TDDC17:
Introduction to Automated Planning

Jonas Kvarnström
Automated Planning Group
Department of Computer and Information Science
Linköping University

jonas.kvarnstrom@liu.se – 2016

I: Introduction to Planning

<u>**One**</u> way of defining planning:

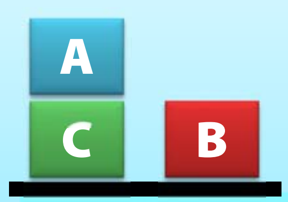
*Using **<u>knowledge</u>** about the world,*
including possible actions and their results,
*to **<u>decide</u>** what to do and when*
*in order to achieve an **<u>objective</u>**,*
***<u>before</u>** you actually start doing it*

Domains 1: Blocks World

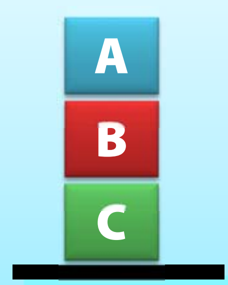
- Classical example: The **<u>Blocks World</u>**

You

Current state of the world



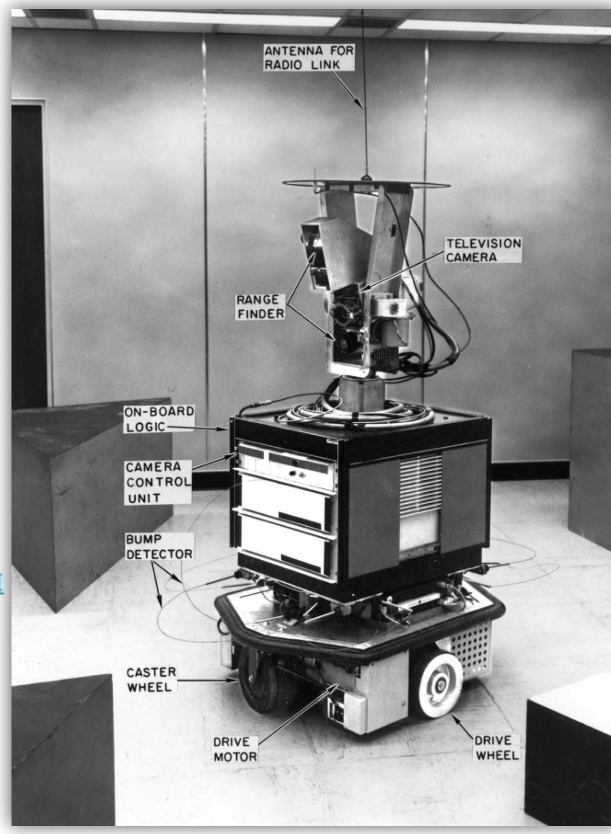
Your greatest desire



Domains 2: Shakey

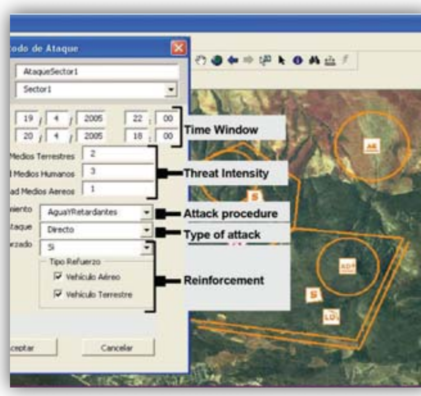
- Classical robot example: **Shakey** (1969)
  - Available **actions**:
    - *Moving* to another location
    - *Turning* light switches on and off
    - *Opening* and *closing* doors
    - *Pushing* movable objects around
    - …
  - **Goals**:
    - *Be in room 4 with objects A,B,C*

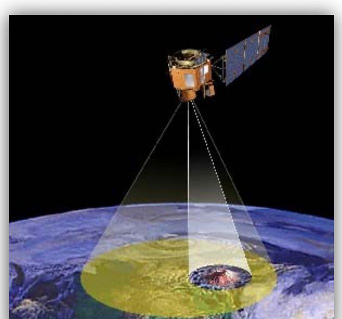
    - http://www.youtube.com/watch?v=qXdn6ynwpiI



Domains 3

Logistics:
Use a fleet of trucks to efficiently deliver packages





On-board planning to view interesting natural events: **http://ase.jpl.nasa.gov/**

SIADEX – plan for firefighting
Limited resources
Plan execution is dangerous!

Domains 4: Dock Worker Robots (DWR)

Containers shipped in and out of a harbor

Cranes move containers between "piles" and robotic trucks

Problem description

↓

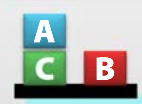
Solver: Planning Algorithm

↓

Solution

Could be a **customized** solver

```
plan         = [ ];
s            = current state of the world;
while (exists b1,b2 [ s.isOn(b1,b2) ]):
    plan     +=   "unstack(b1,b2)"
    s        =    apply(unstack(b1,b2), s)
    plan     +=   "putdown(b1)"
    s        =    apply(putdown(b1), s)
while (exists b1,b2 [ goal.isOn(b1,b2) & !s.isOn(b1,b2) &
                      s.isClear(b1) & s.isClear(b2) ]):
    plan     +=   "pickup(b1)"
    s        =    apply(pickup(b1), s)
    plan     +=   "stack(b1,b2)"
    s        =    apply(stack(b1,b2), s)
return plan
```



Tear down all towers

Rebuild in the right order

Efficient plans → more complex solvers
Complex domains → more complex solvers
Programming is time-consuming
Problem changes → more programming required

We want general + efficient algorithms!

