- Example: Disconnected parts of the state space




**No action for buying a helicopter, no action for losing it**
**➔ Will stay in the partition where you started!**

# Forward State Space Search

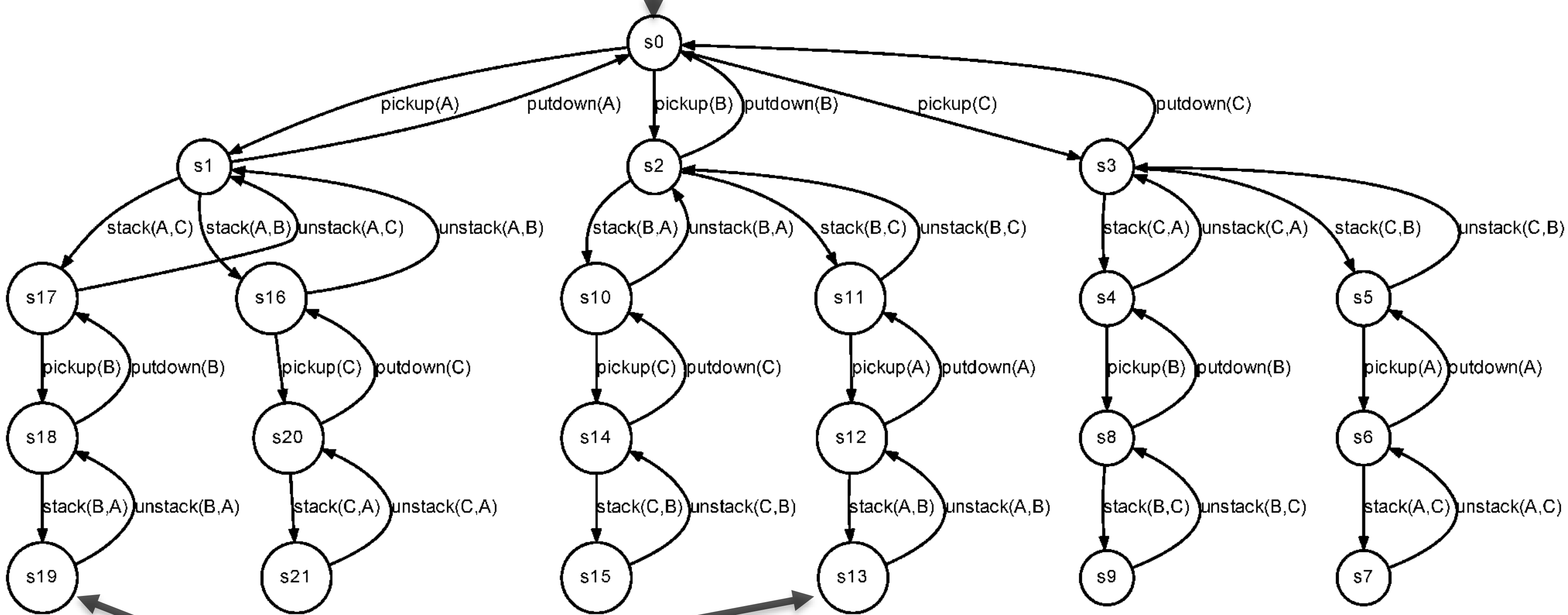
# The Planning Problem





**Find a path <u>in</u> the STS <u>from</u> the initial state <u>to</u> any goal state**

Initial (current) state



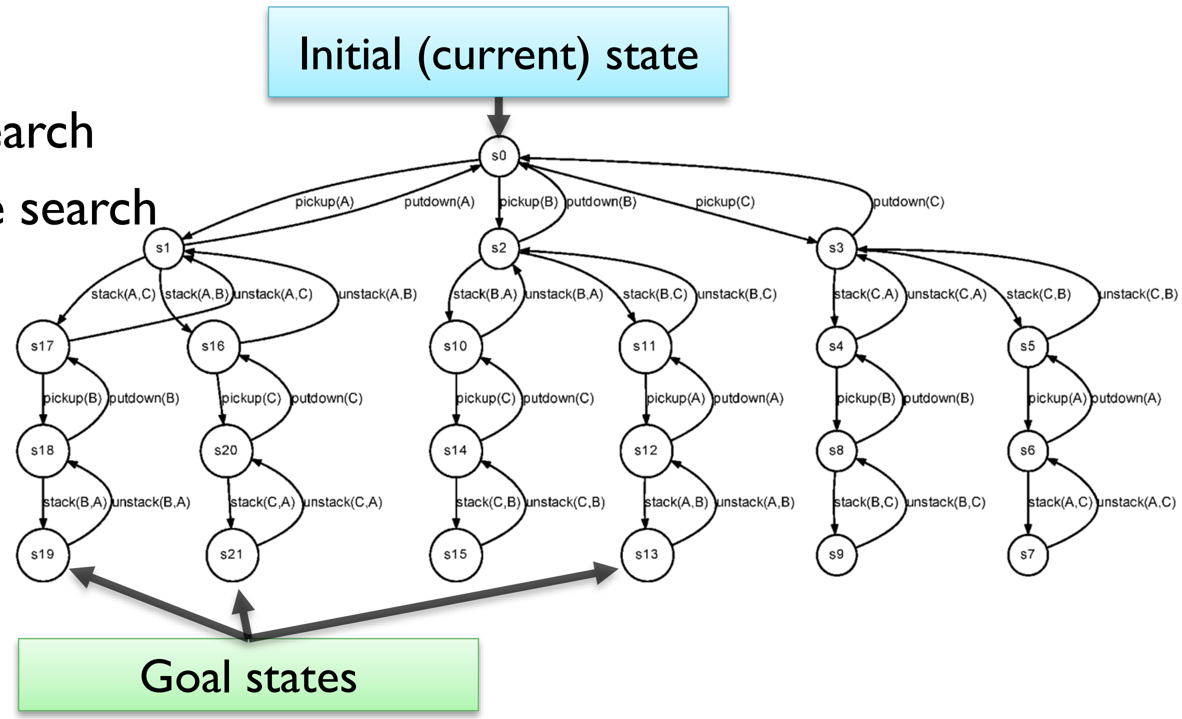

Goal states

*Many graph search methods already exist!*

*How do we apply them to the state space?*

- Can search in **<u>either direction</u>**
  - Most straight-forward: Initial ➔ goal
  - Later: Goal ➔ initial

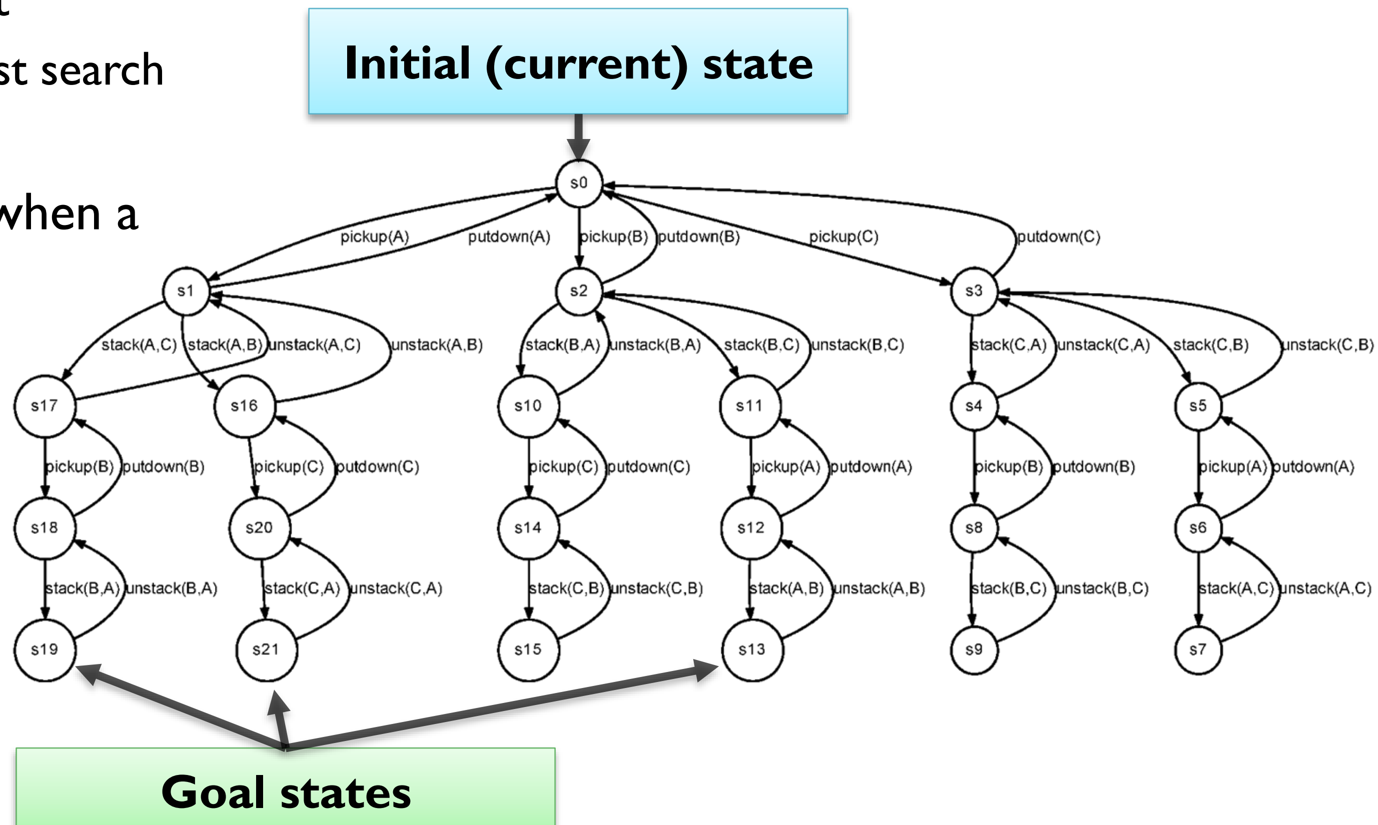
- **<u>Many names:</u>**
  - Forward search
  - Forward-chaining search
  - Forward state space search
  - Progression
  - …



# Forward State Space Search 1



- **Forward search in the state space**
  - **Start** in the initial state
  - Apply a **search** algorithm
    - Depth first
    - Breadth first
    - Uniform-cost search
    - …
  - **Terminate** when a goal state is found

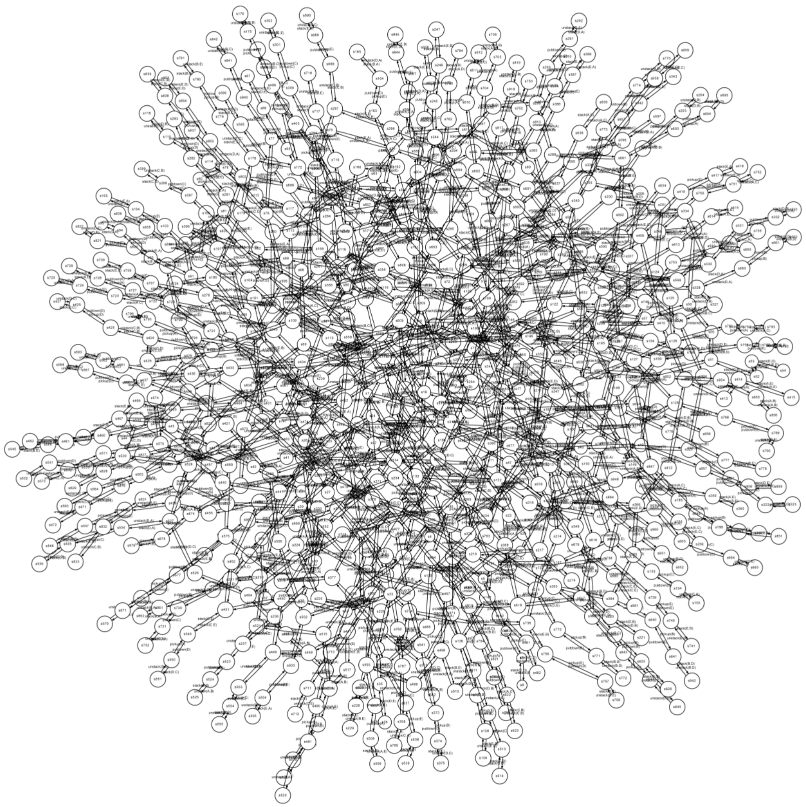


# FSSS 2: Don't Precompute





- The planner is **<u>not</u>** given a complete precomputed search graph!



