

# Developing Autonomous Systems in Artificial Intelligence: Solvers and Learners

**Blai Bonet**

**Universidad Simón Bolívar, Venezuela**

**UC3M. April 2019**



# Challenge

**How to develop agents that make decisions  
on their own?**

# Autonomous behaviour in AI

**Control problem:** at each decision time, select next **action to execute**

Three main approaches:

- **Programming-based:** specify control by hand in form of program

*Pros:* domain-knowledge easy to express

*Cons:* cannot deal with situations not anticipated by 'programmer'

- **Model-based (top-down):** synthesis of control from specification

*Pros:* flexible, clear, and domain-independent

*Cons:* need a model (specification), computationally intractable

- **Learning-based (bottom-up):** learn control from experience

*Pros:* does not require much knowledge in principle

*Cons:* right features needed, incomplete inf. is problematic, slow learning

# Model-based approach: What's in the model?

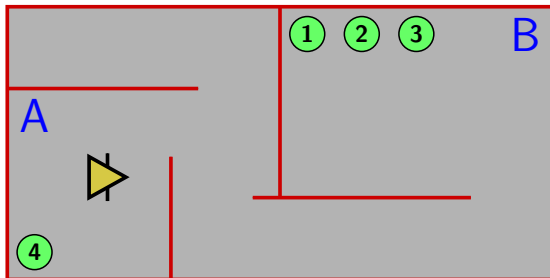
Model of the world (environment) captures:

- Variables make up states; e.g. position of agent(s) and objects, battery life, relations between objects, etc.
- Initial state of the system
- Goals of the agent
- Executable actions (may differ at each state)

It also captures **assumptions about the environment**:

- Deterministic vs. stochastic effects of actions
- Fully vs. partially observable variables
- Cost vs. reward setting
- ...

## Example: Gripper

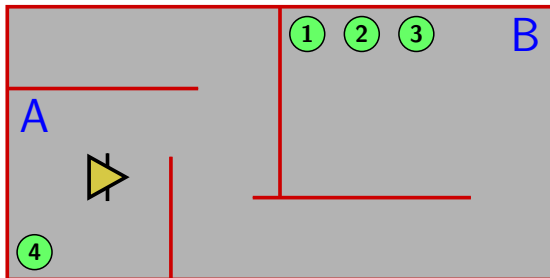


- Bunch of balls in room  $B$
- Robot with left and right gripper, each one may hold a ball
- **Goal:** move all balls to room  $A$

Robot may:

- move between rooms  $A$  and  $B$ ; e.g.  $\text{Move}(A, B)$
- use grippers to pick and drop balls from rooms; e.g.  $\text{Pick}(\text{left}, b_3, B)$

## Example: Gripper



Variables:

- robot's position: room  $A$  or  $B$
- position of each ball  $b_i$ : either room  $A$  or  $B$ , or left or right gripper

States: **valuation for vars** ( $\#states > 2^{n+1}$  for problem with  $n$  balls)

Actions:

- **deterministic transition function**: from state to next state
- may have preconditions; e.g. can drop ① in  $A$  only if at  $A$  and holding it

# Classification of Models

Models classified depending on the assumptions:

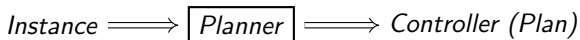
- **Classical planning:** one agent, full sensing (information), det actions
- **FOND/MDP (planning):** one agent, full sensing, non-det actions
- **POND/POMDP (planning):** one agent, partial sensing, non-det act.
- **DEC-POMDP:** multiple collaborative agents in non-det and partially observable setting: joint reward, decentralized partial sensing, and actions
- **Partially Observable Stochastic Games (POSG):** multiple (perhaps competing) agents with decentralized rewards, sensing, and actions

**As model becomes more general, synthesis becomes harder!**

# Planners

A planner is a **solver** for a **class of models**

- input is specification of task (e.g. gripper)
- output is controller for task (e.g. sequence of steps in gripper)



Instance described with **specification language**

**Class of models:** models that can be described with language

The planner must solve **any task** in the class of models



# AI research today

Recent work published in top AI conferences and journals are on:

- Machine Learning
- Natural Language
- **Probabilistic Reasoning**
- Vision and Robotics
- **SAT and Constraints**
- **Search and Planning**
- Multi-Agent Systems
- ...