AI Planning: A Simplified View

Knowledge about the world and how we can affect it

Description of objectives to achieve

General Planning Algorithm

Solution:
Plan with *actions* to perform, *ordering* constraints, …

Execution System

Sensors

Actuators

Simplified, because in reality:

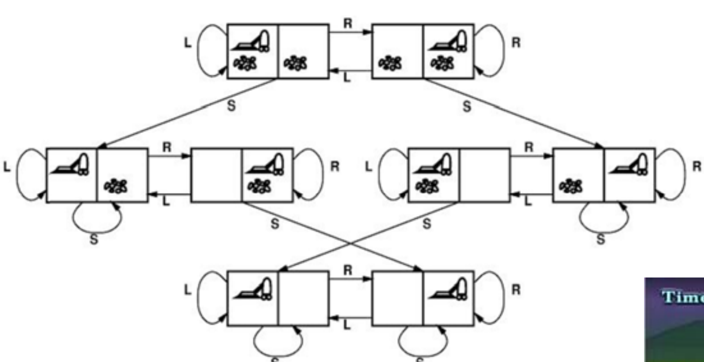
Execution will *fail* ➔ feedback, monitoring, replanning, plan repair

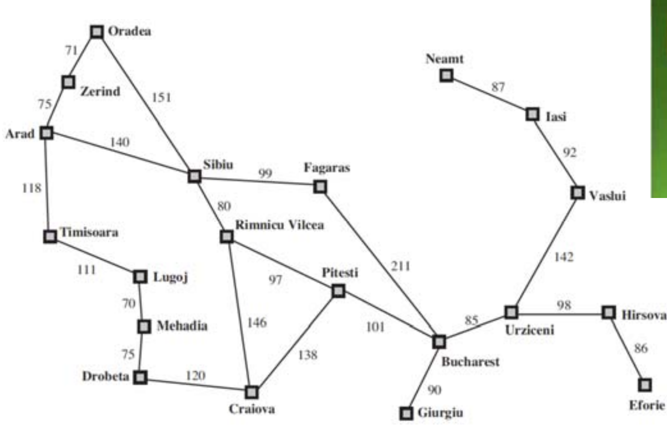
Planning occurs in a *context* ➔ Where are objectives generated?

Multiple agents may plan, interact ➔ negotiation, delegation of tasks, collaboration, competition,

And so on…

You have already planned – using general search algorithms!



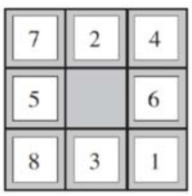


But the **representation** was problem-specific…

States: triple (x,y,z) with $0 \le x,y,z \le 3$, where x, y, and z represent the number of missionaries, cannibals and boats currently on the original bank.

And so was the search **guidance**!

Straight line distance from city n to goal city n’



$h_2(n)$: The sum of the manhatten distances for each tile that is out of place.

(3+1+2+2+2+3+3+2=18) . The manhatten distance is an under-estimate because there are tiles in the way.

How to **generalize**?

Classical Planning

First requirement: A **formal language** for planning problems!

- We want a **general** language:
  - Allows a wide variety of domains to be modeled

- We want good **algorithms**
  - Generate plans quickly
  - Generate high quality plans
- Easier with more **limited** languages

Conflicting desires!

- Many early planners made **similar tradeoffs**
  - Later called ”classical planning”
    Restricted, but a good place to start

Modeling Classical Planning Problems

Planning Domain, Problem Instance


Split knowledge into two parts

Planning Domain

- General properties of DWR problems
  - There are *containers*, *cranes*, …
  - Each object has a *location*
  - Possible actions:
    Pick up container, put down container, drive to location, …

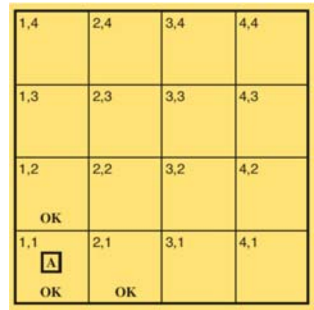
Problem Instance

- Specific problem to solve
  - *Which* containers and cranes exist?
  - Where is everything?
  - Where *should* everything be?
    (More general:
    What should we achieve?)

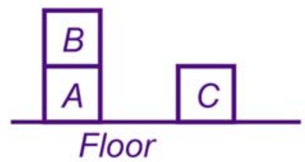
Objects 1: First-Order → Objects Exist

- Many planning languages have their roots in **logic**
  - Great – you've already seen this!

- Propositional syntax
  - Plain **propositional facts**:
    ~B1,1 , ~S1,1 , OK1,1 , OK2,1 , OK1
  - Used in *some* planners to simplify parsing, …



- First-order syntax
  - **Objects** and **predicates** (relations):
    On(B,A), On(C,Floor), Clear(B), Clear(C)
  - Used in most implementations