**Usually too large!**
**➔ Generate as we go,**
**hope we don't actually need the *entire* graph**

# FSSS 3: Initial state




- The **user** (robot?) **observes** the current state of the world
  - The *initial* state


- Must **describe** this using the specified **formal state syntax**…
  - $s_0$ = { **clear**(A), **on**(A,C), **ontable**(C), **clear**(B), **ontable**(B), **clear**(D), **ontable**(D), **handempty** }

- …and give it to the **planner**, which creates **one** search node

**{ clear(A), on(A,C), ontable(C),**
**clear(B), ontable(B), clear(D), ontable(D), handempty }**

# FSSS 4: Successors




- Given **<u>any search node</u>**…

  **{ clear(A), on(A,C), ontable(C), clear(B), ontable(B), clear(D), ontable(D), handempty }**

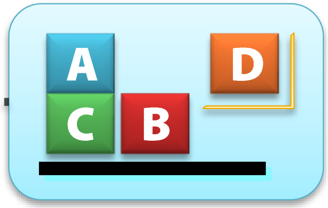
- …we can find **<u>successors</u>** – by appling **<u>actions</u>**!
  - **action** pickup(D)
    - Precondition: **ontable**(D) ∧ **clear**(D) ∧ **handempty**
      Effects: ¬**ontable**(D) ∧ ¬**clear**(D) ∧ ¬**handempty** ∧ **holding**(D)

- This generates **<u>new reachable states</u>**…

**…which can also be illustrated**

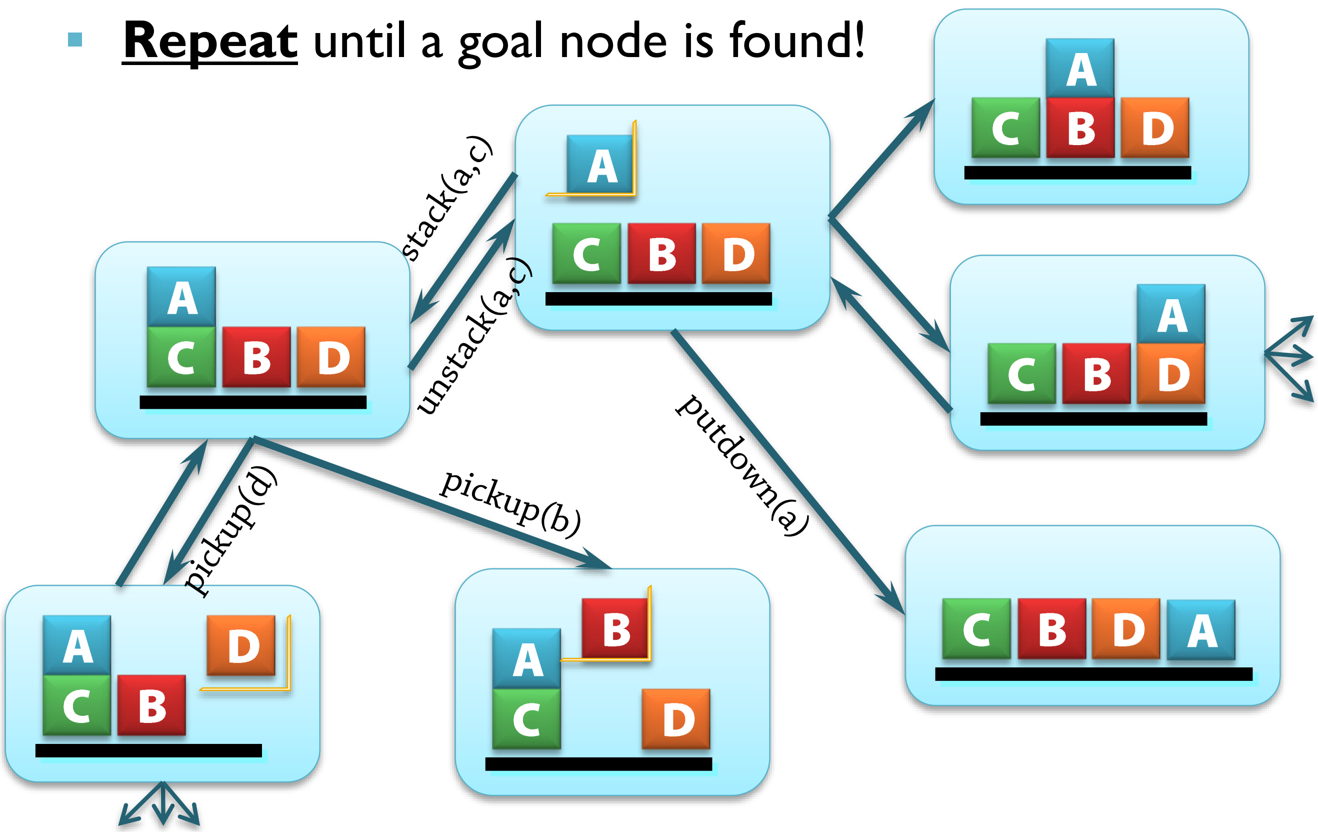
**{ clear(A), on(A,C), ontable(C), clear(B), ontable(B), clear(D), ontable(D), handempty }**


**{ clear(A), on(A,C), ontable(C), clear(B), ontable(B), holding(D) }**



# FSSS 5: Step by step

- A **search strategy** (depth first, A*, hill climbing, …) will:
  - **Choose** a node
  - **Expand** all possible successors
    - *"What actions are applicable in the current state, and where will they take me?"*
    - Generates new states by applying effects
  - **Repeat** until a goal node is found!




This is *illustrated* – the planner works with *sets of facts*

The blocks world is *symmetric*: Can always "return the same way"

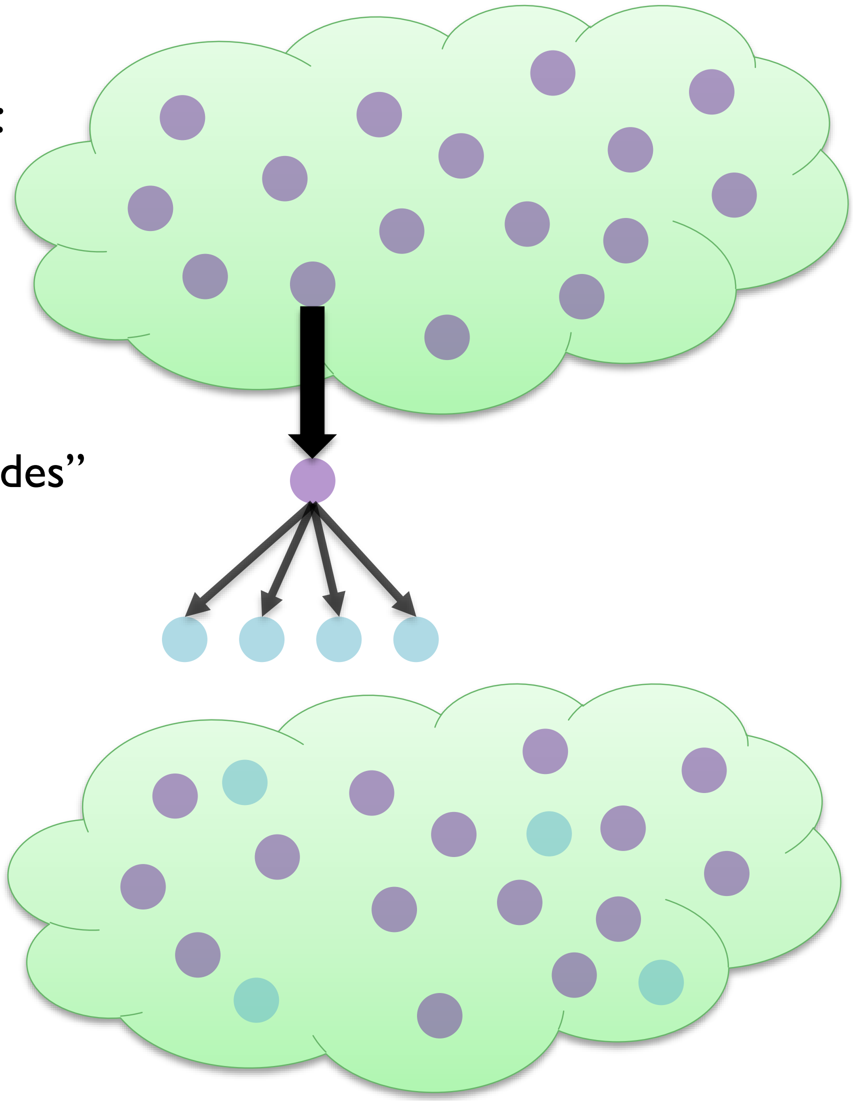
Not true for all domains!

# General Search

- General way of **formalizing** search algorithms:
  - There are some "open" nodes, that we:
    - Know how to reach
    - Haven't explored yet

At first: The **initial state**!

  - Pick / remove one of them
    - Using some *strategy* for picking "good nodes"
  - Find nodes that can be reached in a single step (applying one action)
  - Put **those** back in the set of nodes
    - New options!
  - Repeat until a goal node is found



# Forward State Space Search (4)

- General **Forward State Space Search Algorithm**
  - **forward-search**($A$, $s_0$, g) {

```
    open ← { <s0, ε> }
    while (open ≠ ∅) {
        use a strategy to select and remove one n=<s,path> from open
        if goal g satisfied in state s then return path

        foreach a ∈ A such that γ(s, a) ≠ ∅ {
            {s'} ← γ(s, a)
            path' ← append(path, a)
            add n'=<s', path'> to open
        }
    }
    return failure;
}
```

**Forward** search:
Reach in one *step* =
reach by one *action application*

To simplify extracting a plan,
a state space search node above
includes the plan to reach that state!

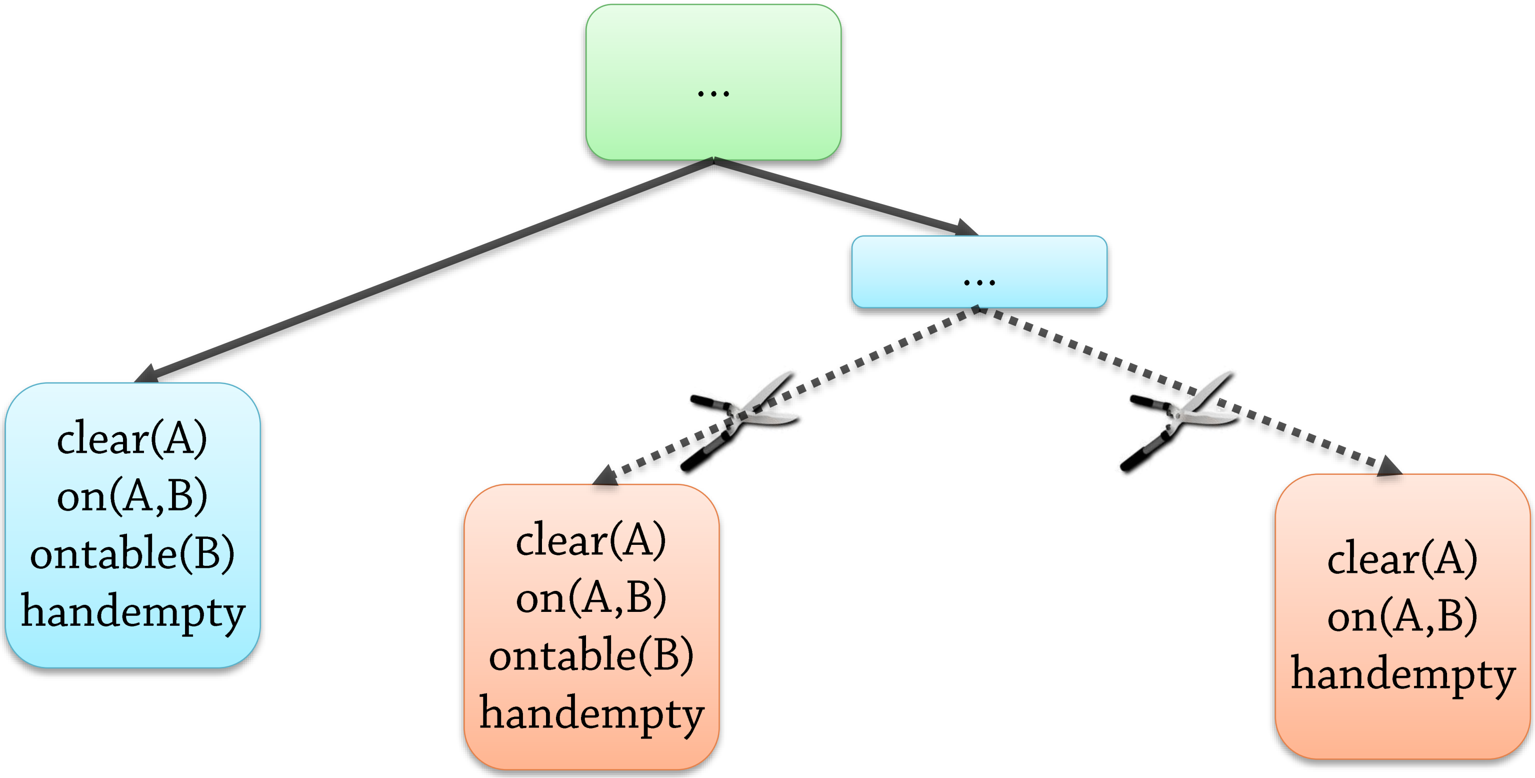
Technically, we search the space of
<state,path> pairs

