Reinforcement Learning and Automated Planning

Class 3: Specifying Models

Ronen I. Brafman

Department of Computer Science

אוניברסיטת בן-גוריון בנגב
جامعة بن غوريون في النقب
Ben-Gurion University of the Negev

MDPs

$$M = \langle S, A, Tr, R \rangle$$

- S – set of states
- A – set of actions
- $Tr : S \times A \rightarrow \Pi(S)$
  - A $|S| \times |S|$ matrix for every $a \in A$
  - Large and difficult to specify...
- $R : S \times A \rightarrow \mathbb{R}$

Models Can Be Very Large

blocks	states
1	1
2	3
3	13
4	73
5	501
6	4051
7	37633
8	394353
9	4596553
10	58941091

How can we specify such large models?

Deterministic MDPs

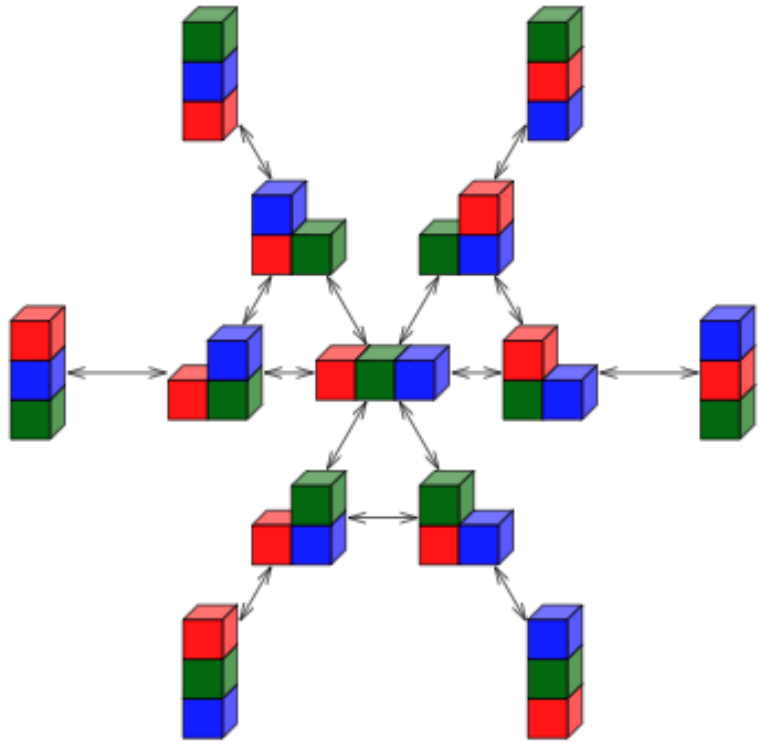
- Suppose Tr is deterministic
- This means that given s and a only one state s' is possible.
- The transition matrix has one non-zero entry in each row
- Tr is simply a function $Tr : S \times A \rightarrow S$
- Assuming $R : S \rightarrow \{0, 1\}$, we are back to finite-state machines
- We still have a problem handling 10 blocks with over 50 million states

Factored Models

- Usually, we cannot control the size of S
- But we can find simple ways to specify it
- And to specify Tr and R
- The key idea is to use state-variables