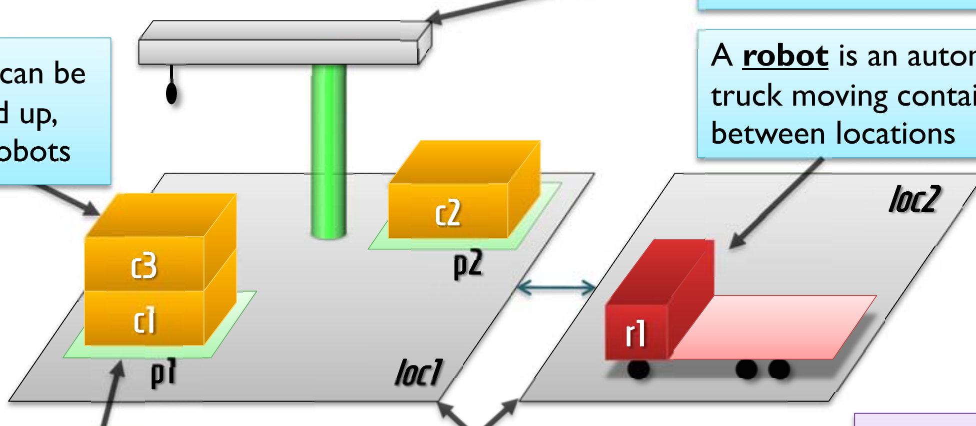
Objects 2: Object Types

- The domain often specifies **object types**

A **crane** moves containers between piles / robots

A **container** can be stacked, picked up, loaded onto robots

A **robot** is an automated truck moving containers between locations



A **pile** is a stack of containers – at the bottom, there is a **pallet**

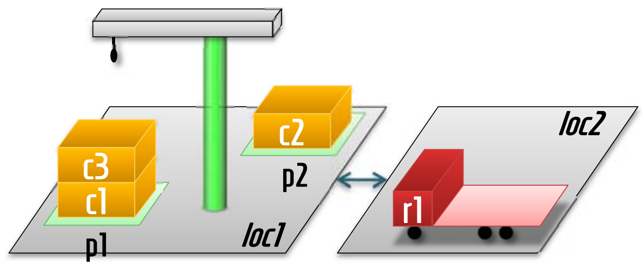
A **location** is an area that can be reached by a single crane. Can contain several piles, at most one robot.

We can skip:
Drivers
Wheels
…

Essential: Determine what is **relevant** for the **problem** and **objective**!

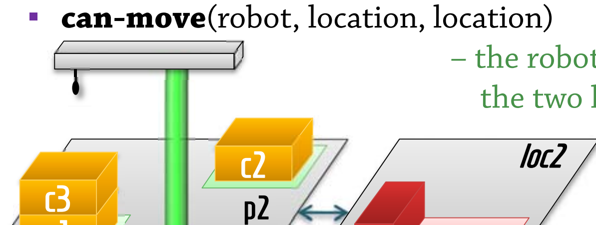
Objects 3: In the problem instance

- **Objects** are generally specified in the **problem instance**
  - robot: { r1 }
  - location: { loc1, loc2 }
  - crane: { k1 }
  - pile: { p1, p2 }
  - container: { c1, c2, c3, pallet }



Facts

- In **first-order representations** of planning problems:
  - Any **fact** is represented as a logical **atom**: Predicate symbol + arguments
  - **Properties** of the **world**
    - **raining** – it is raining [not a DWR predicate!]
  - **Properties** of **single objects**…
    - **empty**(crane) – the crane is not holding anything
  - **Relations** between objects
    - **attached**(pile, location) – the pile is in the given location
  - **Relations** between >2 objects
    - **can-move**(robot, location, location) – the robot can move between the two locations [Not a DWR predicate]



Essential: Determine what is **relevant** for the **problem** and **objective**!

Facts / Predicates in DWR

- All predicates for DWR, and their intended meaning:

"Fixed/Rigid" (can't change)	**adjacent**	(*loc1*, *loc2*)	; can move from *loc1* directly to *loc2*
	attached	(*p*, *loc*)	; pile *p* attached to *loc*
	belong	(*k*, *loc*)	; crane *k* belongs to *loc*
"Dynamic" (modified by actions)	**at**	(*r*, *loc*)	; robot *r* is at *loc*
	occupied	(*loc*)	; there is a robot at *loc*
	loaded	(*r*, *c*)	; robot *r* is loaded with container *c*
	unloaded	(*r*)	; robot *r* is empty
	holding	(*k*, *c*)	; crane *k* is holding container *c*
	empty	(*k*)	; crane *k* is not holding anything
	in	(*c*, *p*)	; container *c* is somewhere in pile *p*
	top	(*c*, *p*)	; container *c* is on top of pile *p*
	on	(*c1*, *c2*)	; container *c1* is on container *c2*

States 1: State of the World

- A **state (of the world)** should specify exactly *which facts* (***ground atoms***) *are true/false in the world* at a given time

We know all **predicates** that exist:
adjacent(location, location), …

We know which **objects** exist for each type

We can calculate all *ground atoms*

adjacent(loc1,loc1)
adjacent(loc1,loc2)
…
attached(pile1,loc1)
…

These are the *facts* to keep track of!

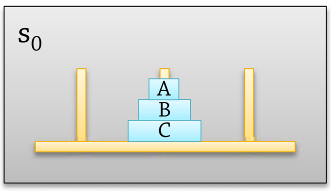
We can find all possible states!

Every assignment of true/false to the ground atoms is a distinct state

Number of states: $2^{\text{number of atoms}}$ – enormous, but finite (for classical planning!)

States 2: Efficient Representation

- **Efficient specification and storage** for a **single state**:
  - Specify **which atoms are true**
    - All other atoms have to be false – what else would they be?
  - ➔ A **state of the world** is specified as a **set**
    containing all **variable-free atoms** that [are, were, will be] true in the world
    - s_0 = { on(A,B), on(B,C), in(A,2), in(B,2), in(C,2), top(A), bot(C) }



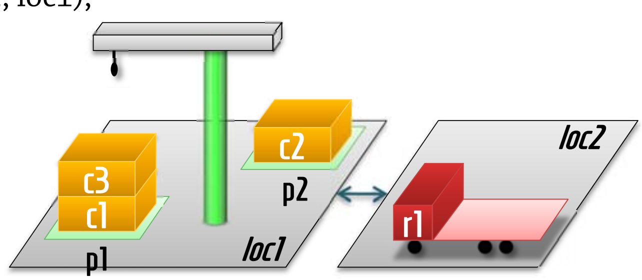
s_0	top(A)	in(A,2)
	on(A,B)	in(B,2)
	on(B,C)	in(C,2)
	bot(C)	

top(A) ∈ s_0 ➔ top(A) is true in s_0
top(B) ∉ s_0 ➔ top(B) is false in s_0

States 3: Initial State

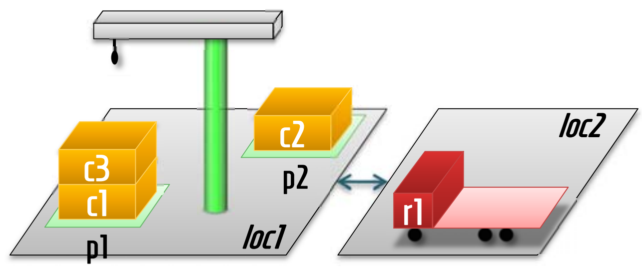
- Initial states *in classical STRIPS planning*:
  - We assume *complete information* about the **initial state** (before any action)
  - So we can use a set of true atoms
    - {
      attached(p1, loc1), in(c1, p1), on(c1, pallet), in(c3, p1), on(c3, c1), top(c3, p1),
      attached(p2, loc1), in(c2, p2), on(c2, pallet), top(c2, p2),
      belong(crane1, loc1), empty(crane1),
      at(r1, loc2), unloaded(r1), occupied(loc2) ,
      adjacent(loc1, loc2), adjacent(loc2, loc1),
      }

Complete *relative to the model*:
We must know everything
about those predicates and objects
we have specified...
But not whether it's raining!