Still generally called **state space** search…

Is always **sound**
**Completeness** depends on the strategy

# Forward State Space Search (5): Pruning






Reach a more expensive node
with the *same* state
➔ can prune
(discard the node, backtrack)

If preconditions and goals are **positive**:

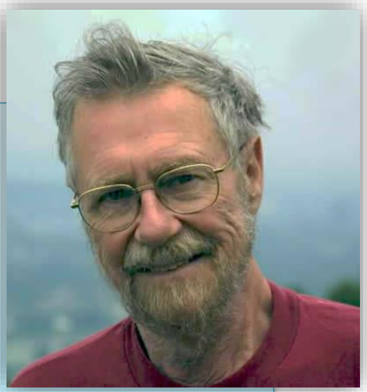
Reach a node with a *subset* of the facts
➔ can prune

# Forward State Space Search: Search Strategies and the Difficulty of Planning

# Forward State Space Search: Dijkstra





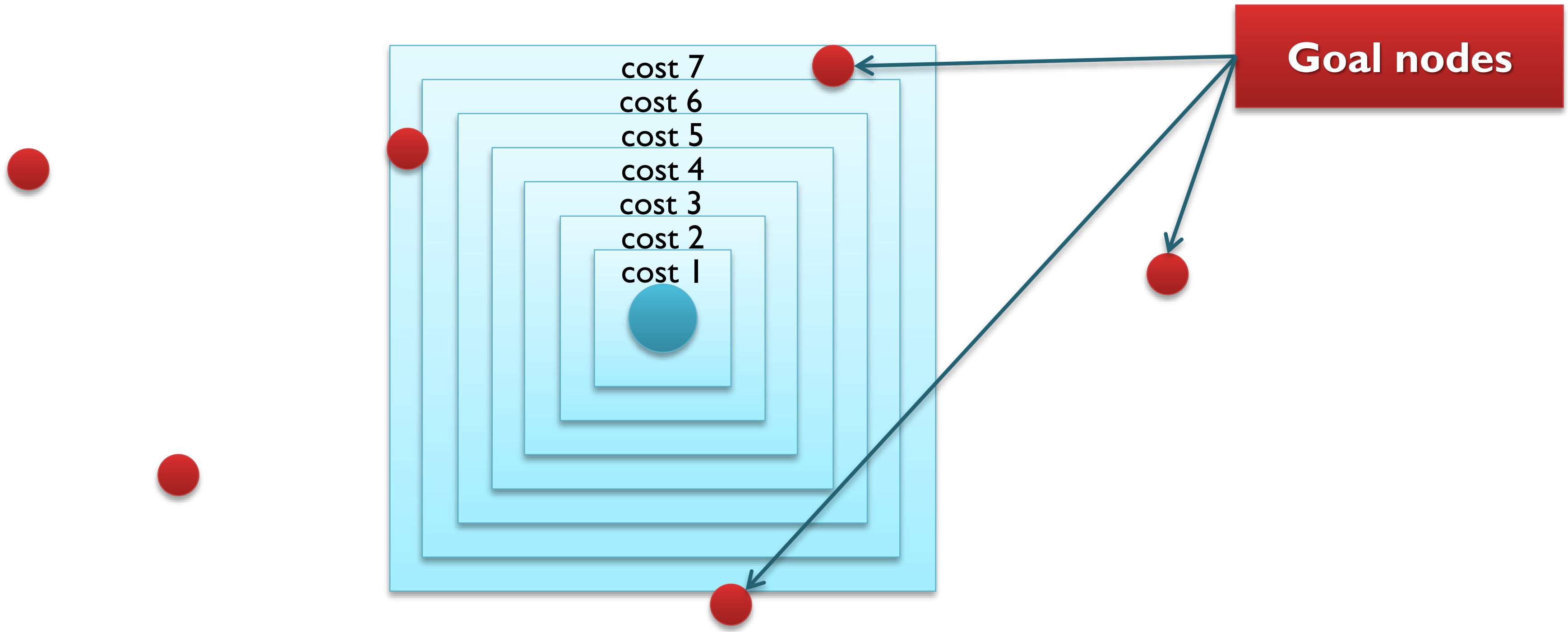


- First search strategy: **Dijkstra's algorithm**
  - **Matches** the given forward search "template"
    - **use a strategy to select** and remove <s,path> from open
    - Selects from *open* a node *n* with minimal g(*n*):
      **Cost** of reaching *n* from the starting point
  - **Efficient** graph search algorithm: O(|E| + |V| log |V|)
    - |E| = the number of edges (transitions), |V| = the number of nodes (states)
  - **Optimal**: Returns minimum-cost plans

# Dijkstra's Algorithm



- Explores states in order of cost
  - Below, we assume $\forall a \in A: c(a) = 1$

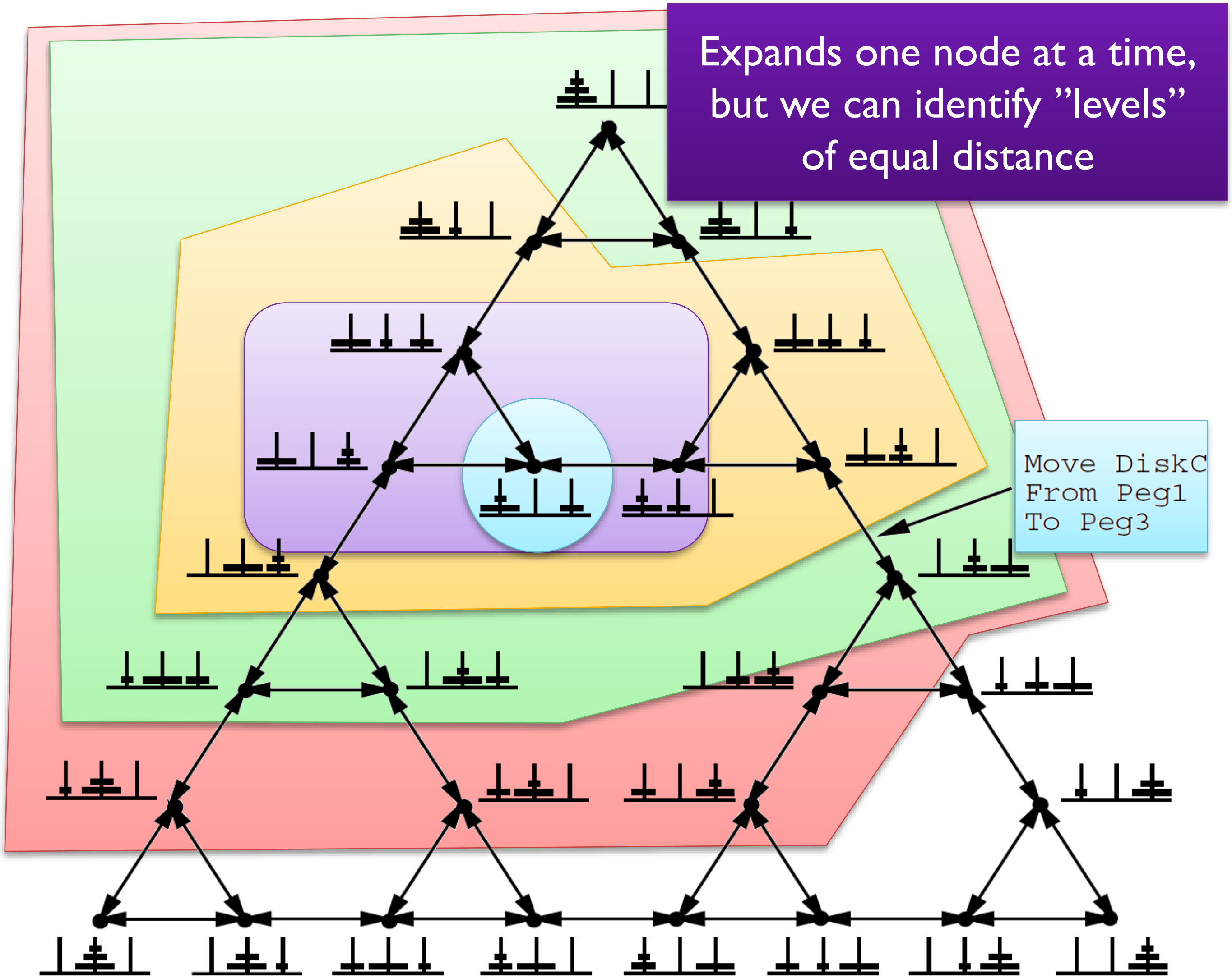


# Dijkstra: ToH





- Running Dijkstra, assuming all actions are equally expensive:

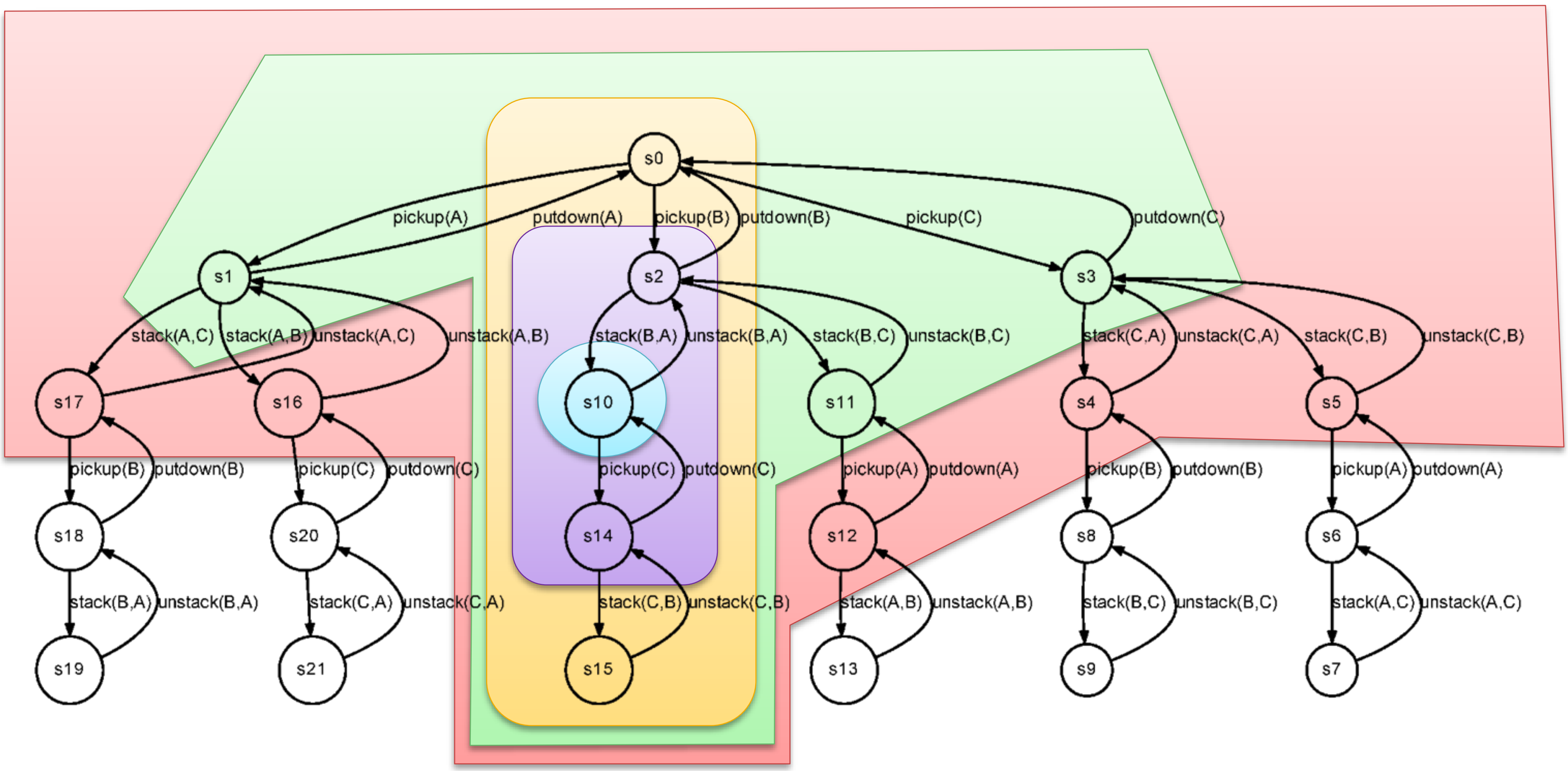


# Dijkstra: Blocks World





- Running Dijkstra, assuming all actions are equally expensive:



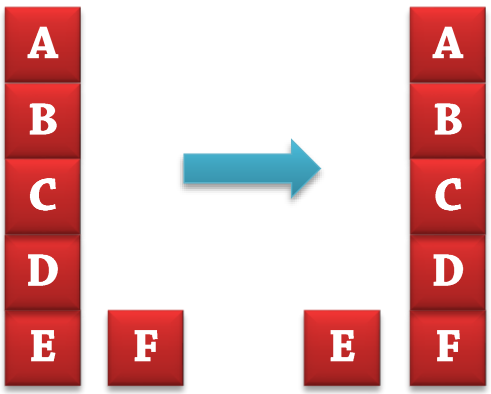
# No problems?