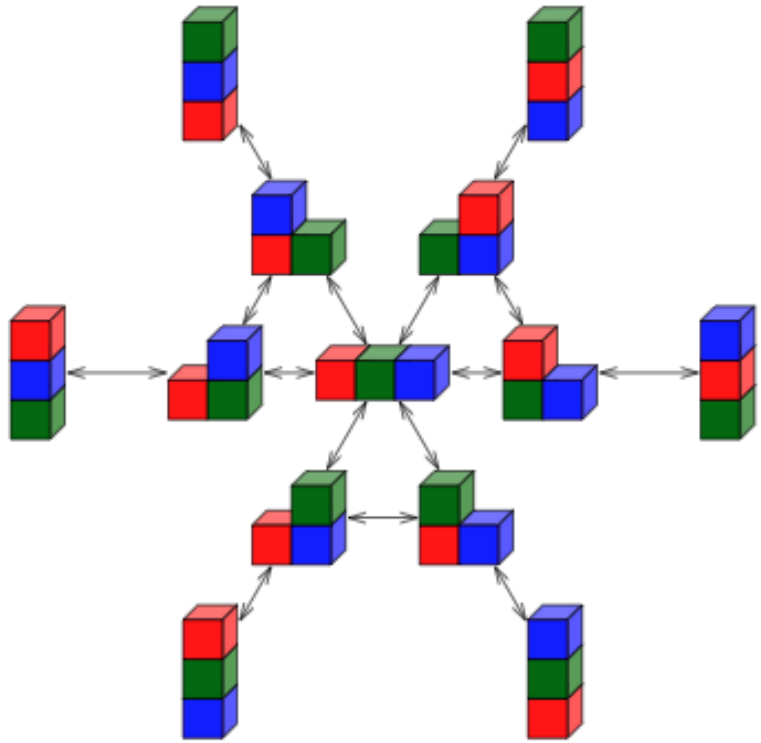
State Variables 1

- A state is usually naturally viewed as specifying the value of state variables (or features)
- What would be natural state variables here?



State Variables 2

- What would be natural state variables here?
- Multi-valued: *On(blue)* = *Table,red,green*, etc.
- Boolean: *On(blue,table)*, *On(blue,red)*, etc.
- S = all possible (or legal) assignments of values to variables



Specifying Deterministic *Tr*

- State variables do not reduce the number of states
- So, in theory, they don't help us reduce *Tr*'s description
  - We still need a $|S| \times |S|$ matrix

- But: consider the action of moving block b_1 from b_2 to b_3
- Which variables does it affect?

Specifying Deterministic Tr

- State variables do not reduce the number of states
- So, in theory, they don't help us reduce Tr's description
  - We still need a $|S| \times |S|$ matrix

- But: consider the action of moving block b_1 from b_2 to b_3
- Which variables does it affect?
- Does it matter how many other blocks there are?

The Frame Axiom

Actions often affect only a few variable

- We can focus on specifying the new values of variables that were impacted by the action
  - If the action is $move(b_1, b_2, b_3)$ and b_1 is on b_2 in s, we can simply write that the effect is $on\text{-}b_1 = b_3$
- *Frame axiom*: what we do not specify does not change
- This allows us to specify the effect of a on s by referring to a very small number of variables
- Viewing each action as a function from S to S, we've reduced the description of the second S

Locality

A variable's new value depends on few other variable values

- Consider again the above example
  - If the action is $move(b_1, b_2, b_3)$ and b_1 is on b_2 in s, we can simply write that the effect is $on\text{-}b_1 = b_3$
- Notice that in this example the effect of the action depends only on one variable in s: $on\text{-}b_1$.
- We can specify the effect of one action on many states, concurrently
  - This takes care of the description of the first S
- We can generalize even farther:
  - If the action is $move(x_1, x_2, x_3)$ we can simply write that the effect is $on\text{-}x_1 = x_3$, where x_i's can be arbitrary blocks
  - Now we're specifying multiple actions at once

Locality

- Consider a different domain in which pupils $p_1, \ldots, p_n$ are sitting on chairs $c_1, \ldots, c_n$. The variables are $x_1, \ldots, x_n$, where x_i denotes the chair p_i is sitting on.
- Consider the action *permute* in which each pupil moves from chair c_i to $c_{i+1(mod\ n)}$
- This action changes the value of all variables
  - So we cannot enjoy the benefits of the frame axiom
- But the effect on each variable depends only on its own current value $x'_i = x_i + 1(mod\ n)$
  - Convention x' denotes the value of x after the action
- There are $n!$ states (assuming pupils always sit), but we can specify *permute* using n equations: $\{x'_i = x_i + 1(mod\ n) | i = 1, \ldots n\}$

Classical Planning

- Invented in the 60's with the first autonomous robots
- Model: finite-state machine
- Everything is specified at the variable level
- Algorithmic challenge: find a sequence of actions that leads us to an accepting state
  - Why is this interesting? We can apply Dijkstra's Algorithm?

Specifying Classical Planning Problems

- States: specify the variables and their possible value
  - Classical planning focuses on Boolean variables (propositions)
  - Blocks world: on(x,y), clear(x)
  - S – all possible assignments
- Actions: specify what variables change
  - Preconditions: propositions that need to be true before we apply the action
  - Effects: new values of propositions that change their value
  - Move(A,B,C): Preconditions – On(A,B),Clear(C). Effects – On(A,C), Clear(A), ¬On(A,B), ¬Clear(C).
- Reward: Conjunction of propositions {On(C,B),On(B,A)}
  - Specifies a set of states
  - Called *Goal states*