Highlighted often considered **techniques**, but often more convenient to think of them in terms of **models + solvers** (also applies to many things in ML)

# Models + Solvers

$$Problem \implies \boxed{Solver} \implies Solution$$

**Solver:** program that solves input problems

**Input:** any problem that belongs to **well-defined class** of problems

**Output:** solution to input

- Input specified in some (fixed) **representation language**
- **Scope** of solver is class of all problems that can be specified in language

## Example: Solver for linear equations

$$Problem \implies \boxed{Solver} \implies Solution$$

**Problem:** the age of John is 3 times the age of Peter. In 10 years, it will be only 2 times. How old are John and Peter?

**Expressed as:**  $J = 3P$  ;  $J + 10 = 2(P + 10)$

**Solver:** Gauss-Jordan (variable elimination)

**Language:** Sets of linear equations

**Solution:**  $P = 10$  ;  $J = 30$

Solver is **general** because it deals with **any instance** of the model

Model is **tractable** (solvable in **cubic time** as a function of number of variables and equations in the model); AI models are **intractable** ...

## Example from AI: Solvers for SAT

$$\text{CNF instance} \Longrightarrow \boxed{\text{SAT Solver}} \Longrightarrow \text{Solution}$$

**SAT** is the problem of determining whether there is a **truth assignment** that satisfies a **set of clauses** (disjunction of literals); e.g.

$$\underbrace{x \vee z}_{\text{clause}} ; \underbrace{\neg y \vee \neg z \vee w}_{\text{clause}} ; \underbrace{\neg x \vee z}_{\text{clause}} ; \underbrace{\neg w \vee \neg z}_{\text{clause}}$$

Satisfying truth assignment:  $w = \text{false} ; y = \text{false} ; z = \text{true} \quad (x = \text{any})$

SAT is **NP-Complete**: **worst-case** running time of SAT solvers is **exponential** in number of variables (100 vars:  $2^{100} \approx 10^{30}$ )

However, SAT solvers tackle problems with **thousands of variables and clauses**, and they are widely used (verification, CAD, etc)

# Complexity of decision problems

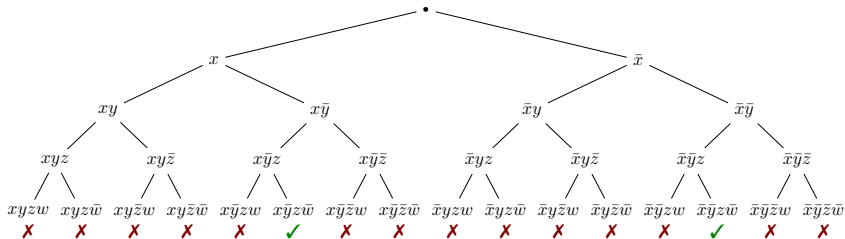
**Decision problems** classified in terms of how many resources (time and space) they require to be solved (resources measured in terms of **input size**):

- **LOGSPACE** = logarithmic space (fully tractable)
  - **P** = polynomial (deterministic) time (“theoretically” tractable)
- 
- **NP** = non-det poly time (exponential time simulation in det computer)
  - **PSPACE** = poly space (using poly space, computer may run by exp time)
- 
- **EXP** = exponential (deterministic) time (provably intractable)
  - **NEXP** = non-deterministic exponential time (provably intractable)
  - **EXPSPACE** = exp space (computer may run by double exp time) (idem)
  - **EXP2** = doubly exponential (deterministic) time (provably intractable)
  - ... hierarchy continues ...