# Dijkstra's Algorithm: Example





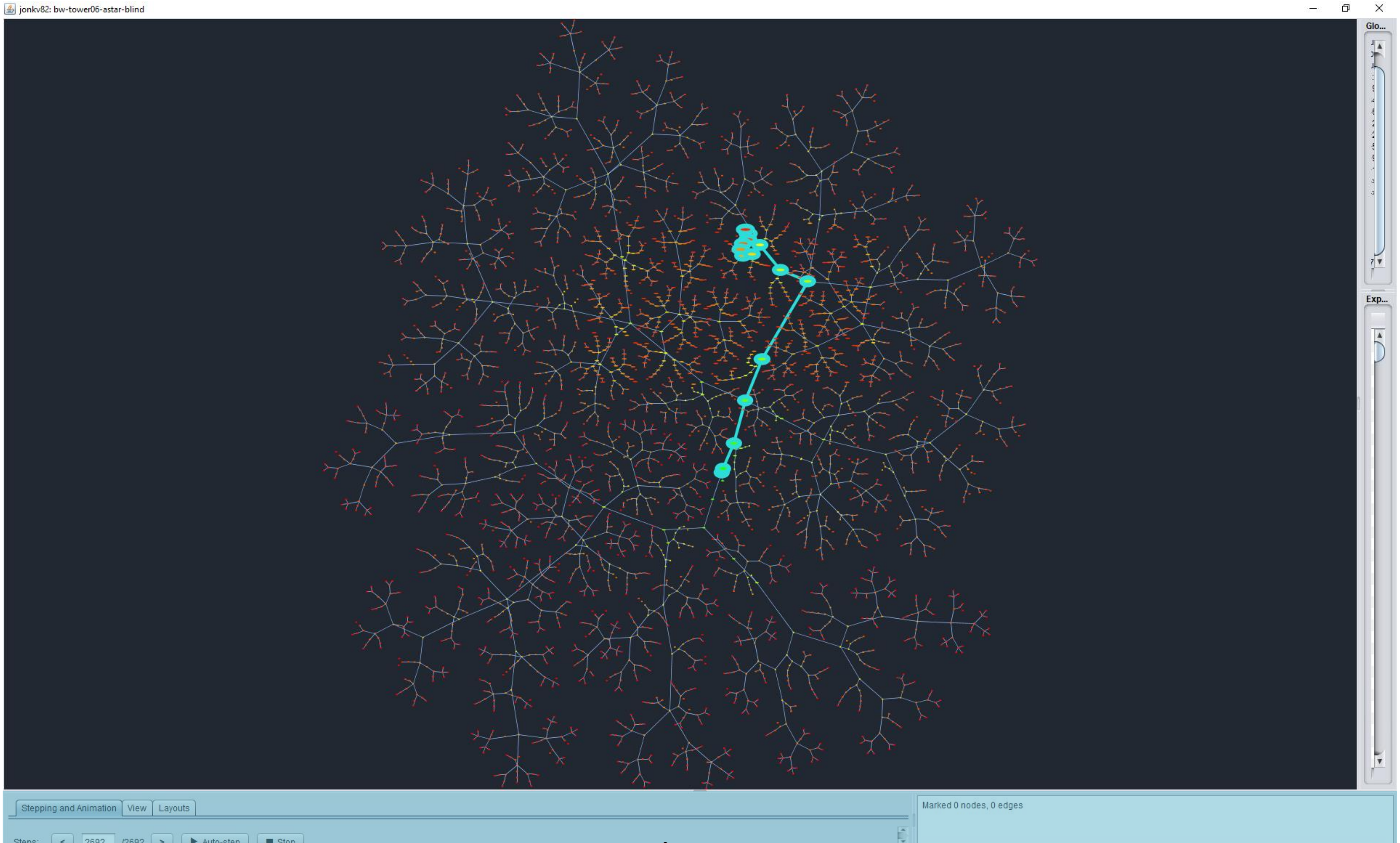
- A simple problem:




| Goal |
|---|
| on(A,B) |
| on(B,C) |
| on(C,D) |
| on(D,F) |
| ontable(E) |
| ontable(F) |

| Optimal solution | |
|---|---|
| unstack(A,B) | pickup(D) |
| **putdown(A)** | **stack(D,F)** |
| unstack(B,C) | pickup(C) |
| **putdown(B)** | **stack(C,D)** |
| unstack(C,D) | pickup(B) |
| **putdown(C)** | **stack(B,C)** |
| unstack(D,E) | pickup(A) |
| **stack(D,F)** | **stack(A,B)** |

# bw-tower06-dijkstra: Only 6 blocks, Dijkstra search, no heuristic




**Actions: 14**

**States: 8706 calculated, 2692 visited**

# 400 blocks



- Blocks world, 400 blocks initially on the table, goal is a 400-block tower
  - Given uniform action costs (same cost for all actions),
    Dijkstra will **always** consider **all** plans that stack **less than 400 blocks**!
    - Stacking 1 block: = 400*399 plans, …
    - Stacking 2 blocks: > 400*399 * 399*398 plans, …
  - More than
    1630569839078931058645796793733472877564594841634782672258624197623042639942079976642582139557665811636541371181631192204882263831691616483204594902834106357987452326989711329392844798003040966743549740387225888734809637192406427243636291547266329397641772360103156941486368193342172528364140014872776180029666087610370180877694906148478874187444026062261348039369352335684180559503711853518371405485159494313093138752108278889433371136136609283180862996179538929537220067341589332765764704756406073917010260309590403035481742212740523295796377736587224525497384594044525865036931693[illegible]0912754853265795909113444084441755664211796274320256992992317773749830373[illegible]4882657444844563187930907779661572990289194810585217819146476629300233604[illegible]1372350568748665249021991849760646988031691394386551194171193333144031542[illegible]1302649432305620215568850657684229678385177725358933986112127352452988033[illegible]308720174243236072916252738750807322557863077768590163743554145844083387870934417498397743743032755753441762912244883519172107733387523069568148099086710905133210482041360782220646563527271107390661180037619441042890007101369543835909464168225385639474333567854582432093210697331749851571100671998530498260475511016725485476618861912891705393354709843502065977868949960690415707700579763228766976414509558156505658981172152043461277059495061370173087930772714109352653432867136000209692448349430242464906145172664594758586010497684553450747960540890382832020613107221778215643420457243461604240437521105232403822580540571315732915984635193126556273109603937188229504400

**1.63 * $10^{1735}$**

Dijkstra is efficient in terms of the **search space size**: O(|E| + |V| log |V|)

The search space is **exponential** in the size of the input description…

# Fast Computers, Many Cores



- But computers are getting **<u>very fast</u>**!
  - Suppose we can check $10^{20}$ states per second
    - >10 billion states *per clock cycle* for today's computers, each state involving complex operations
  - Then it will only take $10^{1735}$ / $10^{20} = 10^{1715}$ seconds…

- But we have **<u>multiple cores</u>**!
  - The universe has at most $10^{87}$ particles, including electrons, …
  - Let's suppose every one is a CPU core
  - ➔ only $10^{1628}$ seconds > $10^{1620}$ years
  - The universe is around $10^{10}$ years old

# Impractical Algorithms



- Dijkstra's algorithm is **completely impractical** here
  - Visits all nodes with cost < cost(optimal solution)
- **Breadth first** would not work
  - Visits all nodes with length < length(optimal solution)
- **Iterative deepening** would not work
  - Saves space, still takes too much time
- **Depth first** search would **normally** not work
  - Always extends the plan if possible, always takes the first applicable action
  - Could work in *some* domains and *some* problems, by pure luck…
  - Usually either doesn't find the goal, or finds **very** inefficient plans

The state space is fine,
but we need some *guidance*!
But first, another *direction*…