States 4: Goal States

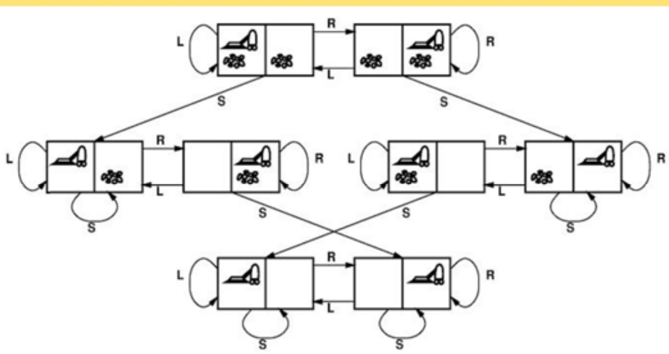
- Classical STRIPS planning: Reach one of possibly many **goal states**
  - Can be specified as a **set of literals** that must hold
  - Example: **Containers 1 and 3 should be in pile 2**
    - We don't care about their order, or any other fact
    - { in(c1,p2), in(c3,p2) }



Actions 1: Intro

- Actions in **plain search** (lectures 2-3):
  - Assumed a *transition / successor function*

Result(*State, Action*) - A description of what each action does (Transition function)



  - But how to **specify** it **succinctly**?

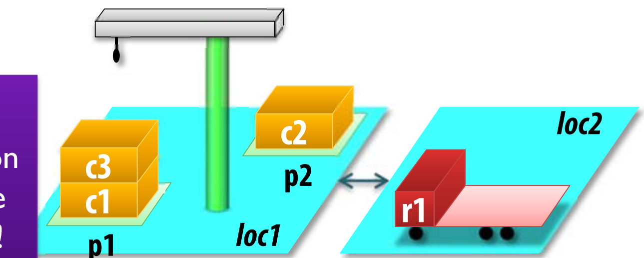
Actions 2: Operators

- Define **operators** or **action schemas**:
  - **move**(robot, location1, location2)
    - Precondition: **at**(robot, location1) ∧
      adjacent(location1, location2) ∧
      ¬ **occupied**(location2)
    - Effects: ¬**at**(robot, location1),
      at(robot, location2),
      ¬**occupied**(location1),
      occupied(location2)

The action is **applicable** in a state *s* if its precond is true in *s*

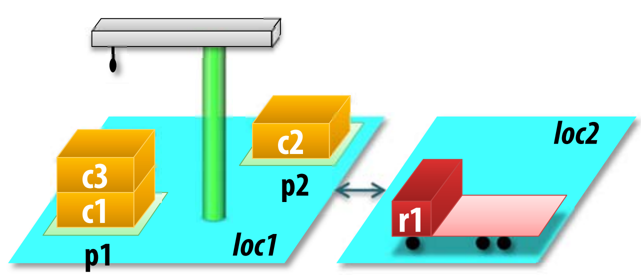
The result of applying the action in state *s*:
s – negated effects
+ positive effects

Classical planning:
Known initial state, known state update function
➔ **deterministic**, can completely predict the state of the world after a sequence of actions!



Actions 3: Instances

- The planner **instantiates** these schemas
  - Applies them to any combination of parameters of the correct type
  - **Example: move**(r1, loc1, loc2)
    - Precondition: **at**(r1, loc1) ∧
      adjacent(loc1, loc2) ∧
      ¬ **occupied**(loc2)
    - Effects: ¬**at**(r1, loc1),
      at(r1, loc2),
      ¬**occupied**(loc1),
      occupied(loc2)



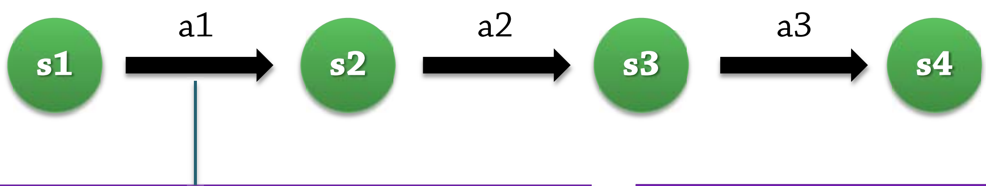
Actions 4: Step by Step

- In **classical** planning (the basic, limited form):

We know the initial state

Each action corresponds to **one** state update

Time is not modeled,
and never multiple state updates for one action



We know how states are changed by actions

→ **Deterministic**, can completely predict the state of the world after a sequence of actions!