# How SAT solvers do it?

**Formula in CNF:**  $\{x \vee z, \neg y \vee \neg z \vee w, \neg x \vee z, \neg w \vee \neg z\}$

One approach: **Brute-force search**



# How SAT solvers do it?

**Formula in CNF:**  $\{x \vee z, \neg y \vee \neg z \vee w, \neg x \vee z, \neg w \vee \neg z\}$

Another approach: **Inference (resolution)**

$$\begin{aligned}(x \vee z) \wedge (\neg x \vee z) &\implies z \\ z \wedge (\neg w \vee \neg z) &\implies \neg w \\ z \wedge \neg w \wedge (\neg y \vee \neg z \vee w) &\implies \neg y\end{aligned}$$

**Brute-force search and inference are both correct, but don't scale up!**

# How SAT solvers do it?

Modern SAT solvers combine **search** with **inference**

Two types of **efficient inference** at every node in search tree:

- Unit propagation (UP)
- Conflict-based clause learning (CDCL) to implement **non-chronological backtracking** and **re-starts**

**Main lesson:** right thing isn't only about correctness; efficiency is crucial!

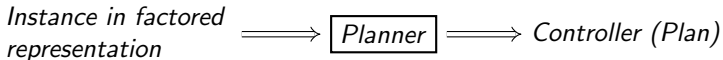


# Specification of models

Models specified using **representation languages**

These languages are **factored languages**

They permit specifications of very large systems (i.e. very large number of states) using very few symbols (bytes)



## Example: Gripper in PDDL

```
(define (domain gripper)
  (:predicates (room ?r) (ball ?b) (gripper ?g) (at-robby ?r) (at ?b ?r)
    (free ?g) (carry ?o ?g))

  (:action move
    :parameters (?from ?to)
    :precondition (and (room ?from) (room ?to) (at-robby ?from))
    :effect (and (at-robby ?to) (not (at-robby ?from))))

  (:action pick
    :parameters (?b ?r ?g)
    :precondition (and (ball ?b) (room ?r) (gripper ?g) (at ?b ?r) (at-robby ?r) (free ?g))
    :effect (and (carry ?b ?g) (not (at ?b ?r)) (not (free ?g))))

  (:action drop
    :parameters (?b ?r ?g)
    :precondition (and (ball ?b) (room ?r) (gripper ?g)
      (carry ?b ?g) (at-robby ?r))
    :effect (and (at ?b ?r) (free ?g) (not (carry ?b ?g))))
)

(define (problem p1)
  (:domain gripper)
  (:objects A B left right b1 b2 b3)
  (:init (room A) (room B) (gripper left) (gripper right) (ball b1) (ball b2) (ball b3)
    (at-robby A) (at b1 B) (at b2 B) (at b3 B) (free left) (free right))
  (:goal (and (at b1 A) (at b2 A) (at b3 A))))
```

# Complexity of planning

Given problem represented in **factored form**, the task of checking whether there is a solution is:

- **Classical planning**: **PSPACE**
- **FOND planning / MDPs**: **EXP** for FOND / **P** for **explicit** MDPs
- **POND planning / POMDPs**: **EXP2** for POND / **PSPACE** for explicit and bounded horizon, **UNDECIDABLE** for unbounded horizon
- **DEC-POMDPs**: **NEXP** for explicit and bounded horizon, **UNDECIDABLE** for unbounded horizon
- **Partially Observable Stochastic Games (POSG)**: at least as difficult as **NEXP(NP)** for explicit and bounded horizon

As model becomes more expressive, task becomes more difficult to solve!

# Beyond classical planning

**Classical planning works:** solvers able to solve large problems

Model is simple, but useful:

- actions may be non-primitive (e.g. abstractions of procedures)
- **closed-loop replanning** able to cope with some uncertainty and limitations

Inherent limitations:

- can't model **general uncertainty on outcome of actions**
- can't deal with **incomplete information** (partial sensing)

Two ways of handling limitations:

- **extend scope** of current solvers (replanning, translations, etc)
- develop **new solvers for more expressive models**

# Online planning in reward-based video games

Methods and ideas from classical planning lifted to play Atari games:

- Collection of deterministic video games that share: screen of  $160 \times 210$  pixels, each up to 128 colors, and 18 actions (joystick movements)
- In majority of games there is no fixed goal; goal is to max accrued reward
- Online game that runs at 60 fps: decisions must be taken at a rate of 4 decisions per second (at least)
- No explicit model, but **simulator is available**