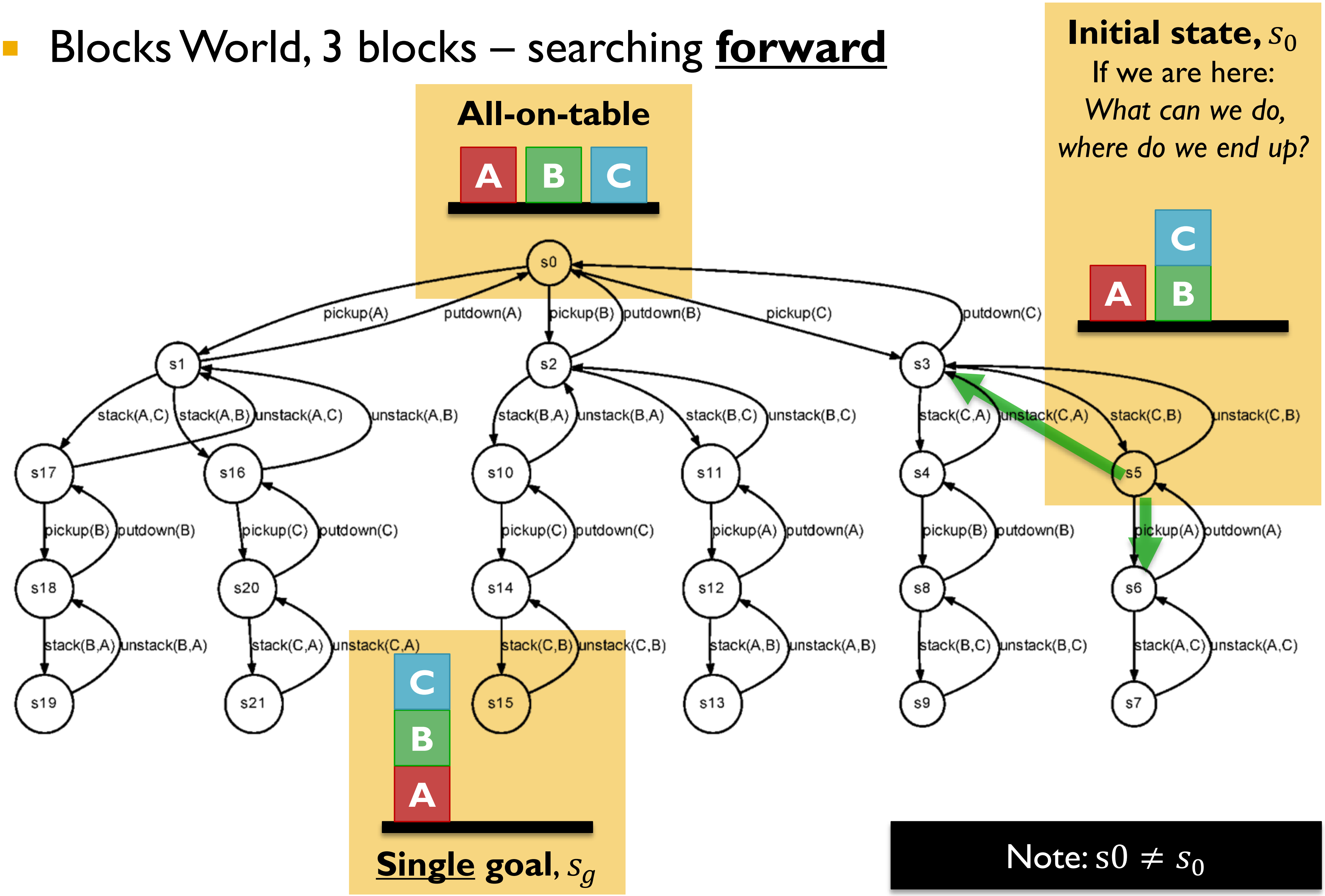
# Backward Search

# Forward Search



jonkv@ida

- Blocks World, 3 blocks – searching **forward**



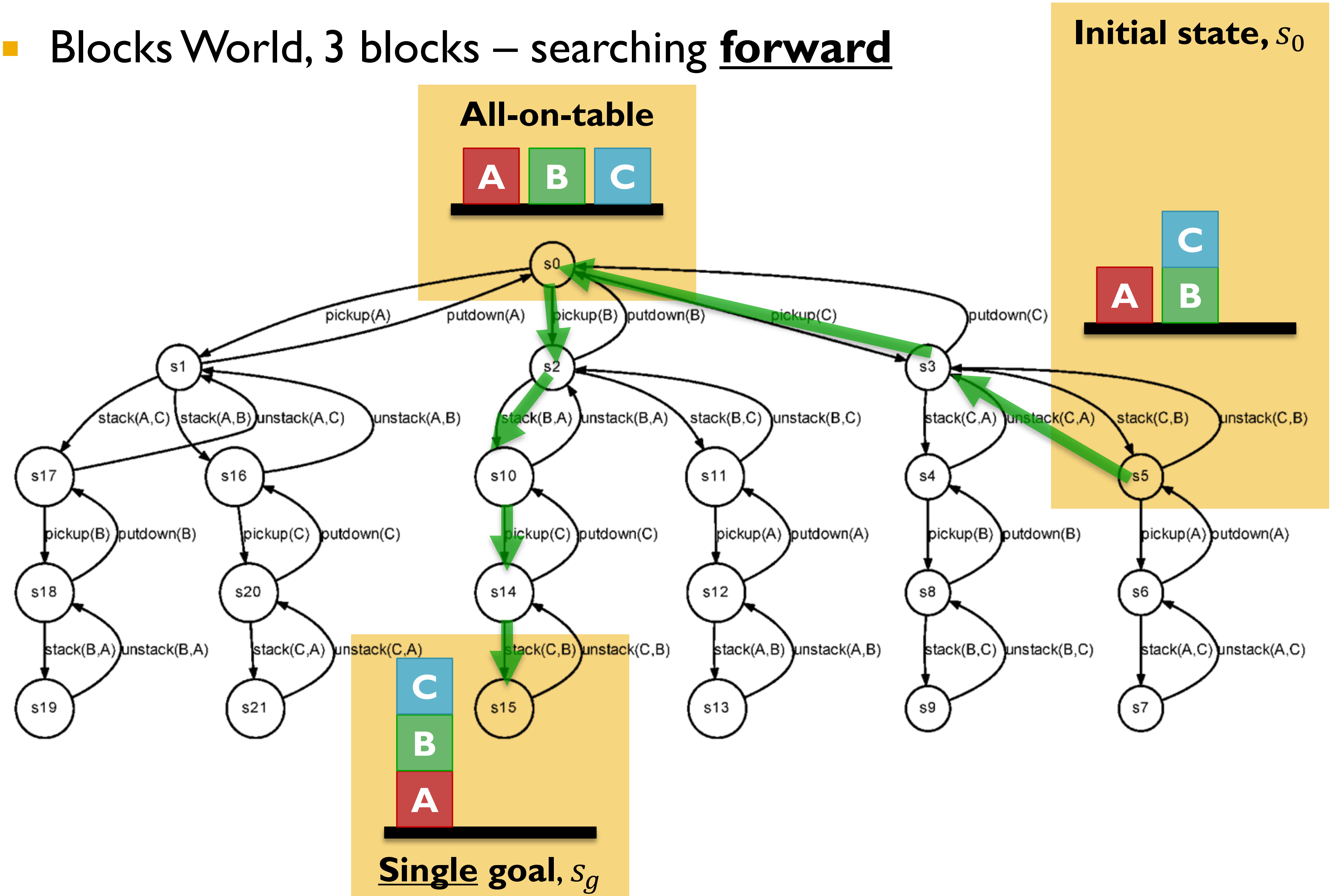

Note: $s0 \neq s_0$

# Forward Search (2)



jonkv@ida

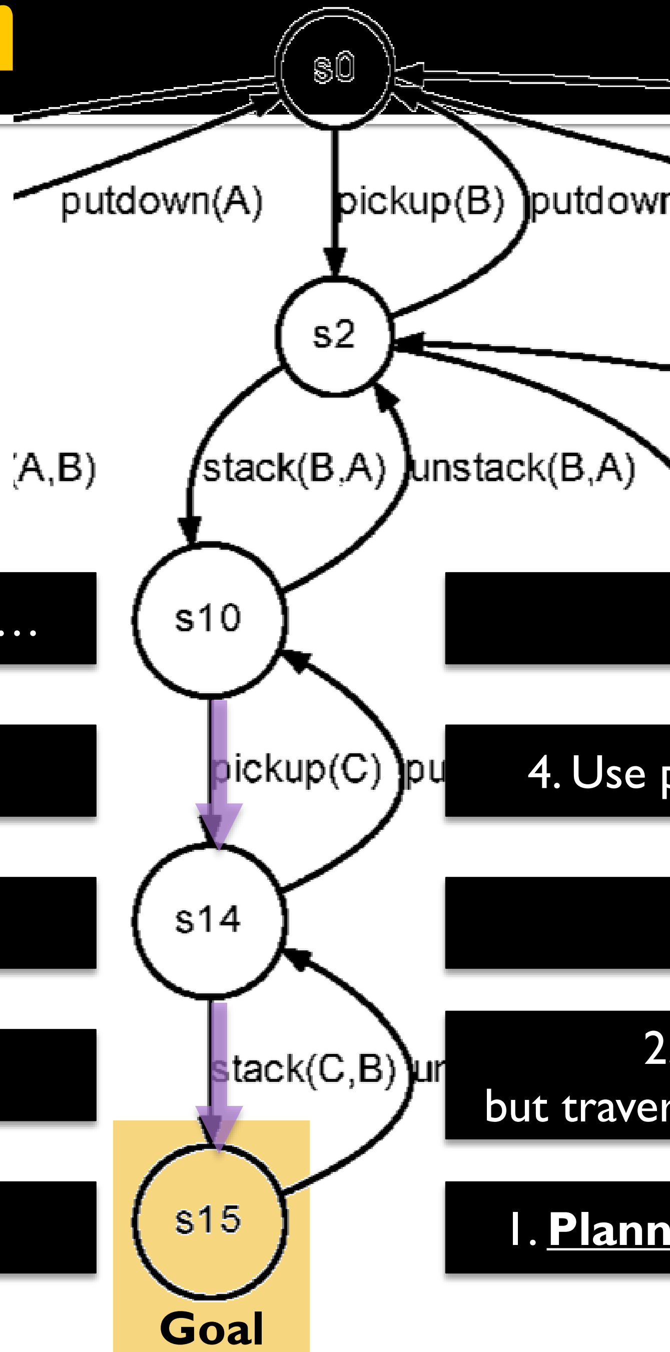
- Blocks World, 3 blocks – searching **forward**



# Backward Search



- Must traverse edges backwards!

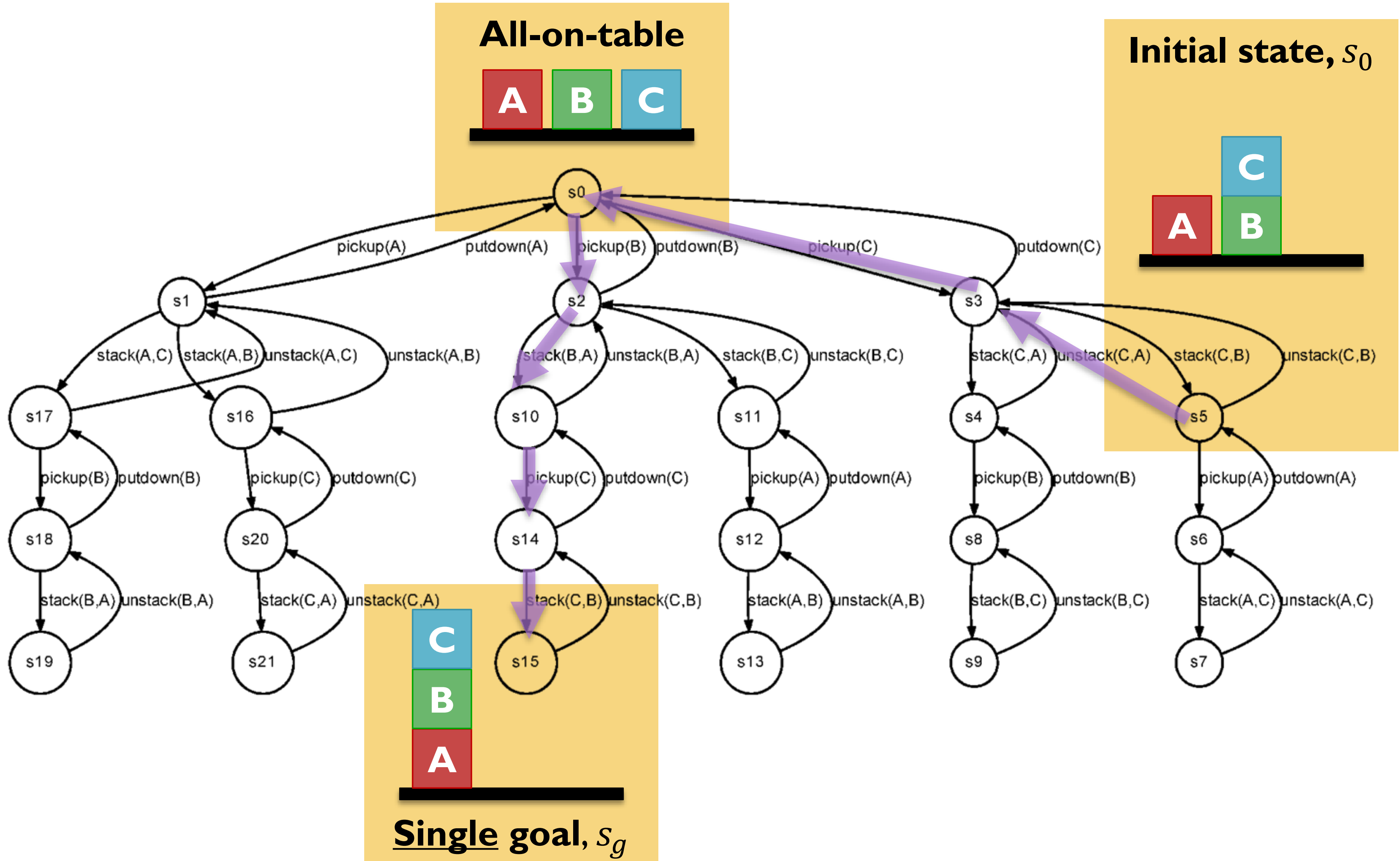



1. **Execution** should pass s10…
2. Execute pickup(C)…
3. Pass s14…
4. Execute stack(C,B)…
5. …and end up in s15

5. Pass s10…
4. Use pickup(C) *backwards*…
3. Pass s14…
2. Use stack(C,B), but traversing the edge *backwards!*
1. **Planning** must start in s15…