The **solution** to the problem will be a **sequence** of actions

Plan Generation

With or Without Search

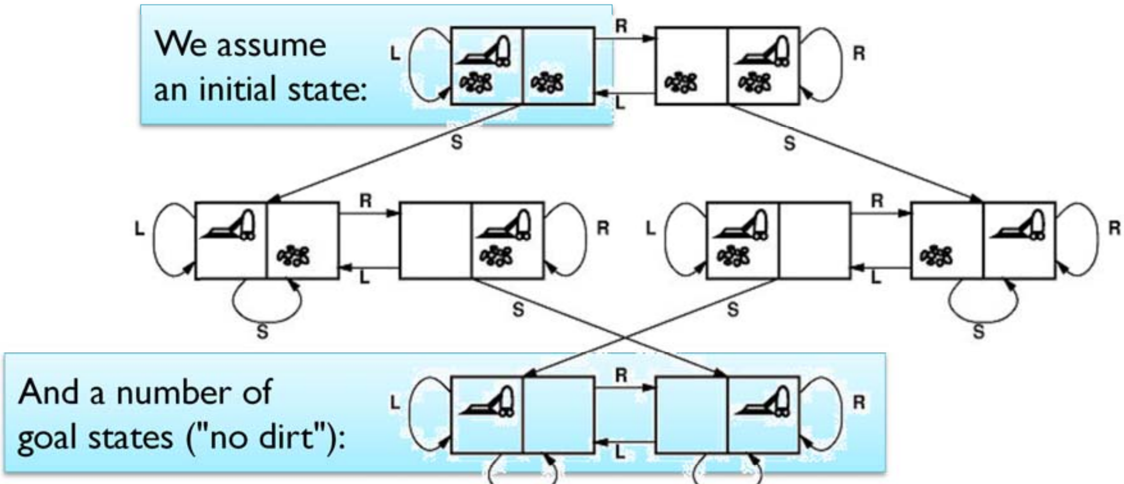
- Important distinction:
  - **Planning** means **deciding in advance** which **actions** to perform in order to **achieve a goal**
  - **Search** will often be a **useful tool**, but you can get by without it

```
plan            = [ ];
s               = current state of the world;
while (exists b1,b2 [ s.isOn(b1,b2) ]):
    plan        +=    "unstack(b1,b2)"
    s           =     apply(unstack(b1,b2), s)
    plan        +=    "putdown(b1)"
    s           =     apply(putdown(b1), s)
while (exists b1,b2 [ goal.isOn(b1,b2) & !s.isOn(b1,b2) &
                       s.isClear(b1) & s.isClear(b2) ]):
    plan        +=    "pickup(b1)"
    s           =     apply(pickup(b1), s)
    plan        +=    "stack(b1,b2)"
    s           =     apply(stack(b1,b2), s)
return plan
```

If we use search:
What search space?

Plan Generation Method 1:
Forward State Space Search

- Forward state space search: As explored in the **Vacuum World**
  - A **search node** is simply a **world state**
  - Successor / transition function:
    - One outgoing edge for every executable (applicable) action
    - The action specifies where the edge will lead



Find a path (not necessarily shortest) – SRS, RSLS, LRLRLSSSRLRS, …

How does a state space **look**?

State Spaces 1: Towers of Hanoi

Toy Problem 1: Towers of Hanoi has a very *regular* state space…

General Knowledge: Problem Domain

- There are *pegs* and *disks*
- A disk can be *on* a peg
- One disk can be *above* another
- Actions:
  Move topmost disk from *x* to *y*, without placing larger disks on smaller disks

Specific Knowledge and Objective: Problem Instance

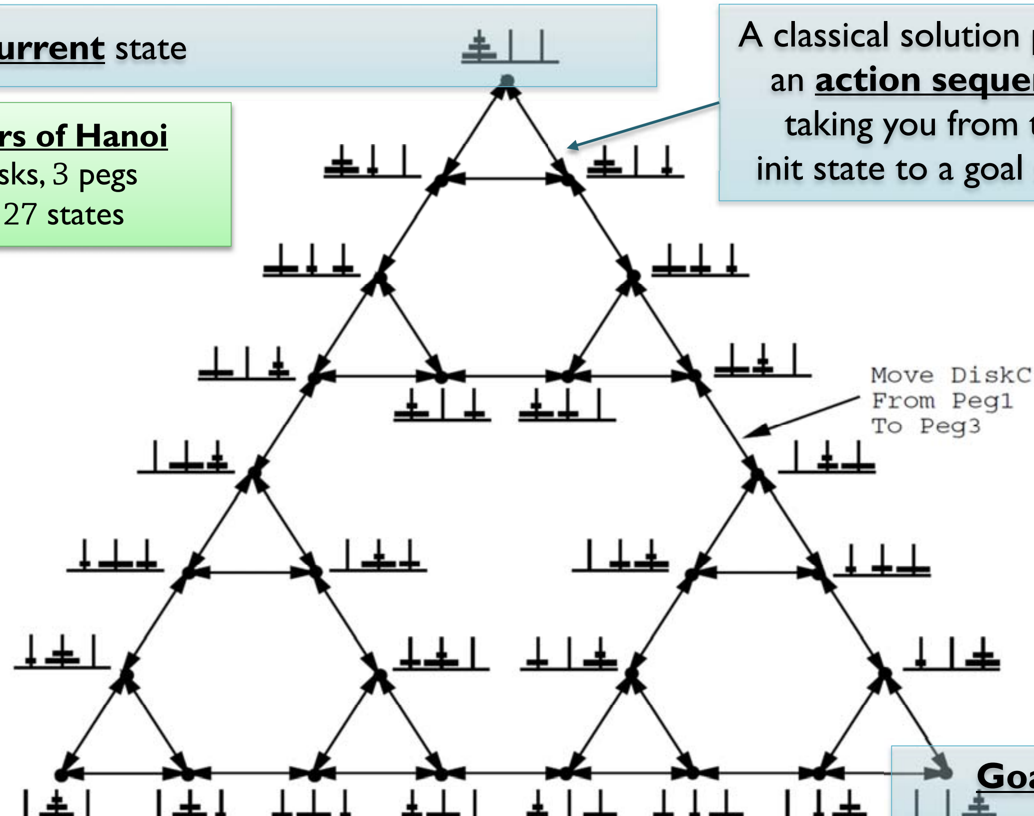
- 3 pegs, 7 disks
- All disks currently on peg 2, in order of increasing size
- We want: All disks on the *third* peg, in order of increasing size