Given **time budget** for choosing next action to apply:

- Use simulator to **explore search tree** of screens within time bound (tree grows exponentially fast:  $18^d$  nodes at depth  $d$ )
- **Select** “best sequence of actions” in explored tree
- Apply **first action** in selected sequence, and **repeat**

## Demo: Playing Atari games

- 60 frames per second, decision made every 15 frames
- Time budget of 1/2 second per decision (almost real time)
- Play for at most 5 minutes or 18k frames
- Screen  $160 \times 210$  pixels of 128 colors each:
  - split into  $16 \times 14$  disjoint tiles
  - $\sim 28k$  features tell which colors contain each tile
  - $\sim 6.8m$  features for relative dist. between tiles with 2 given colors
  - $\sim 13.7m$  features for rel. dist. btw tiles with 2 col. at curr. and prev. screens
  - total of  $\sim 20.5m$  features
- Games in demo:
  - Space invaders
  - Boxing
  - Breakout

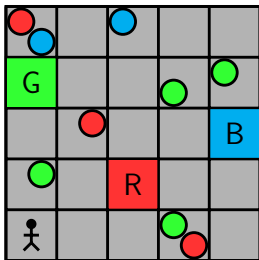
# Agent with partial information

Agent has **partial information** when it doesn't see **full states**

General way to model such problems:

- finite set  $\Omega$  of **observable tokens**
- **environment produces** a token after applying action
- agent **receives token** (it doesn't see state directly)
- token may depend on **current state** and **last applied action**

## Example: Delivering colored balls



Agent knows its position and senses existence of balls in current cell

Observable tokens  $\Omega = \{000, 001, 010, \dots, 111\}$  (i.e. 3-bit word)

- 1st bit tells whether there is a **red ball** in current cell
- 2nd bit tells whether there is a **green ball** in current cell
- 3rd bit tells whether there is a **blue ball** in current cell



# Belief states and information tracking

Agent keeps track of states that are possible given executed actions and observed tokens

Set of possible states is called **belief state**

The initial belief state is  $b_0$  (all possible initial configurations)

Afterwards,

– **belief after executing action  $a$ :**

$$b_a = \{s' : s' \text{ is poss. successor after doing } a \text{ in } s \in b\} \quad (\text{progression})$$

– **belief after executing action  $a$  and receiving token  $z \in \Omega$ :**

$$b_a^z = \{s' \in b_a : s' \text{ is compatible with token } z\} \quad (\text{filtering})$$

# Solvers for problems with partial information

Solvers need to address two fundamental tasks (both intractable):

- information tracking (maintenance of belief states)
- action selection for achieving goal

First task is about how information is tracked, and the **form of solutions**:

- solution is mapping from belief states into actions

Second is about how beliefs, search and inference are combined to find solutions

# Factored belief tracking

Number of system states is exponential in number of variables (e.g. with 9 balls, more than  $9 \times 10^{14}$  states)

Number of belief states is exponential in number of states; that is, **doubly exponential** in number of variables

Factored belief tracking (FBT) is algorithm for efficient **representation and update** of belief states (even at the expense of doing approximations)

FBT decomposes the task into **smaller subtasks** that are tracked **independently** while enforcing some degree of **consistency** among them

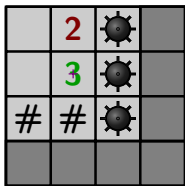
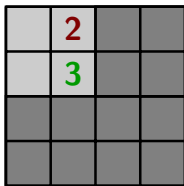
## Example: FBT in Minesweeper

	2	⚡	⚡
	3	⚡	4
1	3	⚡	2
1	⚡	2	1

		+	

- In  $4 \times 4$  board with 5 mines, there are 4,368 different configurations
- Number of belief states (i.e. non-empty subsets of states) is  $2^{4,368} - 1$
- FBT tracks possible contents of  $3 \times 3$  sub-grids **independently**
- Consistency among “local factors” is (approximately) enforced
- **Implemented in linear space + (amort.) const. time per decision**