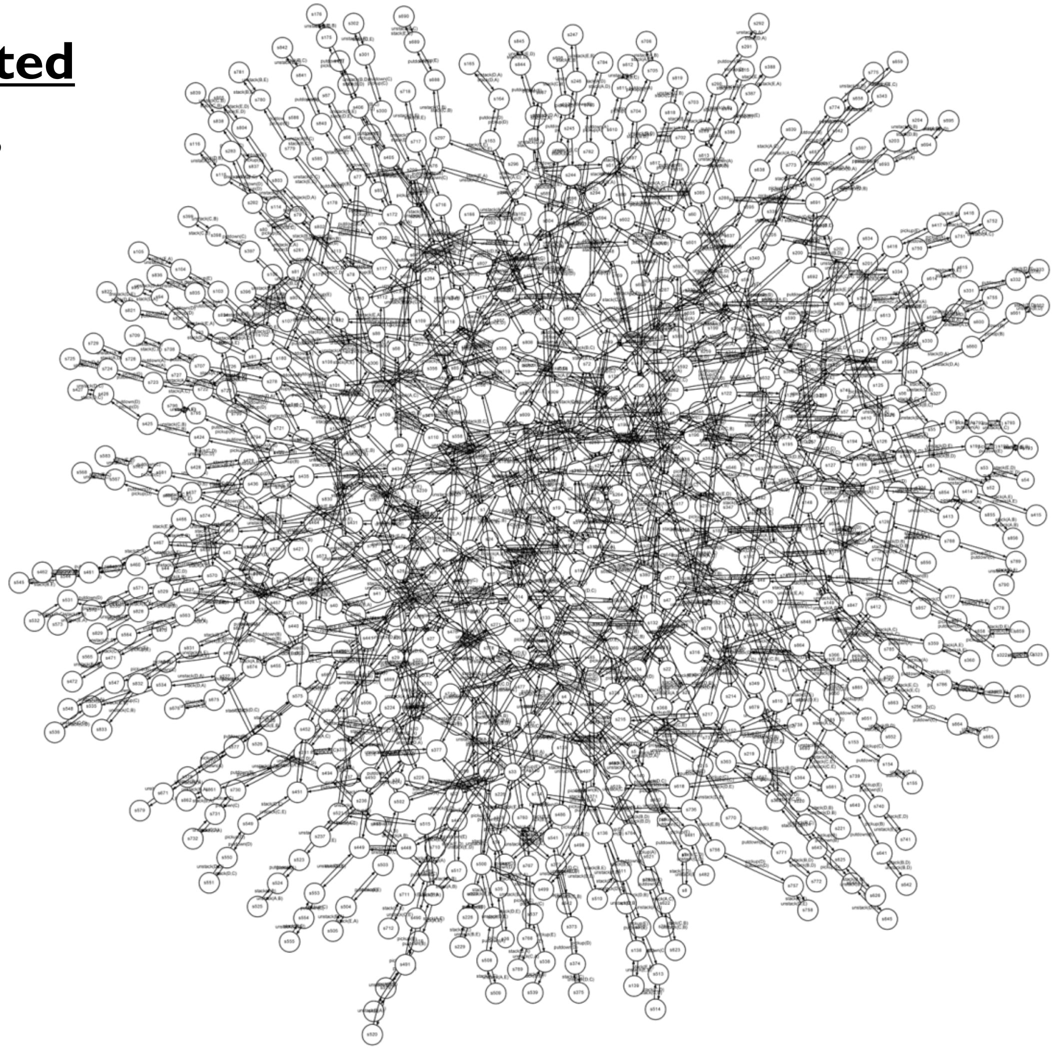
# Backward Search

- Searching **backward**



# Backwards Search: Complication 1

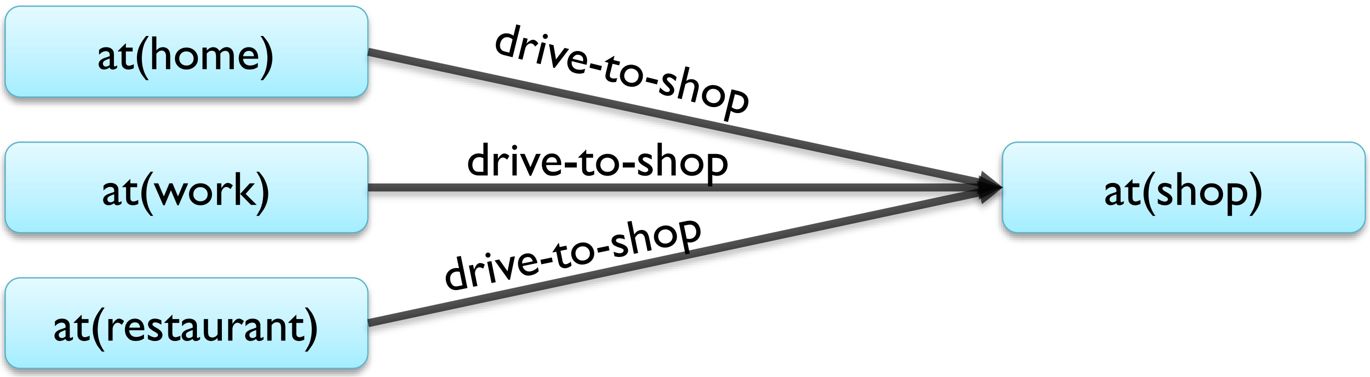
- Complication 1:
  - **<u>The graph isn't precomputed</u>**
    - Must be expanded dynamically, starting in the *goal*
    - Would require an *inverse* of $\gamma(s, a)$: $\gamma^{-1}(s, a)$



# Backwards Search: Complication 2



- Complication 2:
  - **Determinism is unidirectional**, not applicable in backward search
    - Compute $\gamma^{-1}(\{at(shop)\}, \text{drive−to−shop})$:
      If we want to **end up at(shop),**
      what **state** must we be in **before drive-to-shop**?

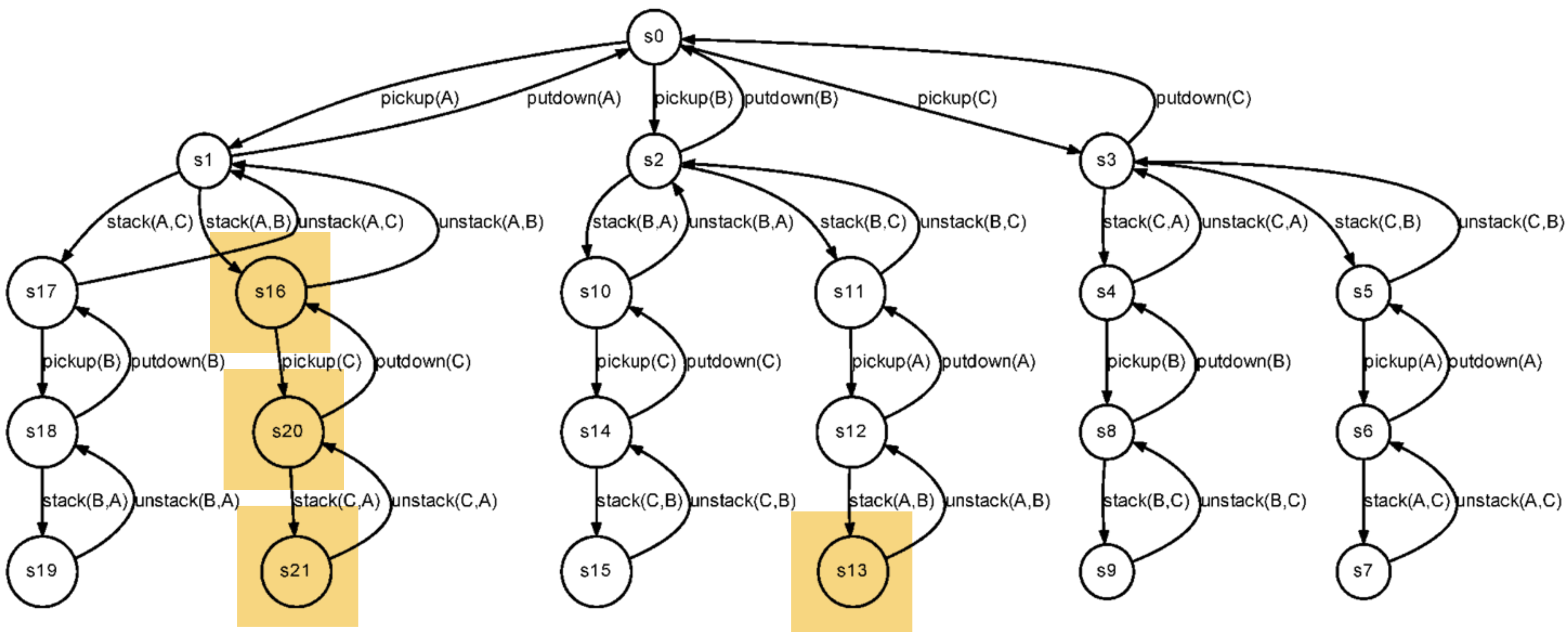


# Backwards Search: Complication 3





- Complication 3:
  - We generally have **<u>multiple goal states</u>** – to **start** in…
    - Goal: on(A,B)



# Backward Search:
# Many complications – same solution

# Repetition: States and goals



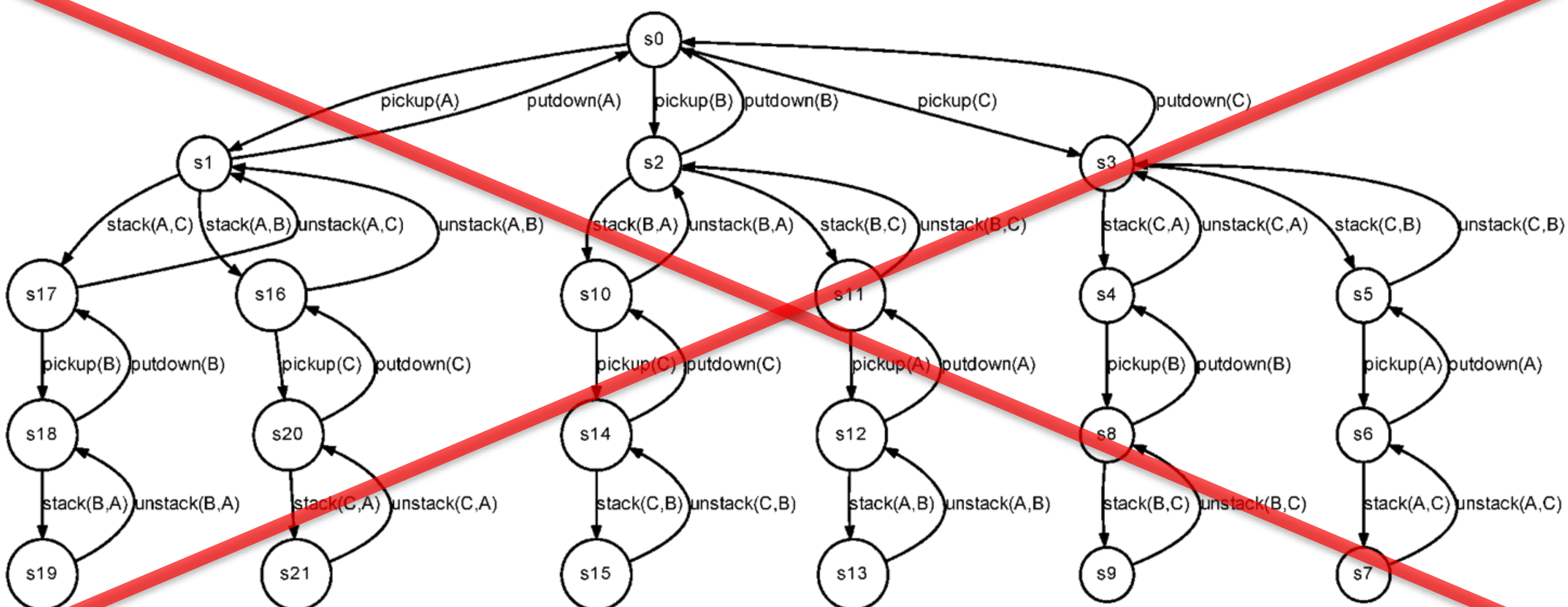
- Recall:
  - A **state** is a set containing **all atoms that are true**
    - **s = { on(A,B), on(C,D) }**
      - No block is clear or ontable:
        If they were, that would have been specified
  - A **goal** is a set of **literals** that **should hold**…
    - **g = { on(A,B), ¬on(C,D) }**
      - A should be on B, and C should *not* be on D
      - We don't care if blocks are clear / ontable or not:
        *If* we cared, that would have been specified
      - Can correspond to **many states**

# Goal Space ≠ State Space





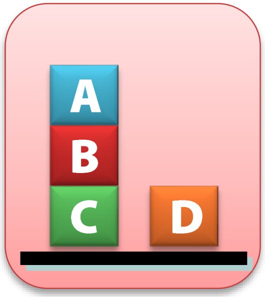
**Backward search uses goal space!**




**Will <u>not</u> construct <u>this</u> graph – use $\gamma^{-1}(g,a)$, not $\gamma^{-1}(s,a)$**

# Goal Specifications

- Suppose we want exactly this:

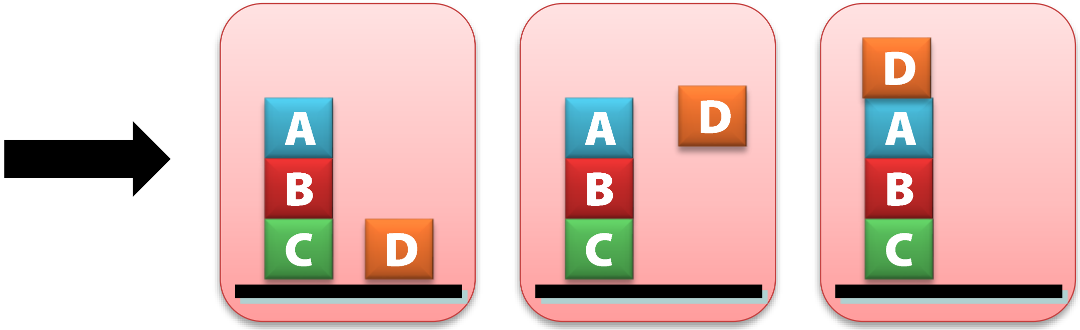



- What is the goal?
  - Could be a **complete** goal (➔ unique state)
    - { clear(A), on(A,B), on(B,C), ontable(C), clear(D), ontable(D), handempty, ¬clear(B), ¬on(A,A), … }
  - But this may be **sufficient**:
    - { on(A,B), on(B,C), ontable(C), ontable(D) }
    - Specifies all positions;
      given a *physically achievable initial state*, other facts follow implicitly

# Goal Specifications (2)



- Usually we **<u>don't care</u>** about all facts (directly or indirectly)!
  - Ignore the location of block D
  - on(A,B)
    ¬clear(B)
    on(B,C)
    ontable(C)



*Relevance:*

*Which actions could achieve part of the goal?*

# Backward Search: Relevance





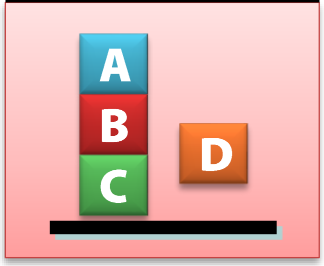
*Later:*
*Where would we have to start?*

**What action $a$ could be the <u>last</u> before achieving the 4 goal facts?**

$g$:
Suppose we want to achieve this…

on(A,B)
¬clear(B)
on(B,C)
ontable(C)

$g$ specifies *some* of the facts we illustrate below…



# Backward Search: Relevance (2)

**No!**

It achieves clear(?top) = clear(B)
The goal requires ¬clear(B)

➔ Destroys part of the goal

stack(B,C) is ***not relevant***
(also *impossible*,
but this is included in ***relevance***)

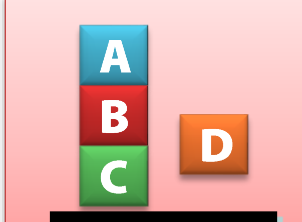
Could **stack(B,C)** be the last action in a plan achieving *g*?

```
(:action stack
  :parameters (?top ?below)
  :precondition (and (holding ?top)
                     (clear ?below))
  :effect
  (and (not (holding ?top))
       (not (clear ?below))
       (clear ?top)
       (handempty)
       (on ?top ?below)))
```

*g*:
Suppose we want to achieve this…

on(A,B)
¬clear(B)
on(B,C)
ontable(C)

*g* specifies *some* of the facts we illustrate below…

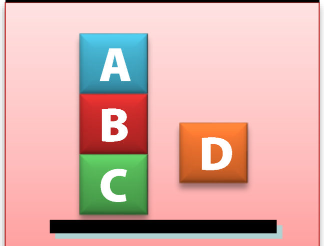


# Backward Search: Relevance (2)

*g*:
Suppose we want to achieve this…

on(A,B)
¬clear(B)
on(B,C)
ontable(C)

*g* specifies *some* of the facts we illustrate below…




Could **pickup(D)** be the last action in a plan achieving *g*?

```
(:action pickup
  :parameters (?x)
  :precondition (and (clear ?x)
         (on-table ?x)
         (handempty))
 :effect
 (and (not (on-table ?x))
   (not (clear ?x))
   (not (handempty))
   (holding ?x)))
```

**Yes**! Effects:
¬ontable(D)
¬clear(D)
¬handempty
holding(D)

Does not contradict the goal

…but also doesn't help us *achieve* any aspect of the goal!

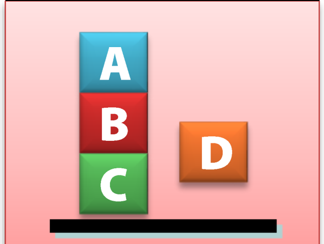
pickup(D) is ***not relevant***

# Backward Search: Relevance (3)

***g*:**
Suppose we want to achieve this…

on(A,B)
¬clear(B)
on(B,C)
ontable(C)

*g* specifies *some* of the facts we illustrate below…




Could **stack(A,B)** be the last action in a plan achieving *g*?

```
(:action stack
  :parameters (?top ?below)
  :precondition (and (holding ?top)
                     (clear ?below))
  :effect
  (and (not (holding ?top))
       (not (clear ?below))
       (clear ?top)
       (handempty)
       (on ?top ?below)))
```

**Yes!** Effects:
¬holding(A)
¬clear(B)
clear(A)
handempty
on(A,B)

Does **not contradict** the goal, **achieves** on(A,B)

stack(A,B) is ***relevant***

# Backward Search: Summary (so far)





## **Forward** search, over **states** $s = \{atom_1, \ldots, atom_n\}$:

$a$ is **applicable** to *current state* s **iff**

$\text{precond}^+(a) \subseteq s$ and

$s \cap \text{precond}^-(a) = \emptyset$

**Positive conditions** are present

**Negative conditions** are not present

## **Backward** search, over **sets of literals** $g = \{lit_1, \ldots, lit_n\}$

$a$ is **relevant** for *current goal* g **iff**

g ∩ effects($a$) ≠ ∅ and

g+ ∩ effects–($a$) = ∅ and

g– ∩ effects+($a$) = ∅

**Contribute** to the goal
(add positive or negative literal)

Do not **destroy** any goal literals

*Regression:*
*What must be true before?*

# Progression and Regression




**Forward** search, over **states** $s = \{atom_1, \ldots, atom_n\}$:

**Progression**: $\gamma(s, a) = (s - \text{effects}^-(a) \cup \text{effects}^+(a))$

I am in state s $\xrightarrow{\text{Action } a \text{ is applicable}}$ I would end up in $\gamma(s, a)$

**Backward** search, over **sets of literals** $g = \{lit_1, \ldots, lit_n\}$

**Regression**: $\gamma^{-1}(g, a) = ???$

I would require $\gamma^{-1}(g, a)$ $\xrightarrow{\text{Action } a \text{ is relevant for } g}$ I need to achieve goal $g$

# Backward Search: Regression



$g' = \gamma^{-1}(g, \textbf{stack(A,B)})$

What facts $g'$
would we require
*before executing a*,
so that for every state $s$
satisfying $g'$:

1) A is **executable** in s
2) $g \subseteq \gamma(s, a)$?

What action *a*
could be the <u>*last*</u>
*before achieving*
*the 4 goal facts?*

$g$:
We want
to achieve
this…

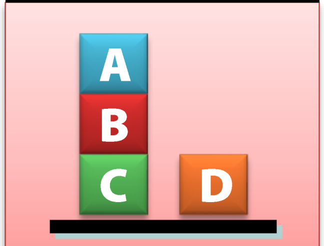
on(A,B)
on(B,C)
ontable(C)
ontable(D)

**<u>Example of subset</u>**

$g$ = { on(A,B), on(B,C), ontable(C), ontable(D) }

$\gamma(a, s)$ = { on(A,B), on(B,C), ontable(C), ontable(D),
clear(A), clear(D), handempty }

$g$ specifies *some*
of the facts we
illustrate below…



# Backward Search: Regression (2)


jonkv@ida

$\gamma^{-1}(g, \textbf{stack(A,B)})$

| | |
|---|---|
| Needed by **stack(A,B)** | holding(A)<br>clear(B) |
| What the goal needs, but **stack(A,B)** did not achieve | on(B,C)<br>ontable(C)<br>ontable(D) |

**stack(A,B)**

```
(:action stack
  :parameters (?top ?below)
  :precondition (and (holding ?top)
                     (clear ?below))
  :effect
  (and (not (holding ?top))
       (not (clear ?below))
       (clear ?top)
       (handempty)
       (on ?top ?below)))
```

$g$:
We want to achieve this…

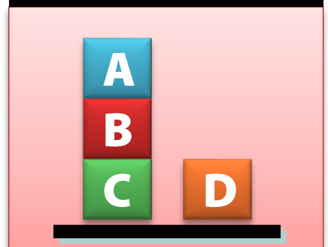
*on(A,B)*
on(B,C)
ontable(C)
ontable(D)

$g$ = {on(A,B), on(B,C), ontable(C), ontable(D) }

$\gamma^{-1}(g, \textbf{stack(A,B)})$ =
{ holding(A), clear(B), on(B,C), ontable(C), ontable(D) }

Corresponds to
*many potential states*

$g$ specifies *some* of the facts we illustrate below…



# Backward Search: Regression (3)




- Formally:

All goals except effects(a) must already have been true

precond(a) must have been true, so that a was applicable

$\gamma^{-1}(g,a) = ((g - \text{effects}(a)) \cup \text{precond}(a))$,
*representing*
$\{ s \mid a \text{ is applicable to } s \text{ and } \gamma(s,a) \text{ satisfies } g \}$

Backward / regression:
Which states could I start from?

Works for:

***Classical goals*** (already sets of ground literals)
***Classical effects*** (conjunction of literals)
***Classical preconditions*** (conjunction of literals)

**What happens**
**if we allow arbitrary (disjunctive) preconditions?**

# Backward Search: Reaching the Goal




$g_2 = \gamma^{-1}(g, \text{stack(A,B)})$

holding(A)
clear(B)

on(B,C)
ontable(C)
ontable(D)

$g_3$

$g_4$

$g_5$

stack(A,B)

$g$:
We want to achieve this…

on(A,B)
on(B,C)
ontable(C)
ontable(D)

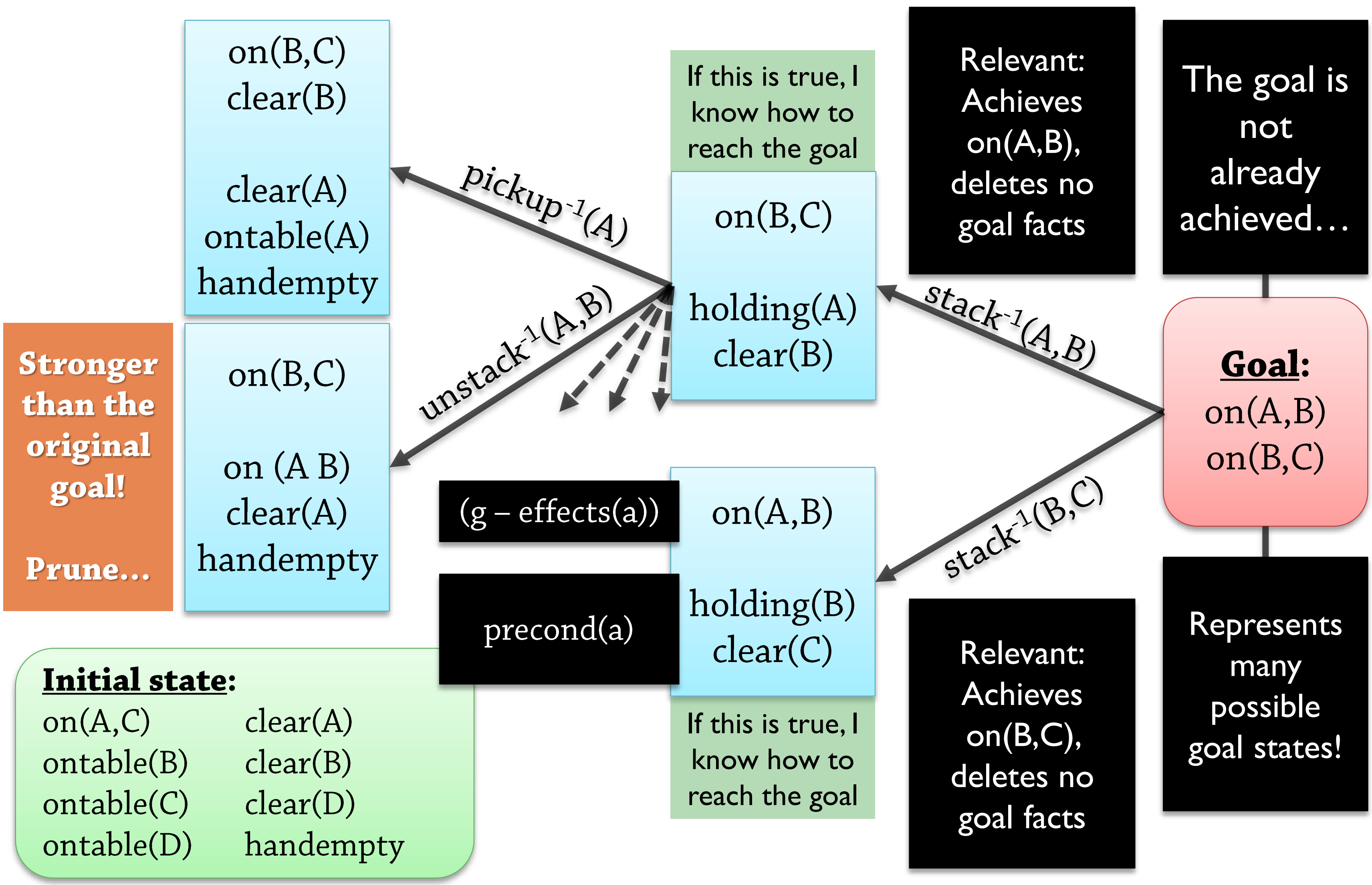
I can reach the goal from *any* state satisfying some $g_i$!