State Spaces 2: ToH, Actions

Initial/current state

Towers of Hanoi
3 disks, 3 pegs
➔ 27 states

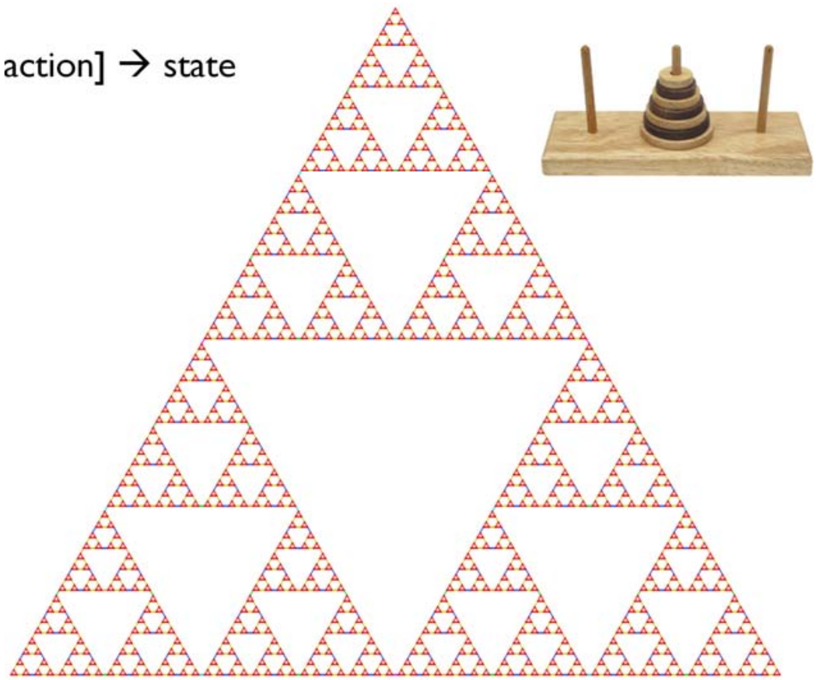
A classical solution plan is an **action sequence** taking you from the init state to a goal state



Goal state

State Spaces 3: Larger Example

- Larger state space – interesting symmetry
  - 7 disks
  - 2187 states
  - 6558 transitions, [state, action] → state



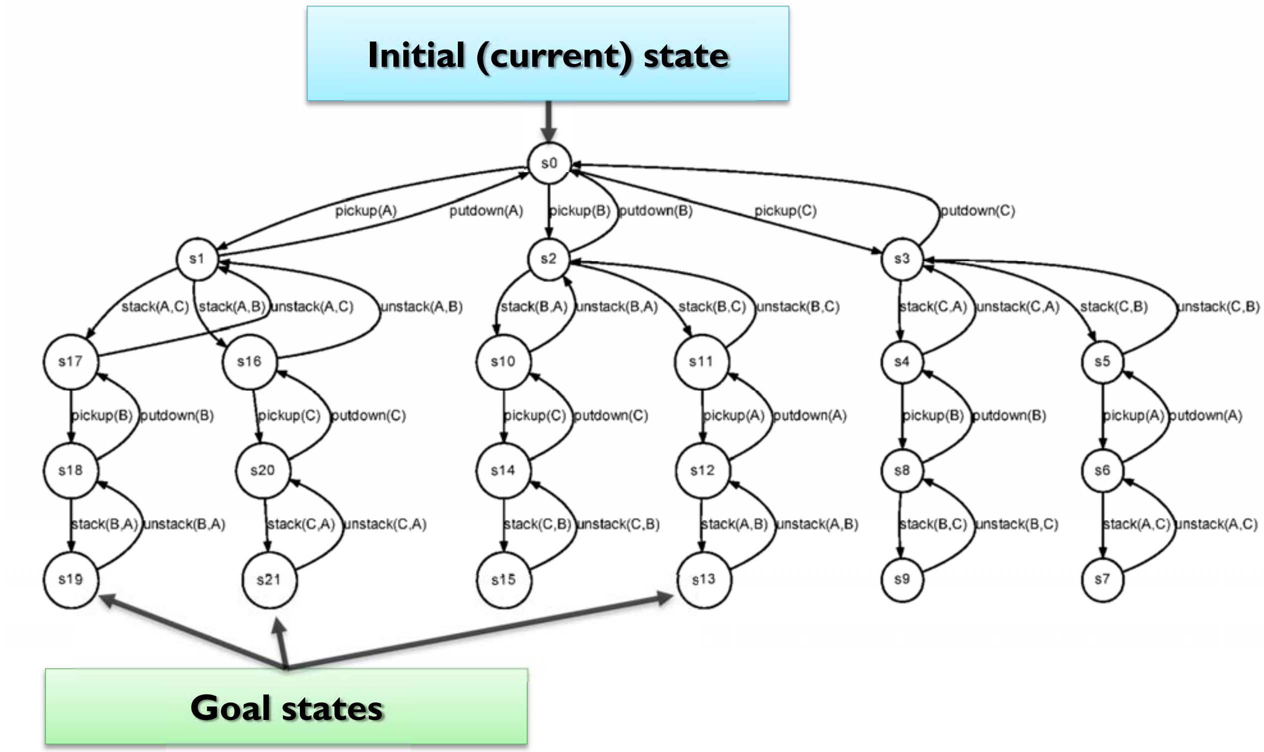
State Spaces 4: Blocks World

- 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 (relations) 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