## Example: FBT in Minesweeper



- In  $4 \times 4$  board with 5 mines, there are 4,368 different configurations
- Number of belief states (i.e. non-empty subsets of states) is  $2^{4,368} - 1$
- FBT tracks possible contents of  $3 \times 3$  sub-grids **independently**
- Consistency among “local factors” is (approximately) enforced
- **Implemented in linear space + (amort.) const. time per decision**

# Demo: Factored belief tracking

State-of-the-art agent for:

- Minesweeper
- Battleship

Both problems are difficult:

- Corresponding decision problems are **NP-Complete**
- Challenge for solvers is to do good belief tracking
- Once beliefs are correctly handled, decision making is trivial!
- Belief tracking done with FBT

# Autonomous behaviour in AI

**Control problem:** at each decision time, select next **action to execute**

Three main approaches:

- **Programming-based:** specify control by hand in form of program

*Pros:* domain-knowledge easy to express

*Cons:* cannot deal with situations not anticipated by 'programmer'

- **Model-based (top-down):** synthesis of control from specification

*Pros:* flexible, clear, and domain-independent

*Cons:* need a model (specification), computationally intractable

- **Learning-based (bottom-up):** learn control from experience

*Pros:* does not require much knowledge in principle

*Cons:* right features needed, incomplete inf. is problematic, slow learning

# (Supervised) Learners

$$x \implies \boxed{\text{function } f_{\theta}(\cdot)} \implies f_{\theta}(x)$$

- $x$  is input; it may be a **high-dimensional vector**
- function  $f_{\theta}$  has **fixed structure** but values determined by parameters in  $\theta$
- output  $f_{\theta}(x)$  may be vector

## Supervised learning:

- Given **anotated** (i.i.d.) sample  $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$
- Find parameter  $\theta^*$  that is **“right fit”**:
  - low “training error”; e.g., low  $\frac{1}{2m} \sum_{i=1}^m (y_i - f_{\theta^*}(x_i))^2$
  - good “generalization”; e.g., low  $\mathbb{E}_{(X,Y) \sim (x_i, y_i)} [(Y - f_{\theta^*}(X))^2]$
- Standard algorithm: **Stochastic gradient descend** (SGD)



# Deep Learning / Deep Reinforcement Learning

$$x \implies \boxed{\text{function } f_{\theta}(\cdot)} \implies f_{\theta}(x)$$

- $f_{\theta}$  is neural network with many **hidden layers**
- number of parameters in  $\theta$  is often quite large
- **DL is about classification:**  $x$ s are objects,  $y$ s are class labels
- **DRL is about control:**  $x$ s are “state features”,  $y$ s are actions (or state values)

Training:

- Network trained with many examples during hours/days using SGD
- DL: training data often available (e.g. tagged images/videos from internet)
- DRL: often insufficient training data
- Once trained, evaluation is fast

# Success and limitations of learners

Big success of deep nets in AI:

- Breakthroughs in image/video understanding, speech recognition, ...
- Superhuman performance in complex games like Chess and Go

Key limitation of trained nets is **fixed input size**:

- No problem for images, Chess or Go, as they correspond to “fixed-size inputs”
- Problematic in simpler domains; e.g. gripper with different # of balls or grippers (there is no deep net able to solve gripper with arbitrary # of balls)

# Solvers vs. learners

Rollout IW **planner** and DeepMind's DQN **learner** perform well in Atari

They illustrate common characteristics of solvers and learners:

- Rollout IW works from scratch for any game without prior pre-processing. It calculates action to apply by doing exploration under given time budget.

Solvers “**think before acting**” at each decision point

- DQN requires a lot of training over many data. Afterwards, DQN plays very fast as decision making is simply to “run network on given input”.