I can reach the goal from *any* state satisfying these 5 literals!

**If the literals are satisfied in $s_0$, I have a solution!**

# Backward Search: Example



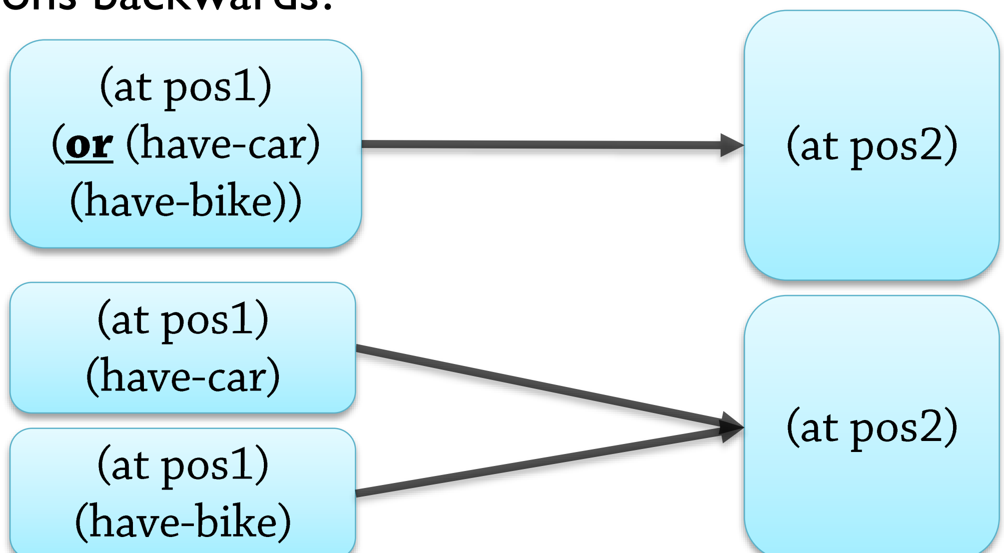


*Forward vs. Backward*

# Backward and Forward Search: Expressivity



- How about **expressivity**?
  - Suppose we have **disjunctive preconditions**
    - (:**action** travel
      :**parameters** (?from ?to – location)
      :**precondition** (and (at ?from) (**or** (have-car) (have-bike)))
      :**effects** (**and** (at ?to) (not (at ?from))))
  - How do we apply such actions backwards?
    - More complicated **disjunctive goals** to achieve?
    - Additional **branching**?




Similarly for existentials ("**exists** block [ on(block,A) ]"): One branch per possible value
Some extensions are less straight-forward in backward search (but possible!)

# Backward and Forward Search: Unknowns



| Forward search | Backward search |
| --- | --- |

**Forward search**

… → …

I can reach this node from the initial state…
But what comes next?
Can I reach the goal?
Efficiently?

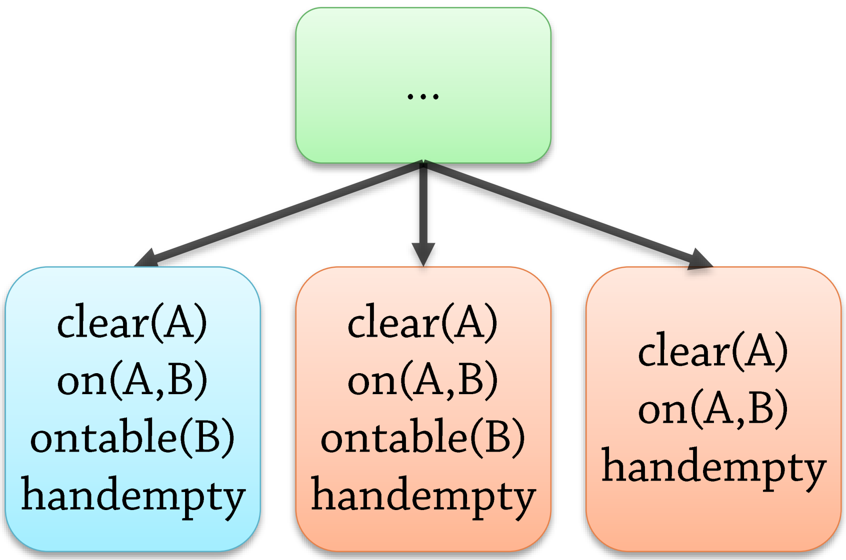
**Backward search**

on(B,C) clear(B) on(A,B) clear(A) handempty ← on(A,B) on(B,C)

I can reach the goal from this node…
But what comes before?
Can I reach it from s0?
Efficiently?

# Backward and Forward Search: Pruning

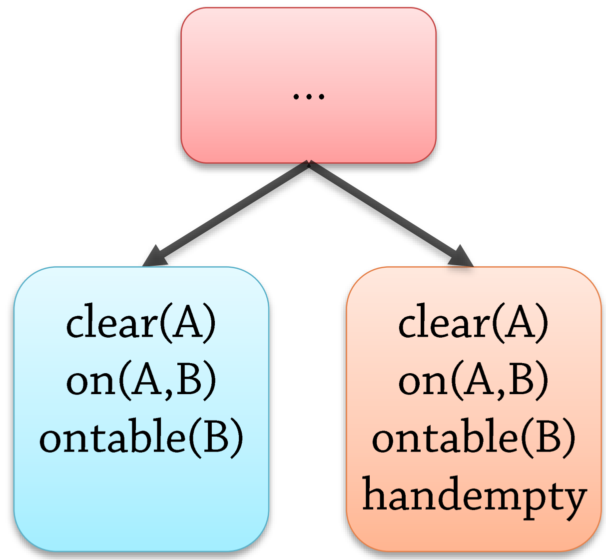
## Forward search




Reach a node with the *same* state
➔ can prune

If preconditions and goals are positive:
Reach a node with a *subset* of the facts
➔ can prune

## Backward search




Reach a node with the *same or stronger* goal
➔ can prune

# Backward and Forward Search: Problems



## FORWARD SEARCH

- Problematic when:
  - There are many **<u>applicable</u>** actions
    ➔ high branching factor
    ➔ need guidance
  - Blind search knows
    if an action is applicable,
    but not if it will contribute
    to the goal

## BACKWARD SEARCH

- Problematic when:
  - There are many **<u>relevant</u>** actions
    ➔ high branching factor
    ➔ need guidance
  - Blind search knows
    if an action contributes to the goal,
    but not if you can achieve its
    preconditions

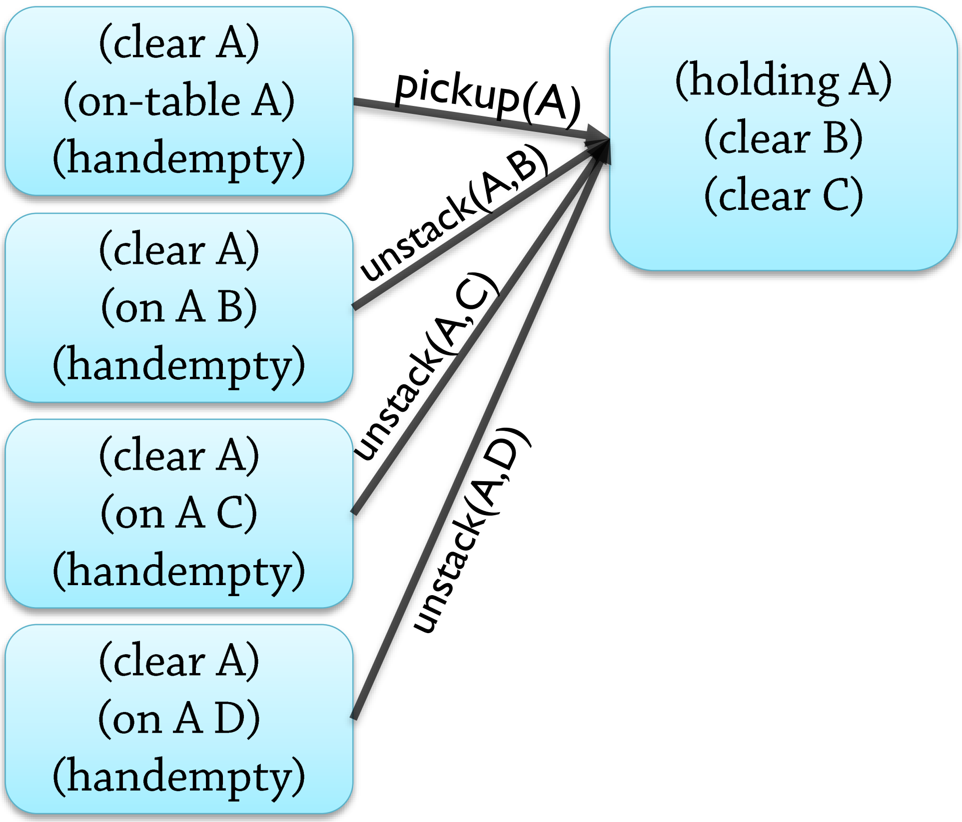
Blind backward search
is **<u>generally</u>** better than blind forward search:
Relevance **<u>tends</u>** to provide better guidance than applicability

This **<u>in itself</u>** is **<u>not</u>** enough to generate plans quickly!

# Lifted Search:
# A general technique

# Lifted Search (1)

- Even with conjunctive preconds:




  - High branching factor
  - No reason to decide *now* which block to unstack A from

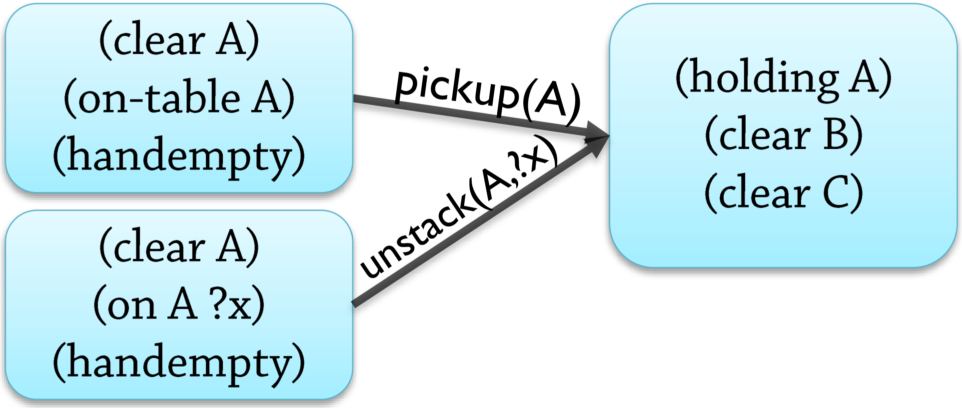
```
(:action pickup
   :parameters (?x)
   :precondition (and (clear ?x) (on-table ?x)
                      (handempty))
   :effect
   (and (not (on-table ?x))
        (not (clear ?x))
        (not (handempty))
        (holding ?x)))

(:action unstack
   :parameters (?top ?below)
   :precondition (and (on ?top ?below)
           (clear ?top) (handempty))
   :effect
   (and (holding ?top)
        (clear ?below)
        (not (clear ?top))
        (not (handempty))
        (not (on ?top ?below))))
```

# Lifted Search (2)



- General idea in *lifted search*:
  - Keep some variables uninstantiated (not *ground* ➔ "lifted")




**Next step**:
How to check for actions achieving (on A ?x)?

Requires *unification* – see the book, fig 4.3

Applicable to other types of planning – will return later!

But isn't *enough* to make unguided backward search efficient…

# Where do we go from here?

# Where do we go from here?



**Forward and backward search are useless without guidance!**

Add **general** guidance mechanisms to the planner

Typically: *Heuristics* to avoid blind search, *judge* which actions seem *promising*

Provide more **specific** information about each domain

*Control formulas*
*Hierarchical Task Networks*

Use a different search space and search algorithm

*Partial Order Causal Link*
*Satisfiability-based planning*
*Planning graphs*