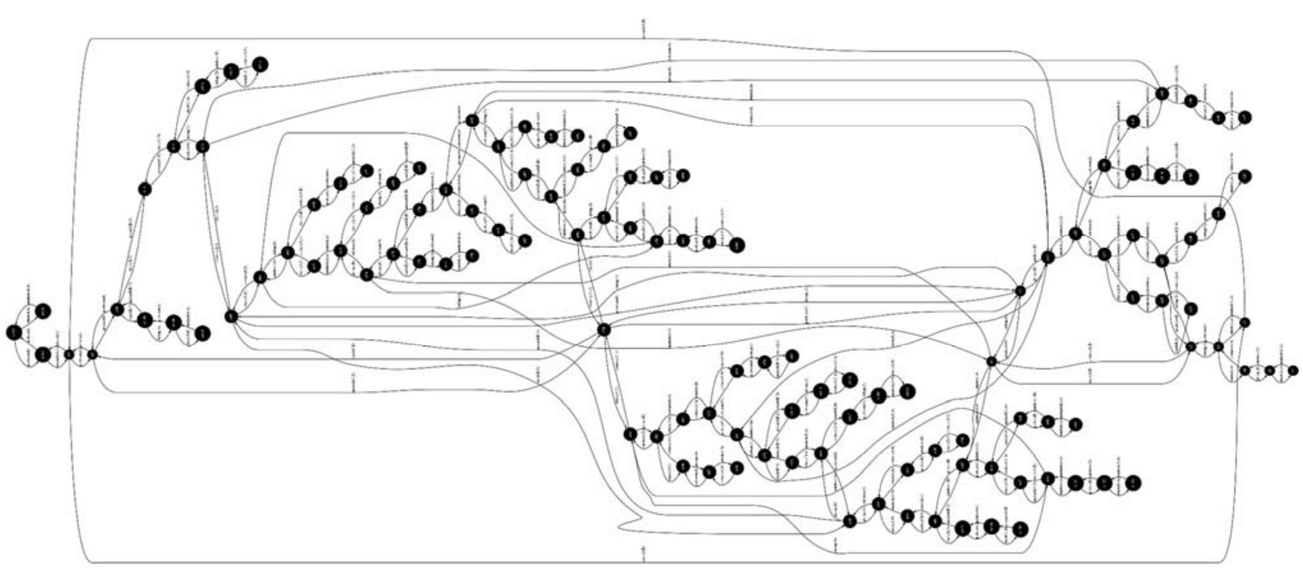
clear(A)
on(A, C)
ontable(C)
clear(B) ontable(B)
clear(D) ontable(D)
handempty

State Spaces 5: Blocks World, 3 blocks

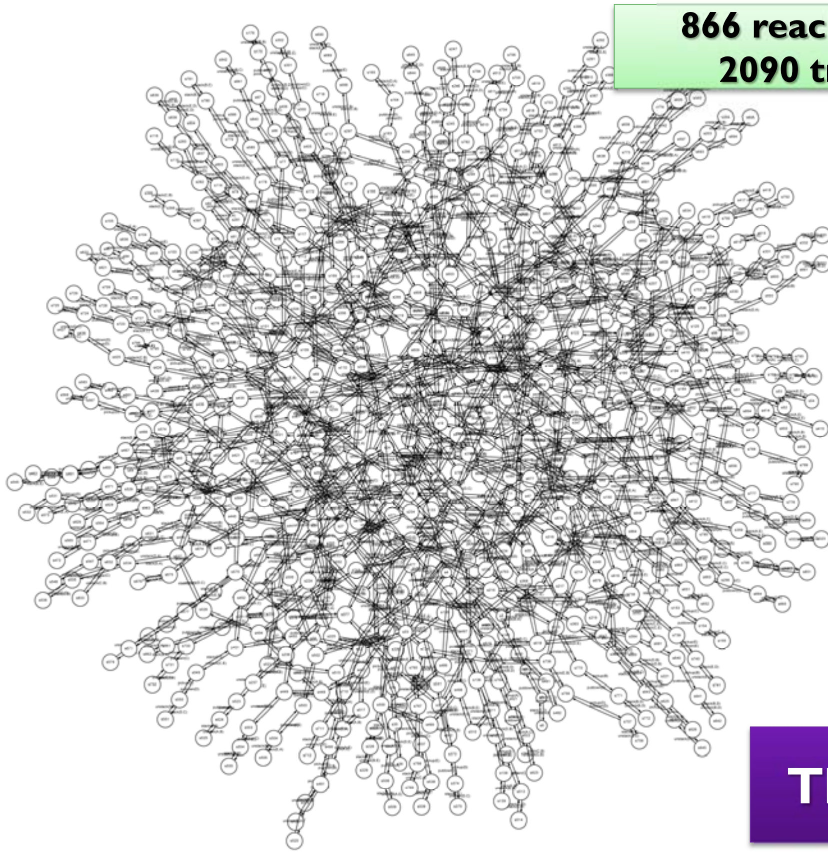


State Spaces 6: Blocks World, 4 blocks

125 reachable states
272 transitions



State Spaces 7: Blocks World, 5 blocks



866 reachable states
2090 transitions

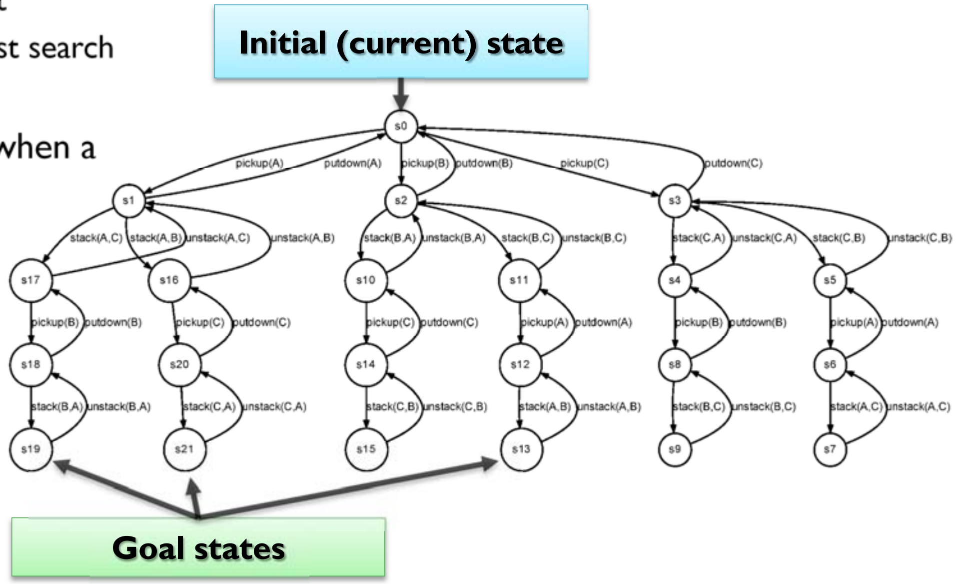
This is tiny!