Learners require a lot of training and tuning. Then, decision making is fast

On the other hand:

- Solvers require models or, in some cases, simulators with given capabilities
- Learners require large training sets and pre-processing time

# Integration of solvers and learners

Better agents by combining solvers and learners

Training data for DRL can be automatically generated using solvers:

- Starting from sample  $\mathcal{M}_i$  for  $i = 0$ :
  - Train DRL function  $f_{\theta_i}$  with sample  $\mathcal{M}_i$
  - Use **self-play** or **stochastic search** to generate new “interesting inputs”  $\{x_k\}_k$
  - Use solver (possibly enhanced by  $f_{\theta_i}$ ) to find labels  $\{y_k\}_k$  for such inputs
  - Make new sample  $\mathcal{M}_{i+1} = \{(x_k, y_k)\}_k$ , increase  $i$ , and **repeat**

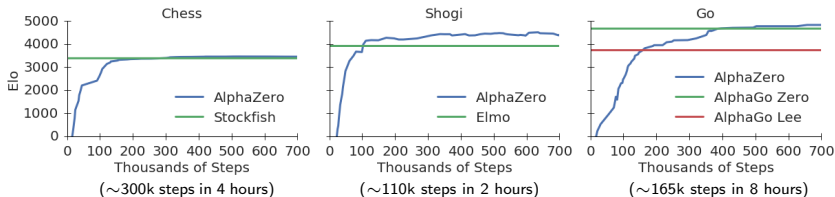
Models can be learned and then fed to solvers:

- From “observed behavior”, use learner to build model
- Use solver to find controller for model
- Use controller to generated new behaviour, and **repeat**

# Solvers + Learners: AlphaZero

DeepMind's AlphaZero:

- **Superhuman performance** in Chess, Shogi (Japanese Chess), and Go
- Learns how to play using only game rules and self play
- Combines DRL with Monte-Carlo Tree Search (MCTS)
- Deep net trained with SGD; it provides evaluation function  $f_\theta$  for states
- Using  $f_\theta$  and MCTS, AlphaZero does self-play to generate new training data

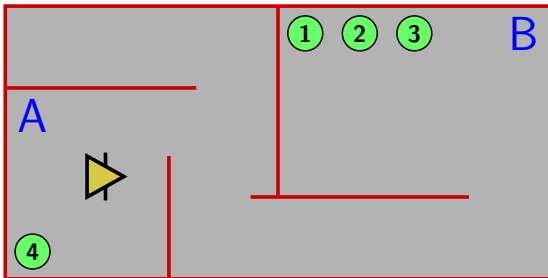


# Solvers + Learners: Generalized planning

Synthesis for **classes** of similar problems rather than for single instances

- **Learn** general abstraction for whole class
- **Solve** abstraction with solver (may need FOND solver)
- Obtained policy works for any instance in class

## Example: General abstraction for Gripper



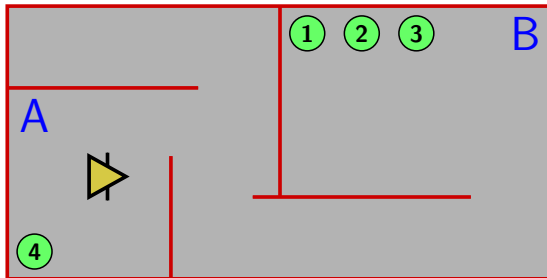
Features:

- $X$  = robot-at-destination-room,  $B$  = #balls-in-other-room
- $C$  = #balls-being-carried,  $G$  = #free-grippers

Abstract actions:

- **Pick** =  $\{\neg X, B > 0, G > 0\} \mapsto \{B\downarrow, G\downarrow, C\uparrow\}$
- **Drop** =  $\{X, C > 0\} \mapsto \{C\downarrow, G\uparrow\}$
- **Leave-destination-room** =  $\{X, C = 0, G > 0\} \mapsto \{\neg X\}$
- **Go-destination-room-fully-loaded** =  $\{\neg X, G = 0, C > 0\} \mapsto \{X\}$
- **Go-destination-room-half-loaded** =  $\{\neg X, B = 0, G > 0, C > 0\} \mapsto \{X\}$

## Example: General abstraction for Gripper



Features:

- $X$  = robot-at-destination-room,  $B$  = #balls-in-other-room
- $C$  = #balls-being-carried,  $G$  = #free-grippers

Starting with  $X = true$ :