Action Schema

- Notice: Move(B,A,C) looks very similar to Move(A,B,C) Preconditions – On(B,A),Clear(C). Effects – On(B,C), Clear(B), ¬ On(B,A), ¬Clear(C).
- Instead of defining a separate action for every choice of three blocks, we can use variables
- This is called an *action schema*
- Move(X,Y,Z): Preconditions – On(X,Y),Clear(X). Effects – On(X,Z), Clear(Y), ¬On(X,Y), ¬Clear(Z).
- The predicate (On(X,Y)) replaces propositions like (On(A,B))
  - We can define types for objects and variables
  - Every assignment of objects to variables (called *grounding*) gives us a proposition On(X,Y) $\rightarrow$ On(A,B)

Classical Planning Domain: Logistics

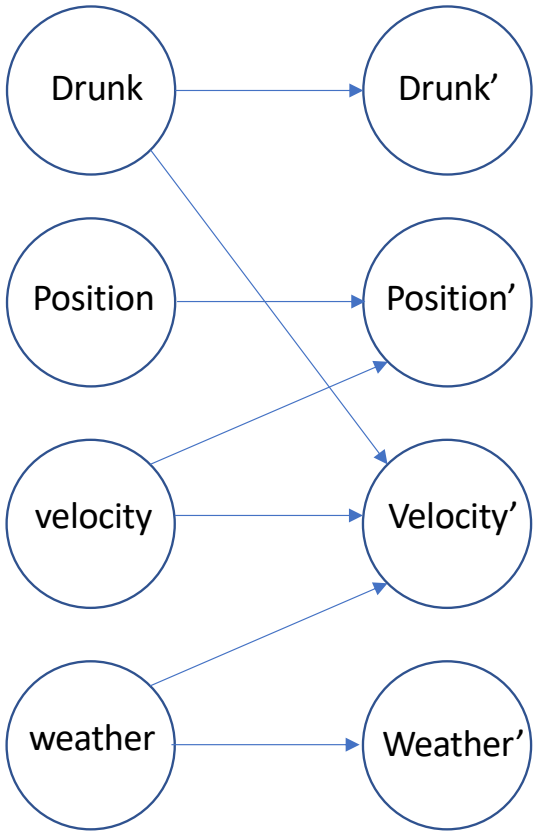
- We have trucks and planes and packages.
- Trucks can move between locations in the same country.
- Planes can move between airports.
- Packages can be loaded and unloaded to/from trucks and planes.
- Actions: moving a truck, flying a plane, loading and unloading a package to a truck or a plane

Logistics Domain

- Fly(X,Y,Z): X is a plane, Y,Z, airports
  - Precondition: At(X,Y)
  - Effect:At(X,Z), ¬At(X,Y)
- Move(X,Y,Z): X is a plane, Y,Z, locations
  - Precondition: At(X,Y)
  - Effect: At(X,Z), ¬At(X,Y)
- Load(X,Y,Z): X is a package, Y a vehicle, Z a location
  - Precondition: At(X,Y), At(Y,Z)
  - Effect: In(X,Y),¬At(X,Y)
- Unload(X,Y,Z): X is a package, Y a vehicle, Z a location
  - Precondition: In(X,Y), At(Y,Z)
  - Effect: At(X,Y),¬In(X,Y)

Example DBN

Action: Drive



Pr(Drunk' | Drunk)

Pr(Postion' | Position, Velocity)

Pr(Velocity' | Drunk, Velocity, Weather)

Pr(Weather | Weather')

Generative Models

- *Tr* specifies the next-state distribution for each action *explicitly*
- We can use it to build a simulator
- A simulator generates the next state (and reward) given the current state with the correct probability
  - All we need is the ability to sample from a distribution
- Models that generate the next state and reward are called *generative models*
  - Such models don't necessarily represent *Tr* explicitly

Generative Models

- Generative models were not popular in the past because solution algorithms needed concrete information about probabilities
  - This information may be hidden in a generative model
    - The simulator can be a black box
    - Or it may implement a complex process from which it is difficult to get the actual probabilities
- Modern methods (including RL, but not only) can use such generative models
- A generative model may provide less information, so why is it useful?

Generative Models

- Generative models were not popular in the past because solution algorithms needed concrete information about probabilities
  - This information may be hidden in a generative model
    - The simulator can be a black box
    - Or it may implement a complex process from which it is difficult to get the actual probabilities
- Modern methods (including RL, but not only) can use such generative models
- A generative model may provide less information, so why is it useful?
- We can describe the model using code
  - Lets us use complex data-structures, loops, functions
  - Can be much easier to write
  - Code can make for more efficient next-state generation