State Spaces 8: Reachable State Space

Blocks	States reachable from "all on table"	Transitions (edges) in reachable part
0	1	0
1	2	2
2	5	8
3	22	42
4	125	272
5	866	2090
6	7057	18552
7	65990	186578
8	695417	2094752
9	8145730	25951122
10	…	…
…30	>197987401295571718915006598239796851	

Forward Search 1

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



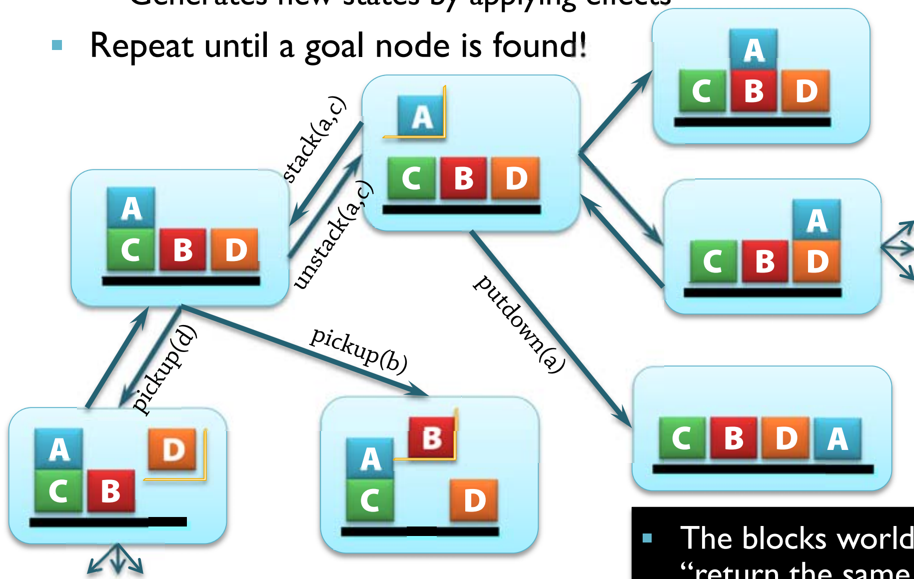
Forward Search 2: Step by step

Extremely many states – must generate the graph as we go

- **<u>Generate</u>** the initial node == initial state
  - From the initial state *description* in the input

Forward Search 3: Step by step

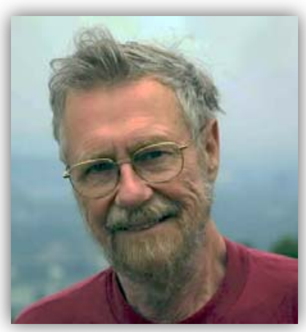
- **<u>Incremental expansion</u>**: **<u>Choose</u>** a node (using **<u>search strategy</u>**)
- Expand all possible successors
  - *"What actions are applicable in the current state, and where will they take me?"*
  - Generates new states by applying effects
- Repeat until a goal node is found!



- The blocks world is *symmetric*: Can always "return the same way"
- Not true for all domains!

Dijkstra

- Which search strategy to use?
  - Breadth first, depth first, iterative deepening depth first, …
- One possible strategy: **<u>Dijkstra's algorithm</u>**
  - Uniform-cost search, but terminate when a goal state is found
  - **<u>Efficient</u>**: $O(|E| + |V| \log |V|)$
    - |V| = the number of nodes
    - |E| = the number of edges
  - And generates **<u>optimal</u>** (cheapest) paths/plans!



Would this algorithm be a solution?

Dijkstra's Algorithm: Analysis

- Blocks world, 400 blocks initially on the table, goal is a 400-block tower
  - Given that all actions have the same cost,
    Dijkstra will first 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
    1630569839078931058645796793733472877564594841634782672258624197623042639942079976642582139557665811636541371181
    631192204882263831691616483204594902834106357987452326989711329392844798003040966743549740387225888734809637192
    406427243636291547266329397641772360103156941486368193342172528364140014872776180029666087610370180877694906148
    478874187444026062261348039369352335684180559503711853518371405485159494313093138752108278889433371136136609283
    180862996179538929537220067341589332765764704756406073917010260309590403035481742212740523295796377736587224525
    4973845940445258650369316934 [illegible] 09127548532657959091134440844417556642117962
    7432025699299231777374983037 [illegible] 4882657444844563187930907779661572990289194
    8105852178191464766293002336 [illegible] 1372350568748665249021991849760646988031691
    3943865511941711933331440315 [illegible] 1302649432305620215568850657684229678385177
    7253589339861121273524529880 [illegible] 3087201742432360729162527387508073225578630
    777685901637435541458440833878709344174983977437430327557534417629122448835191721077333875230695681480990867109
    051332104820413607822206465635272711073906611800376194410428900071013695438359094641682253856394743335678545824
    320932106973317498515711006719985304982604755110167254854766188619128917053933547098435020659778689499606904157
    077005797632287669764145095581565056589811721520434612770594950613701730879307727141093526534328671360002096924
    483494302424649061451726645947585860104976845534507479605408903828320206131072217782156434204572434616042404375
    211052324038225805405713157329159846351931265562731096039371882295044 00

1.63 * 10^1735

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 **very fast**!
  - 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 **multiple cores**!
  - 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

Hopeless? No: We need informed search!