**While**  $B > 0 \vee C > 0$  **do**:

**While**  $X \wedge C > 0$  **do**: **Drop**

**If**  $X \wedge B > 0$  **then**: **Leave**

**While**  $B > 0 \wedge G > 0$  **do**: **Pick**

**If**  $C > 0 \wedge G = 0$  **then**: **Go-destination-room-fully-loaded**

**Elsif**  $C > 0 \wedge G > 0$  : **Go-destination-room-half-loaded**



# Current and future work

- Learners:
  - Learn states, variables and objects from traces
  - Learn abstractions from (learned) state representation
  - Learn abstractions from non-symbolic traces (images)
- Solvers:
  - Devise more efficient solvers
  - Identify tractable subclasses of interesting problems
- Combine learners and solvers into “full pipeline”: from (symbolic or non-symbolic) traces to generalized controllers

(trace = interleaved sequence of actions and observations:  $\langle a_0, z_0, a_1, z_1, \dots \rangle$ )

# Learning state-transition diagrams from traces

ST-diagram: **Labeled directed graph**

- **vertices** are states
- **labeled arrows** are transitions triggered by actions

Learning done by solving **graph coloring** problem with SAT

# Example: Learning ST-diagram in Gripper

**Observable tokens:** 3-bit strings with components for  $B > 0$ ,  $C > 0$ ,  $G > 0$  denoted by b, c, and g

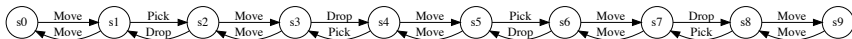
**Action labels:** Move, Pick, Drop

**Traces (sample):** 1,510 traces from single instance with 2 balls and 1 gripper

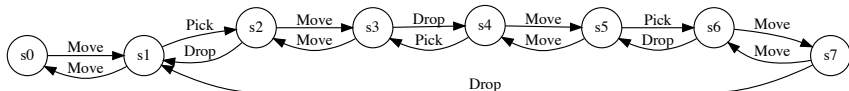
```
b-g Move b-g Move b-g Move b-g Move
b-g Move b-g Move b-g Move b-g Move b-g
b-g Move b-g Move b-g Move b-g Move b-g Move
b-g Move b-g Pick bc- Drop b-g Pick
b-g Move b-g Pick bc- Drop b-g Pick bc-
b-g Move b-g Pick bc- Drop b-g Pick bc- Drop
b-g Move b-g Pick bc- Drop b-g Pick bc-
b-g Move b-g Pick bc- Move bc- Drop b-g Move b-g Move
b-g Move b-g Move b-g Move
b-g Move b-g Move b-g Move b-g
b-g Move b-g Pick bc- Move bc- Drop b-g Move b-g Move b-g
b-g Move b-g Pick bc- Move bc- Drop b-g Move b-g Move b-g Move
b-g Move b-g Pick bc- Move bc- Drop b-g Move b-g Move b-g Move b-g
b-g Move b-g Pick bc- Move bc- Drop b-g Move b-g Move b-g Move b-g Move
b-g Move b-g Pick bc- Move bc- Drop b-g Move
...
```

# Example: Learning ST-diagram in Gripper

**State space:** 10 states, and 18 transitions; “linear shape”



With a smaller sample containing 422 traces only, get **incomplete diagram**:



# Planning and learning at UC3M

Planning and Learning Group (PLG) at Departamento de Informática:

- **Head of group:** Daniel Borrajo Millán
- Carlos Linares López
- Fernando Fernández Rebollo
- Susana Fernández Arregui
- Raquel Fuentetaja Pizán
- Angel García Olaya
- Yolanda Escudero

# Wrap up

- Two main approaches in AI for developing autonomous agents
- Solver-based approach:
  - Automatic synthesis of controller from model
  - Requires model and solver
  - Synthesis task is intractable
  - Usually no pre-processing; at each decision point, the process of decision making requires time
- Learner-based approach:
  - Doesn't require model
  - Requires features and lots of training data
  - Controller is directly learned from data; afterwards, decision making is fast
- Combining the two approaches:
  - Intrinsic limitations of each approach can be overcome
  - Very successful examples
  - Still requires some structure: states, action labels, and observation tokens
- Future work:
  - Learn state structure and control directly from traces